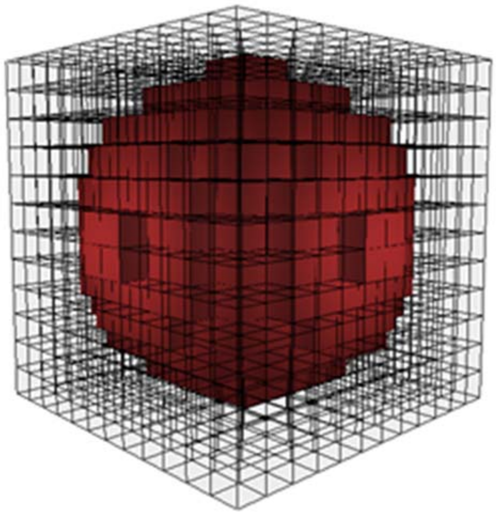
Volumetric rendering and sculpting using WebGL



Name	William Tarrant
T Number	T0001[redacted]
Course	Computing with Games Development
Module	Final Year Project
Supervisor	Dr R Sheehy
Date Submitted	1st May 2014

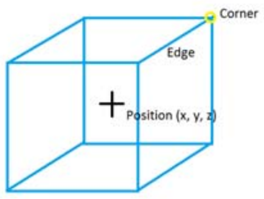
Abstract

With the introduction of WebGL combined with HTML5, feature rich graphical applications that normally only ever resided on desktop machines are now being made possible in the web browser. With web browsers being available as cross platform applications as well being available on mobile devices, the write once, deploy everywhere opportunities are immense.

In this thesis, we aim to explore the possibilities being opened up by this technology by tackling the problem of developing a volumetric rendering engine with the goal of using it as the foundation for a sculpting tool that is accessible by any modern web browser. The volume rendering techniques used will be based on existing volume rendering theory from industry experts but with intent of targeting a cross platform environment including all the main operating systems as well as compatible mobile devices. The purpose of this sculpting application will be to offer the user the ability to manipulate an object that possesses volume as well as having clay like features so that it will mimic “real to life” behaviors. This is an area worth exploring as most modeling applications don’t concern themselves with volume but only the surface mesh. An excellent motive to investigate this area, is exploring the usefulness of volumetric properties of a model, as they may be useful such in areas as 3D printing.

Contents

Terminology

Voxel	Cubic or rectangular element that has a position in space and contains information about itself. In simplest terms it is a 3D pixel. 
2D	Two dimensional (X, Y)
3D	Three dimensional (X, Y, Z)
OpenGL	A multi-platform library that facilitates interaction with the Graphics Processing Unit
OpenGL ES	A subset of OpenGL intended for embedded systems such as mobile phones.
WebGL	Derived from OpenGL ES 2.0 API, WebGL was created for the purposes of rendering graphics in a web browser
Ray casting	A means of projecting a line into the model world and testing for surface intersections
Sculpt	To shape or mould an object
JavaScript	A dynamic scripting language that is part of and interpreted by web browser.
Canvas	An element added as part of HTML5 that allows for the dynamic rendering of 2D graphics
CT	Computed Tomography – medical imaging device used by radiology labs to view internal anatomy. This device operates by having a rotating gantry which has an X-Ray emitter at on side while detectors are located at the opposite side.
MRI	Magnetic Resonance Imaging – medical imaging device used by radiology labs to view internal anatomy. The device uses magnetic fields to temporally alter atoms and measurements are performed as they return to original state. MRI uses no ionizing radiation in contrast to Computed Tomography scans.

1.0 Introduction

In this thesis we aim to demonstrate some of the possibilities being opened since the advent of WebGL and HTML5. To explore these possibilities, we endeavoured to create a 3D modelling application that could either be used for creating renderings or game assets. As modelling applications are not new, with some already being tested as betas for web deployment such as Clara.io, we aim to put a spin on this by doing something which most modelling applications don't do which is encompass volume in there models. Volume rendering is an interesting topic which has offered many benefits in the field of medicine but especially now that 3D printing is becoming more and more popular the possibilities of applying the volume to models makes it a topic worth studying. Combining this theory with WebGL opens further possibilities of making cross platform environment capable of developing a community that can collaborate on, or share their creations.

Volume visualisation is a long established technique that concerns itself with the presenting of empirically sampled data in a more visually appealing manner. This data can be obtained through the use of imagery produced from medical scanners or that of computed geometric models i.e. a mathematical formula for a sphere. Computer Tomography (CT) or Magnetic Resonance Imaging (MRI) scan will produce a series of regularly spaced 2D images; this data can then be visualized in a 3D form by using a technique known as voxelization. The images obtained can be stacked on top of each with a space between each image that matches the space used in the scanning technique to form the basis of a 3D grid. The layers between these images are then subdivided to form a 2 dimensional grid of cubic or rectangular elements that are known as voxels. Voxels can be thought as being the equivalent of a 2D pixel in that it possesses a position and some information about itself (colour) but expanded to a 3D environment. The information stored by these voxels is usually the combined information obtained from images with which the particular voxel is in contact with.

Voxels are then stored in a 3D structure commonly referred to as a volume buffer. The rendering of this volume buffer is what is known as volume visualisation. (Kaufman, Cohen, & Yagel, 1993, p. 51) This technique has proved useful in the area of diagnosis by allowing an extra dimension to physicians with whom they can view a patient's affliction; this issue may not have been so obvious looking at 2D images solely. A case study by Stytz et al, profiled a patient with uncontrollable seizures and it was only later through the use of 3D imaging that the cause was revealed. (Stytz, Frieder, & and Frieder, 1991) Volume visualisation also lends itself to other uses such as representing scientific phenomenon such as fractals, meteorological data or fire which tends to be voluminous. (Kaufman & Mueller, 2003)

While volume visualisation is primarily concerned with the rasterising of sampled data, such as those from medical imaging scans, it is important to form a distinction between it and volume graphics, which is the focus of this thesis. Volume graphics deals with the "manipulation and rendering of volumetric geometric objects". With the great advances in computing technology since the first advent of volume visualisation such as, greater processing power and larger memory, volume graphics is now finding its way into areas of computing graphics that has been dominated by surface graphics, where models are represented by vertices that form a series of triangles. (Kaufman, Cohen, & Yagel, 1993) One of big areas and benefits volume graphics has to offer over surface graphics is for example, terrain generation and or for generating cave structures which is always going to be voluminous in nature. (Cui, Chow, & Zhang, 2011)

Digital sculpting is, in a sense, the digital version of clay sculpting that artists would be familiar with. It allows for the user to be able to perform a manipulation of a structure in 3D space. Many products exist on the market; from the more feature rich Mudbox from AutoDesk to the open source free offering by Blender 3D. (Autodesk, 2013) (Blender Foundation, 2013) While there is a well-established content pipeline for rendering triangle based models, volume graphics is beginning to be recognised as being capable of offering greater geometrical detail when it is required, some commercial packages that currently use a form of volume graphics include 3D-coat and ZBrush who use a volume sculpting technology before the model is converted back to a low poly mesh. (Laine & Karras, 2010)

While most sculpting tools available don't take account of volume, it makes sense when working in a sculpting environment, that the subject or material should be volumetric in nature, similar to how an artist would work with clay on a banding wheel, and it is with this the motivation for exploring this type of application is being exposed with the primary aim being to investigate and present the viability to implement this technique in a volumetric environment. As well as this, it is intended to make this application as accessible as possible by being able to deploy across multiple platforms through the use of the web browser and WebGL technology.

WebGL is a cross-platform, web standard that allows access to the graphics card through a JavaScript API. It is based on the OpenGL ES 2.0 standard and offers 3D programming environment without the need of any plugin through most browsers including Google Chrome, Mozilla Firefox, Safari, Opera and now even Internet Explorer 11. The reason this is the deployment platform of choice is really because of the potential that WebGL opens up the way in which games or online demonstrations can be deployed to multiple platforms, needing only a reasonably modern machine and a compatible web browser. (Khronos, 2013)

Implementing volumetric graphics presents a greater challenge over the traditional polygon rendering and this will from an important section of this thesis, among them is the maintaining of a data structure that is required to track the information on each of the world's voxels, which, depending on the resolution, can involve quite a large data set and because the language being used is JavaScript, providing an efficient algorithm may prove to be more challenging being a lightweight scripting language.

Without rendering, volume graphics would be of little use so part of the investigation is going to be placed on the choice of rendering techniques that can be employed to transform the 3 dimensional datasets to a 2D picture that can be displayed on a user's monitor. Again, as the choice of deployment (WebGL) has already been chosen, the choice of rendering technique may be limited depending on the capability of what the latest release of WebGL has to offer.

2.0 Data visualisation

2.1 Overview

This chapter's aim is to begin to introduce the roles data visualisation has played in various sectors including those of engineering and scientific.

2.2 What is data visualisation?

There are many situations where data on its own, especially empirical data, can be difficult to interpret. But by employing a visualisation technique into their studies; it can help Scientists, Engineers and doctors, with better understanding what their observations mean by now being able to view their data in multiple dimensions on a screen.

Some examples of data that can benefit from data visualisation include the engineering sector such as data obtained from site surveys; others can include numerical computations or the study of fluid dynamics. Other examples include the study of stresses and strains on a material or even that of weather models.

Next a selection of data visualisation categories, as identified from a paper by Schroeder et al, will be examined. The selection will include scalar, vector and tensor data. Taking each one in step, examples will be presented that will demonstrate how various different algorithms can be applied depending on the data being modelled. (Schroeder & Martin, 2005)

2.2.1 Scalar

Scalar data can be simply thought as being a point in space having a value associated with, for example (x, y, z, v) where v represents the value. This value can be anything from a temperature, density or pressure. Discussed next are some techniques which can be applied to the data to aid in its interpretation will now be discussed.

2.2.1.1 Contouring

Contouring is a great example of visualising data and in the example presented below, data, taking an example of a site survey where spot heights levels are taken at 10m intervals. The information depending on the equipment used will store and present the data in single dimension format i.e. an array. This can prove to be difficult to interpret on its own without the aid of some form of visualisation. So taking the survey data example, if contouring is introduced by performing bilinear interpolation by picking a value and calculating where it lies between two data points (if at all) and marking that point.

Below is illustration of how sampled data from a survey can then be converted using the algorithm F(x) = c, where c is the contour value of interest. (Schroeder & Martin, 2005)

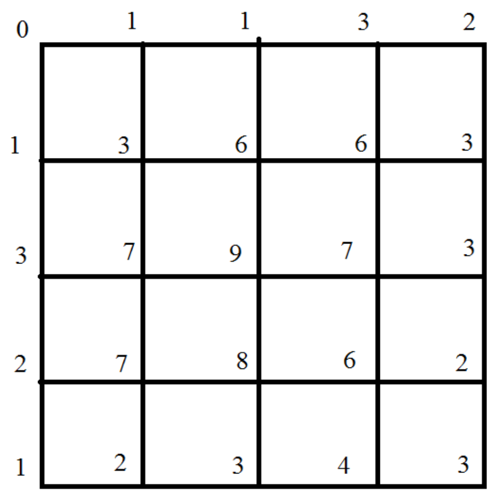


Figure 1 Sampled data on a regular spaced grid

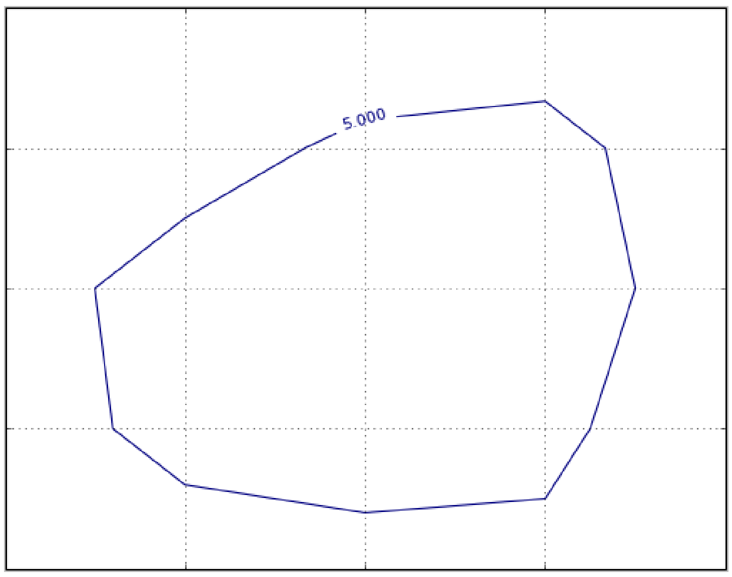


Figure 2 - Visualising scalar data with MathPlotLib (Hunter, 2007)

Contouring will be examined in more detail in the next chapter when the Marching Cube Algorithm is discussed. It forms an important role when it comes to contouring and rendering volumetric structures in 3D.

2.2.1.2 Colour mapping

Colour mapping is a visualisation technique where scalar values are mapped to a colour. A typical use for this could be the study of temperature in fluids dynamics; such a case may be the flow of heated water through a circulation pump.

Colour mapping works by simply taking a scalar value and matching it to a colour value stored in a lookup table. This table may be constructed from Red, Green, Blue and transparency values.

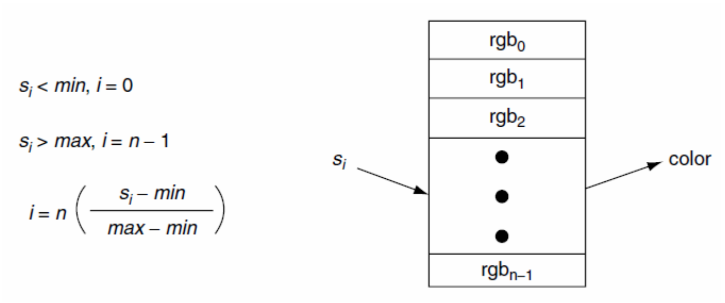


Figure 3 - Scalar to Colour mapping (Schroeder & Martin, 2005)

The key to successful colour mapping is to have an effective lookup table that emphasizes the important features. (Schroeder & Martin, 2005) However in an article by Borland et al, they cite using a rainbow colour mapping scheme as being misleading and are suggesting it should be deprecated. The reasons they cite is that it can obscure the results. They present the scenario that when asked to put colours in order, it is not a cut and dry task. Despite this when it comes to offering suggestions for an alternative, it comes down to the task in hand in the data being presented but they remain highly critical of the rainbow spectrum and wishes that the visualisation community would avoid it. (Borland & Taylor, 2007)

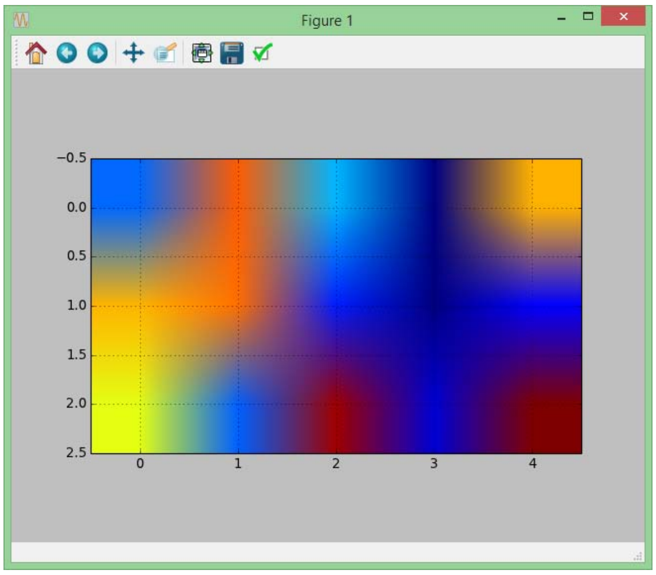


Figure 4 - Colour mapping of an array

2.2.2 Vector field visualisation

Vectors are quantities that represent both direction and magnitude. Having the ability to visualise this form of data makes them suitable for studying vector fields like weather systems or fluid dynamics. When plotting vectors, the length of the vector will represent its magnitude. A scaling factor is usually applied (perhaps based off the vector with the largest magnitude) to ensure that all the vectors can be displayed in the image. Issues that can rise from this are how a particular vector will present itself when projected from the 3D world to a 2D image, so that its true representation is not lost in the projection I.e. scale. (Schroeder & Martin, 2005)

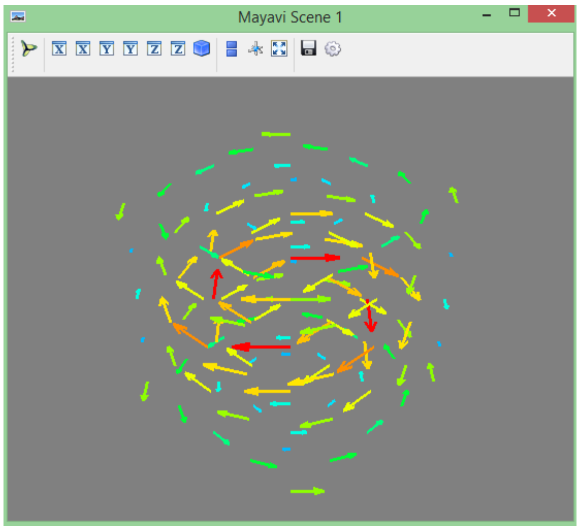


Figure 5 - Visualisation of a vector field using Mayavi

2.2.3 Tensor Visualisation

To analyse the impact stress and strain can have on material, tensors are employed. The issue with dealing with tensors is that they have the same complexity as matrices. By being able to visualise tensor data, the understanding of the impacts of loads can be easier to determine. Stresses and strains are a source of material failure and so by providing a technique to better understand it means it is providing an important tool that can be used from medical or engineering standpoint. (Wunsche, 1999)

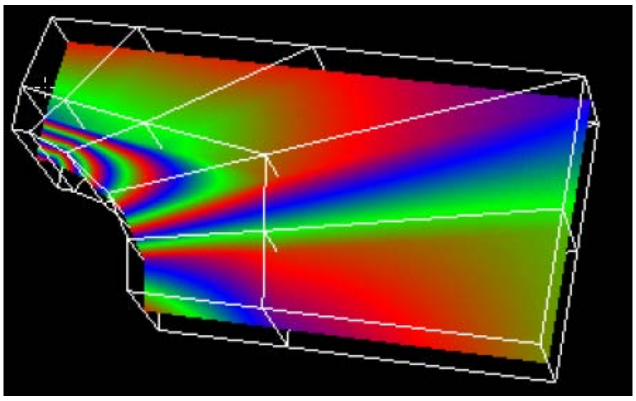


Figure 6 - Using a cyclical colour map to illustrate the maximum principle stress in a holed material (Wunsche, 1999)

3.0 Volume Rendering

3.1 Introduction to volume rendering

The process of three dimensional rendering is a long established technique, having its origins in the 1970's, the Mayo clinic being the early pioneers into its research. Its intent was to solve the problem of information overload which can result from the enormous amount of data presented by a Computer Tomography (CT) scans. (Calhoun BFA, Kuszyk MD, Heath PhD, Carley BS, & Fishman MD, 1999)

In the mid 1980's, now with more advanced technology available at researchers disposal, research into the area of three dimensional rendering was well underway. Some of the leading research was coming from both the University of North Carolina and Pixar. Pixar's intention was to develop a commercially available medical imaging system. When Pixar finally released its technology it was considered to being far too advanced for its time as well as being considerably expensive. (Kurachi, 2011) (Calhoun BFA, Kuszyk MD, Heath PhD, Carley BS, & Fishman MD, 1999)

To illustrate the potential offered by using 3D rendering of medical imaging, Stytz et al, in their paper describe a case where having 2D alone wasn't sufficient in diagnosing the ailment of a patient when they presented with having uncontrollable seizures. In this case they were able to piece together the images obtained from MRI scans and combine it with the results of a PET (Positron Emission Tomography) scan to find the source of the patient's issue. The value of this technique of diagnosis did not just end there but was also instrumental when it came to surgery as the surgeons were able to perform a rehearsal of the craniotomy and produce images that they could later use in the surgery to help guide them. (Stytz, Frieder, & and Frieder, 1991)

The process behind 3D rendering of medical imaging scans involves taking a series of continuous images along an axis, and then stacks each on top of each other to form a volume. The final step is the filling of the void that exists between the images with a 3D matrix of voxels. It is the interpretation of these voxels and presenting a 2D rendered image is the essence of what volume rendering is all about. The research into this area has produced a number of techniques which include Shaded Surface Display, Maximum Intensity Projection and more recently Volume rendering. (Calhoun BFA, Kuszyk MD, Heath PhD, Carley BS, & Fishman MD, 1999)

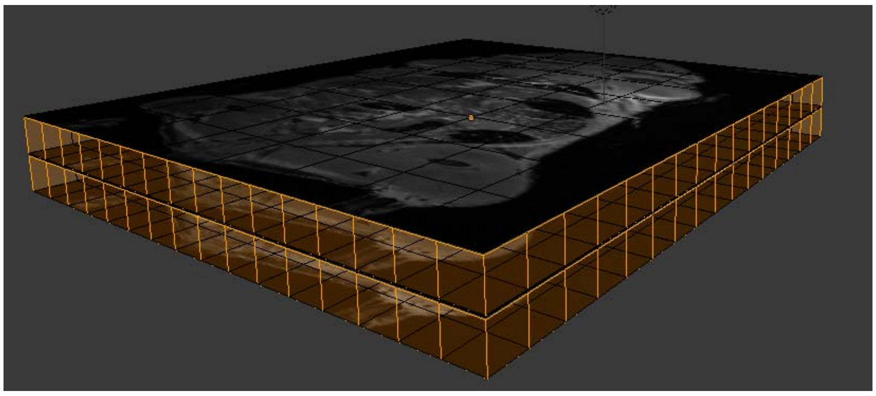


Figure 7 - Simple illustration of stacked images with a matrix of voxels between image slices.

3.1.1 Shaded Surface Display (SSD)

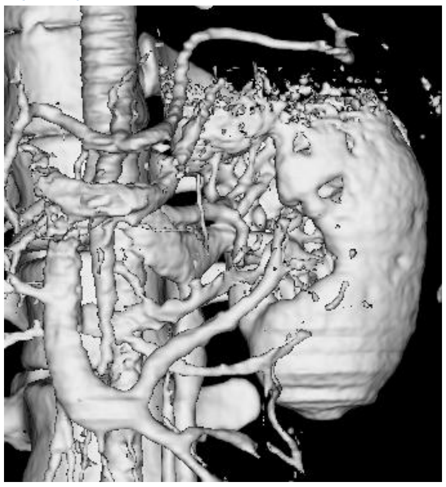


Figure 8 SSD render of a patients left kidney

(Sato PhD, Shiraga MD, Nakajima MD PhD, Tamura PhD, & Kikinis MD, 1998)

This was one of the earliest techniques developed; it worked by abstracting the object of interest by means of thresholding. Different tissue types were assigned different values, a high and low attenuation threshold value. In essence this then became a binary operation in that the area being sampled either contains the tissue or it doesn't. (Siegel, 2008)

The three main steps involved include firstly, the acquiring of the actual image. The second step is the segmentation as mentioned above, involves thresholding to determine foreground objects from the background objects. The final step in the process is the shading procedure an example presented by Magnusson et al discusses the use of Phongs formula to present realistic 3D effects. (Magnusson, Lenz, & Per-Erik, 1991)

Advances in the area of this rendering method include the *marching cube* algorithm which is discussed in more detail later in this chapter, where by including it as an additional process while still making use of thresholding allows for the creation of a contoured mesh surface. (Stytz, Frieder, & and Frieder, 1991)

3.1.2 Maximum Intensity Projection (MIP)

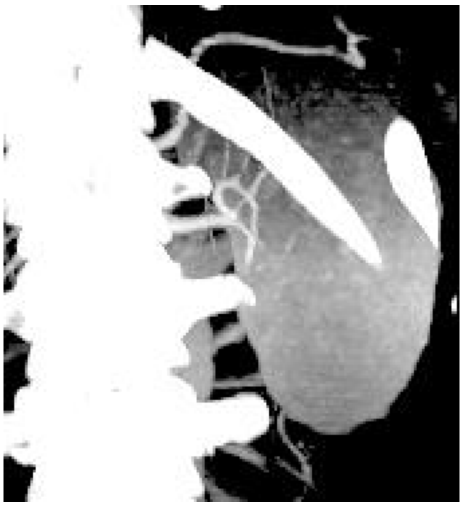


Figure 9 MIP render of patients left kidney

(Sato PhD, Shiraga MD, Nakajima MD PhD, Tamura PhD, & Kikinis MD, 1998)

This rendering technique involves taking a line of voxels and determining which voxel has the maximum intensity. This process of rendering is largely used in the area of angiographic imaging. The process of voxel intensity evaluation here can work both ways. Evaluating from the inside of the dataset to the viewer point of view is referred to as *forward transformation*. An issue with this method is that multiple voxels can be projected to the same pixel leading to an obscured image. The alternative to this is to evaluate from the outside in which is referred to as *ray casting.* The string of values that are encountered is interpolated and it is this value which is placed as the pixel value to be displayed on screen. (Pavone, Luccichenti, & Cademartiri, 2001)

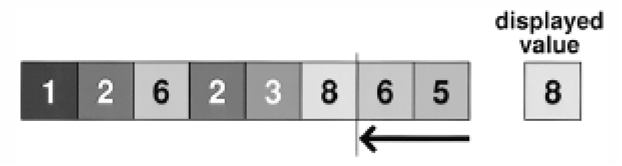


Figure 10 - Maximum Intensity Projection (Calhoun BFA, Kuszyk MD, Heath PhD, Carley BS, & Fishman MD, 1999)

3.1.3 Volume rendering

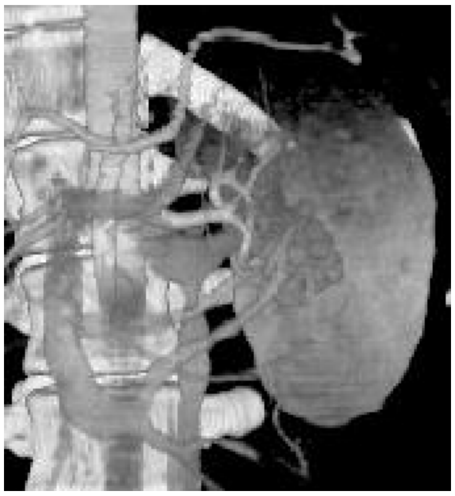


Figure 11 Volume Rendering of patients left kidney

(Sato PhD, Shiraga MD, Nakajima MD PhD, Tamura PhD, & Kikinis MD, 1998)

While Shaded Surface display and Maximum Intensity Projection have an advantage of speed, they also possess some trade-offs. SSD only makes use of 10% of the data obtained from a scan meaning information is lost while MIP lacks perspective without additional processing due to its nature of projection (Calhoun BFA, Kuszyk MD, Heath PhD, Carley BS, & Fishman MD, 1999).

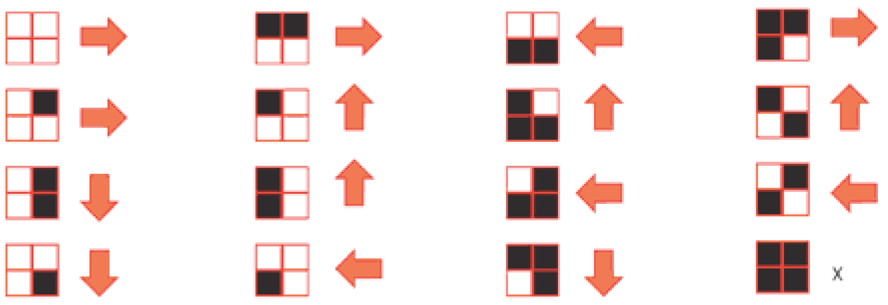
Volume rendering, takes the volume data as a whole and sums the contribution made by each voxel encountered in the make-up of the final 2D rendered image. This does not have the same disadvantages over the previous mention methods in that no information is lost nor is there any thresholding taking place (Pavone, Luccichenti, & Cademartiri, 2001).

The next section of the paper is going to look in depth at different volume rendering techniques with respect to the application of this thesis. The aim will be to present how these techniques can be applied to the area of representing geometry as oppose to just the processing of medical imaging data. The methods will be split between those which are *indirect* and follow the traditional 3D graphics pipeline route and *Direct* which aims to render data by sampling directly from the geometry and projecting directly to the 2D frame buffer i.e. the final rendered image.

3.2 Indirect volume rendering

Surface rendering is a technique that is used to extract isosurface information from volume data. Early techniques involved using contouring (similar to that discussed in chapter 2) of the surface, where contours on sequential slices are connected with triangles. However there were issues with this if there happened to be two or more contours existing on a single slice which leads to ambiguities. Lorensen and Cline in 1987 proposed a new method called the Marching Cube algorithm. The basic premise is that it treats the world it is rendering as 3D grid. Each cell (commonly referred to as a voxel) is examined independently. Each vertex on this cell is considered whether inside or outside and based on this; it determines the topological state of that cell. There are 256 possible topological states, 8 vertices, each having two possible states ($2^8 = 256$), but this is reduced to 14 due to symmetry (Lorensen & Cline, 1987)

To help in better understand the Marching cube algorithm, the following is a demonstration of the marching square algorithm. It is being used in this case to extract a simple outline from a sprite character contained in a bitmap image. The technique and code here is based on the image displayed below and is essentially the 2D implementation of the 3D algorithm. (Spiess, 2010) The algorithm works by sampling pixels contained in a 2 x 2 grid. Based on the various conditions encountered, determines the next move. For the purpose of the demonstration, a pixel is placed at ever move of the cursor (top left corner), leaving in the end, an outline of the sprite character. This illustrates one use of marching squares, in another implementation; it would be possible to have a case table to look up the correct contour. The result in the end would a sprite character traced out with polylines.



16 possible states and the direction this moves the test grid on the next iteration. In this case an object will be traced in an anti-clockwise direction. The state X is illegal.

Figure 12 (Spiess, 2010)

3.2.1 Marching Square python implementation

What follows next is a proof of concept based on the above diagram. As a use of this algorithm can be used to lasso around an object it has uses in image manipulation applications (Spiess, 2010). What is shown here in this demonstration is very high level. Python is the choice here because of the vast availability of modules that make prototyping really fast. The code places a cursor on the original image (all in memory), the neighbouring pixels are each evaluated and based on the outcome, and the next move is decided. The image below on the right shows what is traced out, when a gray pixel is placed on every move.

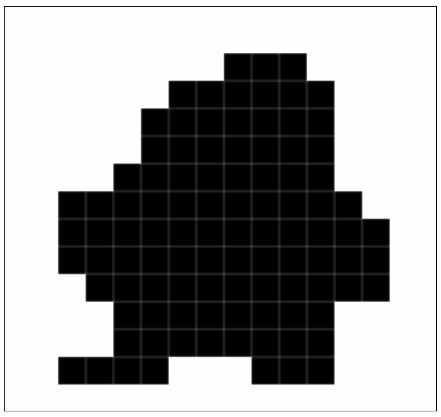


Figure 13 - Original sprite character 16 x 16 - 8 bit

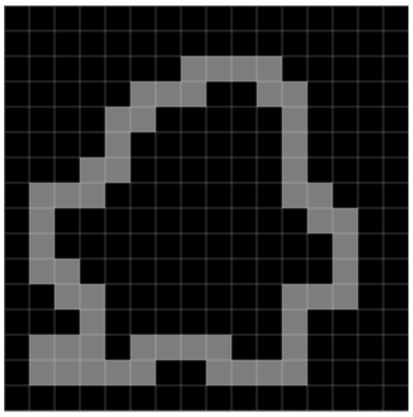


Figure 14 - Extracted outline in a new image

```python
__author__ = 'William'
import Image

white = 255
black = 0

def loadImage():
    im = Image.open("sprite.bmp")
    w, h = im.size
    return im, w, h


def createBlankImage():
    imgNew = Image.new("P", (16, 16))
    return imgNew


def marchingSquare(im, w, h, imgNew):
    pixels = im.load()
    pixelsnew = imgNew.load()
    x = 0
    y = 250

    skip = 0

    while True:
        x0p = pixels[x, y]
        y0p = pixels[x + 1, y]
        x1p = pixels[x, y + 1]
        y1p = pixels[x + 1, y + 1]

        if isALeftState(x0p, y0p, x1p, y1p):
            if x+2 < w:
                x += 1  # go right
            pixelsnew[x, y] = black
        elif isARightState(x0p, y0p, x1p, y1p):
            if x-1 >= 0:
                x -= 1  # go left
            pixelsnew[x, y] = black
        elif isAUpState(x0p, y0p, x1p, y1p):
            if y - 1 >= 0:
                y -= 1  # go up
            pixelsnew[x, y] = black
        elif isADownState(x0p, y1p, x1p, y1p):
            if y + 2 < h:
                y += 1  # go down
            pixelsnew[x, y] = black

        if skip > timeout:
            break
        skip+=1
    imgNew.save('out.bmp')
```

```
def isALeftState(x0p, y0p, x1p, y1p):
            if (x0p is white and y0p is white and x1p is white and y1p is white) or \
            (x0p is black and y0p is black and x1p is white and y1p is white) or \
            (x0p is black and y0p is black and x1p is black and y1p is white) or \
            (x0p is white and y0p is black and x1p is white and y1p is white):
                return True
            else:
                return False

def isARightState(x0p, y0p, x1p, y1p):
    if (x0p is white and y0p is white and x1p is black and y1p is black) or \
       (x0p is white and y0p is black and x1p is black and y1p is black) or \
       (x0p is white and y0p is black and x1p is black and y1p is white) or \
       (x0p is white and y0p is white and x1p is black and y1p is white):
            return True
    else:
            return False

def isAUpState(x0p, y0p, x1p, y1p):
    if (x0p is black and y0p is white and x1p is white and y1p is white) or \
        (x0p is black and y0p is white and x1p is black and y1p is black) or \
        (x0p is black and y0p is white and x1p is black and y1p is white) or \
        (x0p is black and y0p is white and x1p is white and y1p is black):
            return True
    else:
            return False

def isADownState(x0p, y0p, x1p, y1p):
    if (x0p is white and y0p is black and x1p is white and y1p is black) or \
        (x0p is white and y0p is white and x1p is white and y1p is black) or \
        (x0p is black and y0p is black and x1p is white and y1p is black):
            return True
    else:
            return False


def main():
    im, w, h = loadImage()
    imgNew = createBlankImage()
    marchingCube(im, w, h, imgNew)


if __name__ == "__main__":main()
```

3.2.2 Marching cube algorithm

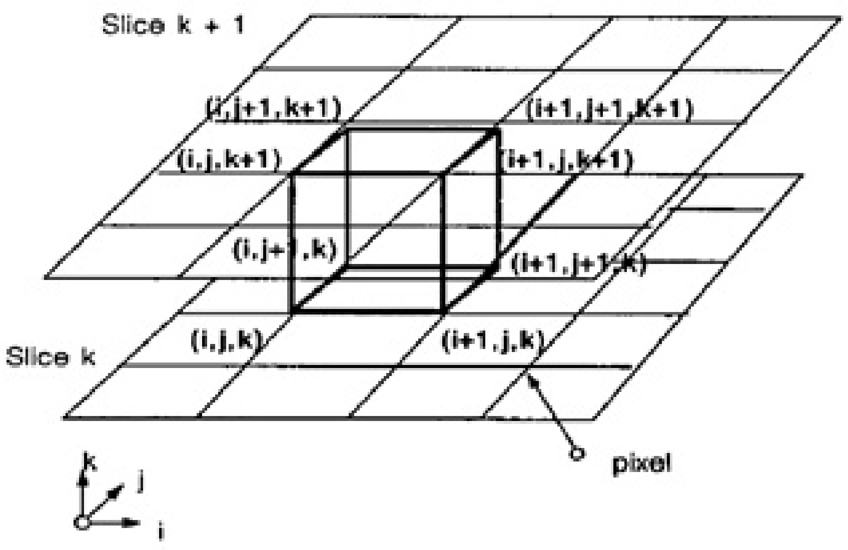


Figure 15 – A voxel being used to sample between two image slices (Lorensen & Cline, 1987)

To implement the 3D algorithm programmatically, first a cell (voxel) is selected for sampling. Each of its 8 vertices as it goes is checked to determine if it is inside or outside the geometric data (by way of some threshold value). If for example it encounters a case where one vertex threshold value is below a certain limit while the rest are above, then it can be said that this one vertex is cutting through the surface. The next step then involves looking up the topological state in a case table. This value gives back the edges that are impacted by this particular case. The final step is to calculate by linearly interpreting where on the marked edges, these cuts occur and place a vertex there. Going with the example of one vertex cutting the surface, the resultant polygon to be rendered would be case 1 from the diagram on the next page. (Bourke, 1994)

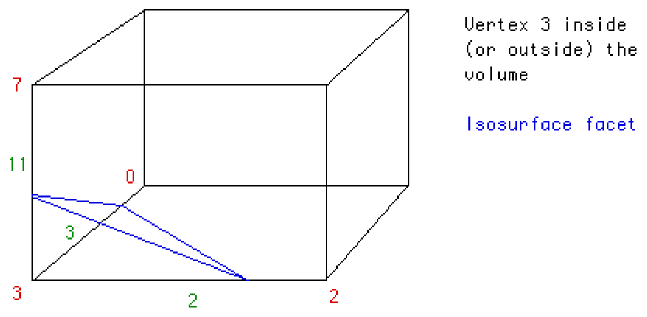


Figure 16 Vertex inside, all other outside (Bourke, 1994)

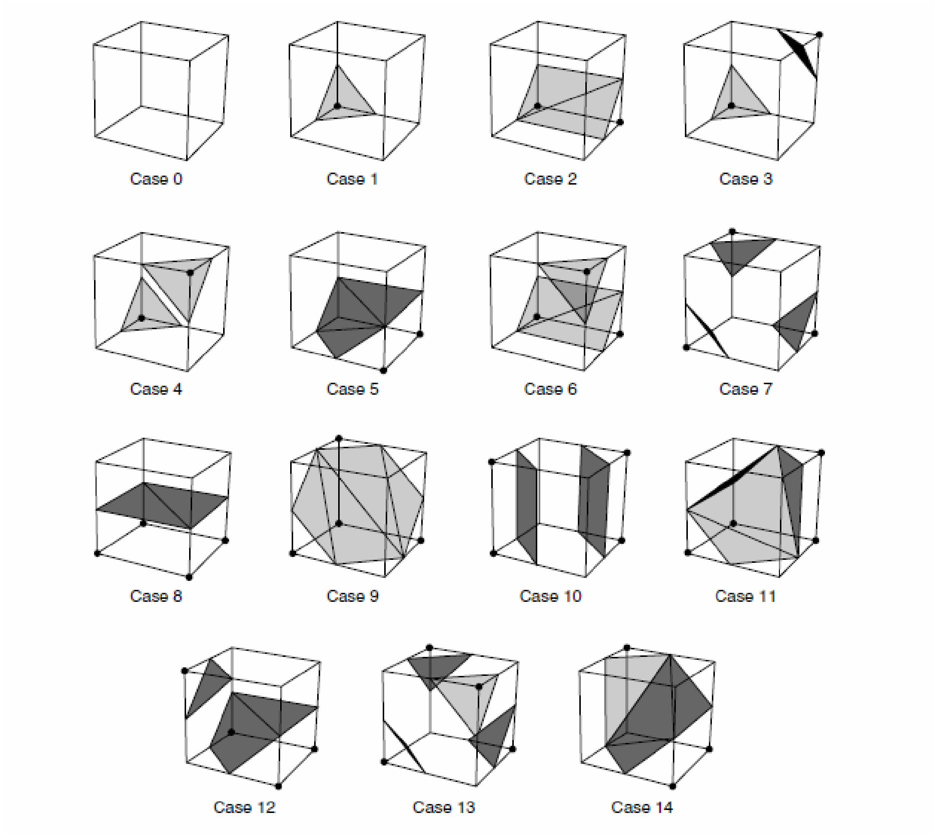


Figure 17 - The marching cube algorithm has 256 possible cases which have been reduced to 14 through symmetry (Schroeder & Martin, 2005)

3.2 Direct volume rendering

3.2.1 Volume Ray casting

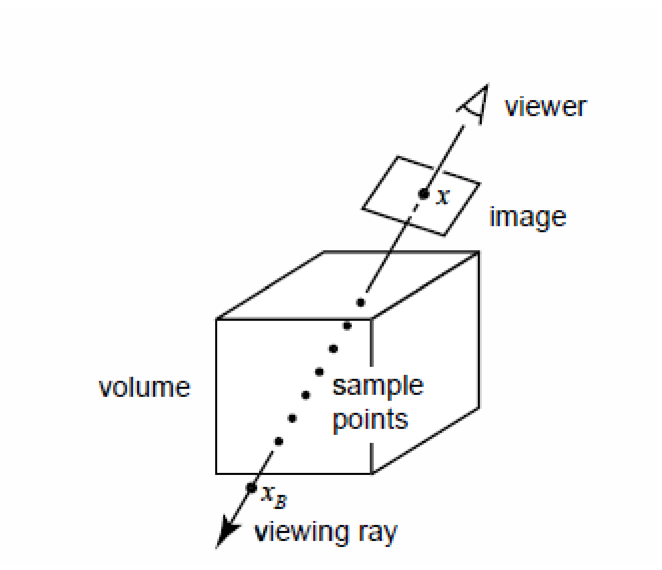


Figure 18 Volume rendering (Lacroute, 1995)

As discussed previously, indirect methods work by extracting surface information from voxels and then displaying it through the use of polygons. This uses the traditional graphics pipeline.

Direct methods, however, work by evaluating the *optical model* where voxels properties are evaluated for how they react with light i.e. emit, reflect, scatter, absorbs or occludes light. The result of this evaluation is then painted directly to the viewing image (Salama, 2006).

Ray-Casting is considered an image order rendering technique because it evaluates for every pixel on the image. There are several algorithms available but it is Marc Levoy's algorithm that is explained here. To perform ray-casting, for each pixel of the image, a ray is cast into the volume. Along the ray at set intervals, the volume around it is sampled. Sampling is performed by taking the surrounding voxels and interpolating colors and opacities which are then merged with the background color by compositing. This can be done either by going front to back or back to front in order to determine the pixel color (Pawasauskas, 1997).

Shown here is a pseudo code algorithm for how ray casting works as detailed in Philippe G. Lacroute paper on fast volume rendering (Lacroute, 1995).

For every pixel on the image a ray is cast into the volume (lines 1 and 2). For defined intervals along the z axis which is the ray cast into the volume raster (line 3). Lines 4, 5 and 6 are performing the sampling of neighbouring voxels to determine the influence color and opacity. Line 7 adds the result to the pixel image.

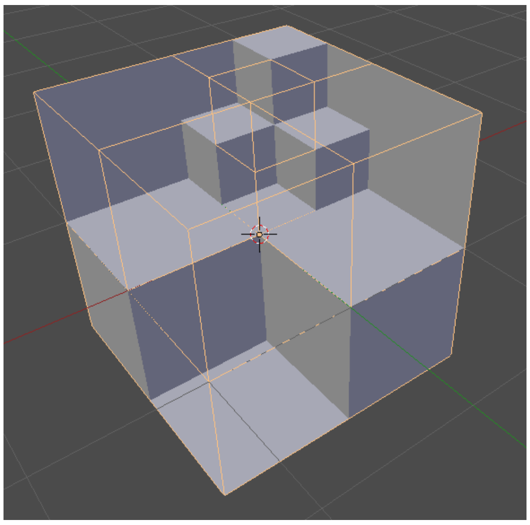
```
1   for $y_i$ = 1 to Image Height
2       for $x_i$ = 1 to Image Width
3           for $z_i$ = 1 to Ray Length
4               For each $x_o$ in Resampling Filter ($xi, yi, zi$)
5                   For each $y_o$ in Resampling Filter($xi, yi, zi$)
6                       For each $z_o$ in Resampling Filter($xi, yi, zi$)
7                           Add contribution of Voxel[$x_o$, $y_o$, $z_o$] to Pixel[$x_i$, $y_i$]
```

3.3 Indirect vs. Direct

When making the choice of rendering technique, there is no real obvious winner. The indirect route which follows the tradition graphics pipeline has a long proven record having being used in games development. But the fact that a pipeline is being followed introduces the possibility of bottlenecks. Using the direct route using rays, has a number of advantages over Rasterization in that it takes into account real world effects such as those of reflections and refractions. It also has the ability to do visual culling automatically because of its line of sight nature. Where the use of rays falls down (currently) is that it is slower to render than that of traditional rasterising techniques (Boulos, et al., 2007).

So back to the choice of choosing which is best to use? What we discover in later chapters is that a combination of both used together depending on the situation works pretty well. But for the purposes of development mainly because of time constraints and the risk that direct may not be supported well enough, indirect will be the choice as it is more reliable. If the opportunity presents itself, direct may be examined.

4.0 Spatial Data structures



From the research so far, numerous techniques have been presented for rendering volumetric objects contained in a voxel world. However, depending on the resolution (sub divisions) of the voxel world, a lot of time can be wasted sampling voxels that may have no contribution to offer to the rendering of the structure. Also the issue arises if there is to be an interaction with the structure (i.e. a manipulation of a voxel), how can the data associated with that particular component be located as quickly as possible? This chapter aims to explore some of the popular spatial data algorithms that exist that could possibly be of use in accomplishing the primary goal of this thesis.

Spatial data algorithms and games are no stranger considering that with the increase in computing power, also brings an increase in scene complexity. An example given is representing a desktop printer would need over 300,000 primitive's or by taking a Boeing-777 and presenting it in a scene, would require 500,000,000 polygons when all the parts are summed together. Taking it further up a notch, when considering a naval submarine, it is tens time more complex than an airplane while an aircraft carrier has ten times more the complexity of a submarine (Cripe & Gaskins, 1998). So it seems clear that the need for an algorithm to speed up queries where any scene renderings or collision detection maybe involved are well justified, and so that we will begin to take a look into spatial data structures.

Spatial data structures

Spatial data structures exist to organise geometry that may be contained in a scene into some n-dimensional space. They are hierarchical in nature and works by each level holding the level below it, continuing recursively until so criteria is satisfied. A popular use of these hierarchical structures exist in graphics and games development to accelerate queries that may be required for the purposes of collision detection, ray tracing or culling algorithms (Haines & Akenine-Möller, 2002).

These structures are of particular interest in the course of a voxelized world when research has shown that “30% and 70% of time spent in isosurface generation was spent examining empty cells” when a cell by cell approach is taken. This applies in particular to the marching cube algorithm as time can be saved if empty cells are partitioned from cells of interest (Wilhelms & Van Gelder, 1992).

Speed improvements that can be obtained from adopting the use of a tree structure can take from being O(n) to being O(log n) (Haines & Akenine-Möller, 2002).

When it comes to analysing what are the different available data structures we find that there are many types of spatial data structures that could possibly suit the needs of this project. These include the Bounding Volume Hierarchies, Binary Search Portioning and Octree which will get a more in depth look because of its nature and how it suits the direction this thesis is moving in.

Bounding Volume Hierarchy

This algorithm works by surrounding objects with simple geometry i.e. taking a complex aircraft model it could be surrounded by a sphere. By doing this it is much easier and faster to look up and deal with an object than to look up the object it is enclosing. Some uses of the BVH includes the area of frustum culling which can be seen in the diagram below, it is useful for finding scene objects with ray intersection as well as collision detection in games (Haines & Akenine-Möller, 2002).

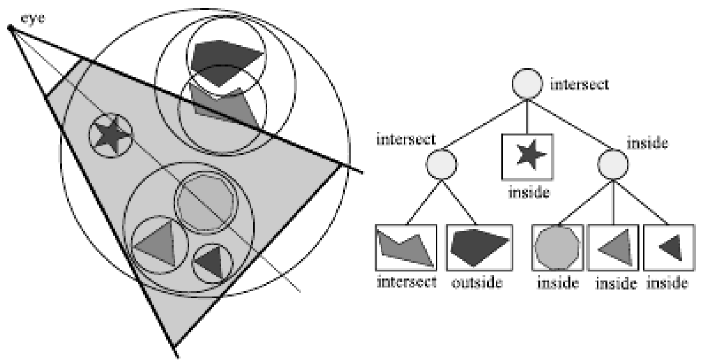


Figure 19 - Frustrum view culling (Haines & Akenine-Möller, 2002)

Binary Space Partitioning trees

Two types of BSP's exist, these are the Axial or the Polygon aligned structure. They work by dividing space in two and then sorting which geometry goes where depending on its location. These offer an advantage over the BVH in that depending on how they are traversed; the contents can be sorted from any point of view.

Axial Aligned BSP Trees

This takes the entire scene world and encapsulates it within an Axial Aligned Bounding Box, each one of these boxes can be recursively sub divided into smaller boxes. The split can be half and half but doesn't have to be. If plane or dividing line is found to split an object it can become part of both or subdivide further splitting into two objects. An alternative use of an Axial Aligned BSP is K-d trees. K-d trees can be thought of a form a BSP trees where a box is split along the x axis, the children are split by y axis and with the grandchildren along the z axis. These types of trees are useful for ray tracing or collision detection because if there is no intersection with a parent bounding box then there certainly won't be an intersection with the children. This saves a level of searching.

Polygon BSP

Similar to the previously described algorithm with the exception that a polygon is used instead to subdivide, this is more time consuming than the AABB, it is normally computed once and the beginning of the program and stored for reuse. It works best when the tree is balanced, unbalanced tends to be in-efficient. A useful use for the polygon BSP is for the use of the painter's algorithm as it is capable of establishing a back to front or a front to back by traversing it from a particular view. In this case, it could substitute the need for a Z-Buffer (Haines & Akenine-Möller, 2002).

4.1 Octree

The Octree is going to get a preferential view because of the suitability it lends to spatially partitioning six sided objects in a hierarchical cubic world. This justification is coming from reading about experiments into papers on optimising volume rendering such as that written by Wilhelms and Van Gelder (Wilhelms & Van Gelder, 1992).

In their paper, they found that the sometimes large data associated with a volume lends itself to being unable to render quickly in real time. They experimented with the marching cube approach (evaluate each and every cell / voxel) and octree traversal. The results found that in all cases, that the use of the Octree was faster than the marching cube.

The technique of spatial representation by using Octrees was pioneered by Donald Meagher, in his paper *Geometric modelling using Octree encoding* he presents his technique that "arbitrary 3-D objects can be represented to any specified resolution in a hierarchical 8-ary tree structure".

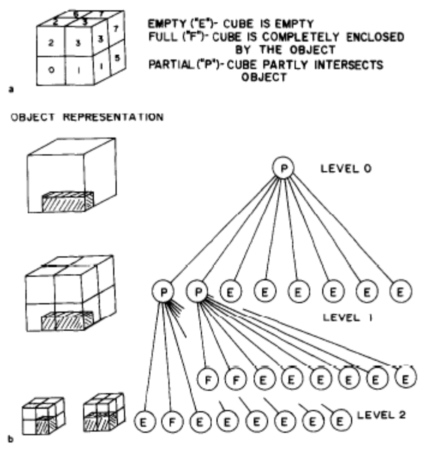

EMPTY ("E")- CUBE IS EMPTY
FULL ("F")- CUBE IS COMPLETELY ENCLOSED BY THE OBJECT
PARTIAL ("P")- CUBE PARTLY INTERSECTS OBJECT
a
OBJECT REPRESENTATION
LEVEL 0
LEVEL 1
LEVEL 2
b

FIG. 1. Simple object represented in Octree Encoding format. (a) Numbering sequence and label definitions. (b) Three-level tree representation of an object.

Figure 20 - Octree representation (Meagher, 1982)

Octree works by having a root which represents the world, represented by level 0 in the figure below. Within this root, providing there objects in the world will be further decomposed into 8 nodes. If the node is describing the complete object then there is no further action and is now considered a leaf. However if there is any discrepancy, the node will further divide into 8 children. Meahger puts forward that the tree structure offers numerous advantages, among them being the ability to represent an object to the precision of the smallest cube. Another advantage offered is that regions of space, if traversed in the right sequence, will be visited in a constant direction. This is of big benefit in application of game engines where hidden surface removal is used as no sorting or searching is required (Meagher, 1982).

5.0 WebGL

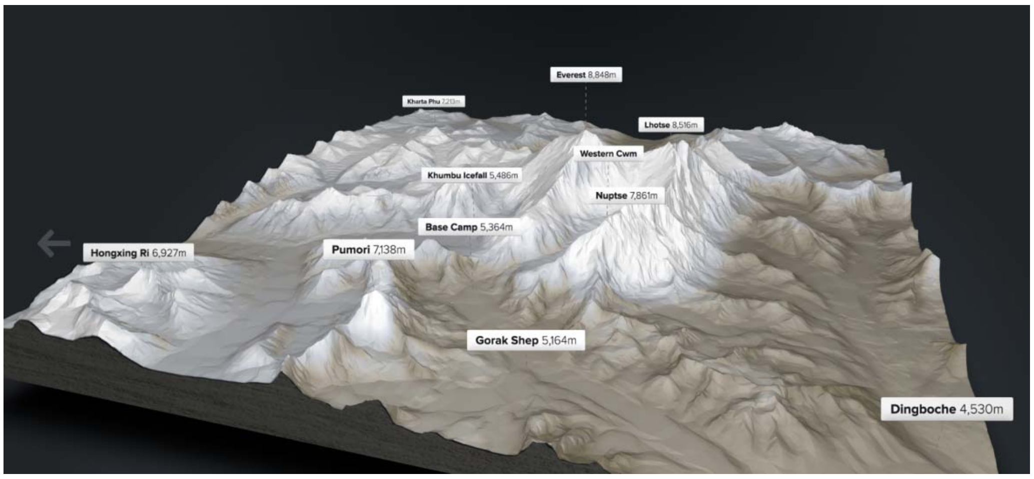


Figure 21 - GlacierWorks WebGL presentation of Mt Everest in the Himalayas - http://explore.glacierworks.org/

5.1 Introduction and brief history

OpenGL, with the GL standing for Graphics Library, is an Application Programming Interface that allows for developers or enthusiasts to write applications that have access to the underlying graphics hardware. By having this interface, it maximises the portability of the users applications allowing them to write once and go anywhere without having to be concerned with platform specifics.

OpenGL came to life by Silicon Graphics Inc. (SGI) who was a developer of high end graphic workstations. Because their competitors were producing much lower cost solutions, SGI decided to focus on portability. They cleaned up their API implementation and released it to the general public to use royalty free. The first version; OpenGL 1.0 was released in 1992. Around the same time SGI set up a group that had input into the OpenGL standard. Today, this is managed by the Khronos Group which consists of over a hundred members including AMD, NVidia, Intel and Sony. (Sellers, Wright Jr, & Haemael, 2013)

OpenGL ES which was derived from the main OpenGL branch is designed to run on embedded systems such as mobile phones. Version 2.0 of this implementation was released in 2007 and it is this release which WebGL 1.0 is based on. The Khronos group released the first version of WebGL in 2011 and as mentioned earlier are also responsible for maintaining the standards of the process.

WebGL allows for the drawing and interacting with three dimensional graphics through a standard browser without the need for any plugins. WebGL which can be combined with HTML5 and JavaScript allows for developers to create web content which is more dynamic (Matsuda & Lea, 2013). Some additional benefits WebGL has to offer from developers point of view is its cross platform deployment, Windows, MAC or Linux are no longer a barrier. Looking for examples of the potential of this technology, we need only look at the dynamic content on the GlacierWorks site (http://explore.glacierworks.org in the image above). Users get a more immersive experience by being able to interact with the 3D model of the Himalayas and experience flybys therefore adding to the learning experience which is can be harder to get from looking at static sites.

5.2 Pipeline

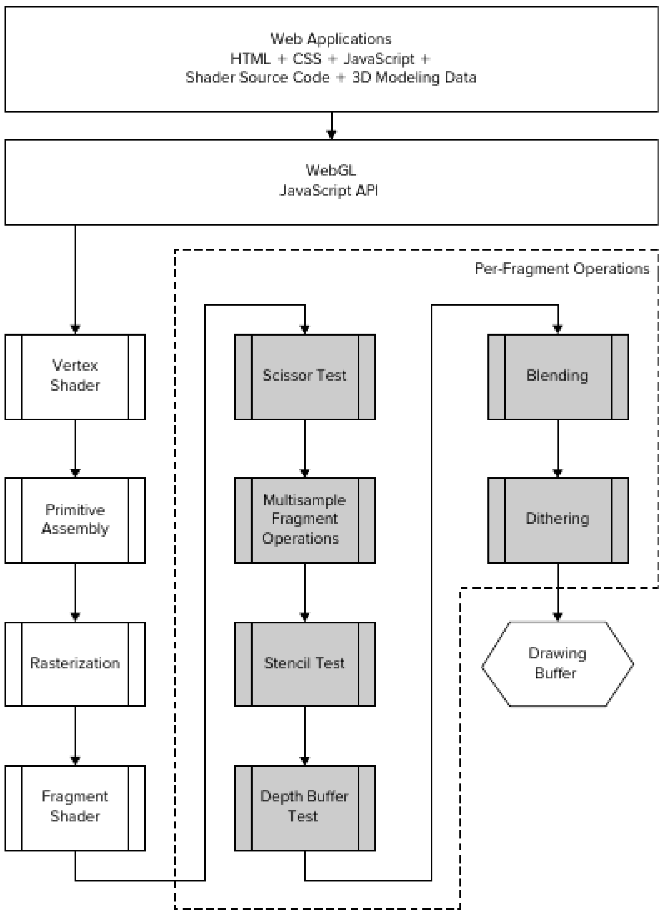


Figure 22 WebGL Pipeline (Anyuru, 2012)

To render a scene, a series of processes must be followed in the WebGL pipeline. The following section aims to breakdown and explain each step in the pipeline process. The following is a an explanation of what happens at the various stages as outlined in Andreas Anyuru's **Professional WebGL Programming : Developing 3D Graphics for the Web** (Anyuru, 2012) as well as making reference to the **OpenGL super bible** (Sellers, Wright Jr, & Haemael, 2013).

Vertex Shader

This is the first stage in the WebGL pipeline. It is a program that describes the position and colour of a vertex. This is user configurable by writing a program in a C like language known as GLSL.

```
// vertex shader program
void main()
{
    gl_Position = vec4 (0.0, 0.0, 0.0, 1.0);
}
```

Primitive Assembly

Now it is time to take the vertices from the previous step and assemble them into primitives such lines or triangles. Also at this stage primitives are sorted from those which exist within the view frustum and those that don't. Basically it is checking to see which primitives are visible to the screen and it is only these that move on to the next stage in the process.

Rasterization

This is a process of taking the primitives received from the assembly stage and turning them into fragments. This can be a visualised as taking a triangle and approximating it to pixels. This information is then passed on to the fragment shader.

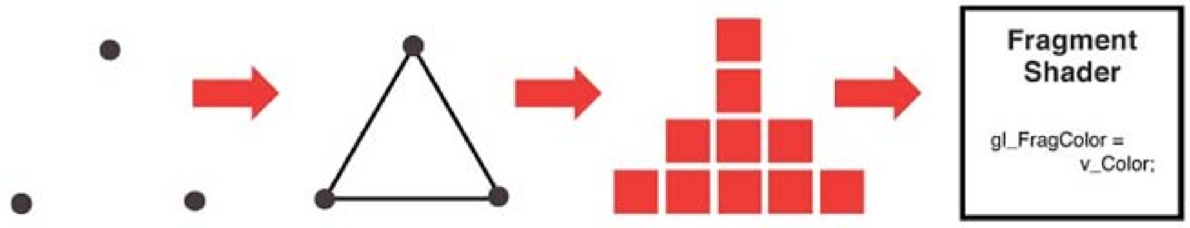
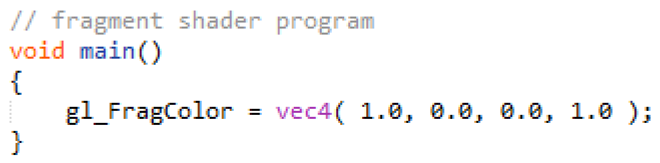


Figure 23 (Matsuda & Lea, 2013)

Fragment Shader

Fragment shaders determine the color of the fragment. Below is an example of a very basic fragment shader program.



```
// fragment shader program
void main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
```

Per Fragment Operations

Scissor Test

This is a culling operation, if the fragment lies within the scissors rectangle it is kept and moves on to the next stage, otherwise it is dropped.

Multisampling Fragment operations

This provides a means of anti-aliasing which is to prevent edges looking jagged when displayed on screen. It works by averaging out pixels on the edge and those adjacent to it to give the smoother appearance.

Stencil Test

This test performs a comparison of a value in the stencil buffer and a reference value. The test is similar to having a card board cut out and placing it over the value to see if they match. Those that fail the test are discarded.

Depth Buffer Test

This test evaluates the depth value of the incoming fragment against the existing content of the depth buffer, again, this determines whether the fragment gets discarded or moves on to the next stage.

Blending

Having survived the depth buffer test, this stage allows combining the color of the incoming fragment color with the color that is already in the color buffer that is at the same position.

Dithering

This is the last step before the draw buffer; the idea here is to arrange the colours in such a way as to give the illusion that there are more colours available than there actually are.

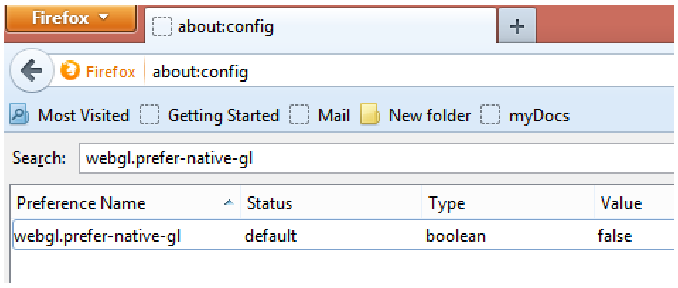
5.3 Browser support

WebGL is designed to run in most of the modern day browsers, this includes; Google Chrome, Mozilla FireFox, Apples Safari and Opera. Up until recently Internet Explorer did not support WebGL but with the release of IE11 has seen a change that could possibly see a wider uptake in WebGL projects. (MSDN, 2013)

An interesting discovery was made during the course of research into how WebGL is supported on Windows operating systems. OpenGL isn't a problem on systems where OpenGL is the main graphics API such as Linux and OSX systems. But due to the fact DirectX is the primary API for windows and to avoid issues with the availability of certain drivers, both Chrome and Firefox use DirectX calls instead of OpenGL to ensure that maximum compatibility with the Windows operating system. WebGL can of course be enabled by appropriate use of flags in chrome or in FireFox by editing the configuration file as shown in the figure below.

To make use of DirectX, both use the ANGLE (Almost Native Graphics Layer Engine) project to convert WebGL's or OpenGL ES 2.0 API calls to DirectX 9.0 or DirectX 11.0 API calls. The library is open sourced under a BSD license because it is still under development. The project is currently the default for both Chrome and Firefox when run on a windows machine.

As well as its primary use above, the tool may also prove to be useful for the purposes of prototyping OpenGL ES applications on windows since it design to handle those API calls (Bridge, 2010).



Search: webgl.prefer-native-gl

Preference Name	Status	Type	Value
webgl.prefer-native-gl	default	boolean	false

Figure 24- Firefox by default has native WebGL disabled and defaults to ANGLE + DirectX instead.

5.4 Security

OpenGL ES being a shader based API means that it has access to the graphics processor in order to render a scene to the screen. While ordinarily, for an embedded system, little security is needed with API calls as they would normally never be any danger of being accessed from outside influences. But with WebGL being capable of making API calls directly through JavaScript, extra precautions are needed to prevent security breaches. One notable security breach which was documented in an early implementation of WebGL in Firefox discovered a way to steal data from a user by capturing screen shots of a user's screen (Forshaw, Stone, & Jordon, 2011).

The above example of security breach was an example of not complying with standards set out by the Khronos Group. The following is a list of the conformance standards in WebGL specification from the Khronos website (http://www.khronos.org/webgl/security/).

Undefined Behaviour

In contrast to OpenGL ES, if a call to read a pixel is made outside of the confines of the frame buffer is defined as 'undefined behaviour' and is of no real risk. However in WebGL to prevent the risk of getting access to pixels that are outside of the frame buffer, the standard specifies that these values are set to 0. Similarly, WebGL defines that range checks are done on areas that have memory access to prevent web code gaining access to data outside of its designated bounds.

Access to non-Initialized Memory

In normal conditions when a graphics API is assigned memory, it fills it with the content for which was designated for; there is little worry for its previous contents. In WebGL however there possibility of accessing old data, for this the specification states that the memory should be zeroed. This of course can impact on performance.

Denial of Service

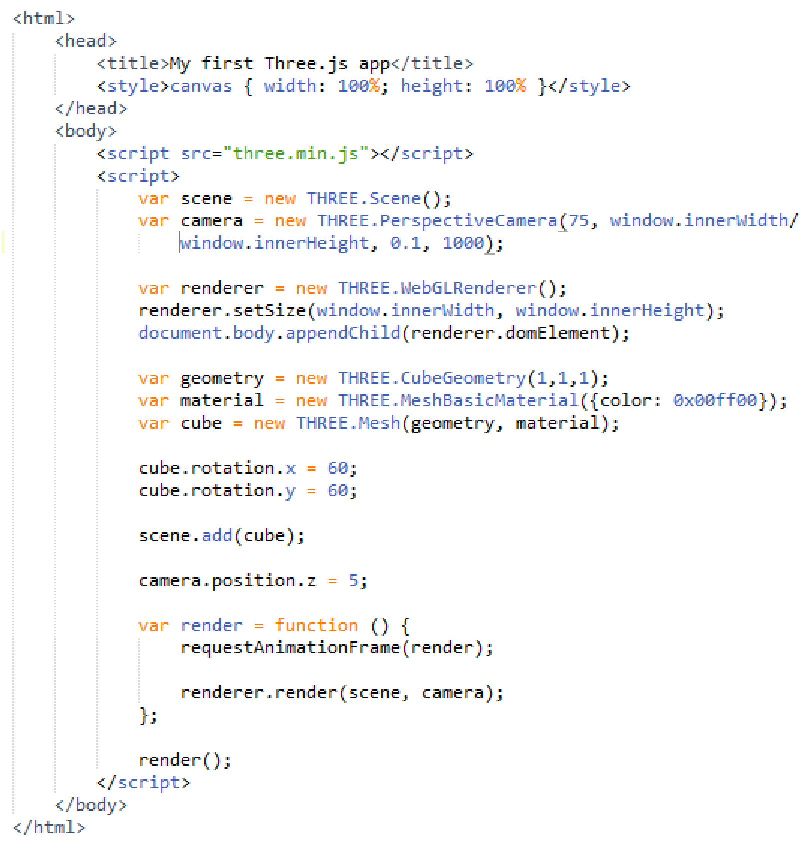
A user's machine may become unresponsive if there are a large number of primitives to be drawn. This is can happen especially if there are complex shaders associated with the calls.

To prevent this checks are normally built into the operating system that will reset operations that are taking too long. The WebGL has in its specification is that a user can be alerted to any reset that takes place and ask them do they wish to continue with the operation.

5.5 THREE JS Library

Three.js is a javascript framework which is under constant development that created by Ricardo Cabello. The purpose of the framework is to abstract WebGL calls to a high level and in doing so allow an easier entry point while also speeding up development times. The library is available under an MIT license which allows reuse as long as the original software license is included. While document and tutorials are available at threejs.org, the source is available on Github for anyone to fork and contribute to the effort.

The following code illustrates best what the library does, in 37 lines of code most of which is self-documenting; a simple cube is orientated and rendered to the screen. The equivalent of writing this in pure WebGL would require substantially more code along with difficulties in debugging.



```
<html>
    <head>
        <title>My first Three.js app</title>
        <style>canvas { width: 100%; height: 100% }</style>
    </head>
    <body>
        <script src="three.min.js"></script>
        <script>
            var scene = new THREE.Scene();
            var camera = new THREE.PerspectiveCamera(75, window.innerWidth/
                window.innerHeight, 0.1, 1000);

            var renderer = new THREE.WebGLRenderer();
            renderer.setSize(window.innerWidth, window.innerHeight);
            document.body.appendChild(renderer.domElement);

            var geometry = new THREE.CubeGeometry(1,1,1);
            var material = new THREE.MeshBasicMaterial({color: 0x00ff00});
            var cube = new THREE.Mesh(geometry, material);

            cube.rotation.x = 60;
            cube.rotation.y = 60;

            scene.add(cube);

            camera.position.z = 5;

            var render = function () {
                requestAnimationFrame(render);

                renderer.render(scene, camera);
            };

            render();
        </script>
    </body>
</html>
```

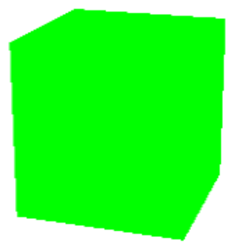


Figure 25 - Simple cube rendered in Web Browser

6.0 Digital Sculpting

6.1 Introduction

This chapter leads into where the project direction is going. Here we will present an overview of what is implied when we speak about working in the 3D environment from an artist/sculpting point of view. We will look at the traditional mesh modelling and sculpting which is a popular choice amongst 3D artists. The chapter will then begin to examine research that has been performed in the area of volume sculpting and also research into the area of virtual clay sculpting where researchers demonstrate their technique into bringing the art of true to life clay sculpting that artists are familiar with into the digital arena. This sense of realism is not available in traditional meshes as they have no density to alter when physically manipulated.

6.2 Working with Traditional Modelling tools

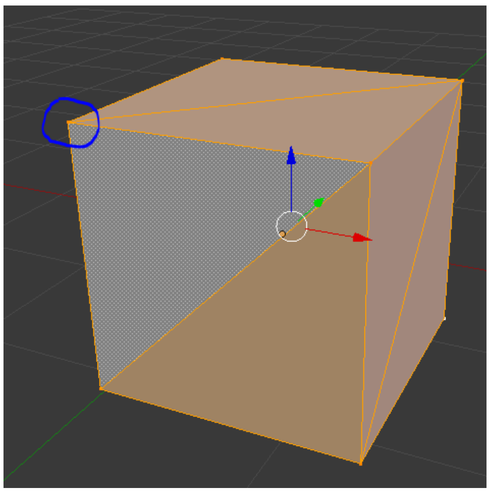
The essence of working with 3D modelling tools is not to recreate the world in 3D; rather it is to present a 2D image similar to which a photographer may capture of real world objects (Hess, 2010).

Many tools exist on the market today that facilitate this art form including Autodesk 3DS Max, Maya and ZBrush to name but a few. However Blender3D is the choice for demonstrative and comparative purposes because of the authors experience with the tool.

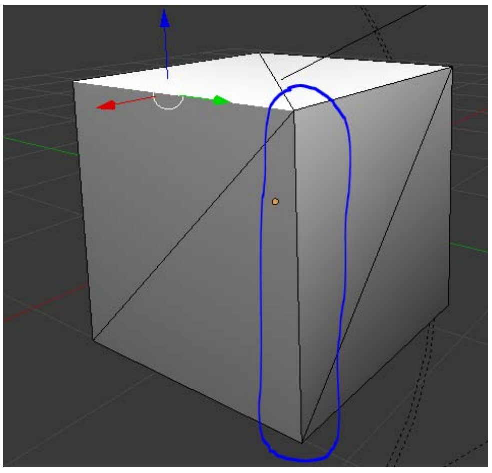
Blender3D has an interesting history having being originally created for in house commercial purposes. The company was a Dutch animation company NeoGeo founded by Ton Roosendal. As the product at the time needed to be re-written, Ton founded another company that marketed and developed products and services around Blender. However with disappointing sales the development was discontinued. Because of the huge interest by artist community, it was decided to try and get Blender open sourced rather than go to waste. A bid to raise 100,000 EUR was made so that investors would allow the tool to be opened sourced. That goal was reached in seven weeks and now Blender exists under the General Public License and is maintained by the Blender Foundation with the help of volunteers and community donations (Blender Foundation, 2013). The tool can be freely downloaded today from www.blender.org

When working with the 3D environment, there are several concepts that need to be understood. The concepts are common across most 3D modelling packages. These include:

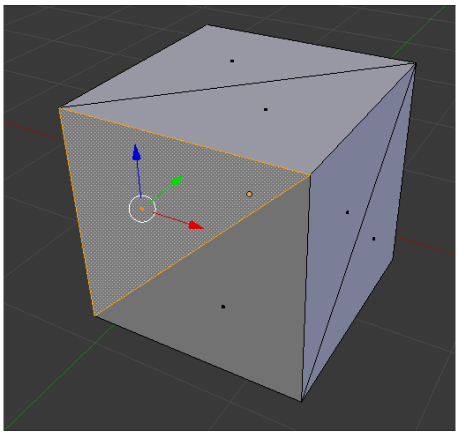
Vertices – Is the simplest of elements, it is a point in 3D space represented by an X, Y and Z coordinate. On its own it has little to offer visually but in combining them they can form simple polygons.



Edge – An edge is classed as being the line drawn between two vertices.

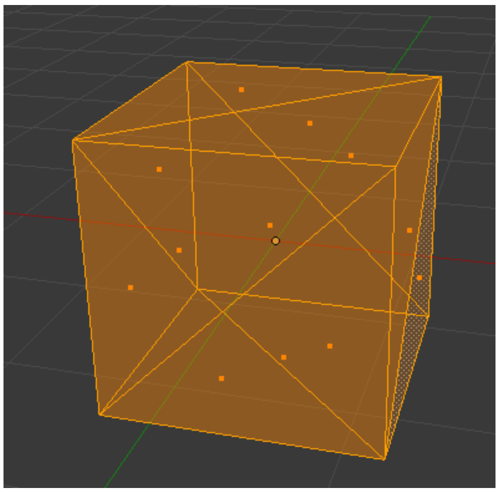


Faces – consists of a minimum of three vertices linked together. Faces have a *normal* associated with it to determine how light will interact with it. The normal are perpendicular to the face represented as a unit vector.

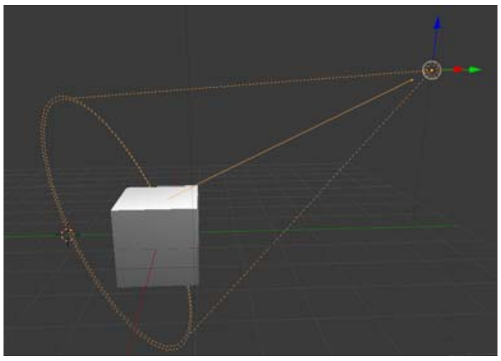


Meshes – Are a combination of all three discussed elements that come together and form the bases of a model.

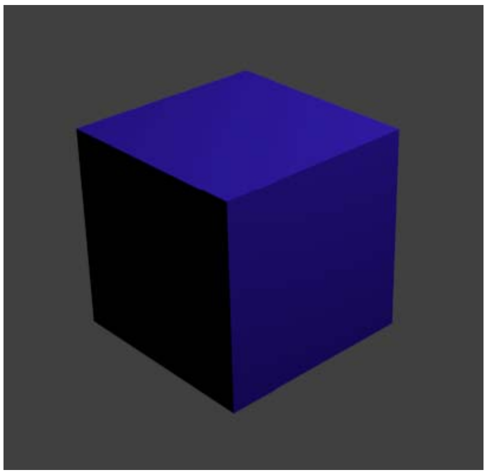
Meshes can be open or closed structures such as the box shown here to the right.



Lighting – To illuminate the scene, there needs to be lights. Lighting in the 3D environment is as much of a science as it is in the real world and is worthy of a discussion in its own right. However for the purposes of giving a general overview here, Blender provides two types of lighting, Directional I.e. A spot light or non-directional lighting which is referred to ambient lighting I.e. a light that has no particular source but affects all objects in a scene (Hess, 2010).

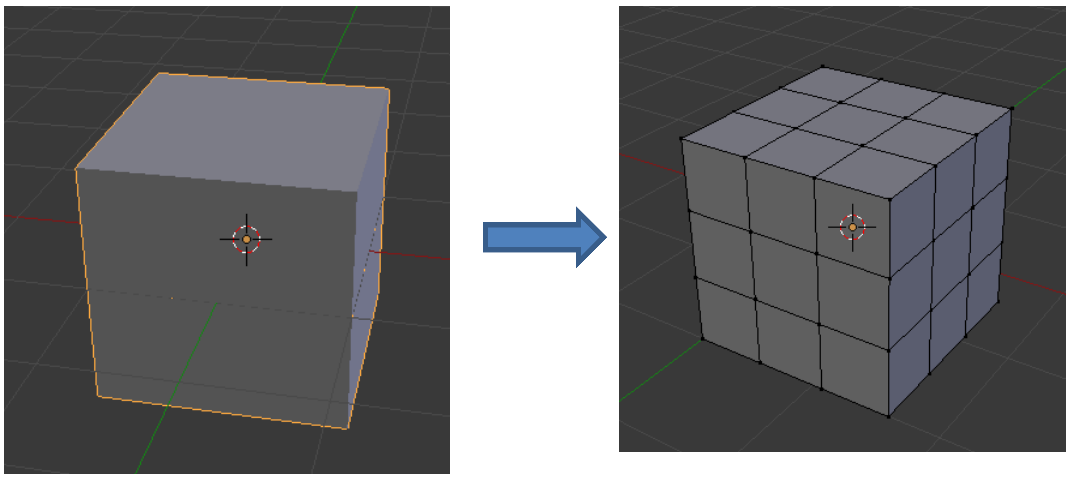


Material – Define how light interacts with a surface. Such properties that can be set are colour, diffuse and specular. (Hess, 2010)



6.2.1 Box modelling

This technique is one of the easiest ways to begin modelling. This iterative process starts with a simple primitive such as a cube. The primitive is subdivided and a subtle amount of detail is added at each stage by moving vertices, edges or faces before adding more subdivisions. It is a skill that requires a lot of practice to perfect but the results can be quite impressive.



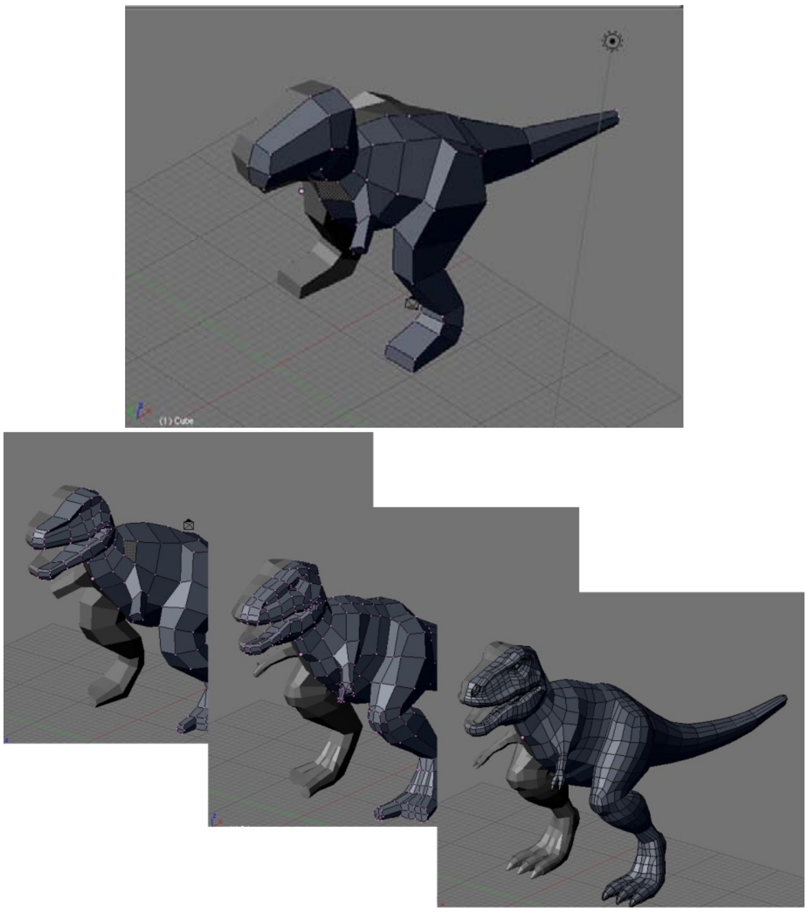
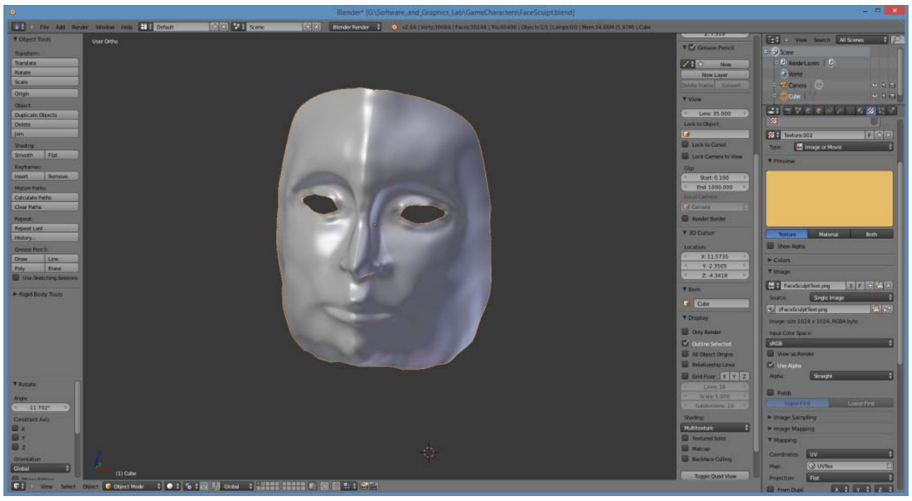


Figure 26 - Step by Step progression of the Box Modelling technique (WikiBooks.org, 2013)

6.2.2 Mesh Sculpting

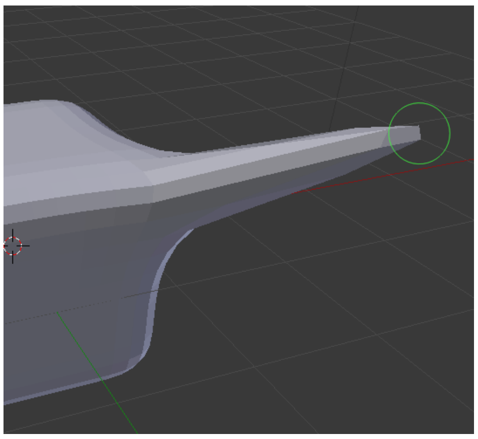


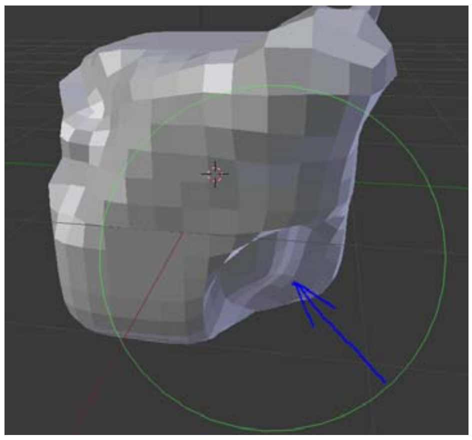
6.2.2.1 Description

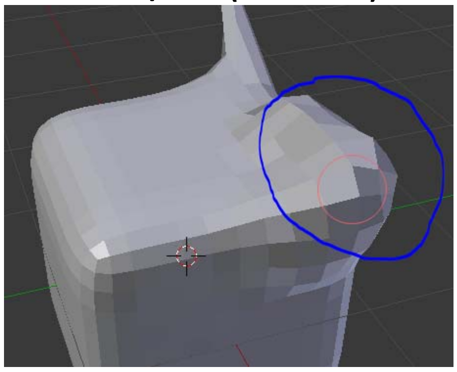
This technique is similar in ways to the box modelling technique described above. However, instead of directly selecting and manipulating individual vertices, edges or faces, a brush is instead used. This brush automatically determines and selects the appropriate vertices.

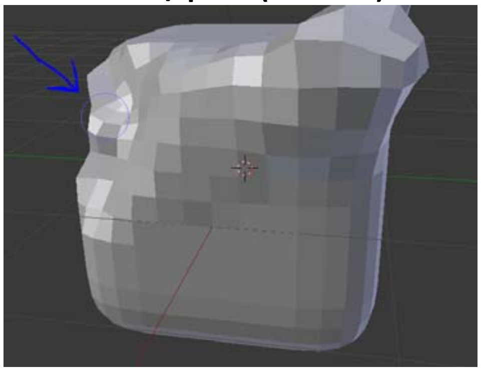
The user is left then to select an appropriate brush and "paint" the relevant action on the mesh (wiki.blender.org, 2013). A Selection of the different brushes on offer by blender is shown below.

Blender sculpt tools


Grab / Pull (Outwards)

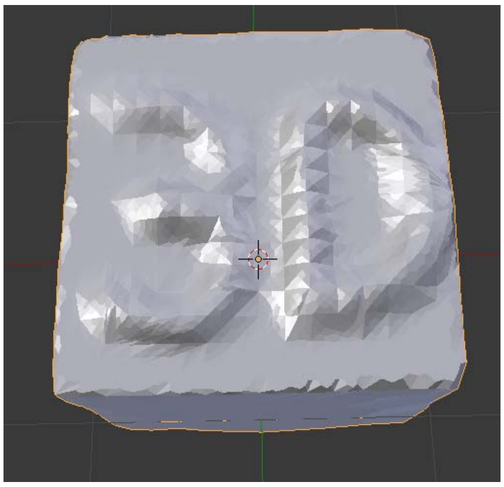

Grab / push (Inwards)

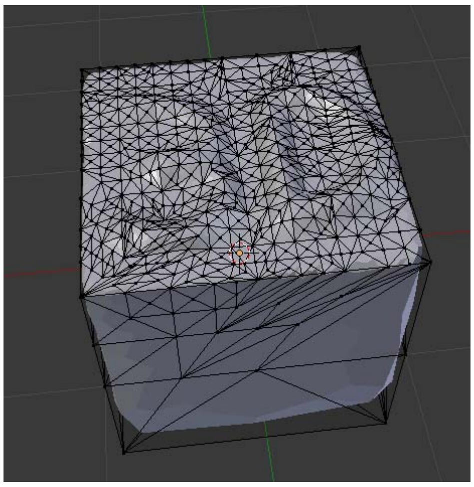

Draw (Add)


Draw (Subtract)

6.2.2.2 Dynamic topology

This is a sculpting tool offered by blender that allows for subdivision of meshes as required. The subdivision occurs when an artist's brush comes into contact with an area of the model. Below shows the influence of the technique as a draw tool is only used on the top face of the model. The top face (with 3D drawn on it) receives numerous sub divisions while the faces further away receive less sub divisions. Having this technique allows to artist's to sculpt complex shapes from simple meshes (Blender.org, 2013).


Draw tool with Dynamic Topology enabled


Effect of Dynamic Topology on the mesh.

6.3 Volume sculpting

Some of the earliest attempts at performing volume sculpting were by **Galyean and Hughes** (Galyean & Hughes, 1991). In their paper they present how they set about developing an application that could perform free form sculpting that takes input from a 3D device. The 3D device controlled the tool that performed manipulation of the object.

The model in their sculpting environment was represented as voxel data, this voxel data is then converted to a polygon structure by use of the marching cube algorithm. The world is described as a grid where the clay model is described by having a value of 1.0 while empty space is described with a 0.0. The tools they used on this model worked by simply changing these values.

Some of the tools they provide to interact with the model were an additive brush which could paint / add on to the object by simply hovering the tool over it. A heat gun style tool was present for the purposes of knocking back or cutting away the material it came in contact with. Sandpaper like tool was also experimented with for the purposes of filling and smoothing ridges and valleys.

To interact with the model they adapted a similar technique to 2D painting. Where painting is performed by moving a brush over a canvas and changing the pixel values beneath, applying to 3D was not a trivial task. 2D is viewed from a single view point whereas a 3D structure can be viewed from any direction. Also the issue of discerning the border meant that some form of thresholding algorithm was required. For displaying purposes they used a 10 x 10 x 10 voxel array which was allowed to have full access to all the drawing tools while for high resolution (30 x 30 x 30) voxel array was given limited tools to prevent overloading the processor.

Computation of the iso-surface was a concern (especially considering this was early 90's) where it $n \times n \times n$ arrays has a big O notation of O (n^3). Performing a draw on the whole model often would be very expensive. What they realised though was that the sculpting tools only ever had an effect on small areas so there was never a need to perform a full re-drawing of the model. To determine which voxels were affected and should be redrawn, they employed a *hashgrid* which is where they divided up the screen into a grid and associated a linked list collection with each cell. If a cube that has polygons that contribute, and if it overlaps it is added to this list. If a cube is determined to have been affected by the sculpt tool it has a flag set and the marching cube algorithm only affects that voxel in the re-draw pass.

Wang and Kaufman later too researched into developing also a user friendly volume sculpting application (Wang & Kaufman, 1995). They noted that Galyean and Hughes research worked fine for clay like structures, but it could not represent true to life realistic objects. Also they used a mouse as oppose to a 3D input device as they felt it was more natural.

Wang et al set about creating an easy to use tool that could be used for giving a "first pass" model design that an artist may choose to use over a 2D drawing. Similar to Galyean and Hughes approach they also decided to minimize computation of the model by performing updates on the areas affected by the modelling tool by way of ray casting as *hashgrid* projection calculations used by Galyean et al.

The employed a 3D volume raster (voxel grid) to store information on the objects being manipulated and also facilitated the modelling of multiple structures in a single scene. They also introduced a world to local coordinate system that allowed rotating of individual structures.

The modelling technique was also an interesting development as they stored the tool also in a 3D volume raster. This meant for a tool action to be performed they simple placed the tool over area and performed a Boolean operation.

The Boolean diff operation for removing material was as follows:

$$A\ diff\ B = A - AB$$

While the union operater allows adding to the volume as follows:

$$A \cup B = A + B - AB$$

For the purposes of rendering, ray casting was employed, where rays were cast into the 3D raster to determine intersections and pixel colours. They also made use of their own special antialiasing algorithm that determined if a ray was close to an edge and in doing so give a smooth transition between object and background. The ray casting technique also made use of progressive refinement, where, a fast low quality image is rendered followed by updates that bring it up to high quality. This is so a user is seeing something on screen rather than having to wait for all calculations to complete.

6.4 Virtual clay modelling

Dewaele et al in their research paper present a method of sculpting clay in a digital environment (Dewaele & Cani, 2004). One of the issues they uncovered and motivated their research was that real time clay manipulation in 3D was by and large still an unsolved problem.

The researchers identify clay as being as “simple and intuitive way to create complex shapes; even children use clay at school” and also that “many artists prefer expressing themselves with real materials instead of using a computer”.

Clay they found was perfect for modelling because it exhibits such properties as plasticity, mass conservation and also surface tension that holds it together for the most part. So getting the "benefits of real clay into a computer based modelling system" was a true motivator for the researchers.

As mentioned, the characteristics of clay exhibit three main properties that make it suitable for sculpting, the researchers found that:

- "Real clay mainly undergoes totally damped, plastic deformations." Means that changes are largely irreversible. Clay does exhibit some elastic properties (a memory that wants to return to original form) but overall its effect is negligible.
- Clay is also uncompressible; it preserves volume during transformation and is referred to conservation of mass.
- Finally, clay generally holds together which can be attributed to its surface tension.

So to implement this, the researchers, developed three layers or rules to follow when designing there application.

The first handles the effect of twists and bends, when implementing, they tried firstly using a Finite Element Mechanism and spring mechanics on the entire modal but found that it was too slow. They instead found employing fluid mechanics theory whereby "when an element in a viscous fluid moves, it pulls nearby fluid elements with it" therefore it exhibits a propagation effect.

The second enforces volume conservation and the third serves to enact a surface tension mechanism that keeps the volume as compact as possible. When performing a manipulation, if they found a cell where its density was greater than 1, the excess would be distributed to the surrounding cells. If the cells absorbed all the excess, then the process stops.

The final control layer aims to keep the gradient of densities near the surface at an acceptable value. Basically the intent here is to avoid expansion whereas the previous step was avoiding contraction. Even still they found that some pieces escaped leaving a bread crumb effect, they absorbed it back in to prevent distracting the user.

For rendering, similar to the techniques detailed earlier, they made use of the marching cube algorithm on areas that were affected by sculpt tool changes. If they found a large scale change had taken place then they employed a fast rendering technique that used ray casting. The ray casting had the affect similar to that of throwing a cloth over an object. However rendering this alone with ray casts coming from a single view point meant that if there two separate objects overlapping then they would be seen as been merged. To overcome this they performed additional ray casts in areas where they found quick changes occurring in the z buffer values.

7.0 Research Questions

The purpose for this research has two primary goals that go hand in hand as of equal importance. These are to investigate the use of WebGL while trying to determine is it a viable option for developing a volume sculpting application.

These form the questions we want to answer:

- How viable is JavaScript, being a dynamic language, for creating large scale applications?
- How can JavaScript deal with large computations, how can bottlenecks be dealt with?
- Can a volume sculpting application be implemented in WebGL?
- Can the use of WebGL help in avoiding a vendor lock in?
- Is THREE.JS a good choice of wrapper library for developing 3D applications? Can it boost productivity over using pure WebGL?
- When creating a volumetric model, how can the volume be manipulated and rendered?
- How can material properties such as elasticity be replicated or mimicked?
- Can the use of spatial data structures be used improve performance in this project?
- What are the possibilities with using volume sculpting?
- Ultimately, is there a future in WebGL?

8.0 Research Methodology

This thesis was written with two goals in mind. We want to design and develop a 3D digital sculpting application that takes volume into account rather than the traditional empty mesh sculpting applications.

With the ability of now being able to develop 3D applications / games with WebGL opens up a whole new cross platform market. This forms the next goal which is to deliver this content through the use of a web browser alone.

With the goal of the thesis defined, it set the path for the research and literary review. Volume rendering/graphics was the first port of call in the investigation as it is a well-established research area into how volume data is taken into account when rendering. This is a huge area having a lot of medical applications when it comes to rendering 3D models based on 2D data acquired from CT/MRI scans.

To find relevant research material, IEEE and science direct provide an excellent source. Other sources included the online college library as well freely available papers on the internet such as Google Scholar.

To begin with a gentle introduction before jumping into 3D volume data rendering, an overview into data visualisation was first researched to present what techniques were used to present information such as scalar data or tensor data.

Next began the investigation into the field of volume rendering. This is a huge area and required quite a bit of reading to establish the key important points so that more targeted research could be focused in those areas. This mainly came down to investigating indirect rendering using traditional pipeline technology or direct rendering techniques.

The previous chapter on volume rendering spurred another area of research which was the use of spatial data structures to optimise searching 3D volume data. The chapter delivers an overview of different data structures but focuses on the Octree due to the popularity in papers researched.

A discussion on WebGL followed next as this will be the product delivery medium. An introduction is given, a brief overview of the pipeline and some security issues associated with the technology and how the risks are mitigated.

The final research chapter deals with digital sculpting. This chapter is in essence the glue that ties all the previous chapters together in the unified goal of the thesis into creating a volumetric sculpting application. The chapter begins by giving a brief introduction into the methodology behind the traditional mesh modelling and sculpting techniques. Following that, the discussion moved into looking at the research that has been performed in the area of implementing volume sculpting. The chapter then closed out by looking at research done into creating virtual clay like modelling tool and examining the techniques used.

9.0 Design

9.1 Test Plan

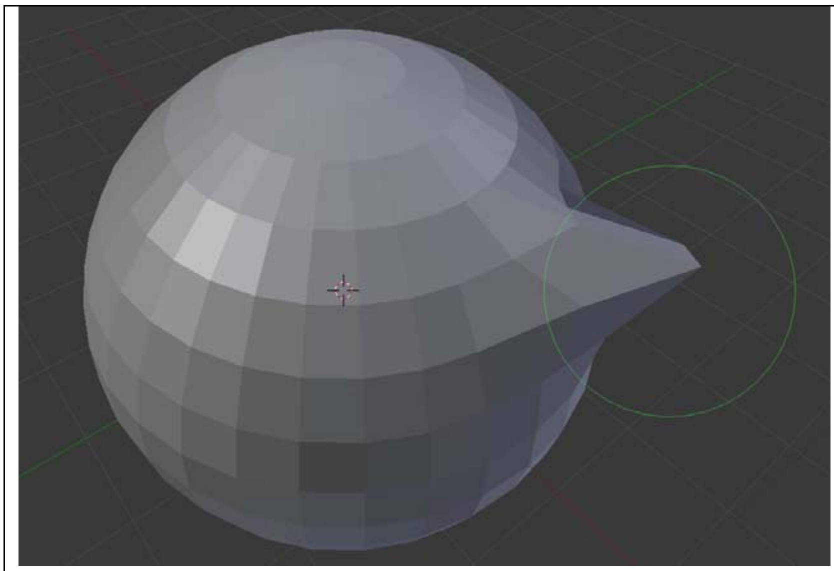
The goal is to implement volumetric sculpting in the web browser. The requirements identified for this for a user to perform digital sculpting is a machine which is reasonably up to current specifications. This machine must run Chrome, Firefox or Internet Explorer 11 and have JavaScript enabled.

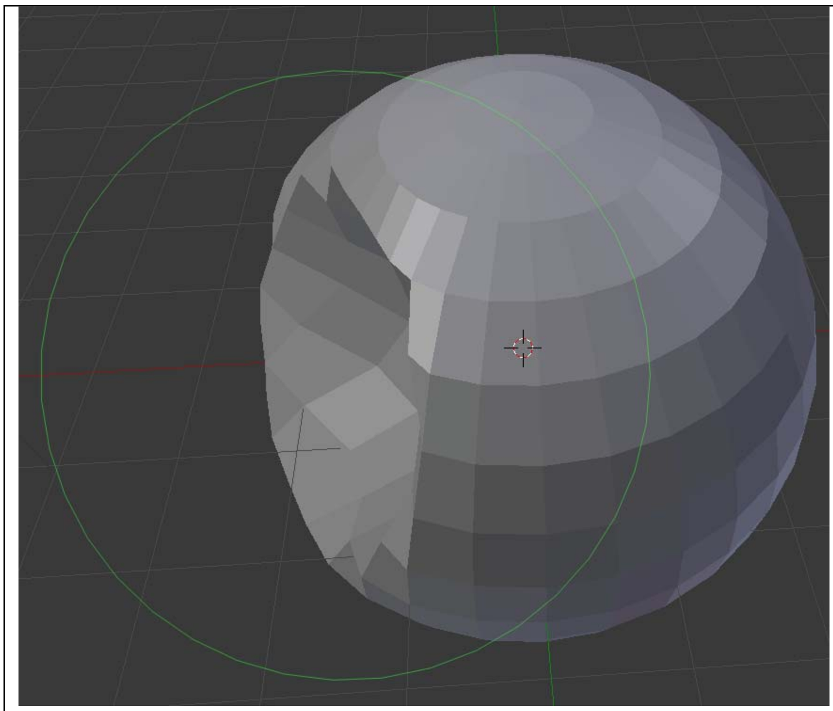
The following are the project objectives from a user's standpoint:

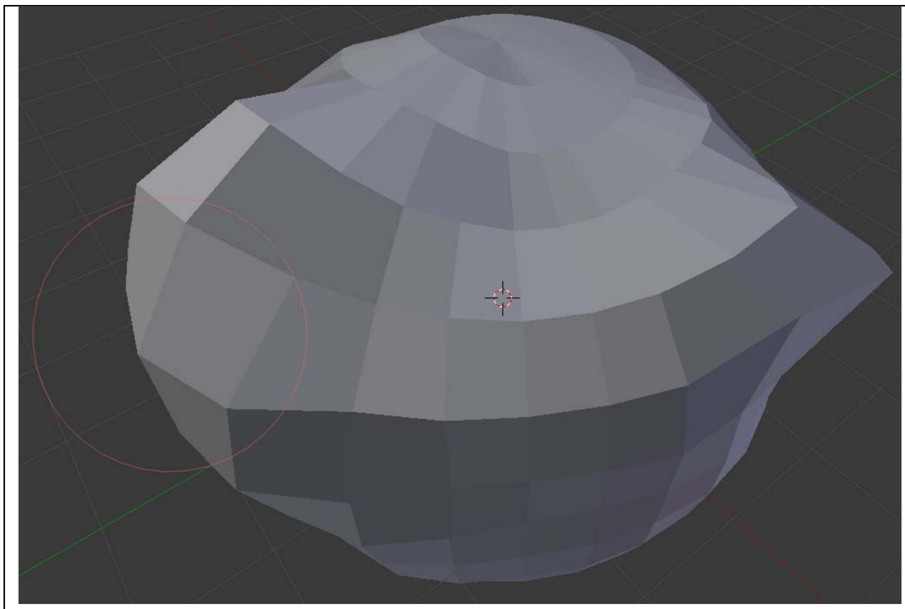
- The user should be able to pan and zoom around a solid object in 3D space.
- The user should be able to add and subtract without causing mesh destruction.
- The user should be able to manipulate the object by pushing and pulling.
- The whole operation should run without any penalty on the machine i.e. slowing down.

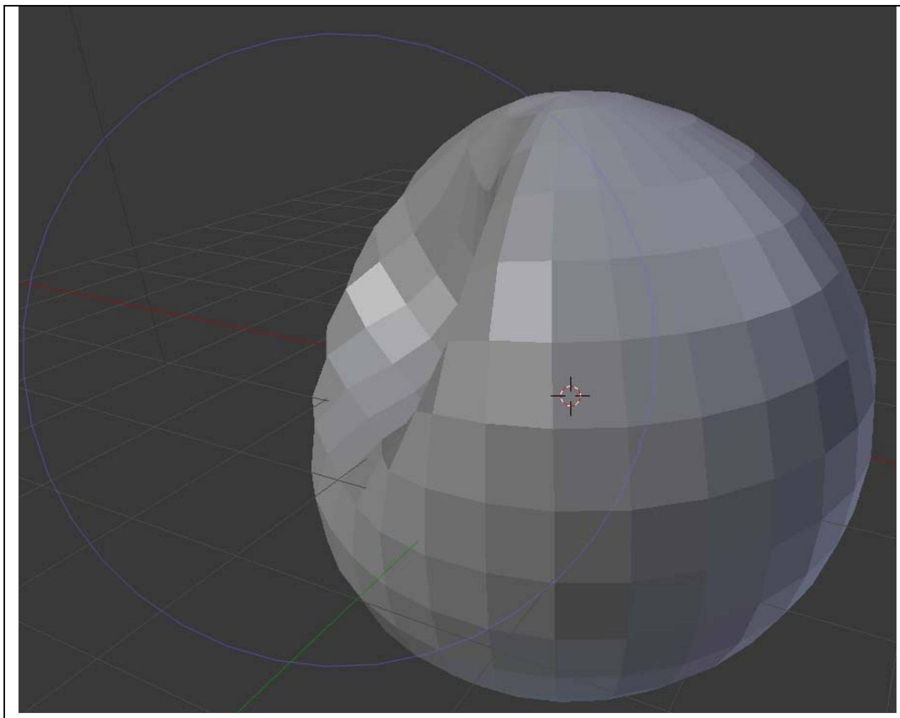
Below is a series of images taken from the sculpting environment in Blender 3D. The image shows a small range of operations that can be carried out on a geometric model.

The goal is to replicate this but with a volumetric structure. The objects shown are empty shells and so are not true to life. Sculpting with clay in real life has volume and it is the aim of this thesis to replicate as best as possible and determine if it is viable and does it offer any benefits.

 Figure 27 - Blender Grab (Pull)	Currently: The blender model when pulled simply gives the impression of skin being stretched. Expected: As it will be volumetric and each voxel tied together, the whole model should distort while maintaining a constant mass.

 Figure 28 - Blender Grab (Push)	Currently: Blender when the push tool is used, the reaction is absorbed with no effect on the other side. Expected: When push is performed, maintaining a conservation of mass, the other side should bulge out.

 Figure 29 Blender Draw (Add)	Current: Blender adds to the model by inflating the geometry resulting in a bulging effect
	Expected: The application should literally draw by adding new structure on top of the existing structure

 Figure 30 - Blender Draw (Subtract)	Current: Blender here performs the opposite to the above. The draw tool with subtract enabled causes the geometry it effects to deflate or shrink.
	Expected: The application should cut away the geometry that the tool influences revealing the internal volume of the model

9.2 Prototype Development

The first experiment of developing a prototype was to get a pure rather crude voxel object that represented scalar data contained in a world. The grid seen below represents the world. The world is divided by a grid with each block being known as a voxel. Within this world a sphere is placed by way of a mathematical representation (Centre and radius). A cursor is placed at the bottom corner of this world and on every hit of the return button, the cursor moves along one block in the grid. On every move, the cursor evaluates whether it is with the bounds of the sphere or not. If the centre is found to be within the bounds of the sphere, a cube is placed at the cursors location, the cursor then simple moves on to the next voxel when the return key is pressed. If a voxel is evaluated as being outside of the bounds of the sphere, no action is taken and the cursor will wait for the return key before moving on.

The result of this process after applying a material and suitable light source is a heavily pixelated but discernable representation of the mathematical sphere.

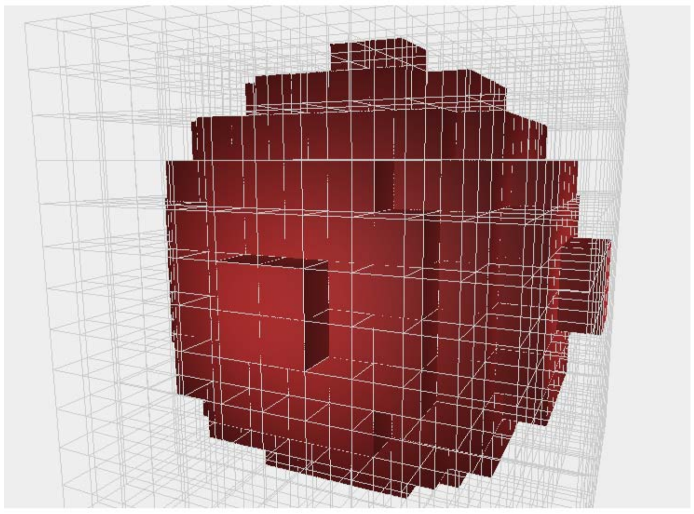
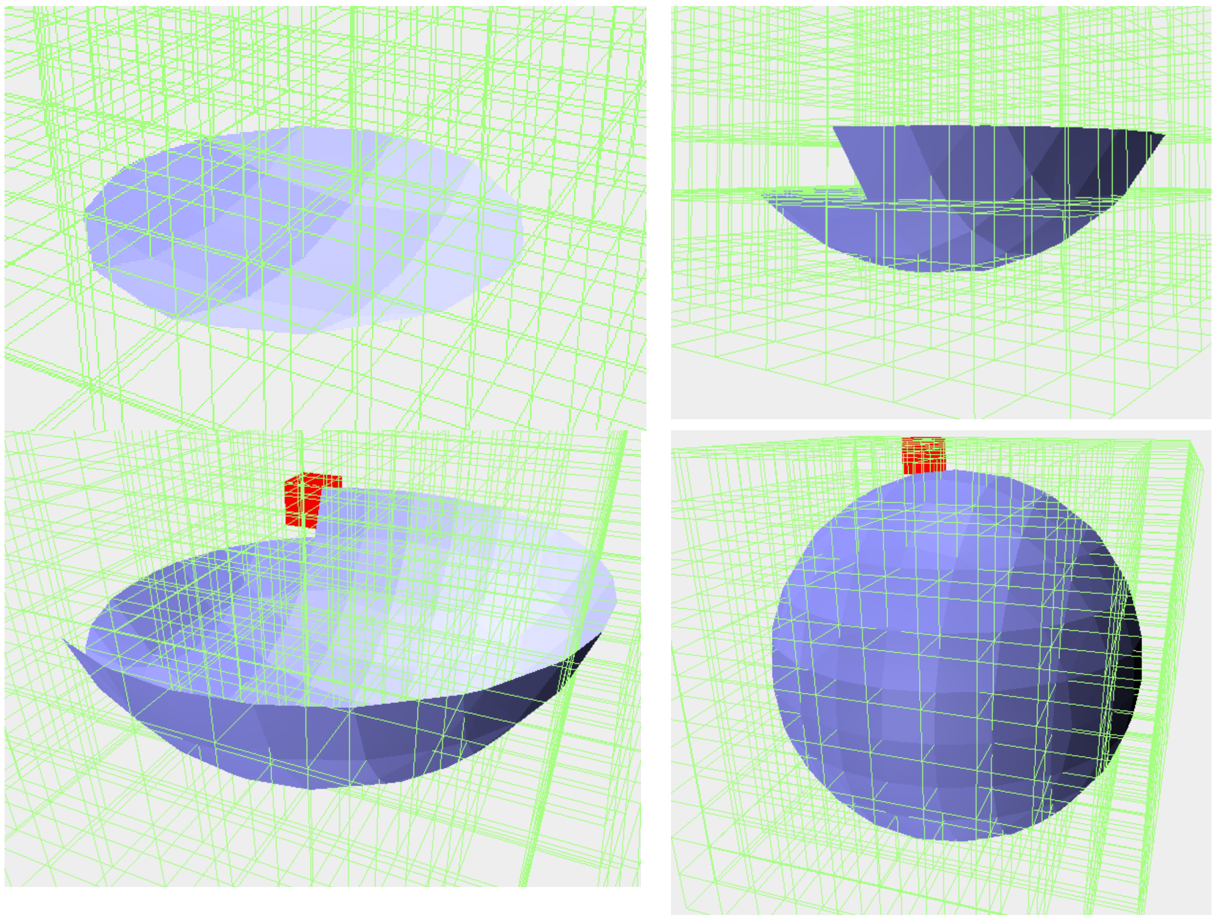


Figure 31 - Results of the first prototype

Moving on from the above experiment, the next prototype is built on the reading of a paper by Paul Bourke on the marching cube algorithm which was explained earlier. (Bourke, 1994). What this does over the previous example, is the ability to offer a contoured geometric shape that doesn't look pixelated while still working with a volumetric structure. Work performed by Lee Stemkosi into ThreeJs and the marching cube was also instrumental in helping to understand the creating of polygons that formed the surface of the volumetric sphere (Stemkoski, 2013).

A "Solid" sphere (represented only by radius and location) again was place in the centre of the world represented by the grid.   The steps of the program are as follows:

1. Move the cursor to the next cell / voxel.
2. For each of the eight vertices of the cursor evaluate whether inside and determine a look up index.
3. Go to the edge table to determine the faces impacted.
4. Perform a interpolation on each edge to determine where a vertex should be placed.
5. Create a face that joins the vertices and add a material to it.
6. Move to next cell.



9.3 Next step

So far, being able to implement a Marching cube algorithm was a huge step forward as it means contouring can be applied to the model rather than having a pixelated model. The next step in development will be to abstract what we have learnt here and apply it towards developing a usable volume rendering engine to which we can apply to our end goal.

Following this, experimentation will begin into trying to introduce the mechanics necessary to offer a realm of realism when the structure is manipulated. This means basically if a push force is applied, then the model should behave as a dense structure similar to the properties of clay while maintaining a conservation of mass.

9.4 Risks

Type	Solution / Mitigation
Using THREE js framework for development as it is an unfinished framework.	The underlining calls are still WebGL, so what's not available can be added as the source is open.
JavaScript code needs to be maintainable as possible, while having a good idea design methodology in a statically typed language. Can JavaScript have the same good coding practices?	Plenty of material is available that teaches good design methodology when it comes to JavaScript so with some study, good practices can be introduced into the code.
Developing in JavaScript will require a good debugging platform	Use of JetBrians WebStorm IDE provides an excellent debugger that provides operations similar to Visual Studio that allows peering into an Object's properties.
Testing is important to insure compliance with requirements, can JavaScript be tested?	There are many packages available that allow the testing of JavaScript code i.e. QUnit
WebGL is not supported on Internet Explorer browsers before version 11	Build some browser detection and warning into code.
Implementing realistic clay like mechanics will require a great deal of time and experiment to give authentic look and feel	This will just require time to examine some more research into the area of fluid mechanics and mass conservation.

10.0 Development

10.1 UML

While still at an experimental design stage, this UML represents the requirements of the project. The THREE JS library is imported into the project and used for scene, lighting, camera and rendering purposes. A custom Octree will be developed and is represented by the world class. The Octree will have children and the node class can be used as either a node or a leaf. The position enum (or JavaScript equivalent) is used to distinguish each of the eight blocks of a subdivided parent block.

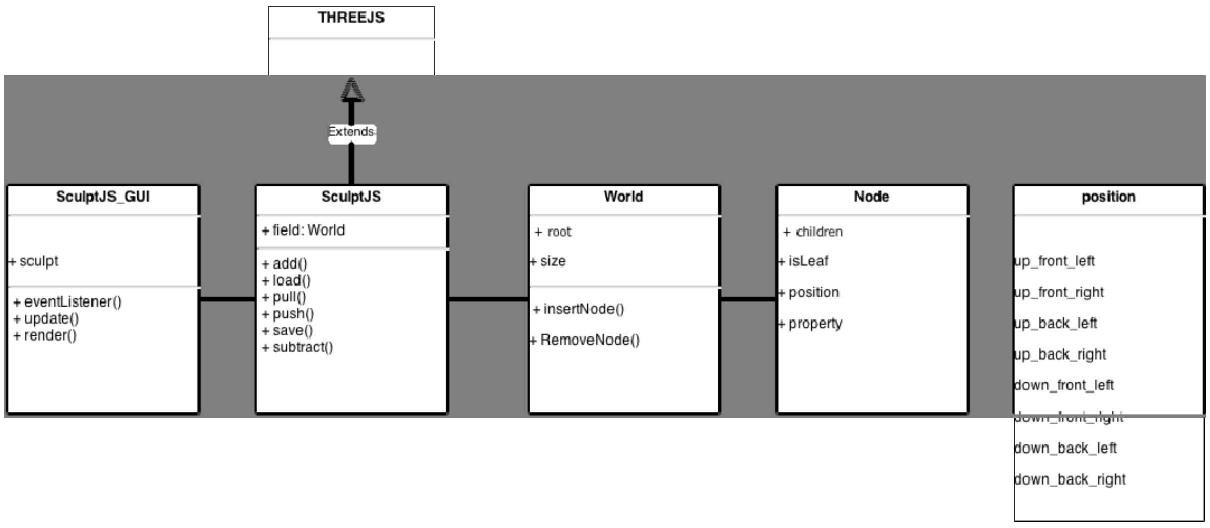
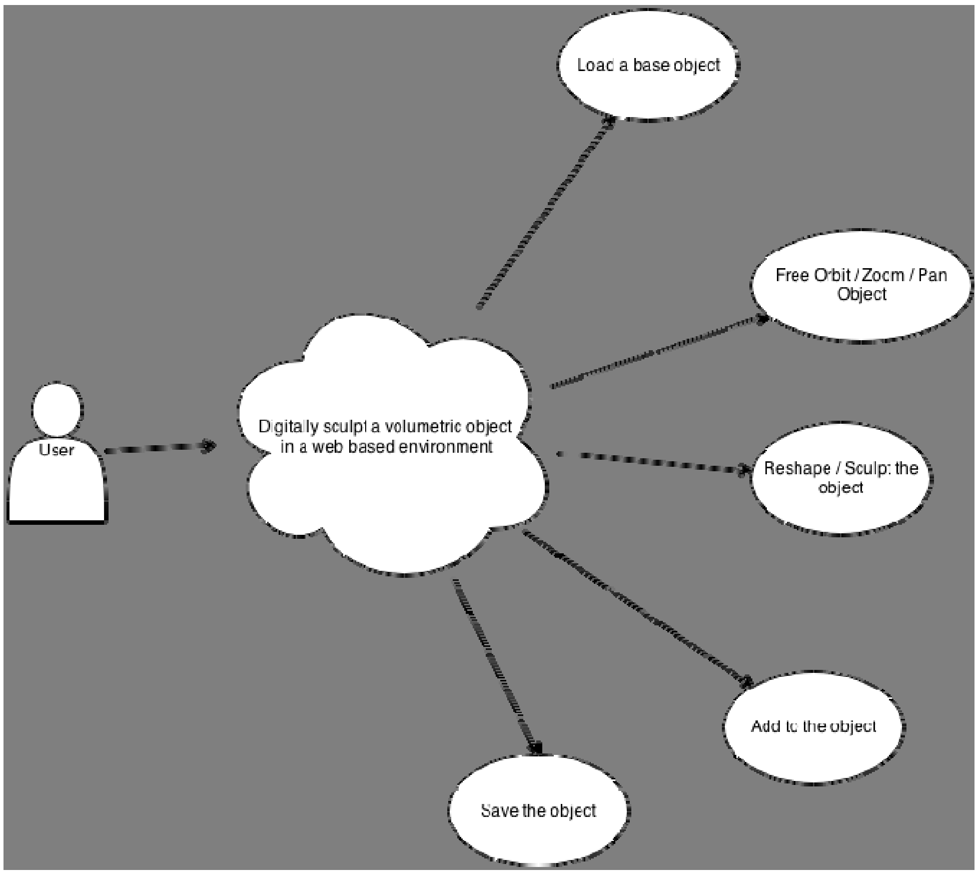


Figure 32 - Preliminary UML Sketch

10.2 Use Case Diagram


Load a base object
Free Orbit / Zoom / Pan Object
User
Digitally sculpt a volumetric object in a web based environment
Reshape / Sculpt the object
Add to the object
Save the object

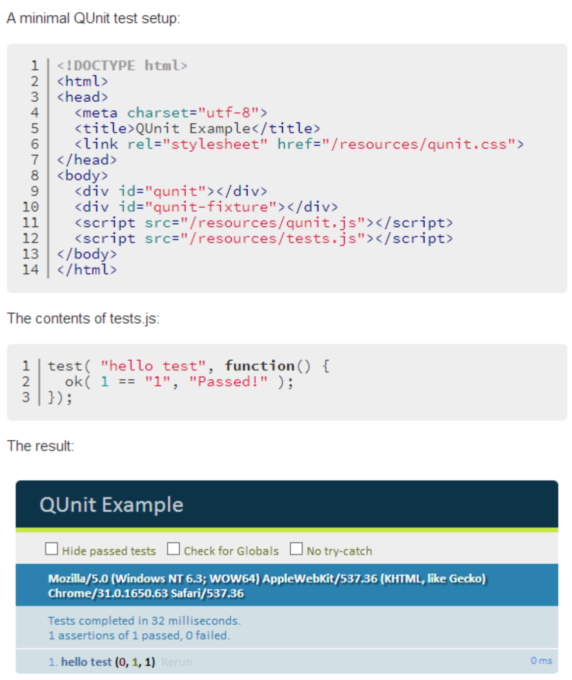
10.3 Development tools

10.3.1 ThreeJS / WebGL

This framework abstracts WebGL calls to a high level and in doing so allow an easier entry point while also speeding up development times.

10.3.2 QUnit - http://qunitjs.com/

QUnit is a testing framework that is provided by the same people who develop JQuery and JQueryUI as they use it for testing their projects. The following is a simple example from there website of how a simple test can be carried out.



A minimal QUnit test setup:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit Example</title>
  <link rel="stylesheet" href="/resources/qunit.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="/resources/qunit.js"></script>
  <script src="/resources/tests.js"></script>
</body>
</html>
```

The contents of tests.js:

```javascript
test( "hello test", function() {
  ok( 1 == "1", "Passed!" );
});
```

The result:

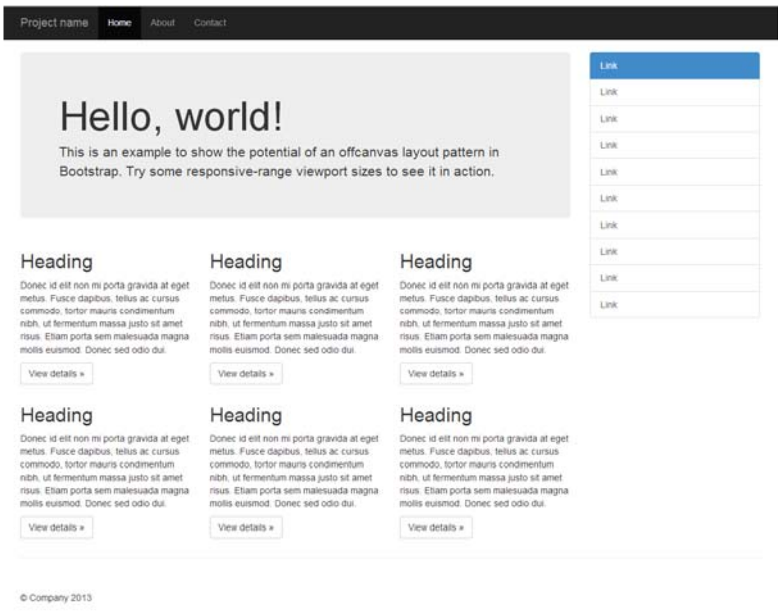
QUnit Example

☐ Hide passed tests ☐ Check for Globals ☐ No try-catch

```
Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/31.0.1650.63 Safari/537.36

Tests completed in 32 milliseconds.
1 assertions of 1 passed, 0 failed.

1. hello test (0, 1, 1) Rerun                                    0 ms
```

10.3.3 BootStrap - http://getbootstrap.com/

This was a project that was originally created by developers at twitter. Bootstrap is now a frontend framework that allows for fast development of responsive webpages that also target mobile web. One of its best features is the way in facilitates easy grid layouts.



Hello, world!

This is an example to show the potential of an offcanvas layout pattern in Bootstrap. Try some responsive-range viewport sizes to see it in action.

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

View details »

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

View details »

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

View details »

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

View details »

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

View details »

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

View details »

© Company 2013

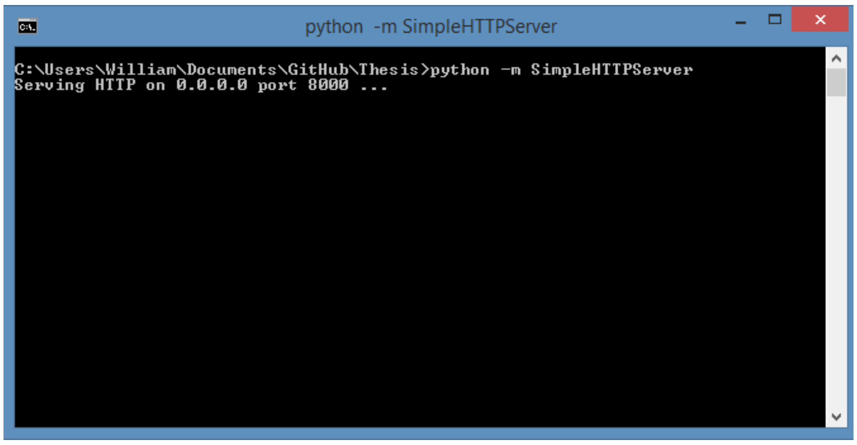
10.3.4 JetBrains WebStorm 7 IDE

WebStorm is a premium IDE developed by JetBrians and targets web development specifically. WebStorm provides excellent JavaScript support as well as an excellent debug facility. The debug alone is enough to warrant the use and cost of this tool in the development of the project.

10.3.5 Python Simple Server

Python 2.7 comes with a web server built-in. This can be accessed by navigating to the relevant root folder that a user wishes to make available for local web testing and simply typing:

```
python –m SimpleHTTPServer
```



```
python  -m SimpleHTTPServer
C:\Users\William\Documents\GitHub\Thesis>python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

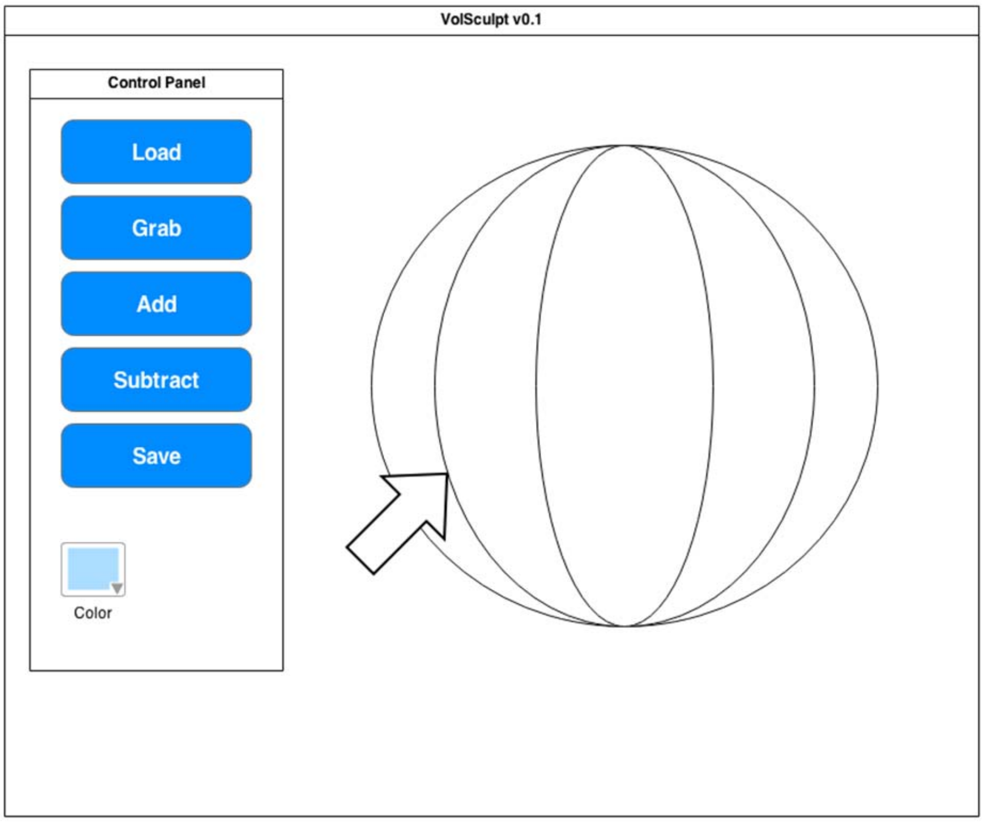
This becomes especially useful when it comes to testing any ajax calls that load local files as this typically violates browsers cross domain origination policy. This happens because the browser sees “file://” as being a different domain. An example of this will when it comes to using web workers which is JavaScripts answer to concurrency. Web workers are stored and loaded as a separate ‘.js’ file, thus the need to setup a local testing server.

10.3.6 Git Source control

GIT is a distributed source control system. Since git has no central repository, it provides an excellent data redundancy in the event of a system failure. Git is well supported with GUI interfaces that take the edge off of learning all the commands and this will be the choice for this project.

10.4 Mock User Interface

This diagram gives a mock view of the intended interface. The main screen will present the model which can have its view rotated by mouse control. A secondary control panel will provide access to different tools and settings.



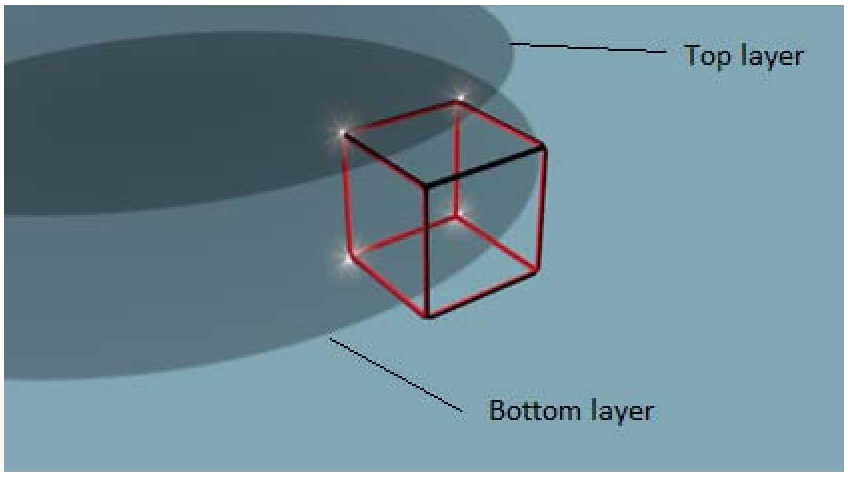
11.0 Developing the Volume rendering and sculpting engine

11.1 How does the Marching Cube code work in detail?

The method used here to implement the marching cube algorithm to solve our problem is based on work produced in a paper by Paul Bourke (Bourke, 1994) which is available at his website (http://paulbourke.net/geometry/polygonise). We as well as that will be adopting Lee Stemkoski (Stemkoski, 2013) method for producing faces to apply to the voxel. The works mentioned here were analysed and modified to suit the nature of the problem we are trying to solve. The method Bourke presents is efficient due to the nature that it is primarily based on look up tables. The first step in our code is to treat each voxel as an individual and produce a resulting mesh (if any) from passing a reference of the voxel to a Marching cube function along with a threshold and the material we wish to apply to it.

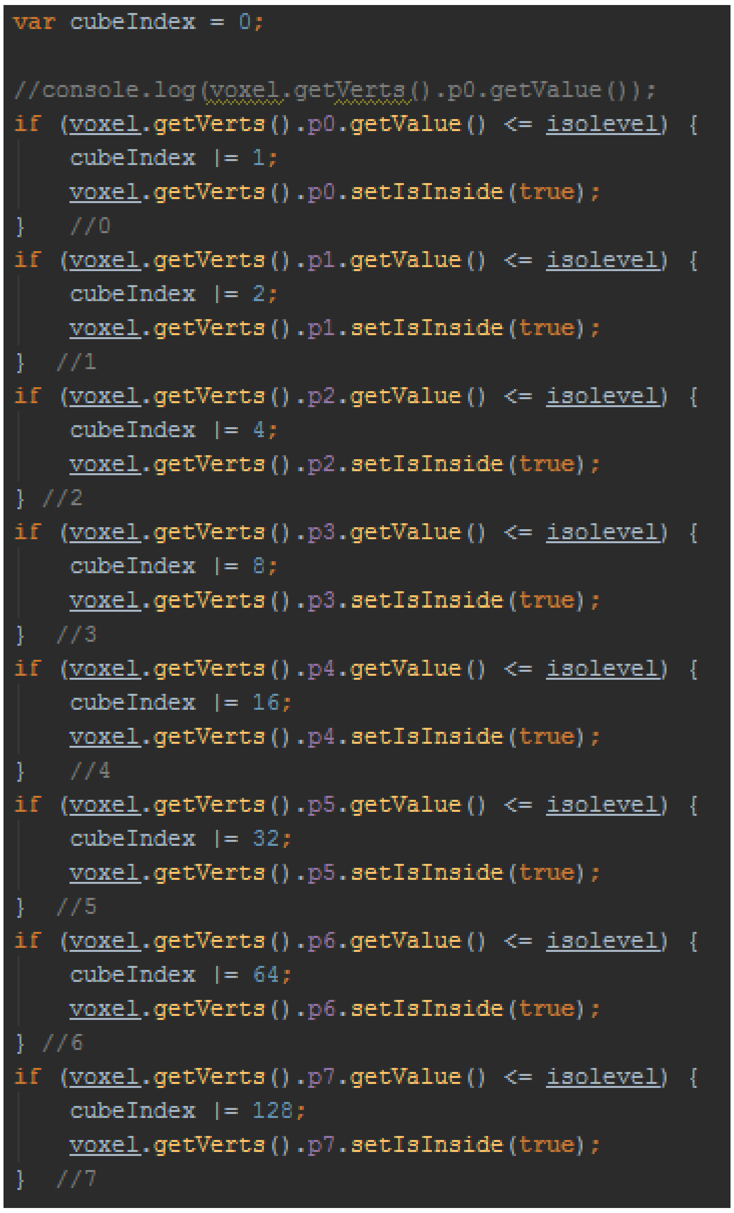
```
public static MarchingCube(voxel:VoxelState2, isolevel:number, material:THREE.MeshPhongMaterial):THREE.Mesh
```

The next step involves determining where each individual corner lies, this can either be inside or outside and is determined by the threshold I.e. if a value is below the threshold, it is said to be inside while a value greater than the threshold is outside. The diagram below shows four of its corners on the inside of a 'volume' (marked with a star) while the four others lie outside.



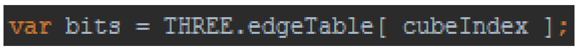
The algorithm shown below produces a number which is then used to look up an 'edge table'. These edge tables were originally produced by Cory Gene Bloyd (see Paul Bourkes page) and demonstrated by Bourke in his paper.

The first step that is required is to 8 bit cube index where each bit represents a vertex.



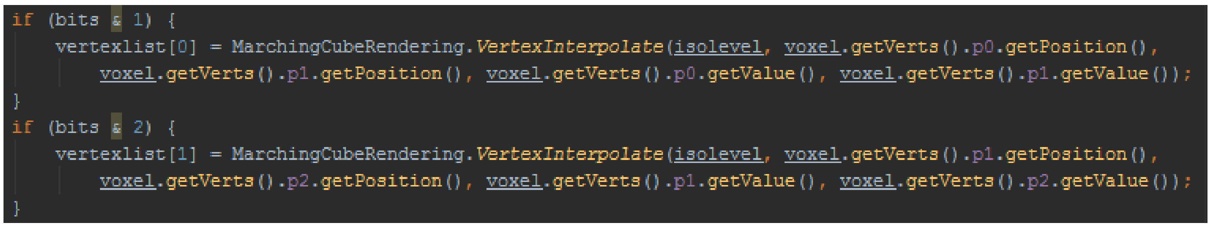
```
var cubeIndex = 0;

//console.log(voxel.getVerts().p0.getValue());
if (voxel.getVerts().p0.getValue() <= isolevel) {
    cubeIndex |= 1;
    voxel.getVerts().p0.setIsInside(true);
}   //0
if (voxel.getVerts().p1.getValue() <= isolevel) {
    cubeIndex |= 2;
    voxel.getVerts().p1.setIsInside(true);
}  //1
if (voxel.getVerts().p2.getValue() <= isolevel) {
    cubeIndex |= 4;
    voxel.getVerts().p2.setIsInside(true);
} //2
if (voxel.getVerts().p3.getValue() <= isolevel) {
    cubeIndex |= 8;
    voxel.getVerts().p3.setIsInside(true);
}  //3
if (voxel.getVerts().p4.getValue() <= isolevel) {
    cubeIndex |= 16;
    voxel.getVerts().p4.setIsInside(true);
}   //4
if (voxel.getVerts().p5.getValue() <= isolevel) {
    cubeIndex |= 32;
    voxel.getVerts().p5.setIsInside(true);
}  //5
if (voxel.getVerts().p6.getValue() <= isolevel) {
    cubeIndex |= 64;
    voxel.getVerts().p6.setIsInside(true);
} //6
if (voxel.getVerts().p7.getValue() <= isolevel) {
    cubeIndex |= 128;
    voxel.getVerts().p7.setIsInside(true);
}  //7
```

The result of the cube index then is used to find the edges impacted by looking up the edge table as described earlier.



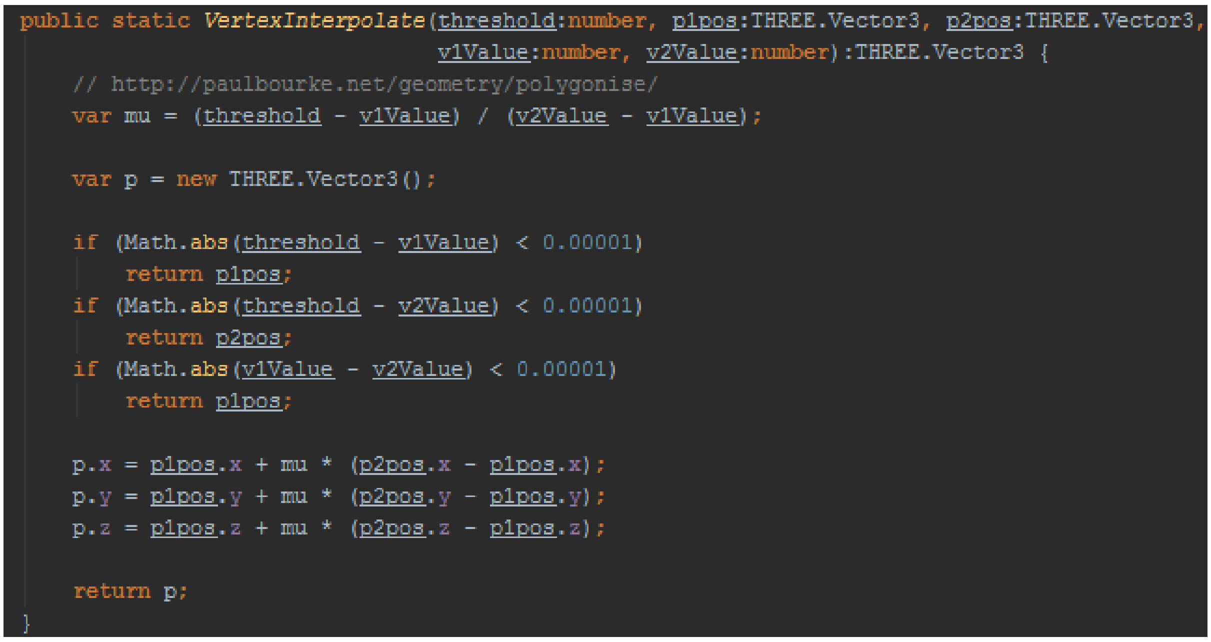
```
var bits = THREE.edgeTable[ cubeIndex ];
```

This results in a 12 bit number where all the individual bits represent an edge, where, 1 indicates it is cut while a 0 indicates it is not. The snippet below shows testing of the edges which were impacted and the result if so are passed to an interpolation function which is discussed next.



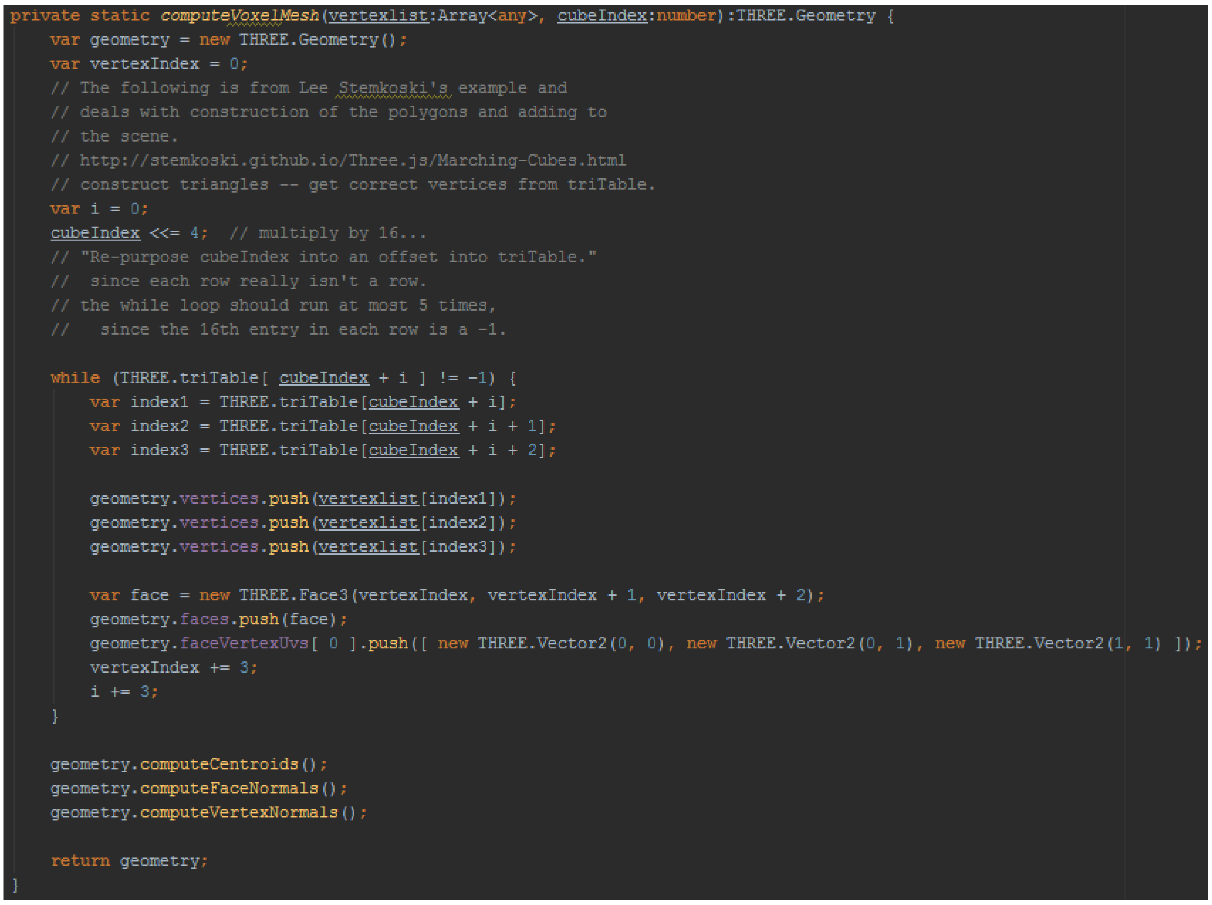
```
if (bits & 1) {
    vertexlist[0] = MarchingCubeRendering.VertexInterpolate(isolevel, voxel.getVerts().p0.getPosition(),
        voxel.getVerts().p1.getPosition(), voxel.getVerts().p0.getValue(), voxel.getVerts().p1.getValue());
}
if (bits & 2) {
    vertexlist[1] = MarchingCubeRendering.VertexInterpolate(isolevel, voxel.getVerts().p1.getPosition(),
        voxel.getVerts().p2.getPosition(), voxel.getVerts().p1.getValue(), voxel.getVerts().p2.getValue());
}
```

Once we have this result, the placement of vertices must then decide. For example, if and edge indicates that one of its vertices is inside and the other is outside, then a vertex must lie somewhere along that edge and this is where an interpolation must be performed.



```
public static VertexInterpolate(threshold:number, p1pos:THREE.Vector3, p2pos:THREE.Vector3,
                                v1Value:number, v2Value:number):THREE.Vector3 {
    // http://paulbourke.net/geometry/polygonise/
    var mu = (threshold - v1Value) / (v2Value - v1Value);

    var p = new THREE.Vector3();

    if (Math.abs(threshold - v1Value) < 0.00001)
        return p1pos;
    if (Math.abs(threshold - v2Value) < 0.00001)
        return p2pos;
    if (Math.abs(v1Value - v2Value) < 0.00001)
        return p1pos;

    p.x = p1pos.x + mu * (p2pos.x - p1pos.x);
    p.y = p1pos.y + mu * (p2pos.y - p1pos.y);
    p.z = p1pos.z + mu * (p2pos.z - p1pos.z);

    return p;
}
```

This function now aims to position a vertex appropriately between the two corner positions. A factor 'mu' is calculated and is multiplied by the distance that lays between the position 1 and position 2 of the edge. This result is then marked on the edge between Position 1 and Position 2 at the calculated distance from Position 1.



```
private static computeVoxelMesh(vertexlist:Array<any>, cubeIndex:number):THREE.Geometry {
    var geometry = new THREE.Geometry();
    var vertexIndex = 0;
    // The following is from Lee Stemkoski's example and
    // deals with construction of the polygons and adding to
    // the scene.
    // http://stemkoski.github.io/Three.js/Marching-Cubes.html
    // construct triangles -- get correct vertices from triTable.
    var i = 0;
    cubeIndex <<= 4;  // multiply by 16...
    // "Re-purpose cubeIndex into an offset into triTable."
    //  since each row really isn't a row.
    // the while loop should run at most 5 times,
    //   since the 16th entry in each row is a -1.

    while (THREE.triTable[ cubeIndex + i ] != -1) {
        var index1 = THREE.triTable[cubeIndex + i];
        var index2 = THREE.triTable[cubeIndex + i + 1];
        var index3 = THREE.triTable[cubeIndex + i + 2];

        geometry.vertices.push(vertexlist[index1]);
        geometry.vertices.push(vertexlist[index2]);
        geometry.vertices.push(vertexlist[index3]);

        var face = new THREE.Face3(vertexIndex, vertexIndex + 1, vertexIndex + 2);
        geometry.faces.push(face);
        geometry.faceVertexUvs[ 0 ].push([ new THREE.Vector2(0, 0), new THREE.Vector2(0, 1), new THREE.Vector2(1, 1) ]);
        vertexIndex += 3;
        i += 3;
    }

    geometry.computeCentroids();
    geometry.computeFaceNormals();
    geometry.computeVertexNormals();

    return geometry;
}
```

The above code which was originally produced by Lee Stemoski (http://stemkoski.github.io/Three.js/Marching-Cubes.html) is being abstracted here to form its own function. What happens where is we are constructing geometry which consists of faces from all the vertices produced during the previous interpolation stage. Three vertices form a face and this process continues until all vertices from the complied list in the earlier function has been used to construct a geometry which can then be returned. A short cut is been taken here by using a double face as the material which will be combined with this geometry that will then form the mesh that will eventually be rendered. Ordinarily, meshes only have one face to reduce computation time, and if it is the case of using a single face then it is the order of the vertices that decide this. Essentially meaning, care must be given to the order in which they are added. Using double face, this is now not important.

11.2 Developing the Volume rendering engine and testing with image stacks

This next experiment aims to produce a volume rendering engine leveraging on the theory presented above. What is hoped to be achieved is to produce a geometric model from a series of images similar to the medical approach. For this experiment we will use a number of images that were produced in Photoshop by using the gradient tool. The advantages of using a gradient over a solid image in volume rendering becomes clear soon as it allows for the use of a threshold which is demonstrated below.

As mentioned, the images were created in Photoshop and while we could pass these images (or upload at runtime) we decided instead that to reduce the burden on the client. We would pre extract the data by using a program that we wrote in Python that loads images from a file and stacks them according to file name, and then samples based on a grid that we have specified which should match the same grid format (voxel width and the number of voxels) as being used in our voxel rendering engine. This was purely a choice to reduce computation time; we have no doubt that this could have been implemented client side using JavaScript. The extracted data from the output of our python program was then placed in a JSON file which the rendering engine obtains from an Ajax request. It then processes this file and applies the values to each of the voxel corners in our defined voxel world.

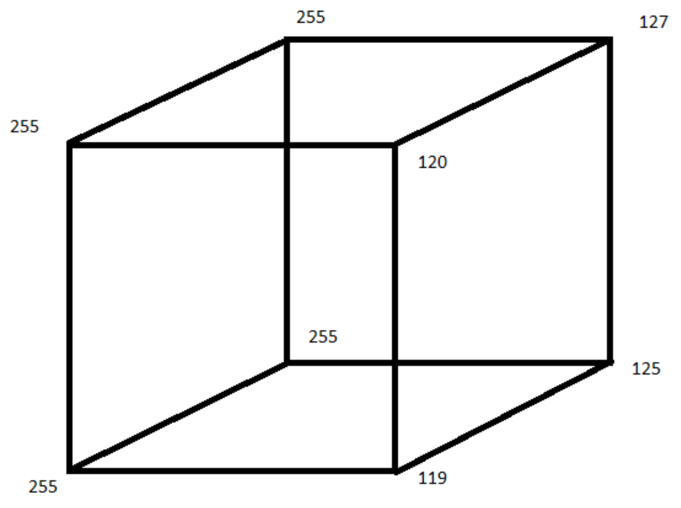


Figure 33 - Some sample pixel values applied as voxel corner values

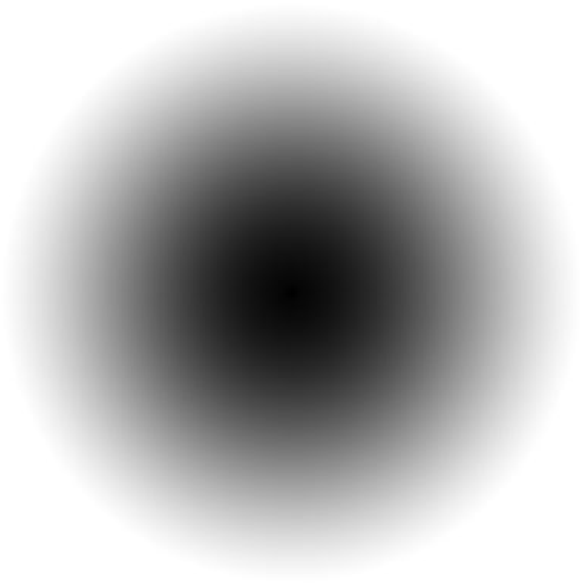
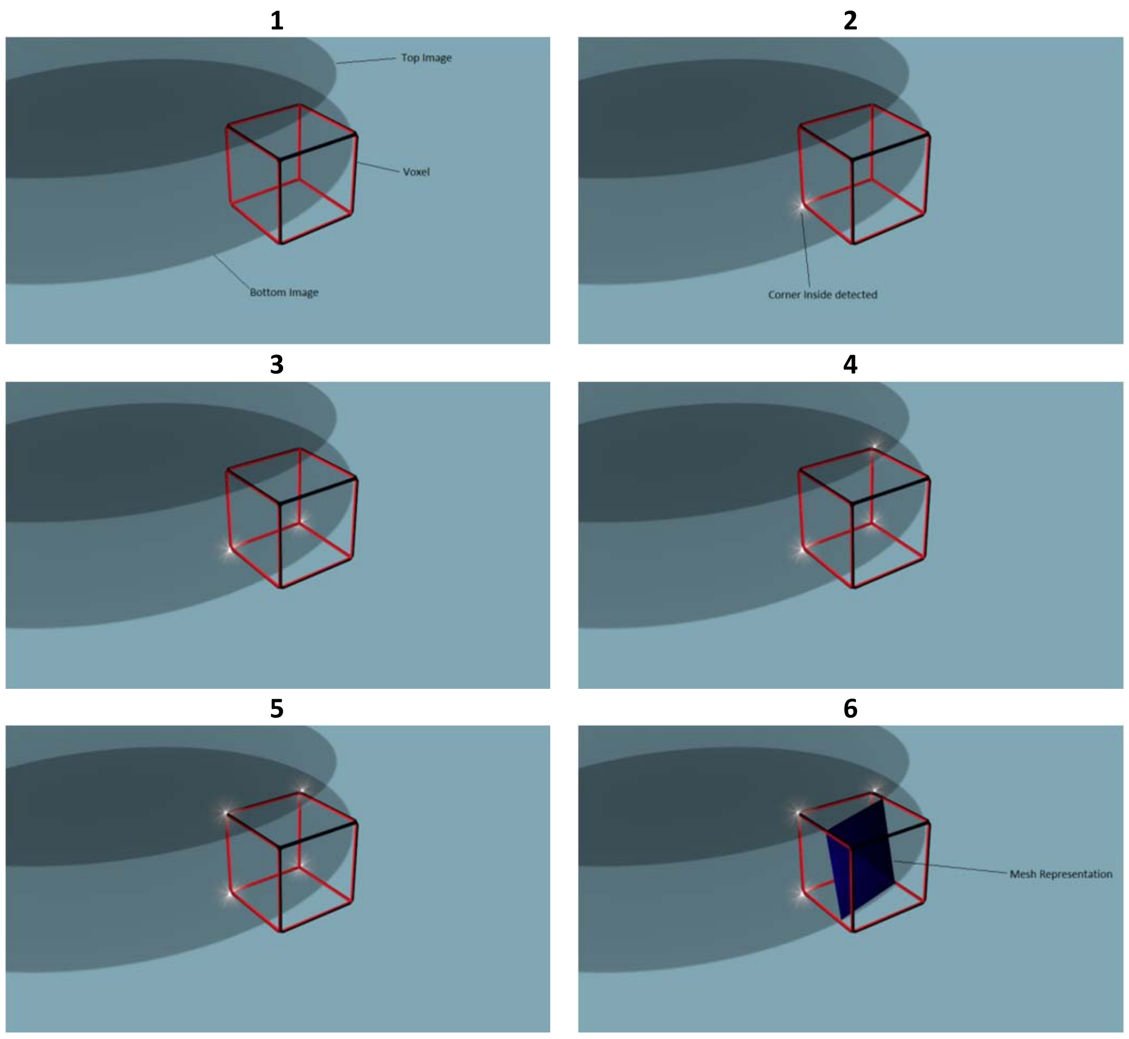


Figure 34 Sample image

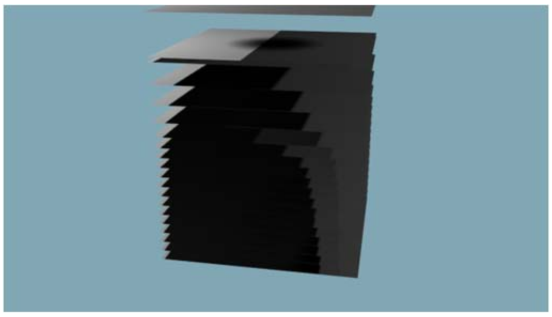


The above sequence of images aims to demonstrate next the steps taken when evaluating a voxel that lies between too image layers.

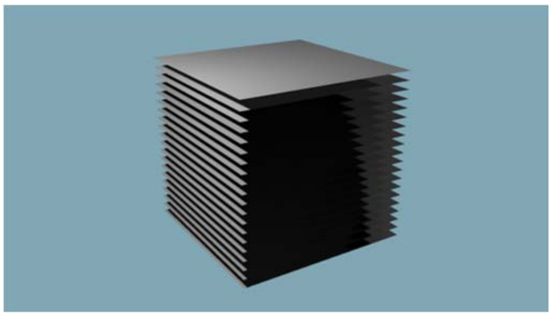
1. We deal with 2 images at a time stacked with a space between that are divided up into a grid of voxels (one voxel is highlighted here for clarity).
2. Next for each corner we need to determine whether it is inside or outside by applying some form of threshold. In the example above, it is an 8 bit image with pixel values from 0 (black) to 255 (white). If a value is below this threshold it is marked to be inside.
3. Inside Corner detected
4. Inside Corner detected
5. Inside Corner detected
6. Based on having a case of 4 corners inside, and 4 outside, the procedure is then to perform a lookup in the edge table to determine which edges are impacted. With that result, we then need to determine where to position vertices using an interpolation function so we can build up a surface geometry.

1

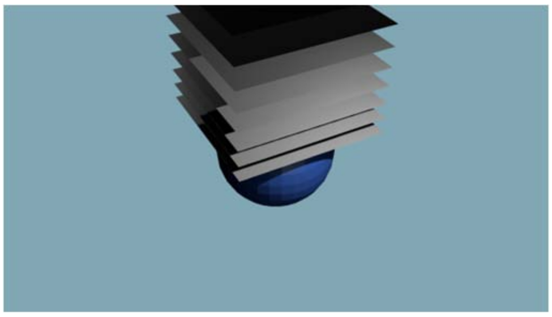
2



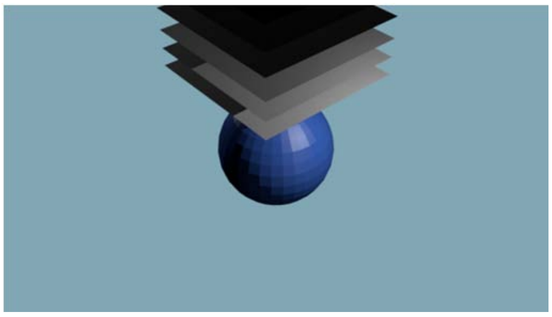
3



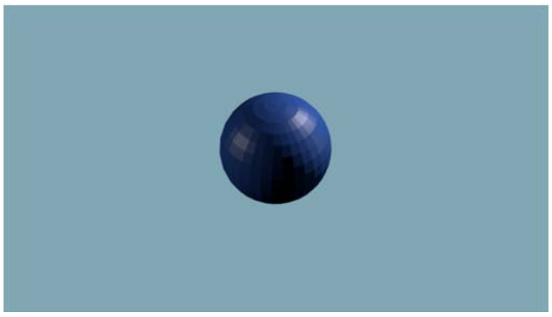
4



5



6



The above now illustrates the end goal of how our stack rendering engine aims work. We will build an image stack with a layer of voxels between the gaps. We then evaluate each voxel against a threshold; we then calculate and place the representing surface geometry. What we are then left with is a surface representation of that volume.

The diagram in the next page illustrates the path we took in constructing our solution. We will make use of both AJAX and Web Workers (described in detail in the next chapter) in order to introduce concurrency to make the program run as fast and as smooth as possible without impacting on the client with waiting times.

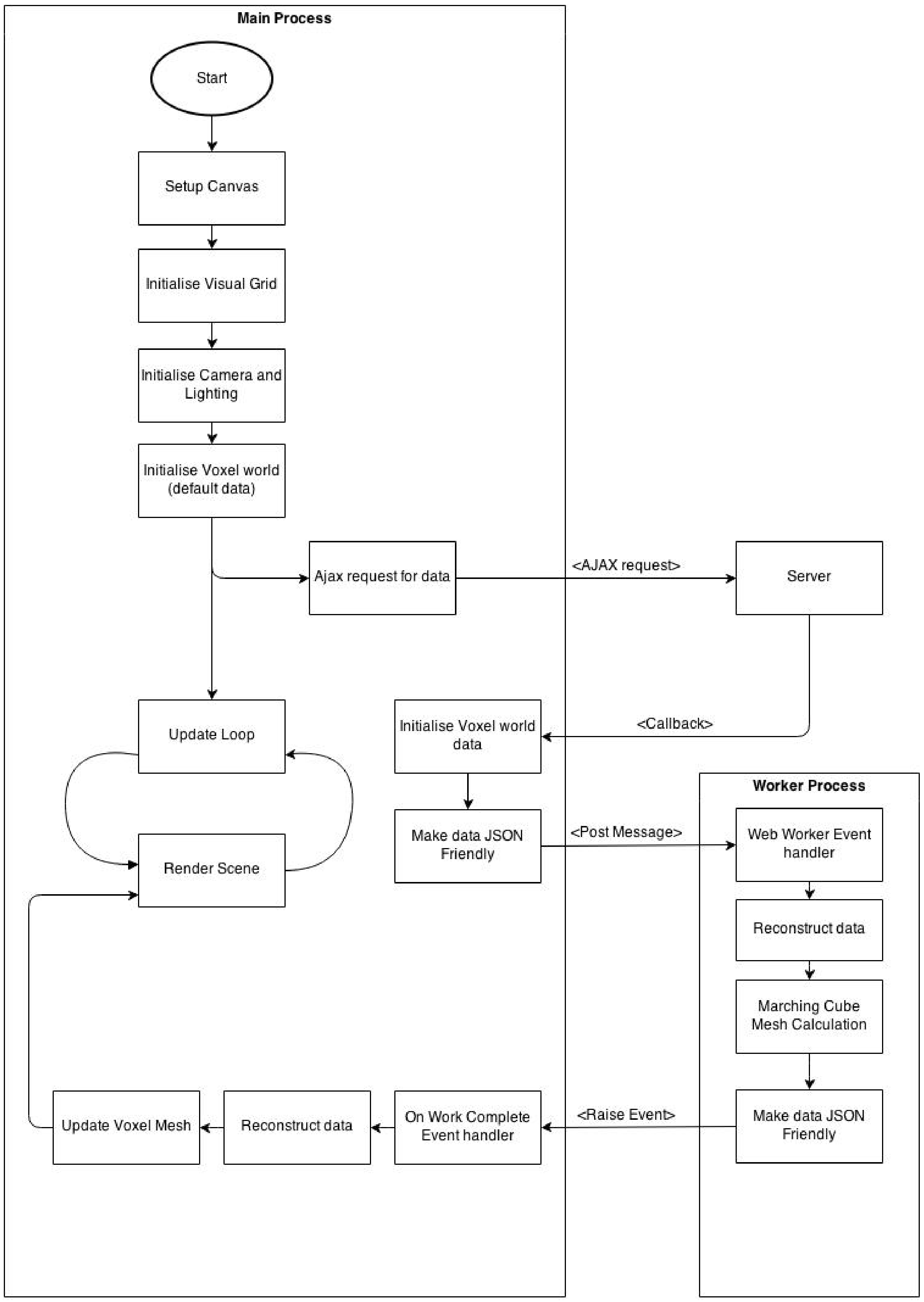


Figure 35 - On load sequence diagram

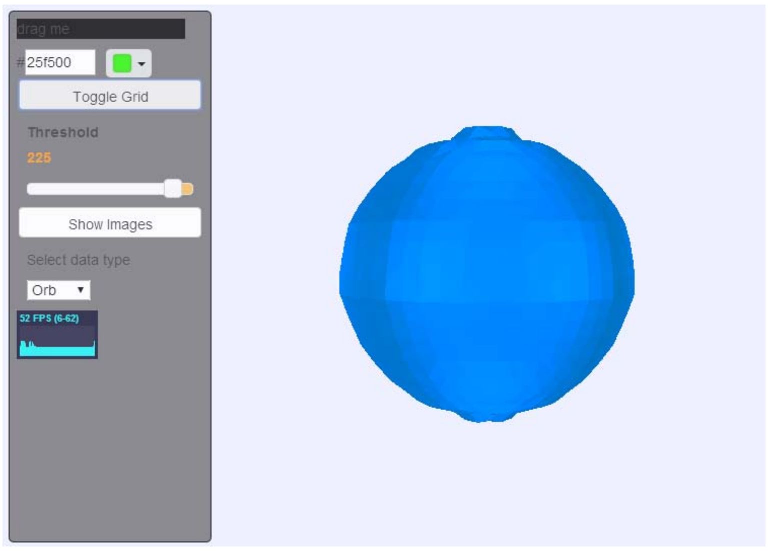


Figure 36 - Resultant mesh model

The above image shows what we termed as an “Orb” which is constructed from a series of images produced in Photoshop using a gradient tool with the intent of mimicking CT sampled data of a sphere. We can now leverage the fact that we are dealing with volume represented by the gradient as we can now adjust the threshold value by moving the slider in the menu bar. By adjusting this, a call will be passed once again to the web worker mentioned earlier with all the world data along with the threshold. Upon completion, the resultant meshes for each individual voxel will be updated which can be seen below. This technique is possible because of the darker nature as we approach the centre of the images. This can be thought of again as mimicking the same type of images produced from an X-Ray/ CT scan. The outer layers could represent soft tissue, while the centre could represent denser material such as bone.

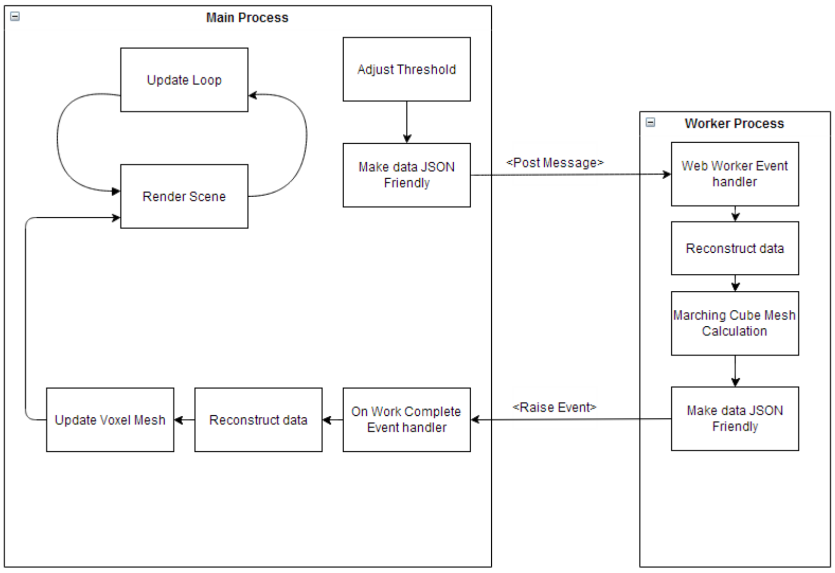


Figure 37 - Sequence for adjusting the threshold

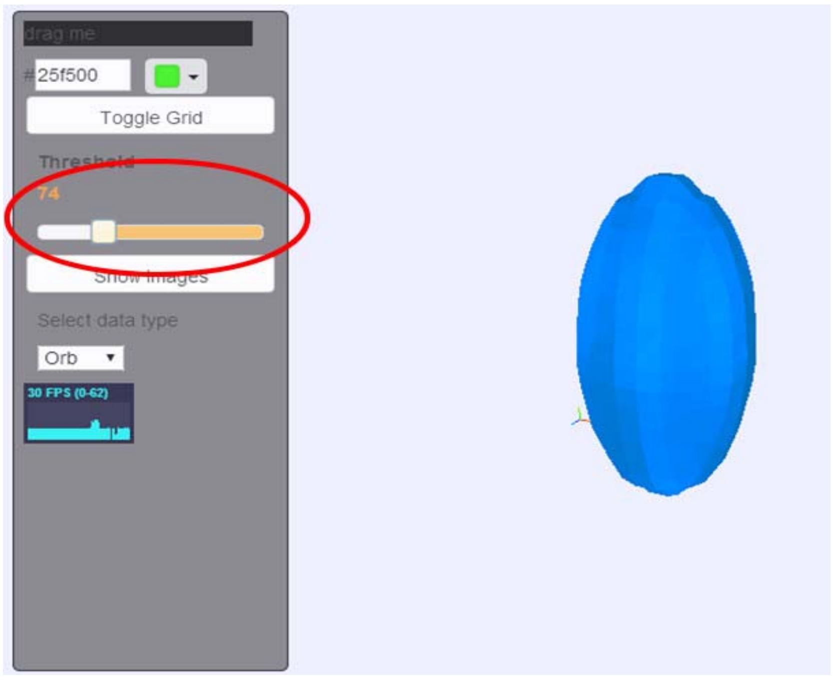


Figure 38 Applying threshold

11.2.1 Other examples

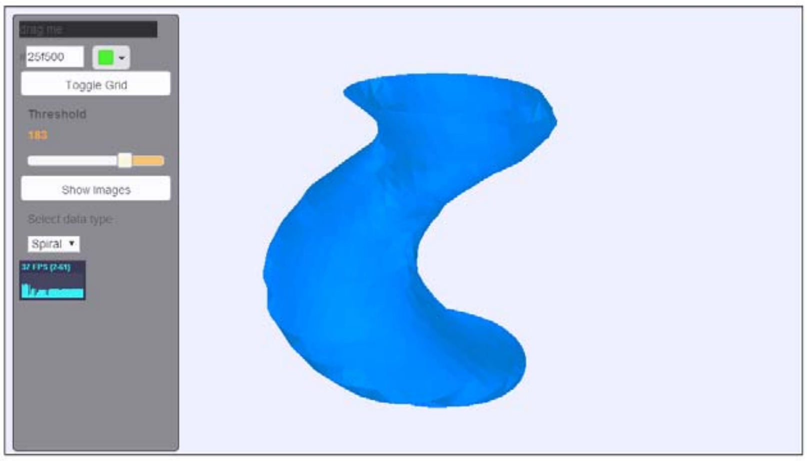


Figure 39 - Spiral Volume

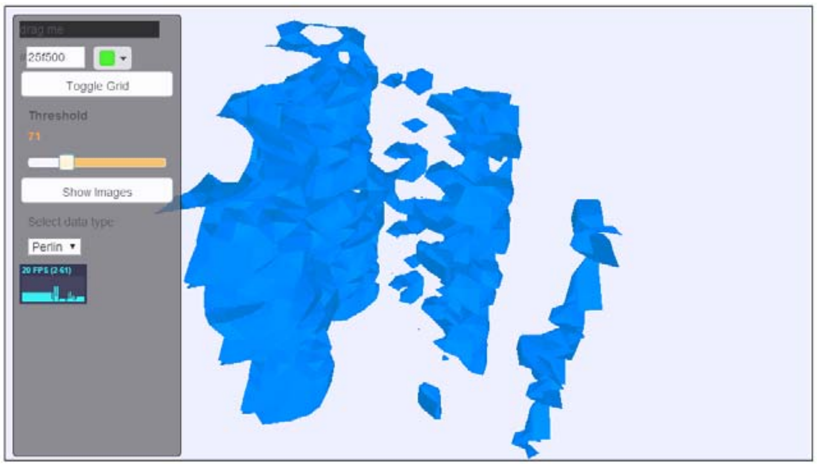


Figure 40 Perlin Noise

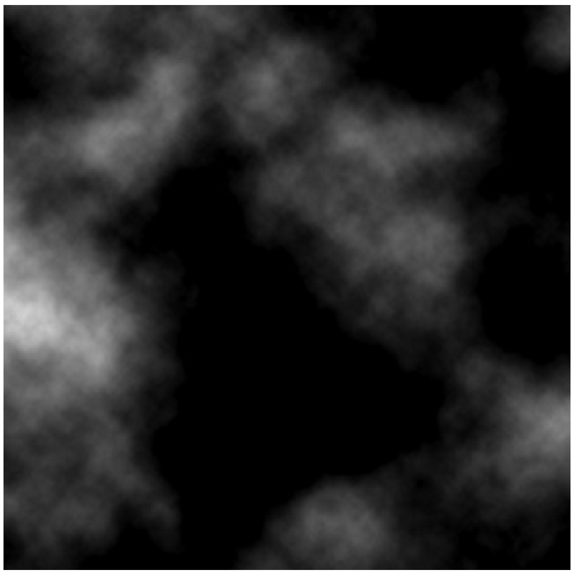


Figure 41 Example of a perlin image used

11.3 Developing the Sculpting application

11.3.1 The problem

Representing a sphere is easy mathematically and worked well in the prototypes where the theoretical sphere to be rendered was defined as simply having a centre and radius. The values to be used in the Marching Cube algorithm were then derived simply by determining how far a voxel corner was from the centre of that sphere. But what happens when we want to apply a manipulation or distortion on that sphere? How can it be now represented mathematically? How can any material properties i.e. elasticity be applied to it? A very difficult prospect, and well beyond the scope of what this thesis is about, mathematically speaking of course.

11.3.2 The solution

Instead of attempting any form of mathematics, the approach to be taken is to instead represent the base object as geometry where we can let the computer handle the maths as appose to us dealing with the complexity. For our proposed solution, it will be made up of nodes/vertices with connecting edges. To make the object behave more like a real world material and have elastic properties, each connecting edge will actually be a spring model making use of Hook's law. Each node will have the ability to be grabbed and moved by way of mouse selection and drag and release actions. One important point must be made here is that the intention is **none of this model** will be visible to the user in the end product, for this proof of concept it will remain visible to illustrate our objective. What should happen is the geometry that the user sees, will that which is generated by the Marching Cube algorithm. So sum up, its main serving purpose is to represent a distortable object that would otherwise be impossible (within the scope of this thesis anyway) to represent mathematically.

To make this approach work we need to return to the grass roots of what volume rendering is all about. The geometry of the controller sphere can be thought as an object, or a patient for that matter, being scanned in a medical scanner with the resultant sampled image slice data being rendered as a 3D volumetric object. The exception being of course the object we are going to scan will have its rendered output in the same place.

The sampling and rendering process is now where this will be won or lost, the task in hand is now to somehow produce a function to generate the correct values that can be assigned to the voxel corners and be used to render the model. What would be ideal is, if we could simulate the same results obtained from the stack rendering solution.

11.3.2.1 Experiment 1:

This was one the first attempts at producing the base object by making use of the prototype example to generate a mesh by using the centre and radius approach.

Because representing a distorted sphere mathematically is far too complex. The aim is to wrap the sphere in a (non-visible) mesh linked by **springs** (internally and externally) and then manipulate the sphere by pulling nodes and having the springs represent elasticity. After this, the plan is to use the Marching Cube algorithm to re render the sphere. This rendering will be achieved by now testing on which side the voxel corners lie in relation to the planes of the surface of the (non-visible) controller object.

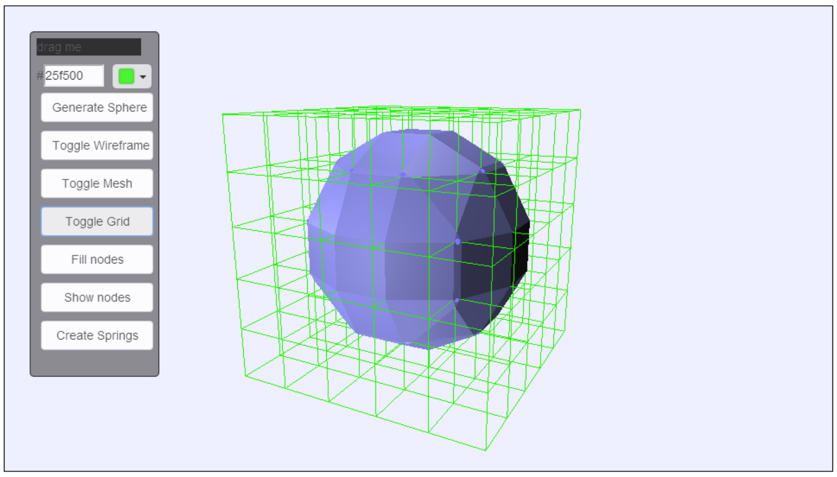
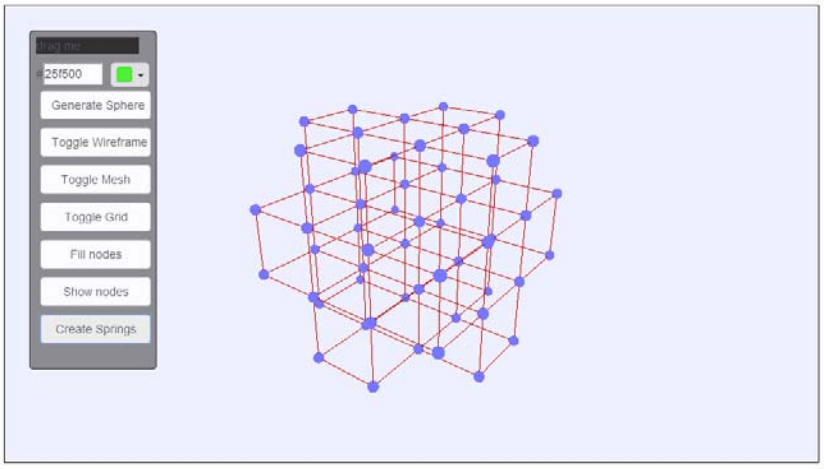
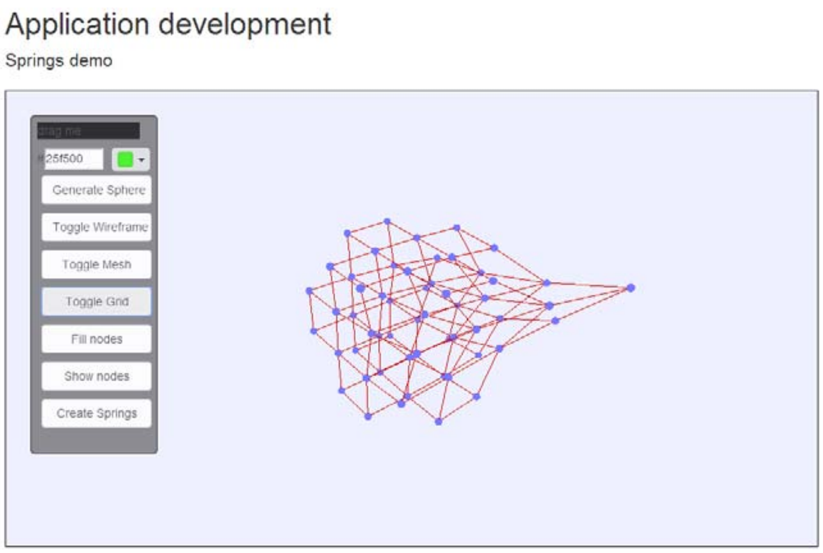


Figure 42 - Marching cube rendered sphere

Result




Application development

Springs demo

We actually abandoned this approach because of the complexity of the base sphere generated by the marching cube algorithm. The above was easy part to generate but it is only using the voxel corners depending on whether they were marked inside which we determined during the original Marching Cube pass. These nodes were then connected together with a spring joint. The image below shows the surface that the Marching Cube algorithm renders. The next step would be to include the surface that was generated and connect it to the nodes we produced above.

However, as it is a per voxel operation, there is a lot of resultant duplicate vertices. This makes it more difficult than it needed to be to use the vertices to generate nodes on the surface and connect them. Instead, of wasting too much time on this and not getting any return for investment, we dropped this idea in favour of the next approach which is to just procedurally generate our own sphere thus giving more control and flexibility.

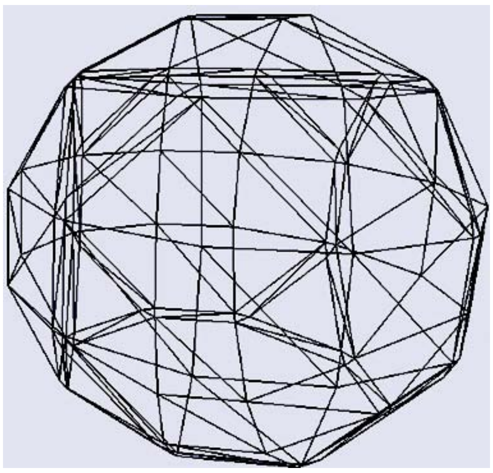


Figure 43 The complex wire frame mesh produced by the Marching Cube pass

11.3.2.2 Experiment 2:

Having abandoned the first approach because of its inherent complexity, the decision next was to procedurally generate our own sphere as we gain the benefit of having more control over it, as we determine how it is produced. The same idea of using springs will be applied here again to connect the outer nodes. However, this time we will not connect the internal volume with spring joints. A mesh for the purposes of producing volumetric data will then be applied to the model. The final step in the plan is to then use the Marching Cube algorithm to render the output of the data obtained from scanning this controller base model. This scanning function will be discussed shortly as it involved its own trial and error approach.

While it may seem counterproductive to what the thesis is all about, to have a sphere with a mesh in place that can be spring jointed and manipulated. The idea behind the controller sphere is that it will be of sufficiently low resolution to allow selecting and manipulating of the nodes. The Marching Cube algorithm, which will run in the background, will then be expected to render the object to a sufficiently high level of detail; depending of course on the resolution we decide set on the voxel world i.e. voxel size/voxel numbers.

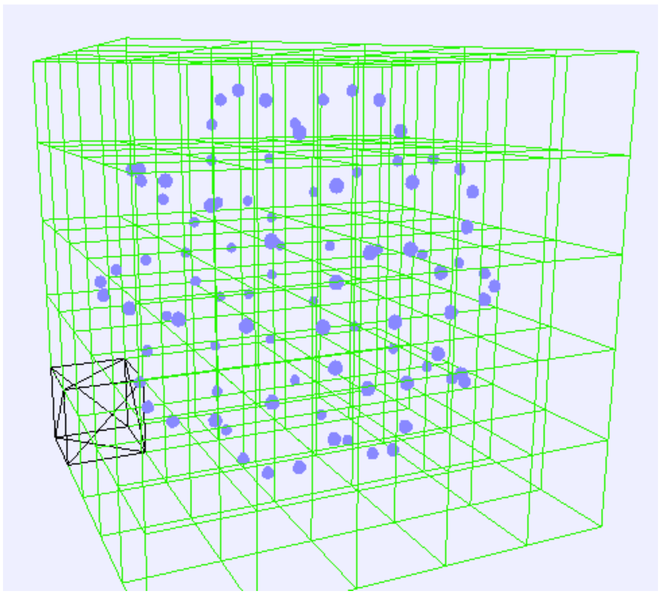
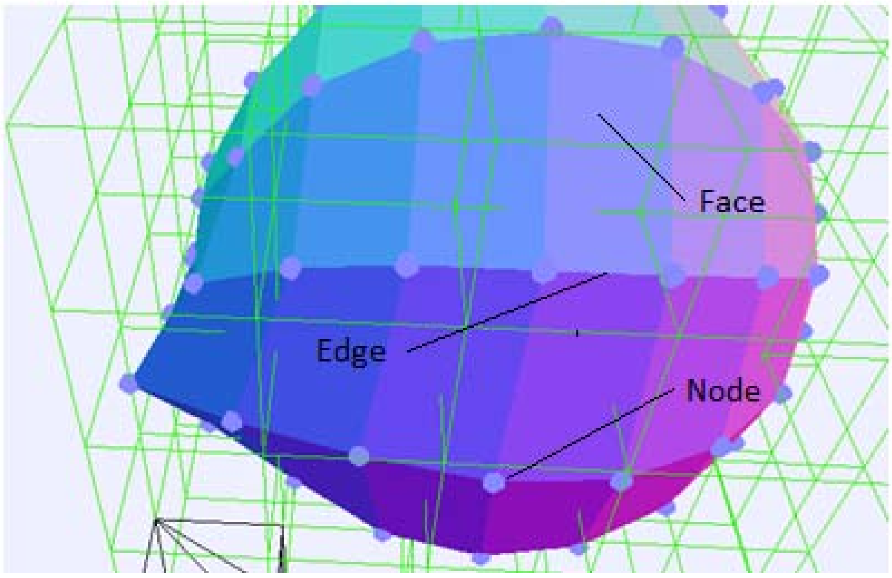


Figure 44 Calculated nodes

Producing the base sphere

```
var x = (Math.sin(Math.PI * m / this._m) * Math.cos(2 * Math.PI * n / this._n)) * this._radius;
var y = (Math.sin(Math.PI * m / this._m) * Math.sin(2 * Math.PI * n / this._n)) * this._radius;
var z = (Math.cos(Math.PI * m / this._m)) * this._radius;
```

Using the above adopted formula from a stack overflow response (User: Jonathan @ stackoverflow.com, 2010), we were able to generate the nodes shown in the image above. We then implement a 'for loop' that generates a specified number of nodes for every longitudinal line. This does result in duplicates at the poles which do need to be eliminated.



From there we proceed to implement an algorithm that connects each of theses nodes as well as calculate faces and our own custom normals as each face will make use of a 'double' face material for ray casting purposes. This is a computionally intense task which we need to introduce concurrency by calling upon the web workers to alleviate. Once the web worker returns the calculated geometry, we render it on the main process (web workers cannot access the DOM). We also at this stage, add the faces calcuated to an octree for efficient look up when it comes to performing the sampling process.

11.3.3 Creating spring joints using Hook's Law

In order to replicate or at least mimic an elastic material, we connected each of the nodes of our controller object with a spring joint connection. To implement this, we followed a physics tutorial presented by Peter Collingridge who demonstrated the use of springs in a Python simulation using Hooke's law.

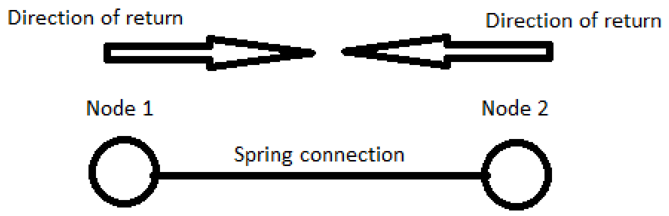
To summarise Hooke's law, it states that:

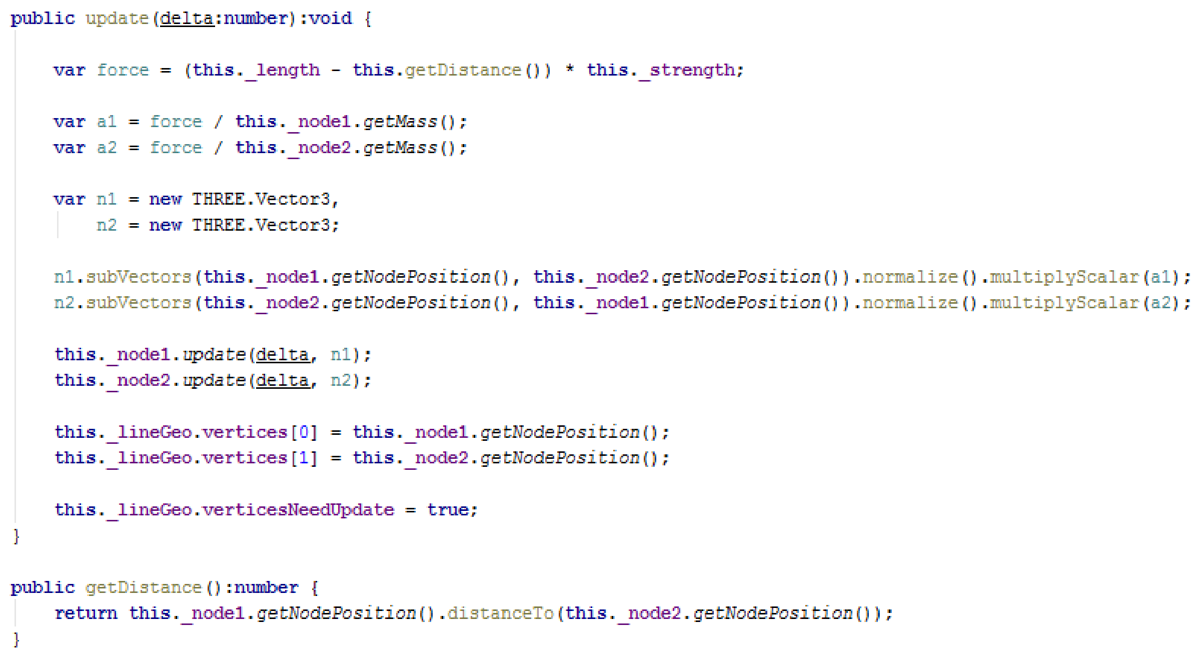
Force = extension in length or displacement * Spring constant

(Collingridge, 2011)

In our example we follow Peter's lead but make several adaptions to bring it in tune with the 3D environment in which we are working. Below is the code used in our update method which gets run every loop for every spring connection we have created. The following is the sequence of what the method is actually performing.

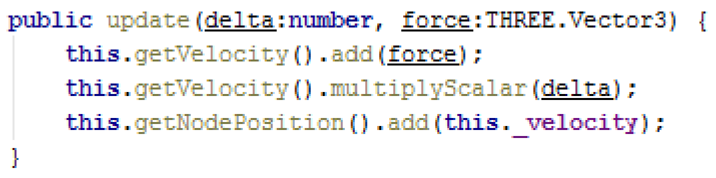
- We calculate a force using the above mentioned formula (strength is used as the spring constant).
- Next we calculate the Acceleration by manipulating the formula Force = Mass * Acceleration
- We then construct new vectors consisting of the normalised direction of return (for node 1 – the direction from node 1 to node2 and vice versa for node 2) and multiply by the force we calculated which will then be added to the velocity of our node.





```
public update(delta:number):void {

    var force = (this._length - this.getDistance()) * this._strength;

    var a1 = force / this._node1.getMass();
    var a2 = force / this._node2.getMass();

    var n1 = new THREE.Vector3,
        n2 = new THREE.Vector3;

    n1.subVectors(this._node1.getNodePosition(), this._node2.getNodePosition()).normalize().multiplyScalar(a1);
    n2.subVectors(this._node2.getNodePosition(), this._node1.getNodePosition()).normalize().multiplyScalar(a2);

    this._node1.update(delta, n1);
    this._node2.update(delta, n2);

    this._lineGeo.vertices[0] = this._node1.getNodePosition();
    this._lineGeo.vertices[1] = this._node2.getNodePosition();

    this._lineGeo.verticesNeedUpdate = true;
}

public getDistance():number {
    return this._node1.getNodePosition().distanceTo(this._node2.getNodePosition());
}
```

Figure 45 - Spring update method



```
public update(delta:number, force:THREE.Vector3) {
    this.getVelocity().add(force);
    this.getVelocity().multiplyScalar(delta);
    this.getNodePosition().add(this._velocity);
}
```

Figure 46 - Individual node update

11.4 Application of the Octree

The original intentions when developing this concept application was to use the Octree data structure for managing all the voxels in our voxlelized world. However, this didn't materialise as such as we found it was easier to reference the world by breaking it up into levels with each level consisting of an array of voxels.

However, that did not mean that the use of this rather efficient data structure (as presented in our earlier research) didn't have a place in this project, in fact, quite the opposite.

During development it was found that one of the community members implemented an Octree that integrates into THREE.JS. This structure ties in very well to the ray casting system that is used in THREE.JS. What this meant for the project is that it reduces the number of searches required when casting a search ray into a scene. Instead of testing the entire scene of objects, first, only objects that lie in general direction within a specified radius are found. Once this list is compiled the ray test is carried out against this list. This has worked extremely well in this project and is used extensively in the sculpt concept application for finding collisions with faces of the control object when performing a volume sample.

11.5 Volume sampling and rendering

The key to the success of this solution will be how we sample or scan the controller object so that we can voxelise it with the Marching Cube algorithm. This section describes the path, through trial and error to finding the best solution to this problem. One item that is central to all the attempts is the use of ray casting. The image on the next page is a visual example of ray casting which is basically an intersection test. Different frameworks have different ways of implementing ray casting, but, for THREE.js we can determine which object was intersected through scene references. In our controller sphere, we are using double faces so that the ray can hit either side of the face and determine an intersection. This however presents an issue because the face has two normal vectors now which conflict in indicating what is the intended direction it should be facing. This is why earlier we stated that we created our own normal when procedurally generating the sphere, as we can now apply a dot product function against the ray line to determine the 'intended' facing direction.

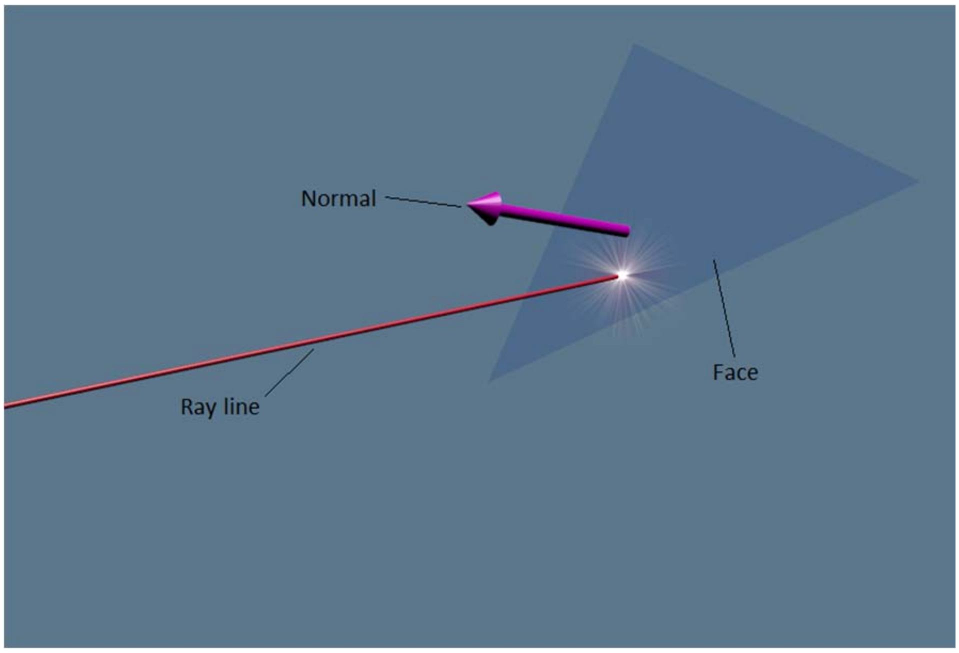
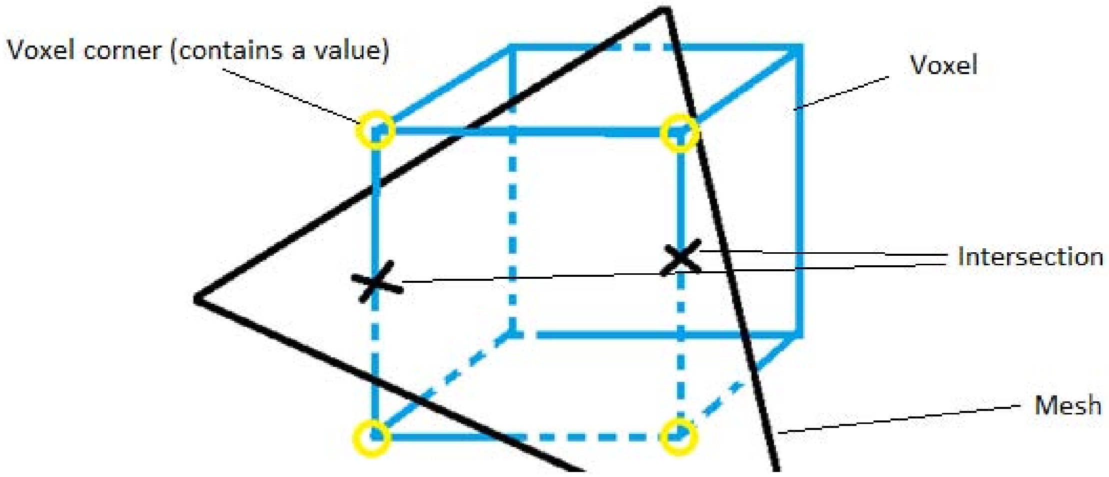


Figure 47 Ray cast test visualisation

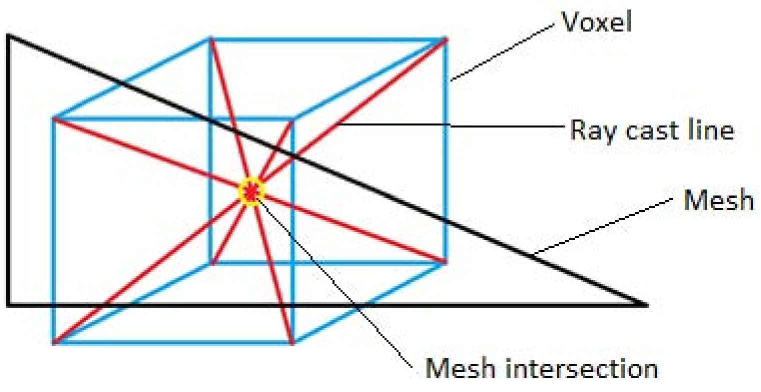
Obtaining voxel corner values



So the problem next is we need values to assign to each voxel corner that reflects the control object that was generated earlier. The idea is to somehow choose an appropriate algorithm that will sample the amount of the controller which is contained within the voxel. For this to work an number or criteria must be satisfied, first an appropriate value must be assigned to each corner and secondly an appropriate threshold must be selected that will decide where along an edge between two voxel corners an vertices will lie if there is an intersection with a mesh belonging to the controller

11.5.1 Approach 1

Process



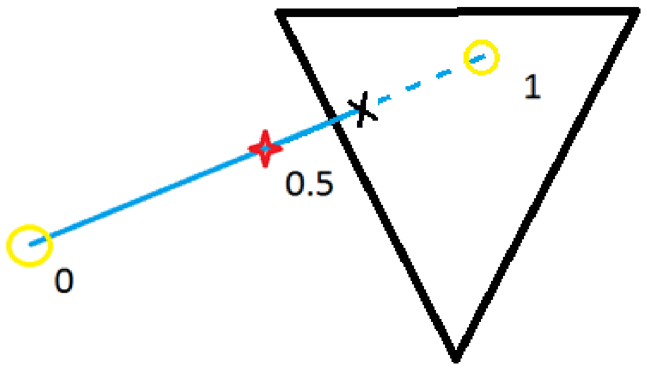
This approach was to ray cast at diagonals which is easy enough to implement. If an intersection occurs, we assign distance from that corner to the intersection point as the voxel corner value.

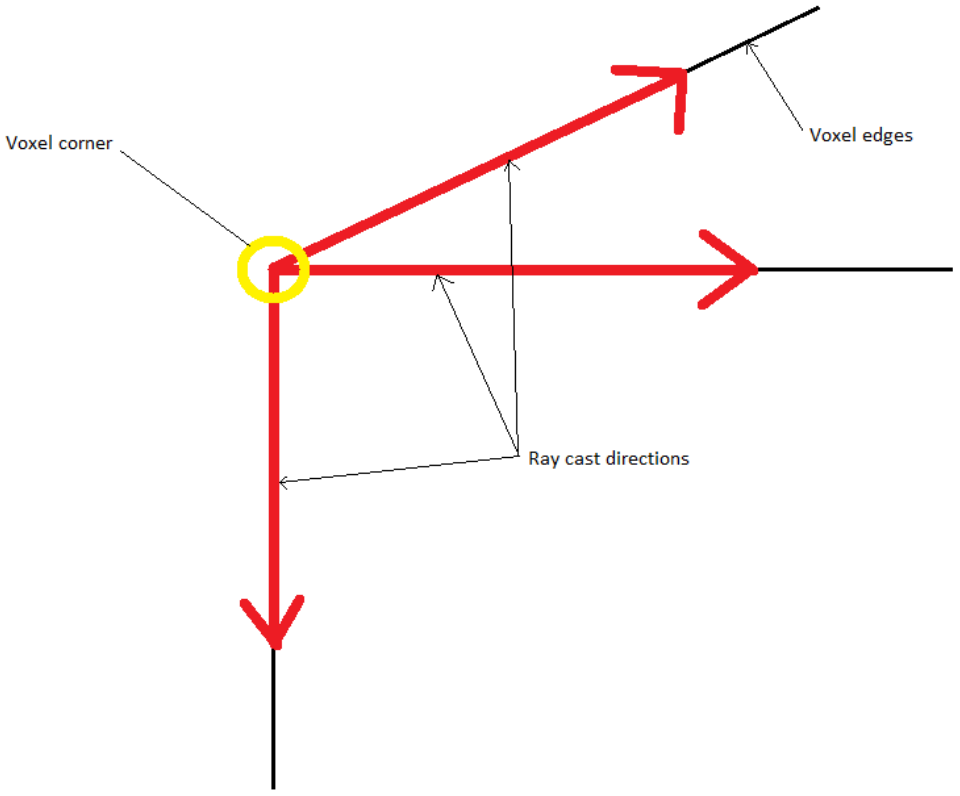
Result

As this was the first approach, it was almost expected from the outset that this wouldn't present the ultimate desired outcome, but as we need to start somewhere it formed a foundation of how we should progress next in our attempt to voxelise our control sphere. So to conclude, the unreliable results produced from this experimentation resulted in the abandonment early as mentioned, but did provide an opportunity to experiment with ray casting for sampling so it wasn't a total loss.

11.5.2 Approach 2

Process





For this attempt we will adopt a simple binary approach. For each corner we will use a ray casting technique that will cast rays in three directions and determine if there is an encounter with the mesh. If a corner is determined to be inside we will assign a value of 0 otherwise we assign the corner a value of 1 to signify outside. For the purposes of the Marching Cube algorithm we will pass the value of 0.5 as the threshold.

Results

A positive start but nowhere near to the desired level of detail we require. This was to be expected if the vertex positions were always going to be places mid edge on the voxel.

As expected to gleam any sort of decent result, the controller sphere needs to be of a high resolution if this sampling algorithm is to select appropriate values for the voxel corners. Even at this, it becomes apparent pretty fast that this will not lead to the final desired outcome. What follows are series of screenshots that show the outcome of this test approach.

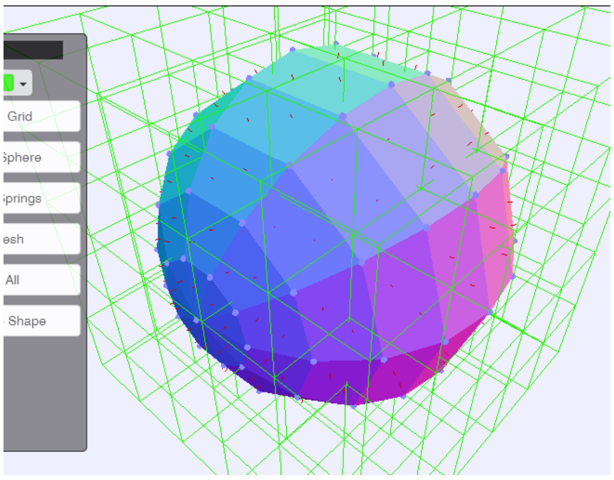


Figure 48 Low resolution controller

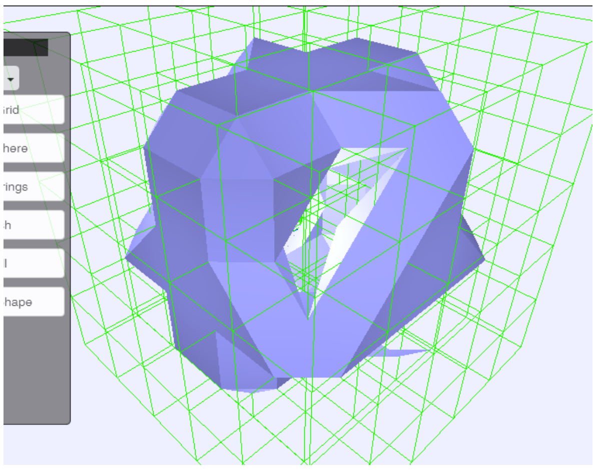


Figure 49 – Marching Cube output

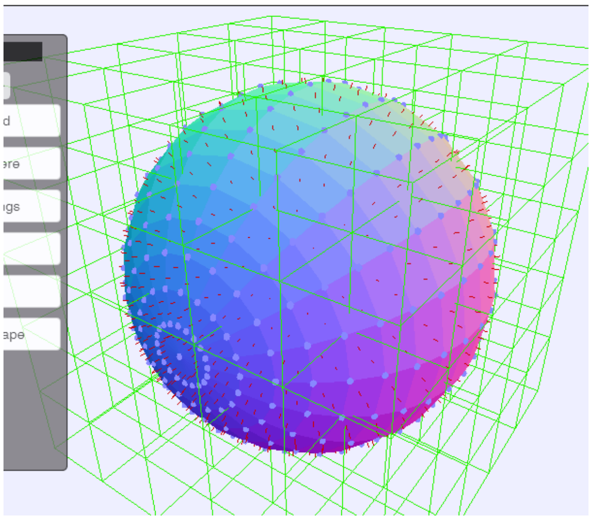


Figure 50 High resolution controller

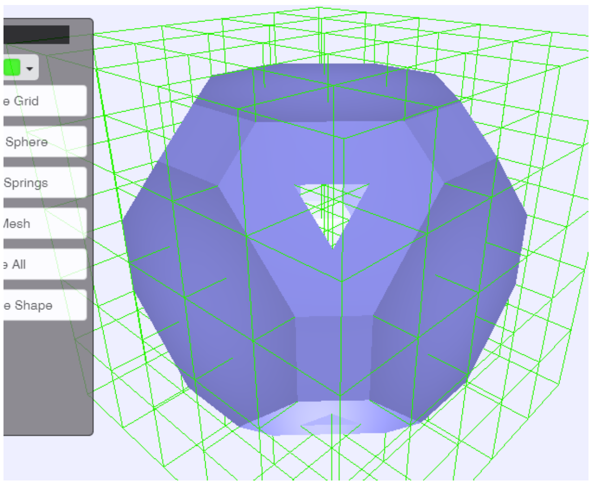


Figure 51 Rendered result

11.5.3 Approach 3

Process

This attempt aims to build on the work of the previous experiment but aims to extract better corner values than just using binary 0 or 1. So, same as the previous we pick a voxel corner and test it in its three possible directions for intersections with the controller mesh - There will be 4 possible scenarios to be dealt with here:

0 Hits	set voxel corner value to some safe number that indicates it is well outside
1 Hits	set voxel corner value equal distance to intersection point
2 Hits	Form a line and do a distance to perpendicular of the line calculation and set voxel corner value equal to this value.
3 Hits	form a triangle of the three points and perform a perpendicular to plane calculation and set voxel corner value equal to this value

Results

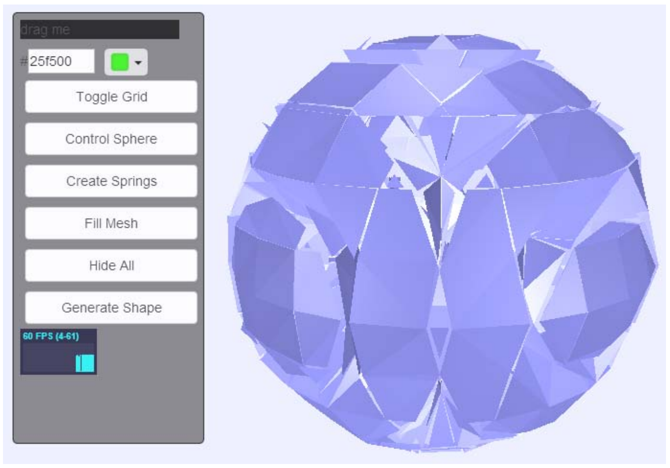


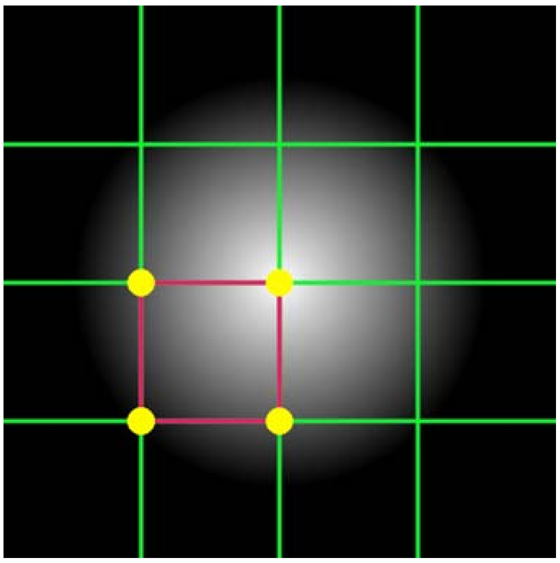
Figure 52 Result of the March Cube pass with the values acquired from the above approach

As can be seen from above, this method doesn't produce a reliable method of volume sampling. The result above indicates that by using this approach results in gaps between the meshes meaning we are losing out on data. The conclusion that can be drawn from this is perhaps sampling from individual voxels is not the best approach, but rather an attempt should be made to adapt a global sampling method.

11.5.4 Approach 4

Process

All of the previous attempts were trying to use per voxel sampling, however if we consider the prototype and the stack renderer, it worked well because all values were global based. This now seems to be a better approach than the previous per voxel sampling method. Especially looking at the roots of the use of the algorithm, taking an x-ray and sampling the colors at the corners of the voxel, those colors are connected outside of the voxel essentially meaning that values are global. So we should adopt and approach that attempts to build a global image and render them similar to the stack render solution.



To build our images, we are essentially going to have to x-ray our controller object similar to how a patient gets scanned during a medical assessment. We will again make use of ray casting and our specially built control object that has double faces, using our custom defined normal to tell the true direction of the face.

Below is an illustration of a ray cast against our control object. The entry and exit is marked below with the particle effect, it is this information we wish to collect and process.

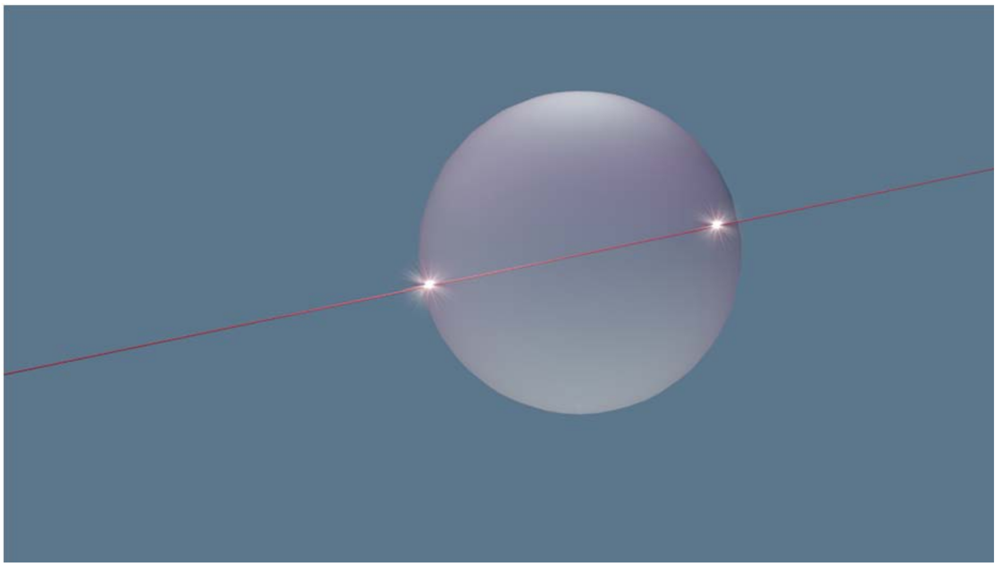
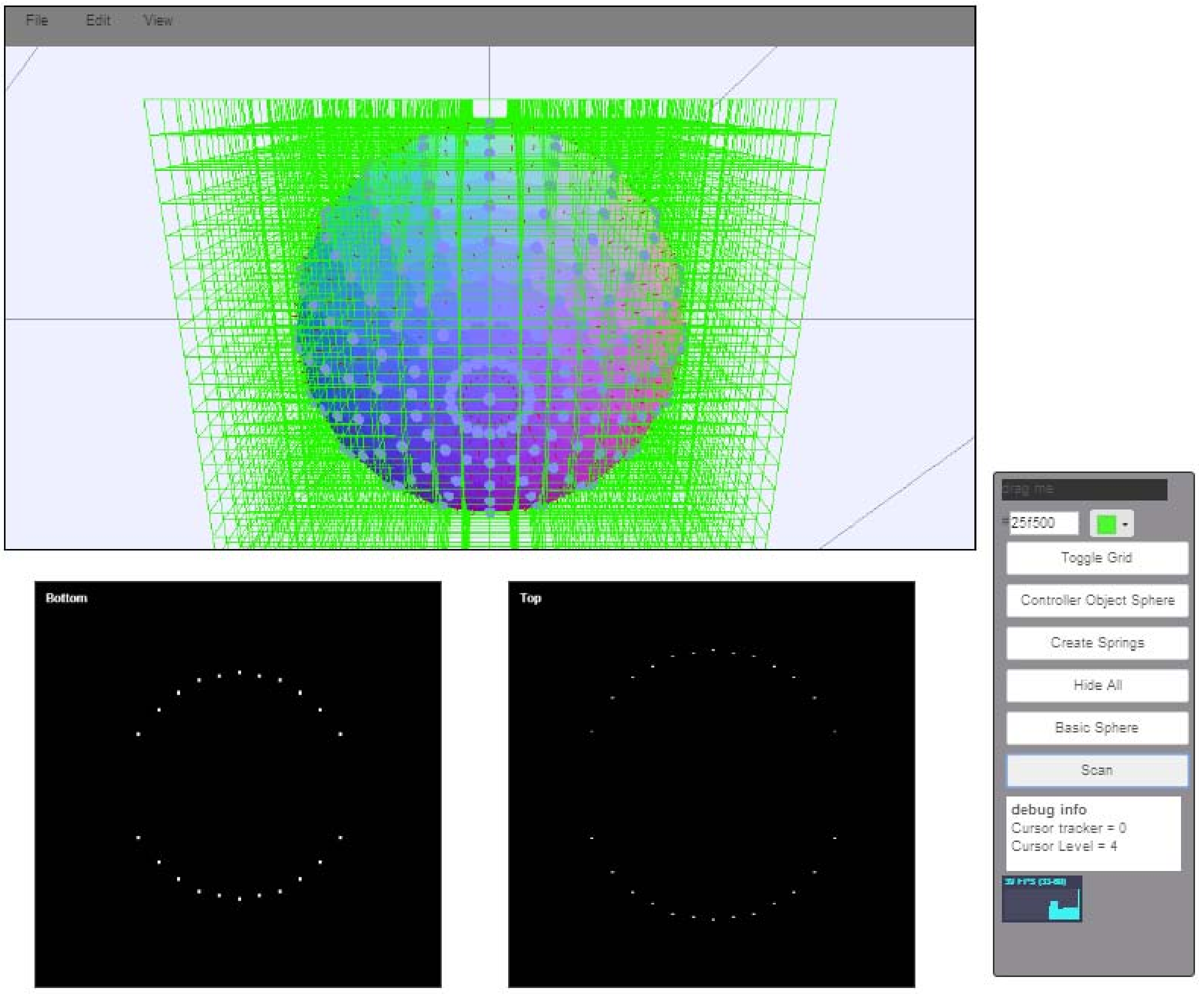


Figure 53 - Visualisation of the entry and exit marks of a ray cast against the control object



```
debug info
Cursor tracker = 0
Cursor Level = 4
```

Figure 54 the results of the sampling

The above shows the result of this approach. The sampling is only being taken from one direction. What is notable is the loss of detail when we approach the extremes of our control object. Also, all we have captured here are points which by themselves are not much use i.e. we are not determining volume.

The next attempt at this we will take more samples from different directions. These will be taken with 2 on the horizontal (front and side) and 1 from the vertical. Along with this, we will trace the route as it passes through a volume and store this.

The series of images below illustrate what should happen. This time we are including a secondary control object that has its normal facing inwards. The outcome we expect from this is a volumetric sphere with a hollow core.

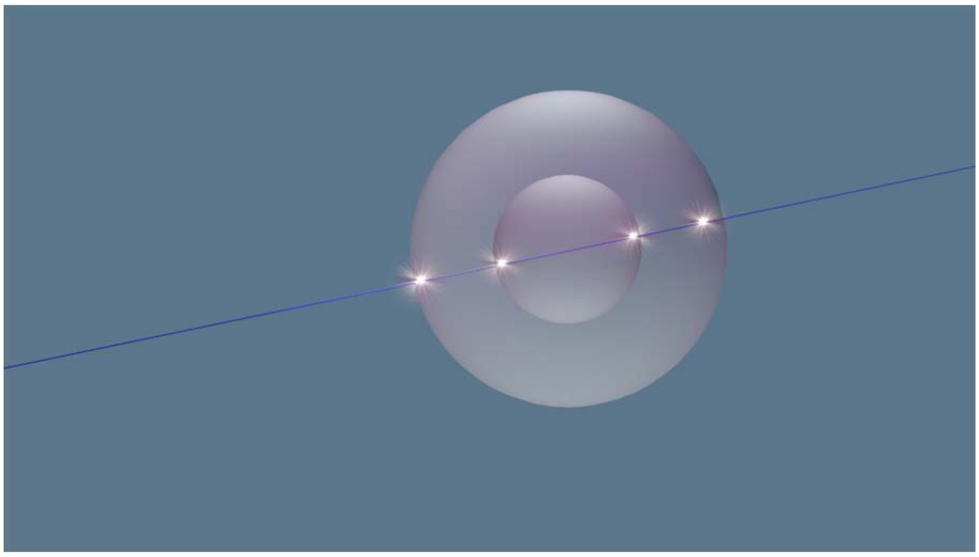


Figure 55 Visual aid of ray casting through a volume that has a hollow core

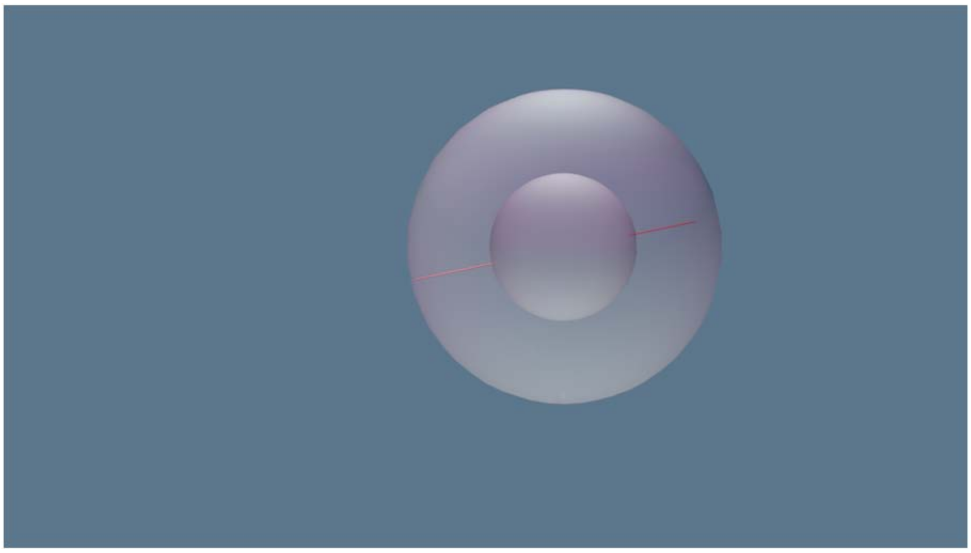


Figure 56 Resultant volume trace which is stored

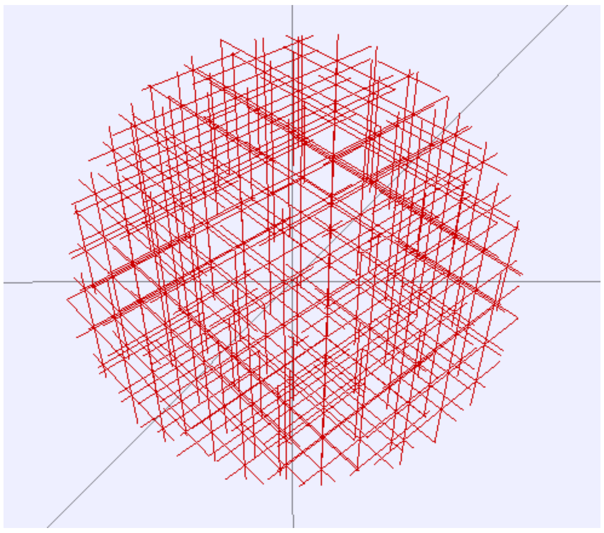


Figure 57 Application shows the lines denoting volume

The above taken from our application now shows all the captured trace lines as it passed through the volume.

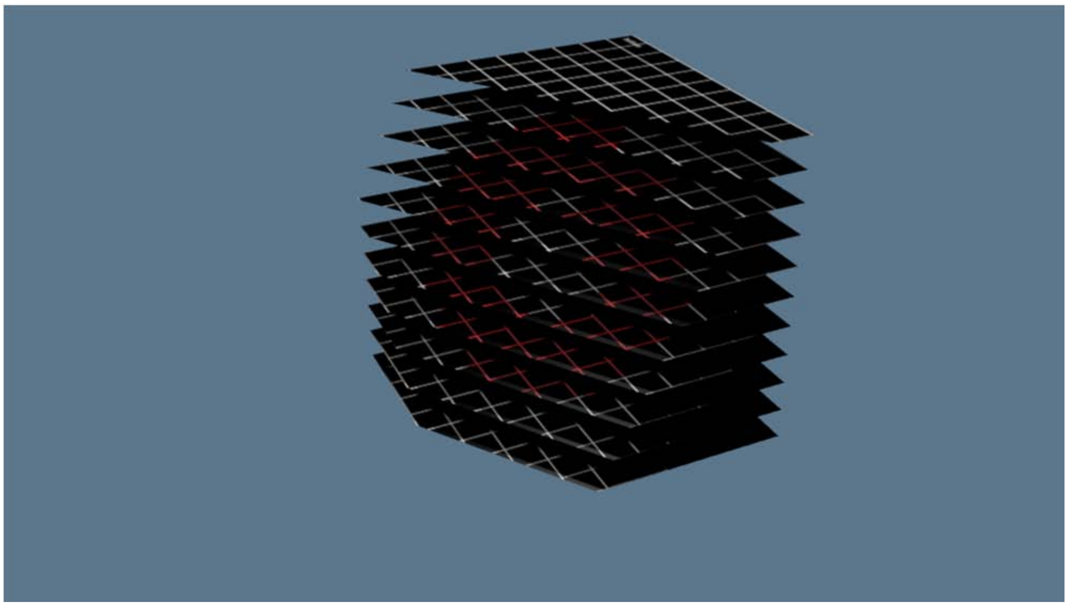
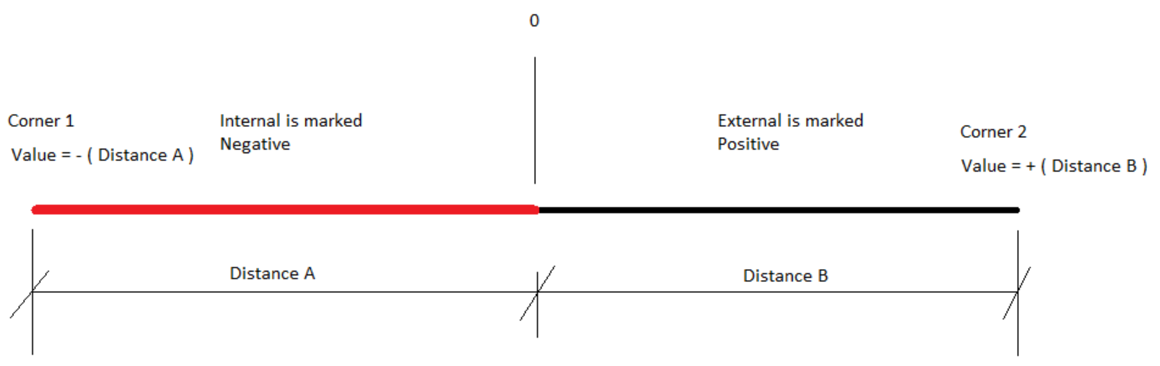


Figure 58 Visualisation of captured data as an image stack



Once we have completed the volume sampling step. We can now use this information to render using the Marching Cube algorithm. The approach taken is to first build the case for a voxel by determining which corners are inside and which corner are outside. This is done by examining if a corner lines on one of the collected trace lines i.e. the red lines.

Once we have our Marching Cube case, we must perform the interpolation to determine where to place a vertex. The diagram above shows how we establish a value for each corner. We examine our collection of captured volume traced lines (red) and find the line that aligns with the edge which connects the two voxel corners under consideration (corner 1 & corner 2 shown above). The value that is assigned to the corner is the shortest distance it takes to get to the edge of that matched volume trace line. If that corner is within the volume (lies on the red portion) we mark it negative to signify it is inside. If it is external (lies on the black) we mark it positive which signifies it is outside. To ensure correctness, the absolute values of the two corners should add up to equal the voxel width.

These are the values now passed to the Marching Cube algorithm, with a threshold of 0 to denote the surface or placement of a vertex. The rendering in progress can be seen in the next image.

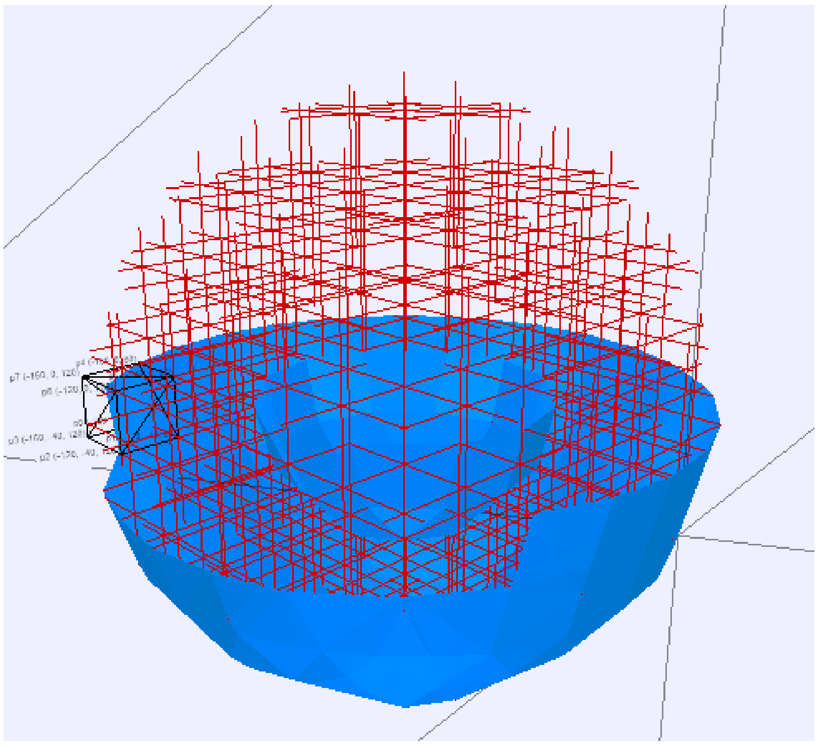


Figure 59 The Marching Cube rendering process of the standard untouched controller

11.5.4.1 Extrusion demonstration

The following now is a demonstration of the extrusion functionality taking place. The captions under the images explain what is happening here.

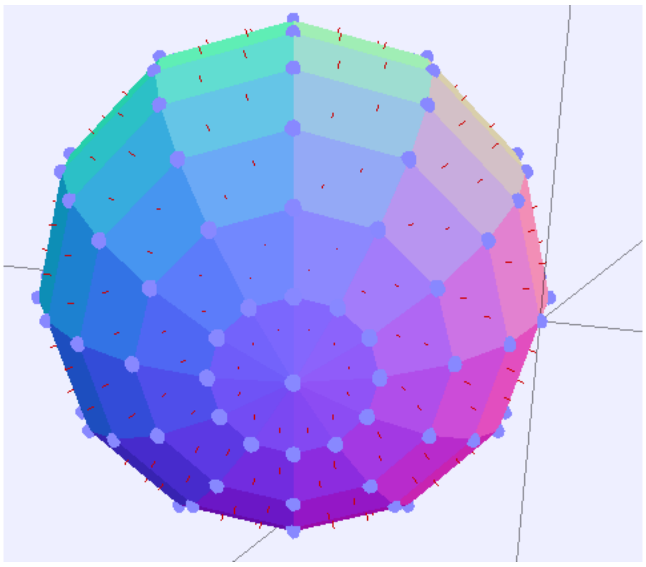


Figure 60 - We begin with the base controller

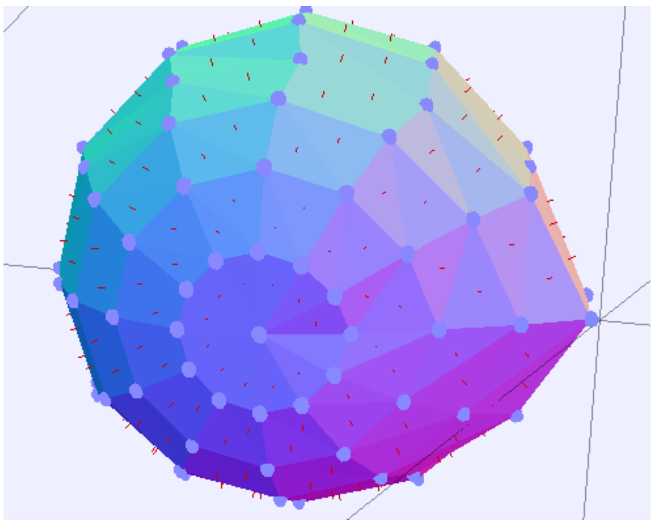


Figure 61 - We apply spring joints and grab and extrude a node outwards

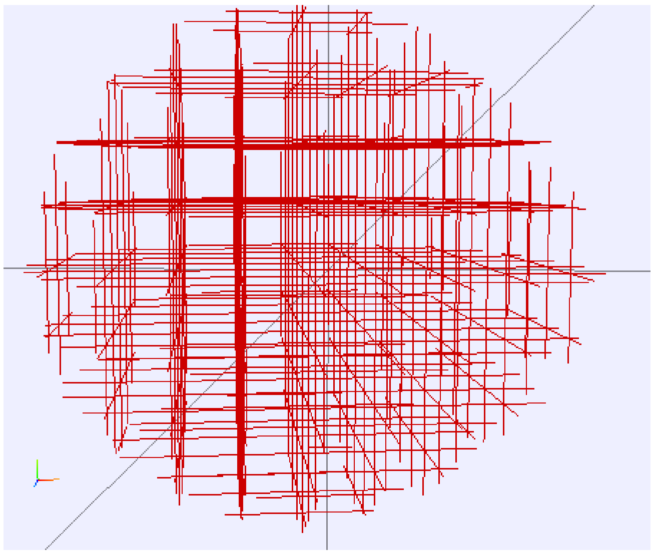


Figure 62 - We apply the volume sampler which results in the lines traced here

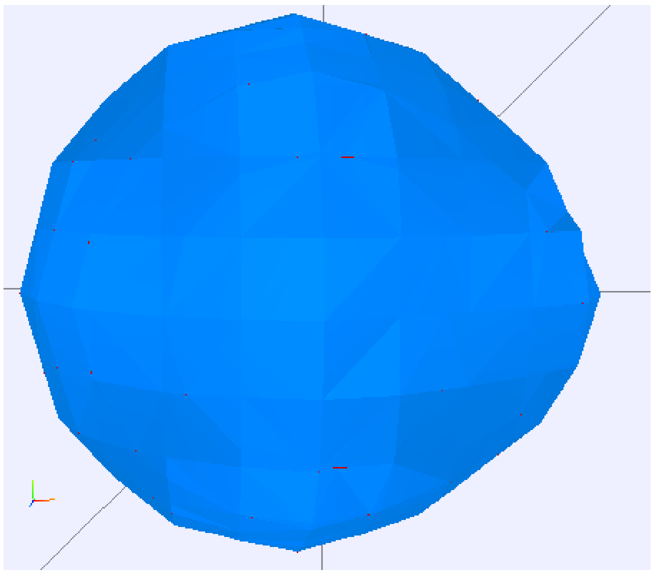
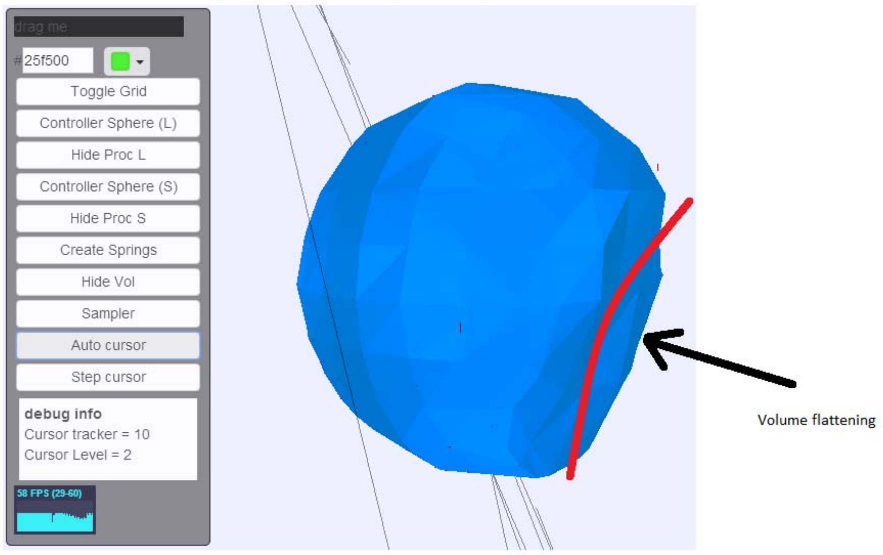


Figure 63 - We apply the Marching cube algorithm to the acquired volume data to render the above output

11.5.4.2 Compression demonstration

This follows the same procedure as above except we press the nodes inwards before applying the sampler and rendering with the Marching Cube algorithm.



debug info
Cursor tracker = 10
Cursor Level = 2

11.6 Results

11.6.1 What worked well?

The volume rendering application worked extremely well, we now have an application that was written once and is capable of being distributed to multiple devices as can be seen from the below image. Although the 2 year old android smart phone does require more time than the desktop/laptop to process the data as well as render the volumetric model. It must also be noted that no effort was made to optimise for mobile devices. So, using touch control doesn't behave always as it should.

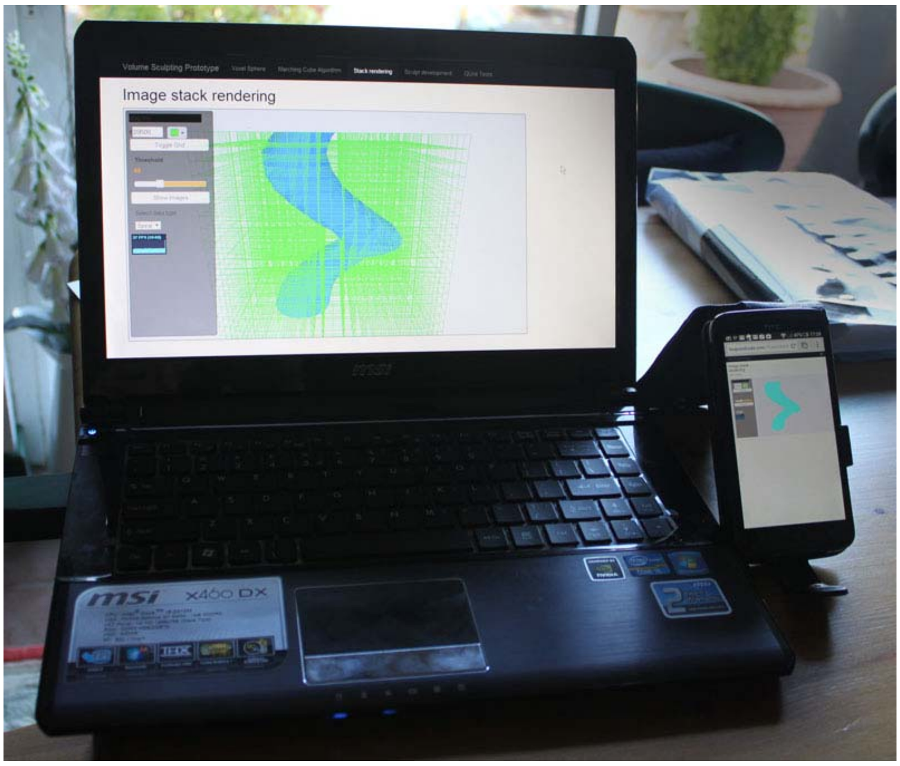


Figure 64 – Same application on laptop and Android smart phone

From a sculpting perspective, we had mixed results but regardless, still established an excellent foundation to progress on. Mesh flattening seems work very well as was demonstrated above. This was mainly due to the fact that we are pushing into the volume and not resulting in any ray sampling lines being caught between voxels which forms the next part of our discussion.

11.6.2 What didn't work so well?

As mentioned, mesh extrusions posed a bit of trouble, which is demonstrated in the below series of images. The issue arises primarily if a vertex is pulled in such a fashion that no collisions occur with the voxel edges which are the primary source for capturing volumetric data.

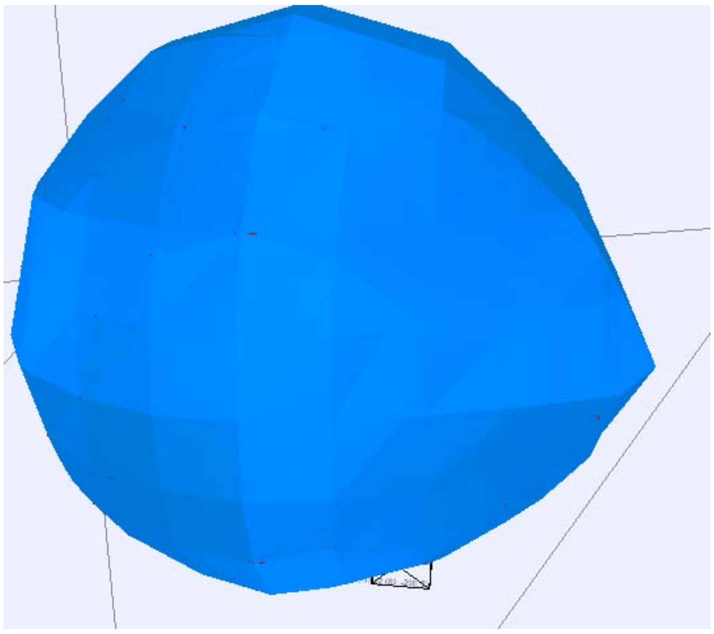


Figure 65 - What looks like a good extrusion?

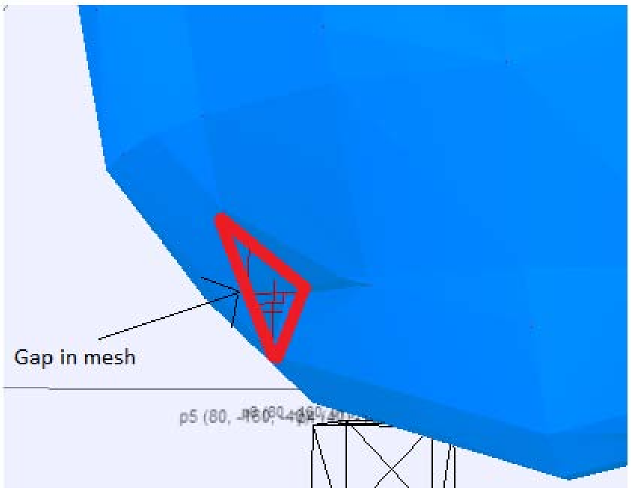


Figure 66 – Sometimes result in gaps in the mesh

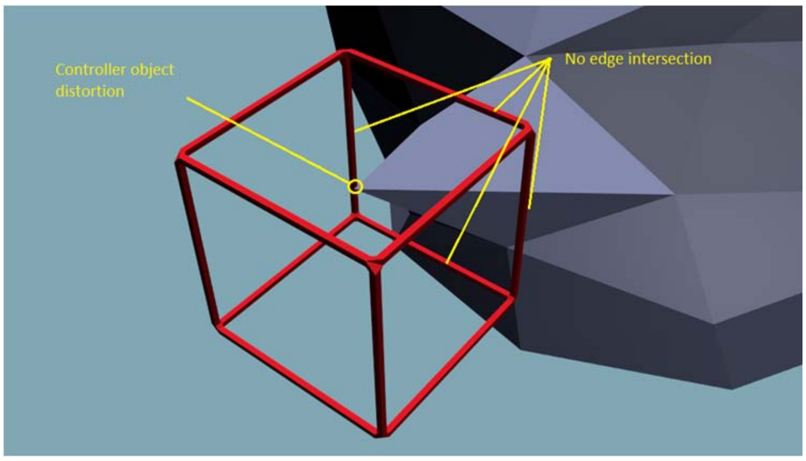


Figure 67 - Visualisation of the source of cause

11.6.3 What could be tried next?

For the stack rendering application, it would be nice to develop it further to maybe handle more detailed images such as CT scan images. This would obviously require increasing the resolution and require further optimisations as well as leveraging more out of web workers for concurrency.

For the sculpting application, it would be nice also if we could also introduce a larger resolution to help alleviate some of the issues we were having above. Along with that, it would be interesting to also investigate the possibility of introducing ray tracing as part of the rendering process.

11.6.4 Applications of the work

One major application for this type of application lies in the area of 3D printing or additive manufacturing as it also referred to. In an article written for the Wall Street Journal, the author quotes a senior figure of General Electric who states that “Manufacturing is undergoing a change that is every bit as significant as the introduction of interchangeable parts or the production line”. One of this changes includes the use of additive manufacturing where it possible to create a product in a single pass without the need of traditional methods such as using milling machines or casting. This also offers the advantage in not having to deal with the problem of waste or the need for assembly.

The following figures the author quotes are for America but give a good reflection of the way manufacturing with 3D printing is going.

- "27% Annual growth rate of 3-D market over the past three years"
- "$2.2b Worldwide sales of 3-D printers and services 2012"
- "$10.8b Projected sales of 3D printers and services, 2021"

Also according to that same article, even Nike are finding the implications of employing 3D printing by having less waste and reduced delivery costs. The profound consequences of this new technology for Nike was "Almost seemingly out of the blue, the reason for making shoes in low-wage countries begins to evaporate and the advantages of locating the machine closer to the customer—in part for faster delivery—begin to loom much larger".

This type of manufacturing has now come so far that in an English university, they have managed to get a printer to self-replicate. The consequences being that it may be possible to buy a printer and spawn more printers from it, only having to rely on consumables (Koten, 2013).

Thinking of other applications for this type of solution, since we are developing this for the web environment, it opens up the possibility of having an online create and share community similar to that on offer by unity's asset store. It could be possible to develop this further to have backend support for handling storage and payments if the venture was to be ever monetized.

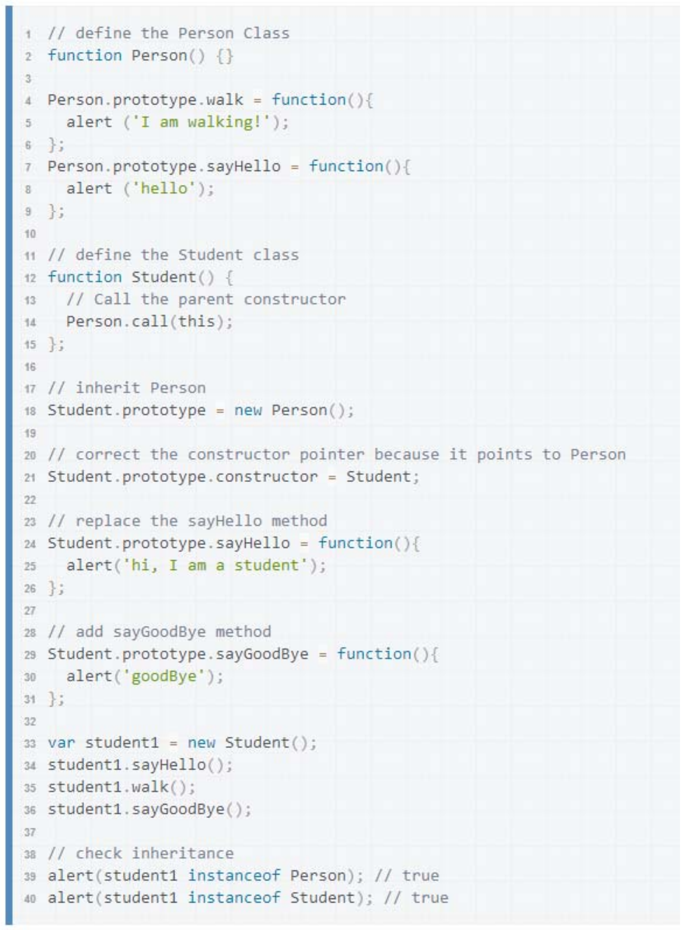
Another possible application for this could be for use in a game engine that uses voxels instead of traditional mesh approach. In traditional game engines a meshes position is update and rendered every game loop. In a voxel engine, it would be a case of switching voxels on or off or manipulating their properties i.e. color or opacity. It all depends on the effect wanted, however the more detail that is required as encountered here, can have an impact on performance.

12.0 Developing for the Web Environment

12.1 Developing with JavaScript

We began the development stage and produced the early prototypes using pure JavaScript. JavaScript is described as being the language of the web. An even better description is it "is a high level, dynamic, un-typed interpreted programming language that is well suited to object orientated and functional programming styles" (Flanagan, 2011).

The language attempts to be Object Oriented through simulating/mocking inheritance by use of the prototype. The following is an example taken from the Mozilla Developer Network page describing object oriented programming in JavaScript. The Person class here implements a walk and a say hello function. The student inherits then from the Person class and overwrites the Person's say hello while implementing their own 'say good bye' function.



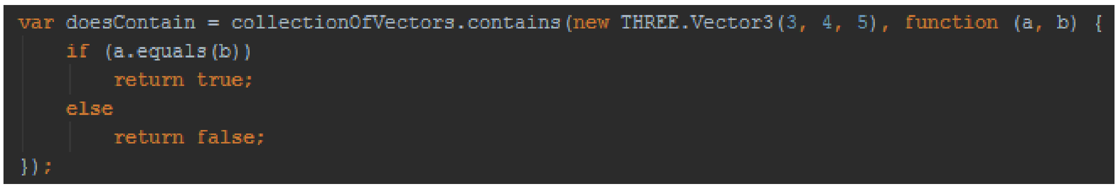
```
// define the Person Class
function Person() {}

Person.prototype.walk = function(){
  alert ('I am walking!');
};
Person.prototype.sayHello = function(){
  alert ('hello');
};

// define the Student class
function Student() {
  // Call the parent constructor
  Person.call(this);
};

// inherit Person
Student.prototype = new Person();

// correct the constructor pointer because it points to Person
Student.prototype.constructor = Student;

// replace the sayHello method
Student.prototype.sayHello = function(){
  alert('hi, I am a student');
};

// add sayGoodBye method
Student.prototype.sayGoodBye = function(){
  alert('goodBye');
};

var student1 = new Student();
student1.sayHello();
student1.walk();
student1.sayGoodBye();

// check inheritance
alert(student1 instanceof Person); // true
alert(student1 instanceof Student); // true
```

(Mozilla Developer Network, Introduction to Object-Oriented JavaScript, 2014)

The student class is then instantiated and the functions tested producing the following outputs:

Function call	**Output**
student1.sayHello()	“hi, I am a student”
student1.walk()	“I am walking!” as implemented by Person
student1.sayGoodBye()	“goodbye”

As mention, JavaScript is also able to behave as functional programming language by being able to take functions as parameters or return them. The following example from our own application shows how we are able to pass a comparator function to a ‘contains’ function which we are using to test if an object is contained in collection.

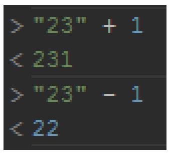


```
var doesContain = collectionOfVectors.contains(new THREE.Vector3(3, 4, 5), function (a, b) {
    if (a.equals(b))
        return true;
    else
        return false;
});
```

Figure 68 - Higher order function in JavaScript

JavaScript has come a long way since the days of making scrolling text and fancy menus and today by leveraging Googles V8 engine, JavaScript has extended itself into being capable of offering itself as a highly scalable server with the advent of Node.js (Hughes-Croucher & Wilson, 2012).

However, despite its maturity, it still does come with its idiosyncrasies.



```
> "23" + 1
< 231
> "23" - 1
< 22
```

Figure 69 - Some of JavaScript’s interesting features

Some of the pitfalls in JavaScript discovered in the early stages were the lack of type safety. This became noticeable with the rapidly growing complexity of the project and prompted a move to try an alternative system that could address this issue. With a tight deadline and some quick investigation the final choice came down to adopting Typescript into the project, other choices would have included either Coffee Script or Google Dart. Coffee Script was not chosen because of its unusual syntax while Dart is not yet widely adopted raising the question of support. Typescript won through its familiar C#/Java like syntax.

12.2 The move to TypeScript

Typescript is an open source initiative developed by Microsoft to aid programming teams that must build and maintain large scale JavaScript projects. Typescript in large part is a syntactical sugar for JavaScript but offers the benefit that always compiles back into JavaScript while also allowing a mixture of JavaScript (Microsoft, 2014). One of the key benefits which can be quickly deciphered form its name is that it offers static typing. This combined with a powerful IDE such as Visual Studio or WebStorm allows the developer to experience real time compilation and alerts to any type defects they may be introducing.

Typescript replicates a lot of the features that are available in Object Orientated languages by introducing variable visibility with public and private, the ability also to have interfaces which allows for creating contracts.

Some other neater features of the language is that it implements generics (<T>) as well as making use of the lambda symbol (=>) which .Net programmers are familiar with for leveraging the power of functional programming which JavaScript implements by default.

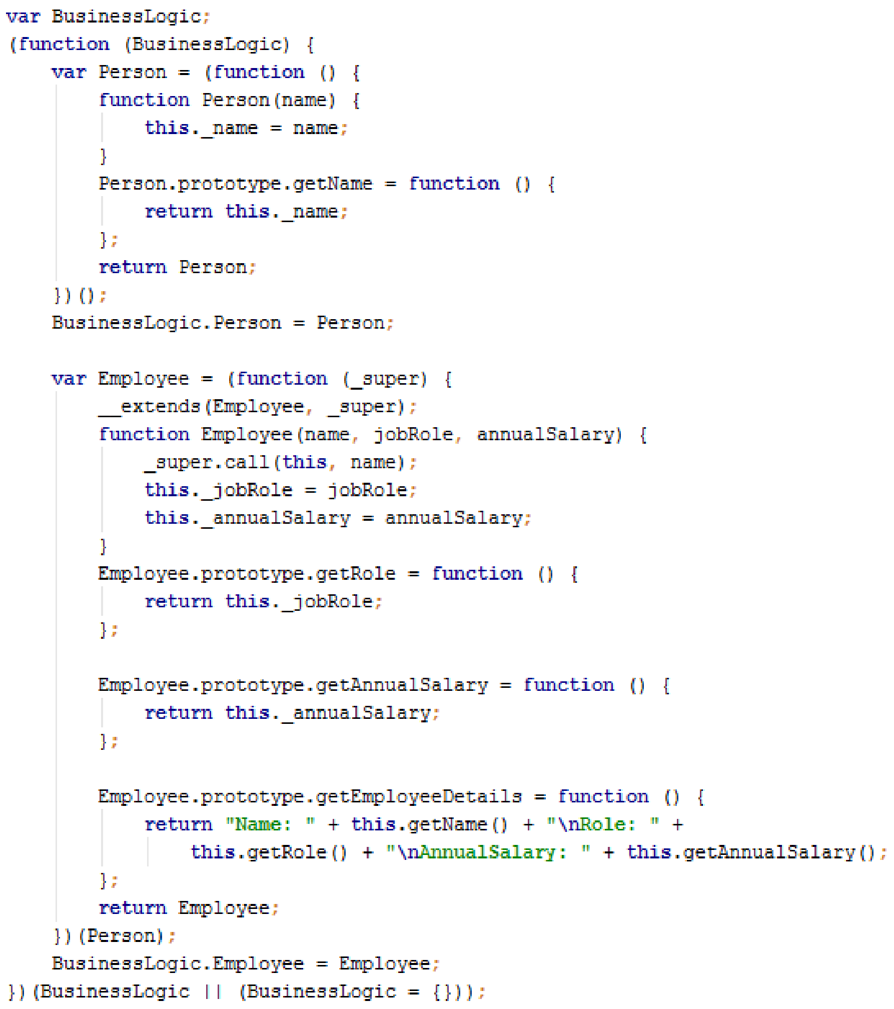
What follows next is a simple example of Typescript followed by its compiled output to JavaScript. One could almost be forgiven for thinking the example was written in Java or C# with its close syntactical resemblance.

```
module BusinessLogic {

    export interface IPerson {
        getName() : string;
    }

    export class Person implements IPerson {
        private _name:string;

        constructor(name:string) {
            this._name = name;
        }

        public getName():string {
            return this._name;
        }
    }


    export class Employee extends Person {
        private _jobRole:string;
        private _annualSalary:number;

        constructor(name:string, jobRole:string, annualSalary:number) {
            super(name);
            this._jobRole = jobRole;
            this._annualSalary = annualSalary;
        }

        public getRole():string {
            return this._jobRole;
        }

        public getAnnualSalary():number {
            return this._annualSalary;
        }

        public getEmployeeDetails():string {
            return "Name: " + this.getName() +
                "\nRole: " + this.getRole() +
                "\nAnnualSalary: " + this.getAnnualSalary();
        }
    }
}
```

Just explain some of the unfamiliar key words, a ‘**module’** can be thought as being namespace in .Net or package in Java.

The use of **‘export’** before a class is for visibility to other classes, almost like making a class public.

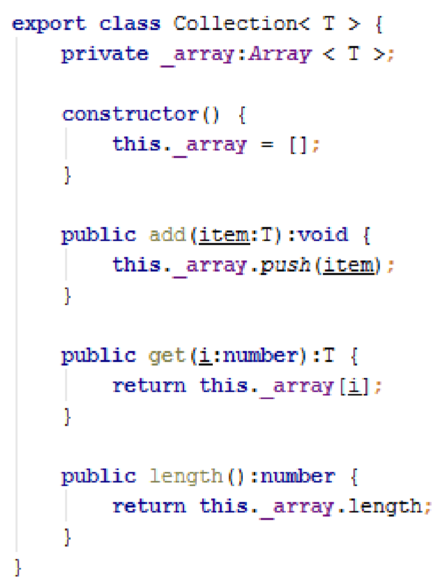
The following now is the compiled JavaScript output from the Typescript compiler (v1.0).



```
var BusinessLogic;
(function (BusinessLogic) {
    var Person = (function () {
        function Person(name) {
            this._name = name;
        }
        Person.prototype.getName = function () {
            return this._name;
        };
        return Person;
    })();
    BusinessLogic.Person = Person;

    var Employee = (function (_super) {
        __extends(Employee, _super);
        function Employee(name, jobRole, annualSalary) {
            _super.call(this, name);
            this._jobRole = jobRole;
            this._annualSalary = annualSalary;
        }
        Employee.prototype.getRole = function () {
            return this._jobRole;
        };

        Employee.prototype.getAnnualSalary = function () {
            return this._annualSalary;
        };

        Employee.prototype.getEmployeeDetails = function () {
            return "Name: " + this.getName() + "\nRole: " +
                this.getRole() + "\nAnnualSalary: " + this.getAnnualSalary();
        };
        return Employee;
    })(Person);
    BusinessLogic.Employee = Employee;
})(BusinessLogic || (BusinessLogic = {}));
```

The one caveat with using Typescript is that everything must be declared to avoid the compiler giving errors even when using external libraries. This can circumvented by declaring the type as “any” in advance or acquiring what is known as a type definition file which is even better as it declares all the methods of the external library ensuring type safety when using that library. An excellent open repository exists on GitHub titled ‘Definitely Typed’ which provides these typed files for most popular libraries, the THREE.js library here makes use of this typed file to allow it be used without generating compiler errors. https://github.com/borisyankov/DefinitelyTyped

Typescript also supports generic’s which is demonstrated in this custom collection that was created for the purposes of this implementation. The purpose of this class was to extend an array adding additional functionality and so the choice was to create a custom solution.



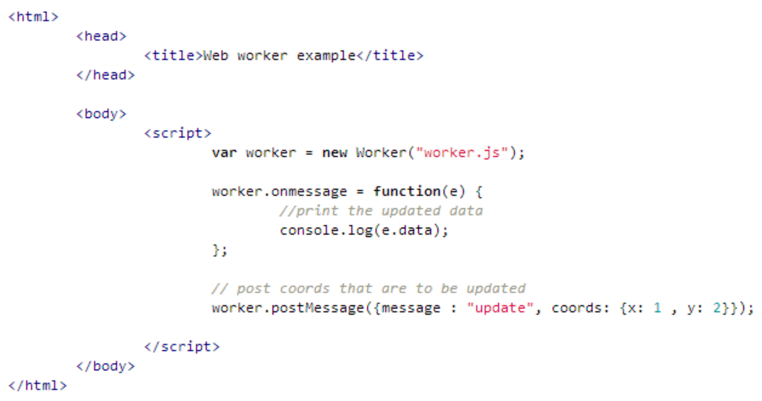
```
export class Collection< T > {
    private _array:Array < T >;

    constructor() {
        this._array = [];
    }

    public add(item:T):void {
        this._array.push(item);
    }

    public get(i:number):T {
        return this._array[i];
    }

    public length():number {
        return this._array.length;
    }
}
```

12.3 Web Workers

JavaScript by itself is essentially runs as a single process all in its own thread. This was never a problem before, but as web development matures, it is proving more and more to being capable of delivering rich content similar to that which is normally experience in native applications. Prior to the introduction of multi-processing in JavaScript, a browser trying to handle a large volume of work would warn the user through alert messages that it is under heavy load. This, from a user's perspective is undesirable as they won't know if this is meant to happen or if they are under attack from a malicious script. This problem was encountered early in the development phase and the bottlenecks became apparent and so it became apparent that the introduction of concurrency was needed.

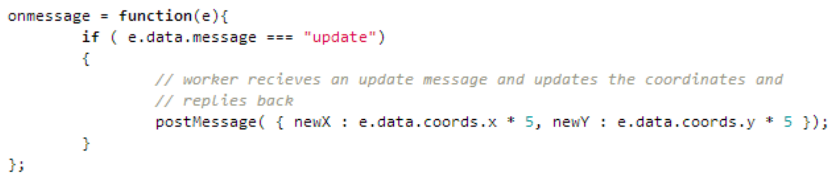
Workers are created in separate files and are run as a separate thread/process. Web Workers also have the ability to spawn other workers to further improve the processing ability. It utilises message passing by use of JSON or simple objects and communicates asynchronously through the use of events. This is an asynchronous process so the main thread will continue and the output from the worker process is dealt with through an event. Web Workers have no access to the DOM or the main process which is why there is a shift in thinking in that typical OO languages have access to objects outside their process but with Web Workers the only means of communication are through the use of messages (Bidelman, 2010).

The following is a basic implementation of a web worker. It should be noted that the worker itself is created in a separate file and is loaded at run time. The two processes communicate by way of events, ensures that this is a non-blocking process flow.



```
<html>
        <head>
                <title>Web worker example</title>
        </head>

        <body>
                <script>
                        var worker = new Worker("worker.js");

                        worker.onmessage = function(e) {
                                //print the updated data
                                console.log(e.data);
                        };

                        // post coords that are to be updated
                        worker.postMessage({message : "update", coords: {x: 1 , y: 2}});

                </script>
        </body>
</html>
```

Figure 70 - Worker instantiation and call

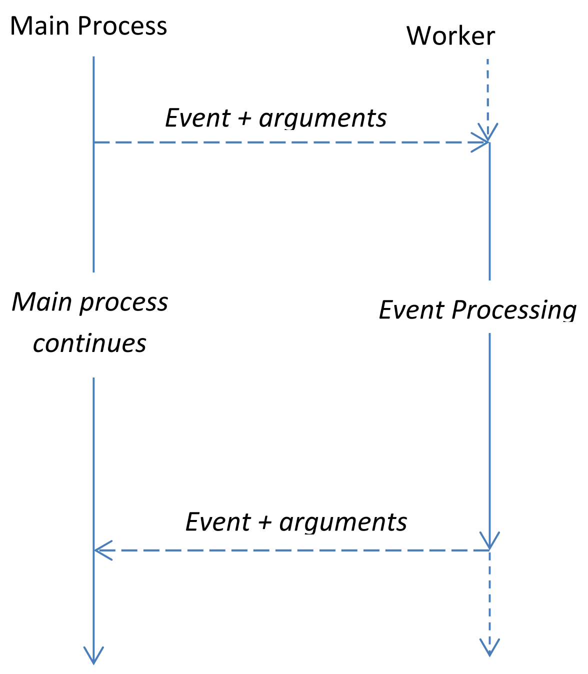


```
onmessage = function(e){
        if ( e.data.message === "update")
        {
                // worker recieves an update message and updates the coordinates and
                // replies back
                postMessage( { newX : e.data.coords.x * 5, newY : e.data.coords.y * 5 });
        }
};
```

Figure 71 Background worker script

As Web Workers are based around message passing as appose to object passing, care is needed to what information can be passed. What has been encountered during the development phase is that there was a need to create an “adapter” that basically is a slimmed down version of the object that requires processing or information that the threaded process requires. This is necessary as JSON doesn’t have the ability to convert complex objects that have circular dependency. Basically this means having composite objects (A has a B) can’t be converted so there is a need to create some form of adapter that passes only the need to know information.

It should also be noted that Web Workers do not have any access to the main “window” meaning they are completely isolated from accessing any variables in the main process or the UI. This in some ways is good in that there is little need to worry about issues that normally need to be considered when performing concurrency such as thread safety and locking activities, however, it does need a bit of thought in its implementation (Mozilla Developer Network, Using web workers, 2014).



12.4 Deployment through GitHub

For this project as well as being developed locally, we are going to make this project live to the world as it allows for us to experience the various delays that go hand in hand when content is being passed across continents rather than the instantaneous delivery experienced on the local loop.

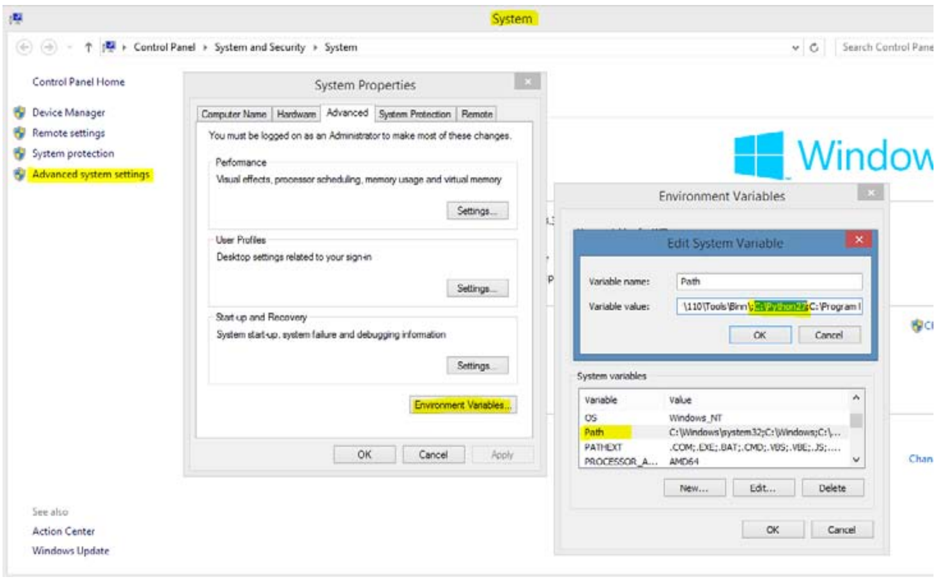
GitHub, where the source repository is being hosted also has a facility for hosting basic project pages related to that repository. This works perfect for us as we are only delivering HTML, CSS and JavaScript with the client doing all the rendering on their side

To create this, all that is needed is create a branch titled ‘gh-pages’ and because this was a web project anyway, this became the main branch of our project. To see any changes, all that was needed is to commit to main ‘gh-pages’ branch for them to go live. GitHub provides an address for your project which usually follows the line of http://username.github.io/repository.

12.5 Running the project locally

Because the project uses AJAX to request certain files, the browser will not allow for it to happen on a local machine because of Cross-Origin Resource Sharing (CORS). This is basically when a request is being made for content from a domain that is not the same as which the domain from which the originating request is being made from (Mozilla Developer Network, HTTP access control (CORS), 2014). In order to run the project locally, we can instead run a local HTTP server. The easiest one to install and which will be described here actually comes bundled with Python which is available from http://www.python.org/downloads/.

As mentioned, first download and install Python (2.7 preferred) and ensure that its path gets added to environment path which is located in Windows system settings.



You must be logged on as an Administrator to make most of these changes.

Performance: Visual effects, processor scheduling, memory usage and virtual memory

User Profiles: Desktop settings related to your sign-in

Start-up and Recovery: System start-up, system failure and debugging information

Variable name: Path

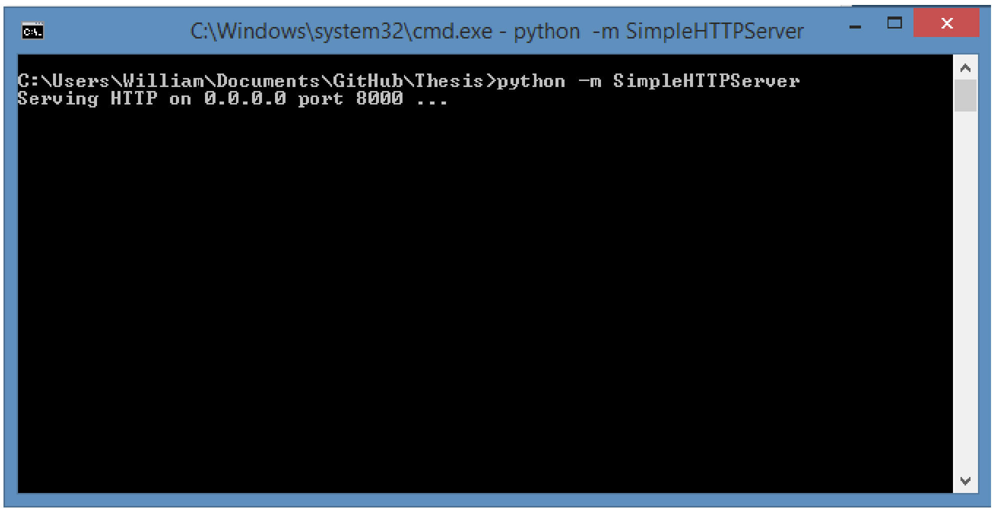
Variable value: \110\Tools\Binn\;C:\Python27;C:\Program I

System variables

Variable	Value
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	AMD64

Next, once this is done, navigate to folder from which the server is to be launched from (this will effectively become the root folder which will be indexed in the browser).

Run the following command form the command line: python –m SimpleHTTPServer



```
C:\Users\William\Documents\GitHub\Thesis>python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Finally, point the browser at address given and the contents of that folder will be displayed in the browser. If it detects an 'index.html' then that will be the page that will be displayed.

13.0 Conclusion

So as we approach the end, we have uncovered quite a lot on our journey to implement a volumetric sculpting application in the browser using WebGL. What is left for us now is to draw a conclusion on the research questions that were raised and which have largely been answered over the development phase of this thesis.

Is **WebGL a viable option for developing serious applications?**

Having WebGL certainly allows for the creation of feature rich applications or games for the web browser, however, saying it is viable compared traditional platforms presents a bit of debate. When we create desktop applications say in Java, we deploy to the Java Virtual Machine, if we develop in C# (mono withstanding) we are deploying to the .Net environment. Both of these have the ability to handle the interaction with the hardware using the JVM or CLR. So, under normal circumstances you expect the application to work out of the box. However, developing for the web is far more volatile. We are not simply developing for the browser but in fact several. We currently have Chrome, Firefox, Internet Explorer, Opera and Safari. Then we have to consider the mobile browsers which are their own kettle of fish. Browser developers strive to adhere to web standards but the fact of the matter is it occurs at different rates of development and different methods of implementation for each browser. This is what makes developing for every browser a scary prospect.

How viable is JavaScript, being a dynamic language, for creating large scale applications?

Our experience was that it wasn’t, however that is not a general statement as there are plenty of developers who are well versed in JavaScript that would be confident enough to tackle large projects using JavaScript as the primary langauge.

However, coming from a static typed background, the transition and sense of security offered by TypeScript made it a good choice for someone who was new to web development to be confident that they are writing good type safe code.

How can JavaScript deal with large computations, how can bottlenecks be dealt with?

JavaScript runs largely as a single process even AJAX is not truly asynchronous, it just lets the main process continue and only reacts by invoking a call back when the server returns. The draw backs of running large computational tasks definitely showed when running on the same process. However by introducing web workers, we found significant improvements by being able to have volumes of work broken down and off loaded to worker threads to process without impacting the main process.

Can a volume sculpting application be implemented in WebGL?

The results of our proof of concept application would suggest that it is possible. Although there is quite a bit more work and hopefully we will be reopened following the completion of this thesis. It would be hoped that further optimisations can be introduce in the solution obtained as well as leveraging the power of JavaScripts ability of concurrency to deliver a more streamline end product.

Can the use of WebGL help in avoiding a vendor lock in?

WebGL certainly can, as the only requirement is a compatible browser, a modern machine, a text editor and some basic knowledge to create fully feature rich 3D application. A danger that exists lies with changing web standards, what works today may not work in a years' time. Also a lot of browsers are still playing catch up with adhering to the WebGL standard. So, it's a little early yet as to whether a commitment be made to stick with WebGl.

Is THREE.JS a good choice of wrapper library for developing 3D applications? Can it boost productivity over using pure WebGL?

Yes, the contributors to this library have put an extra ordinate amount of effort into abstracting and creating wrappers of common graphical functions that are provided by WebGL. To think of producing the same application we produced in raw WebGL code, well, we would rather not think about it to put as mildly as possible.

When creating a volumetric model, how can the volume be manipulated and rendered?

We found using Hooke's law, we were able to mimic elastic properties rather well. However we only implemented this on the surface mesh for the proof of concept stage so we can't truthfully say that this would work if it was connecting the internal volume also, but there is no reason to suggest why it shouldn't.

Can the use of spatial data structures be used improve performance in this project?

Originally we had intended on using the Octree to store the voxels created in our volumetric world, however that didn't pan out as we found dealing with levels (world array contains a array of voxels) to be easier. However, the use of Octrees played a large part in the use of ray casting and sampling for our controller objects to produce volumetric data. We were able to store a reference to the faces in the Octree structure and that use it for efficient ray cast tests. Although we have no metrics to back up this statement, it is clear that by reducing the amount of objects by location, fewer tests were required of the ray cast. This and the earlier referenced research on Octree's, suggest yes, that spatial data structures can be used to improve performance in this project.

What are the possibilities with using volume Rendering/Sculpting?

We uncovered quite a few during our research but one of the more interesting possibilities included the latest phenomenon of 3D printing which is "possibly" going to revolutionise the manufacturing industry. We also looked at the possibility of setting up an online create and share venture. Finally, we looked at the possibility of applying this to developing an online game engine that uses voxels and the marching cube algorithm instead of traditional mesh modelling approach.

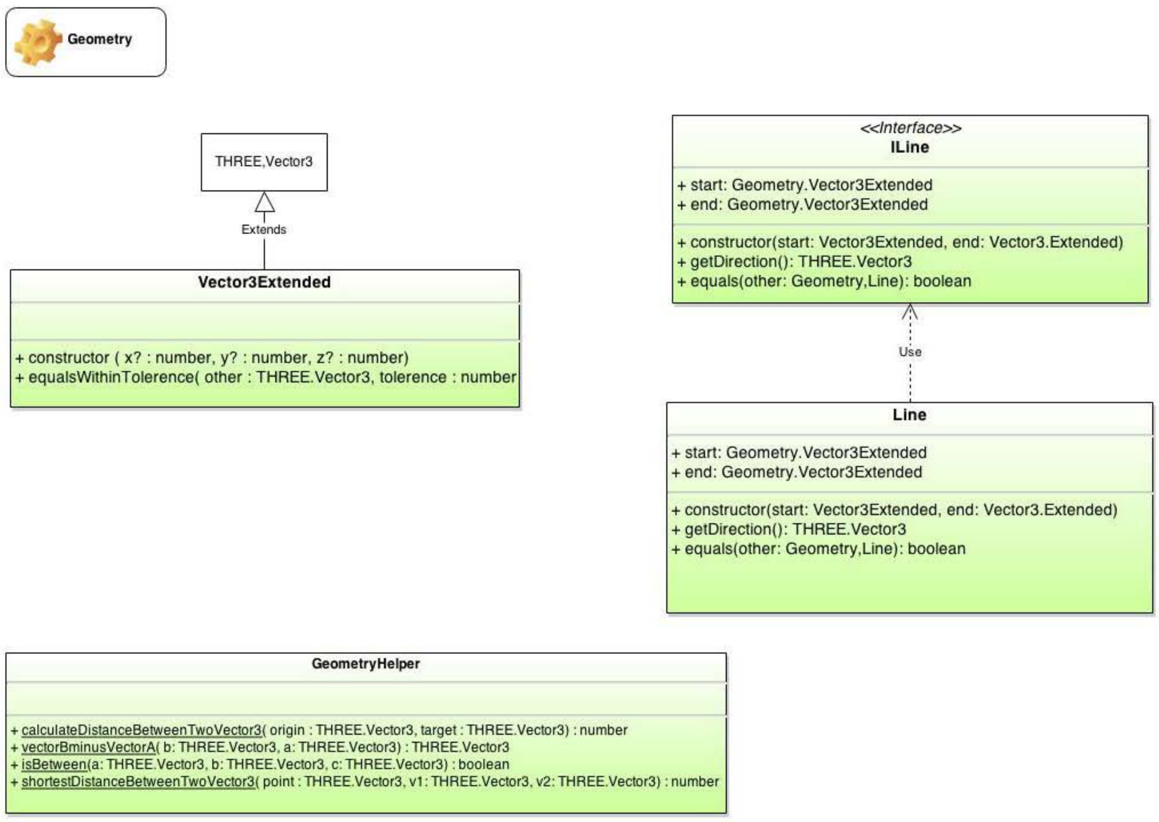
So, is there a future in WebGL?

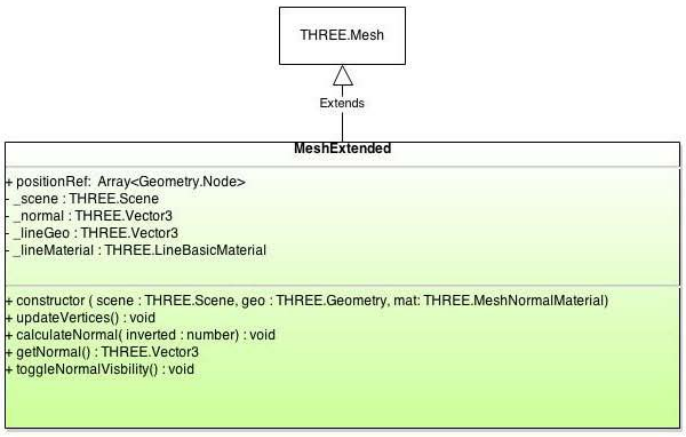
One final comment on the future of developing with WebGL for interactive content on the web is, from a personal point of view, an issue that needs to be examined is loading times for large projects. Having looked at a number of Three.js/WebGL examples, a lot of them suffer from long download times (testing on 3Mb broadband). A question that has to be asked is how patient will the client be to await deliver of content? A suggestion is possibly to

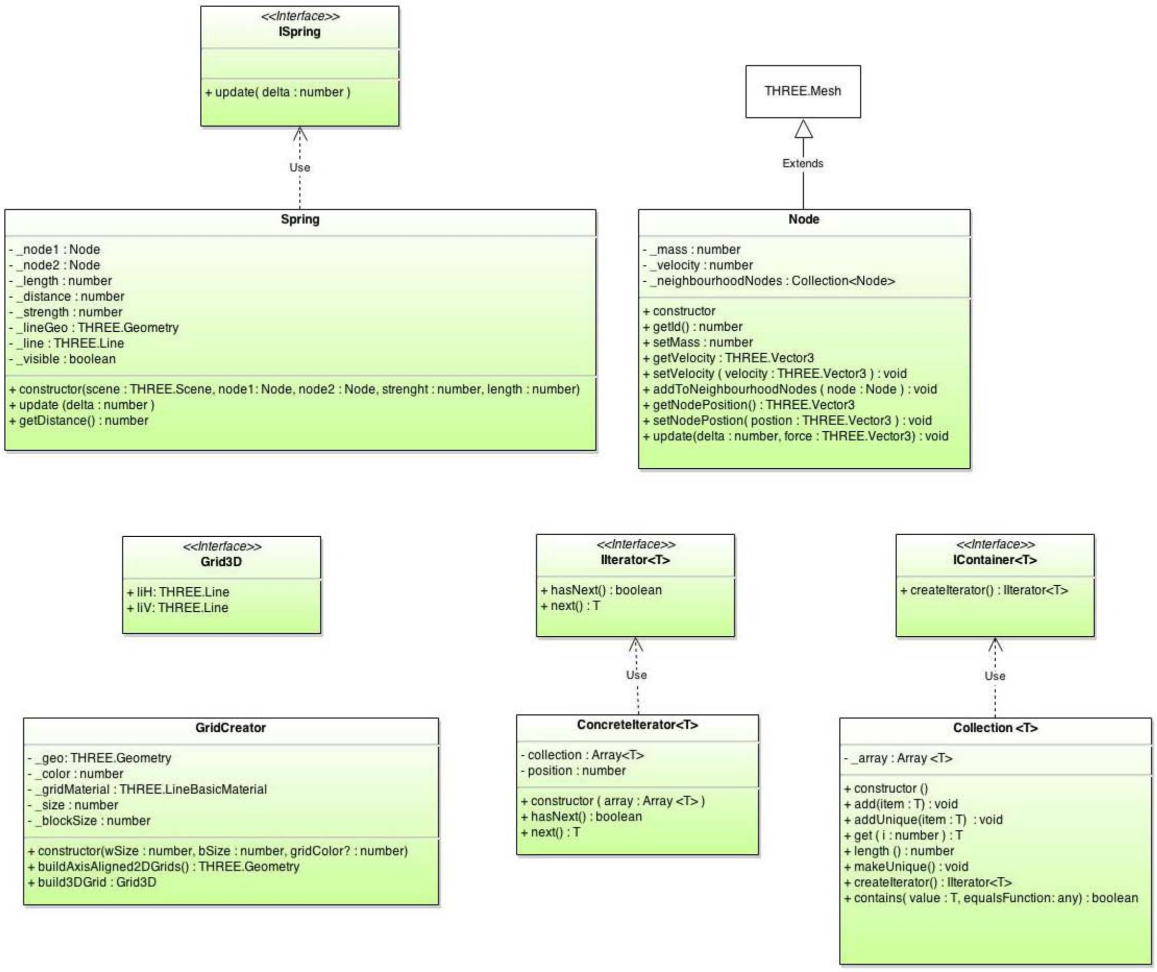
have some interactive content while waiting or possibly only load content progressively that is needed.
To get an industry professionals opinion on the matter of the future of WebGL we make reference to comments made by John Carmacks the co-founder and former employee of id software who now currently heads up Oculas rift development. John speaks from an optimisation point of view, but he makes references to writing high performance graphic applications using JavaScript as almost being “offensively wrong”, although the comment seemed meant in jest. Speaking of the future of delivery of content via the web, he seems to favour the cloud for delivering value (QuakeCon2011, 2012). All that remains to be seen is he right and is this just another fad. Although as this conclusion was been written, Unity, one of the more popular game engines available, have announced that in their Unity 5 game engine that they intend to support porting to WebGL which indicates at least in their eyes, WebGL has a future (Echterhoff, 2014).

14.0 Appendix

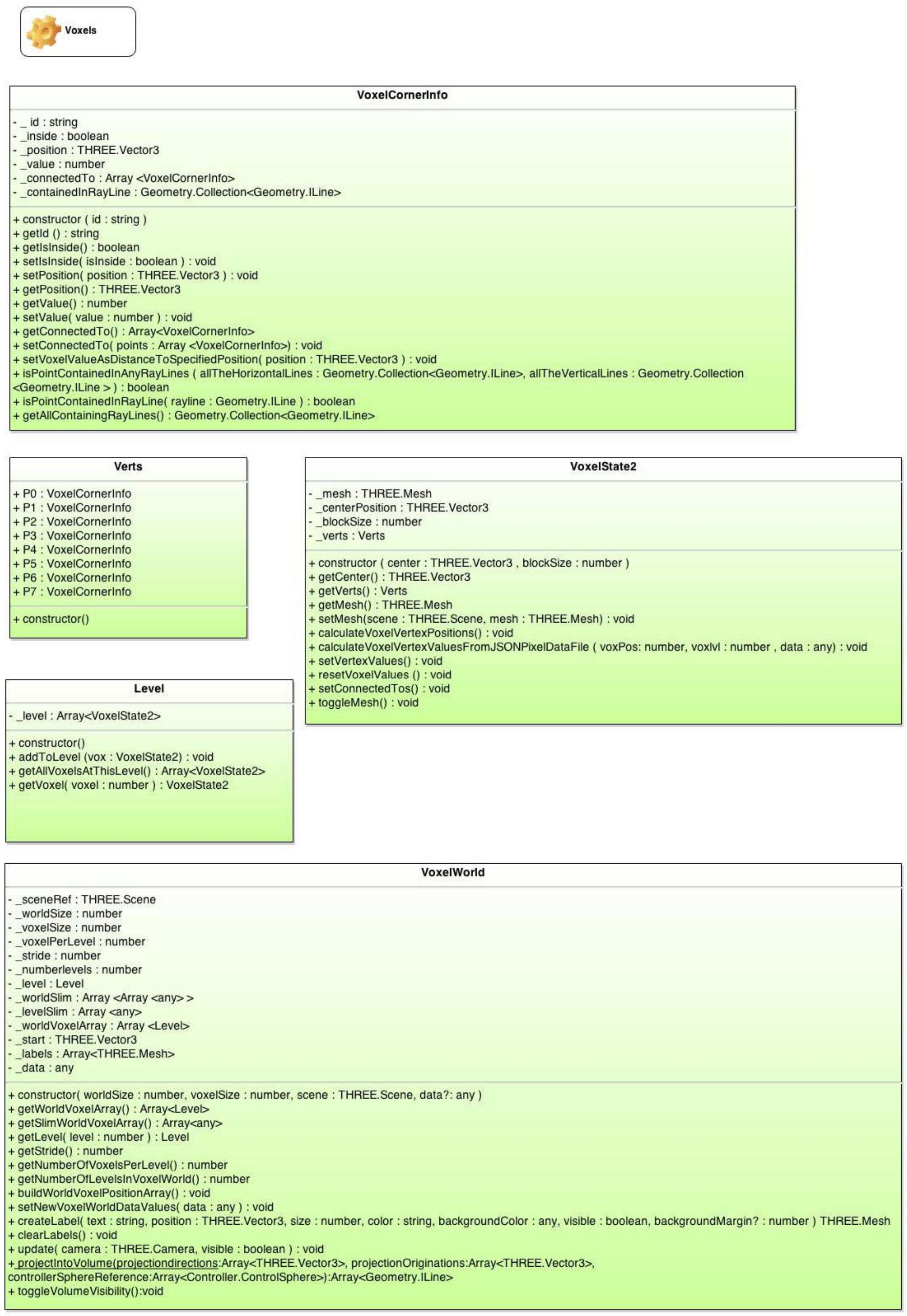
14.1 UML Diagrams


Geometry
THREE,Vector3
Extends
Vector3Extended
+ constructor (x? : number, y? : number, z? : number)
+ equalsWithinTolerence(other : THREE.Vector3, tolerence : number
<<Interface>>
ILine
+ start: Geometry.Vector3Extended
+ end: Geometry.Vector3Extended
+ constructor(start: Vector3Extended, end: Vector3.Extended)
+ getDirection(): THREE.Vector3
+ equals(other: Geometry,Line): boolean
Use
Line
+ start: Geometry.Vector3Extended
+ end: Geometry.Vector3Extended
+ constructor(start: Vector3Extended, end: Vector3.Extended)
+ getDirection(): THREE.Vector3
+ equals(other: Geometry,Line): boolean
GeometryHelper
+ calculateDistanceBetweenTwoVector3(origin : THREE.Vector3, target : THREE.Vector3) : number
+ vectorBminusVectorA(b: THREE.Vector3, a: THREE.Vector3) : THREE.Vector3
+ isBetween(a: THREE.Vector3, b: THREE.Vector3, c: THREE.Vector3) : boolean
+ shortestDistanceBetweenTwoVector3(point : THREE.Vector3, v1: THREE.Vector3, v2: THREE.Vector3) : number


THREE.Mesh
Extends
MeshExtended
+ positionRef: Array<Geometry.Node>
- _scene : THREE.Scene
- _normal : THREE.Vector3
- _lineGeo : THREE.Vector3
- _lineMaterial : THREE.LineBasicMaterial
+ constructor (scene : THREE.Scene, geo : THREE.Geometry, mat: THREE.MeshNormalMaterial)
+ updateVertices() : void
+ calculateNormal(inverted : number) : void
+ getNormal() : THREE.Vector3
+ toggleNormalVisbility() : void



<<Interface>>
ISpring
+ update(delta : number)
Use
Spring
- _node1 : Node
- _node2 : Node
- _length : number
- _distance : number
- _strength : number
- _lineGeo : THREE.Geometry
- _line : THREE.Line
- _visible : boolean
+ constructor(scene : THREE.Scene, node1: Node, node2 : Node, strenght : number, length : number)
+ update (delta : number)
+ getDistance() : number
THREE.Mesh
Extends
Node
- _mass : number
- _velocity : number
- _neighbourhoodNodes : Collection<Node>
+ constructor
+ getId() : number
+ setMass : number
+ getVelocity : THREE.Vector3
+ setVelocity (velocity : THREE.Vector3) : void
+ addToNeighbourhoodNodes (node : Node) : void
+ getNodePosition() : THREE.Vector3
+ setNodePostion(postion : THREE.Vector3) : void
+ update(delta : number, force : THREE.Vector3) : void
<<Interface>>
Grid3D
+ liH: THREE.Line
+ liV: THREE.Line
GridCreator
- _geo: THREE.Geometry
- _color : number
- _gridMaterial : THREE.LineBasicMaterial
- _size : number
- _blockSize : number
+ constructor(wSize : number, bSize : number, gridColor? : number)
+ buildAxisAligned2DGrids() : THREE.Geometry
+ build3DGrid : Grid3D
<<Interface>>
IIterator<T>
+ hasNext() : boolean
+ next() : T
Use
ConcreteIterator<T>
- collection : Array<T>
- position : number
+ constructor (array : Array <T>)
+ hasNext() : boolean
+ next() : T
<<Interface>>
IContainer<T>
+ createIterator() : IIterator<T>
Use
Collection <T>
- _array : Array <T>
+ constructor ()
+ add(item : T) : void
+ addUnique(item : T) : void
+ get (i : number) : T
+ length () : number
+ makeUnique() : void
+ createIterator() : IIterator<T>
+ contains(value : T, equalsFunction: any) : boolean



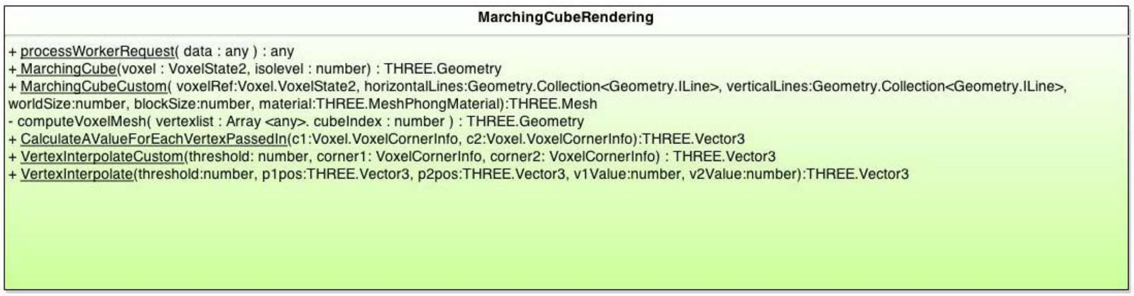
```
VoxelCornerInfo
- _id : string
- _inside : boolean
- _position : THREE.Vector3
- _value : number
- _connectedTo : Array <VoxelCornerInfo>
- _containedInRayLine : Geometry.Collection<Geometry.ILine>
+ constructor ( id : string )
+ getId () : string
+ getIsInside() : boolean
+ setIsInside( isInside : boolean ) : void
+ setPosition( position : THREE.Vector3 ) : void
+ getPosition() : THREE.Vector3
+ getValue() : number
+ setValue( value : number ) : void
+ getConnectedTo() : Array<VoxelCornerInfo>
+ setConnectedTo( points : Array <VoxelCornerInfo>) : void
+ setVoxelValueAsDistanceToSpecifiedPosition( position : THREE.Vector3 ) : void
+ isPointContainedInAnyRayLines ( allTheHorizontalLines : Geometry.Collection<Geometry.ILine>, allTheVerticalLines : Geometry.Collection
<Geometry.ILine > ) : boolean
+ isPointContainedInRayLine( rayline : Geometry.ILine ) : boolean
+ getAllContainingRayLines() : Geometry.Collection<Geometry.ILine>
```

```
Verts
+ P0 : VoxelCornerInfo
+ P1 : VoxelCornerInfo
+ P2 : VoxelCornerInfo
+ P3 : VoxelCornerInfo
+ P4 : VoxelCornerInfo
+ P5 : VoxelCornerInfo
+ P6 : VoxelCornerInfo
+ P7 : VoxelCornerInfo
+ constructor()
```

```
VoxelState2
- _mesh : THREE.Mesh
- _centerPosition : THREE.Vector3
- _blockSize : number
- _verts : Verts
+ constructor ( center : THREE.Vector3 , blockSize : number )
+ getCenter() : THREE.Vector3
+ getVerts() : Verts
+ getMesh() : THREE.Mesh
+ setMesh(scene : THREE.Scene, mesh : THREE.Mesh) : void
+ calculateVoxelVertexPositions() : void
+ calculateVoxelVertexValuesFromJSONPixelDataFile ( voxPos: number, voxlvl : number , data : any) : void
+ setVertexValues() : void
+ resetVoxelValues () : void
+ setConnectedTos() : void
+ toggleMesh() : void
```

```
Level
- _level : Array<VoxelState2>
+ constructor()
+ addToLevel (vox : VoxelState2) : void
+ getAllVoxelsAtThisLevel() : Array<VoxelState2>
+ getVoxel( voxel : number ) : VoxelState2
```

```
VoxelWorld
- _sceneRef : THREE.Scene
- _worldSize : number
- _voxelSize : number
- _voxelPerLevel : number
- _stride : number
- _numberlevels : number
- _level : Level
- _worldSlim : Array <Array <any> >
- _levelSlim : Array <any>
- _worldVoxelArray : Array <Level>
- _start : THREE.Vector3
- _labels : Array<THREE.Mesh>
- _data : any
+ constructor( worldSize : number, voxelSize : number, scene : THREE.Scene, data?: any )
+ getWorldVoxelArray() : Array<Level>
+ getSlimWorldVoxelArray() : Array<any>
+ getLevel( level : number ) : Level
+ getStride() : number
+ getNumberOfVoxelsPerLevel() : number
+ getNumberOfLevelsInVoxelWorld() : number
+ buildWorldVoxelPositionArray() : void
+ setNewVoxelWorldDataValues( data : any ) : void
+ createLabel( text : string, position : THREE.Vector3, size : number, color : string, backgroundColor : any, visible : boolean, backgroundMargin? : number ) THREE.Mesh
+ clearLabels() : void
+ update( camera : THREE.Camera, visible : boolean ) : void
+ projectIntoVolume(projectiondirections:Array<THREE.Vector3>, projectionOriginations:Array<THREE.Vector3>,
controllerSphereReference:Array<Controller.ControlSphere>):Array<Geometry.ILine>
+ toggleVolumeVisibility():void
```



```
MarchingCubeRendering

+ processWorkerRequest( data : any ) : any
+ MarchingCube(voxel : VoxelState2, isolevel : number) : THREE.Geometry
+ MarchingCubeCustom( voxelRef:Voxel.VoxelState2, horizontalLines:Geometry.Collection<Geometry.ILine>, verticalLines:Geometry.Collection<Geometry.ILine>,
worldSize:number, blockSize:number, material:THREE.MeshPhongMaterial):THREE.Mesh
- computeVoxelMesh( vertexlist : Array <any>. cubeIndex : number ) : THREE.Geometry
+ CalculateAValueForEachVertexPassedIn(c1:Voxel.VoxelCornerInfo, c2:Voxel.VoxelCornerInfo):THREE.Vector3
+ VertexInterpolateCustom(threshold: number, corner1: VoxelCornerInfo, corner2: VoxelCornerInfo) : THREE.Vector3
+ VertexInterpolate(threshold:number, p1pos:THREE.Vector3, p2pos:THREE.Vector3, v1Value:number, v2Value:number):THREE.Vector3
```

Helper

JQueryHelper

+ getScreenWH(id : string) : Array <number>
+ appendToScene(id : string, renderer : THREE.WebGLRenderer) : void

Imaging

<<Interface>>
IHorizontalImageSlice

+ top: HTMLCanvasElement
+ bottom: HTMLCanvasElement

<<Interface>>
IVerticalImageSlice

+ near : HTMLCanvasElement
+ far : HTMLCanvasElement

CanvasRender

+ drawCanvas(name:string, arrayOfLines:Array<Geometry.ILine>, translateTo:THREE.Vector3, orientation:number, drawGrid:boolean, worldSize:number, blockSize:number):HTMLCanvasElement
+ drawImage(canvasID:string, imageToSuperImpose:any) : void
+ drawImage2(canvas : HTMLCanvasElement, imageToSuperImpose: any) : void
+ drawAllImages(arrayOfHorizontalSlices:Array<IHorizontalImageSlice>, arrayOfVerticalSlices:Array<IVerticalImageSlice>, horizontalElemID:string, verticalElemID:string):void
+ clearAllImages(horizontalElemID:string, verticalElemID:string):void

Controller

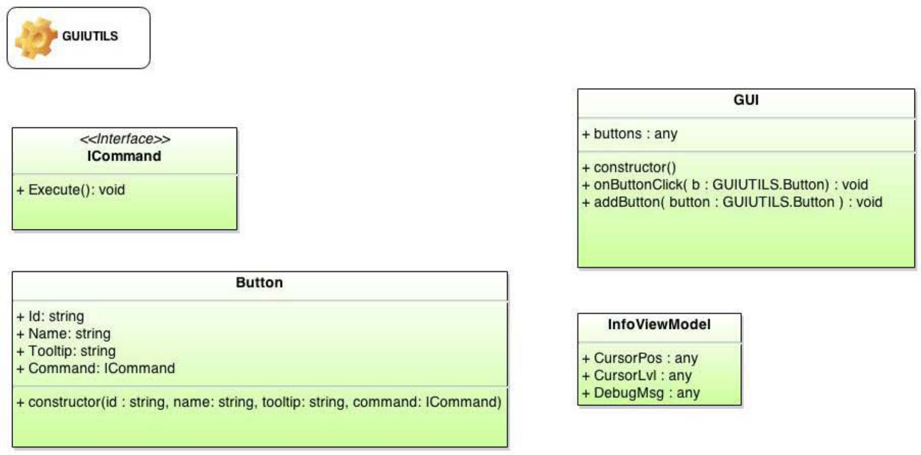
<<Interface>>
ISphereSkeleton

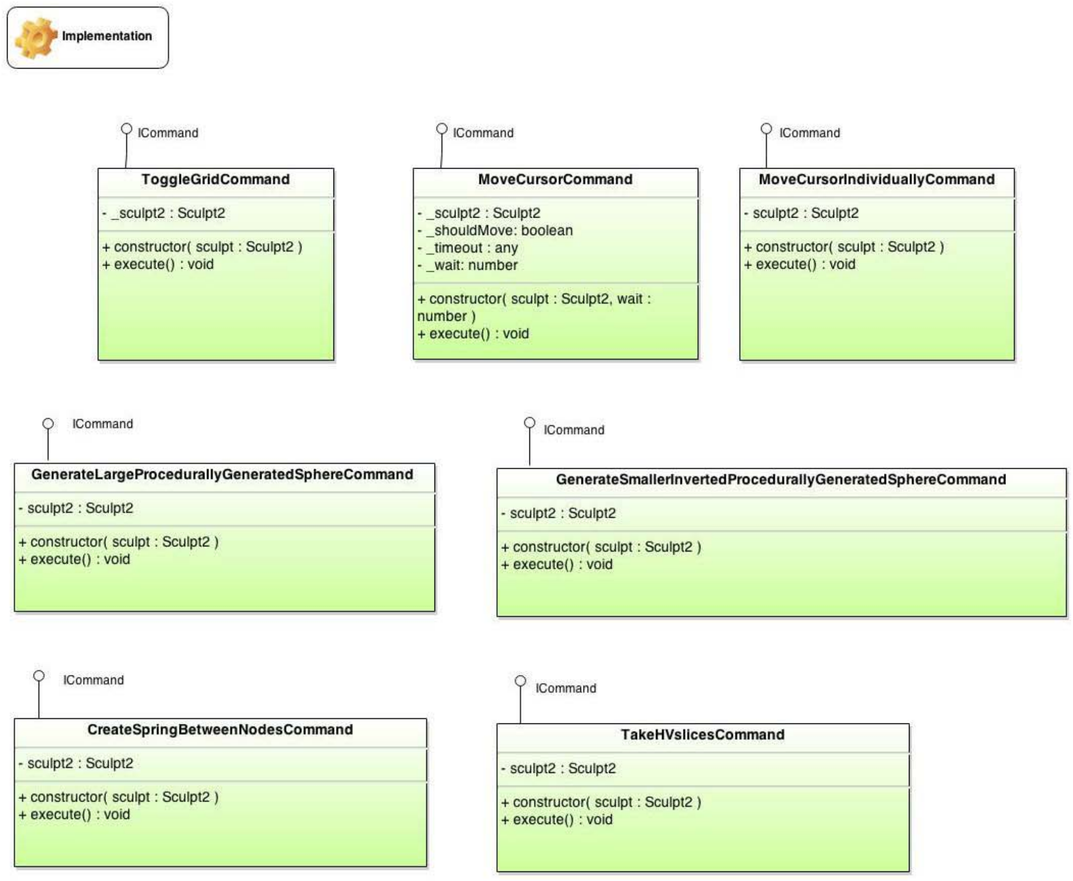
+ points : Array<THREE.Vector3>
+ lines : Array<THREE.Lines>

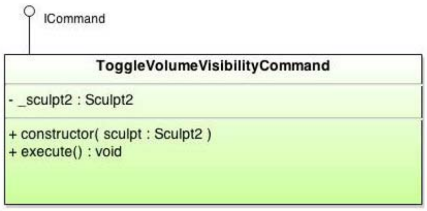
ControlSphere

- _id:number;
- _n:number;
- _m:number;
- _radius:number;
- _scene:THREE.Scene;
- _nodeSize:number;
- _nodeVelocity:THREE.Vector3;
- _nodeMass:number;
- _nodes:Array<Geometry.Node>;
- _faces:Array<Geometry.MeshExtended>;
- _octreeForNodes:any;
- _octreeForFaces:any;
- _sphereSkeleton:ISphereSkeleton;
- _alreadyGenerated:boolean;

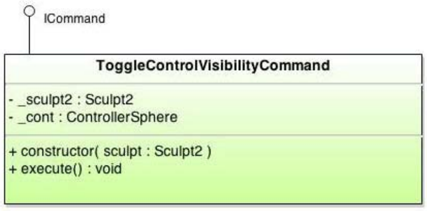
+ constructor(id:number, segments:number, radius:number, scene:THREE.Scene, size:number, velocity:THREE.Vector3, mass:number)
+ getNodes():Array<Geometry.Node>
+ getSphereSkeleton() : ISphereSkeleton
+ getOctreeForNodes() : any
+ getOctreeForFaces() : any
+ toggleVisibility() : void
- generateSphereVerticesandLineConnectors() : void
+ generateSphere() : void
+ calculateFaces() : void
+ calculateMeshFacePositions(particles : any, segments : any) : Array<any>
+ addFaces(verts : any) : void
+ update (inverted : number) : void


GUIUTILS
<<Interface>>
ICommand
+ Execute(): void
Button
+ Id: string
+ Name: string
+ Tooltip: string
+ Command: ICommand
+ constructor(id : string, name: string, tooltip: string, command: ICommand)
GUI
+ buttons : any
+ constructor()
+ onButtonClick(b : GUIUTILS.Button) : void
+ addButton(button : GUIUTILS.Button) : void
InfoViewModel
+ CursorPos : any
+ CursorLvl : any
+ DebugMsg : any

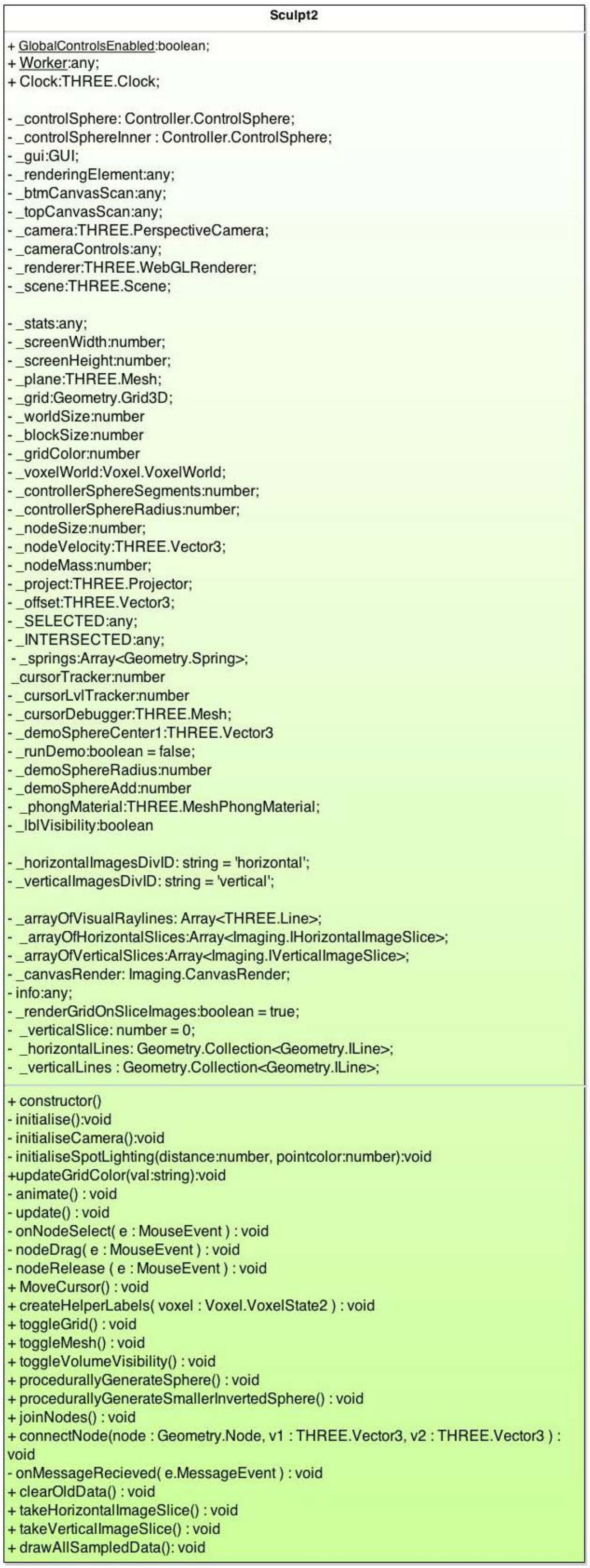

Implementation
ICommand
ToggleGridCommand
- _sculpt2 : Sculpt2
+ constructor(sculpt : Sculpt2)
+ execute() : void
ICommand
MoveCursorCommand
- _sculpt2 : Sculpt2
- _shouldMove: boolean
- _timeout : any
- _wait: number
+ constructor(sculpt : Sculpt2, wait : number)
+ execute() : void
ICommand
MoveCursorIndividuallyCommand
- sculpt2 : Sculpt2
+ constructor(sculpt : Sculpt2)
+ execute() : void
ICommand
GenerateLargeProcedurallyGeneratedSphereCommand
- sculpt2 : Sculpt2
+ constructor(sculpt : Sculpt2)
+ execute() : void
ICommand
GenerateSmallerInvertedProcedurallyGeneratedSphereCommand
- sculpt2 : Sculpt2
+ constructor(sculpt : Sculpt2)
+ execute() : void
ICommand
CreateSpringBetweenNodesCommand
- sculpt2 : Sculpt2
+ constructor(sculpt : Sculpt2)
+ execute() : void
ICommand
TakeHVslicesCommand
- sculpt2 : Sculpt2
+ constructor(sculpt : Sculpt2)
+ execute() : void



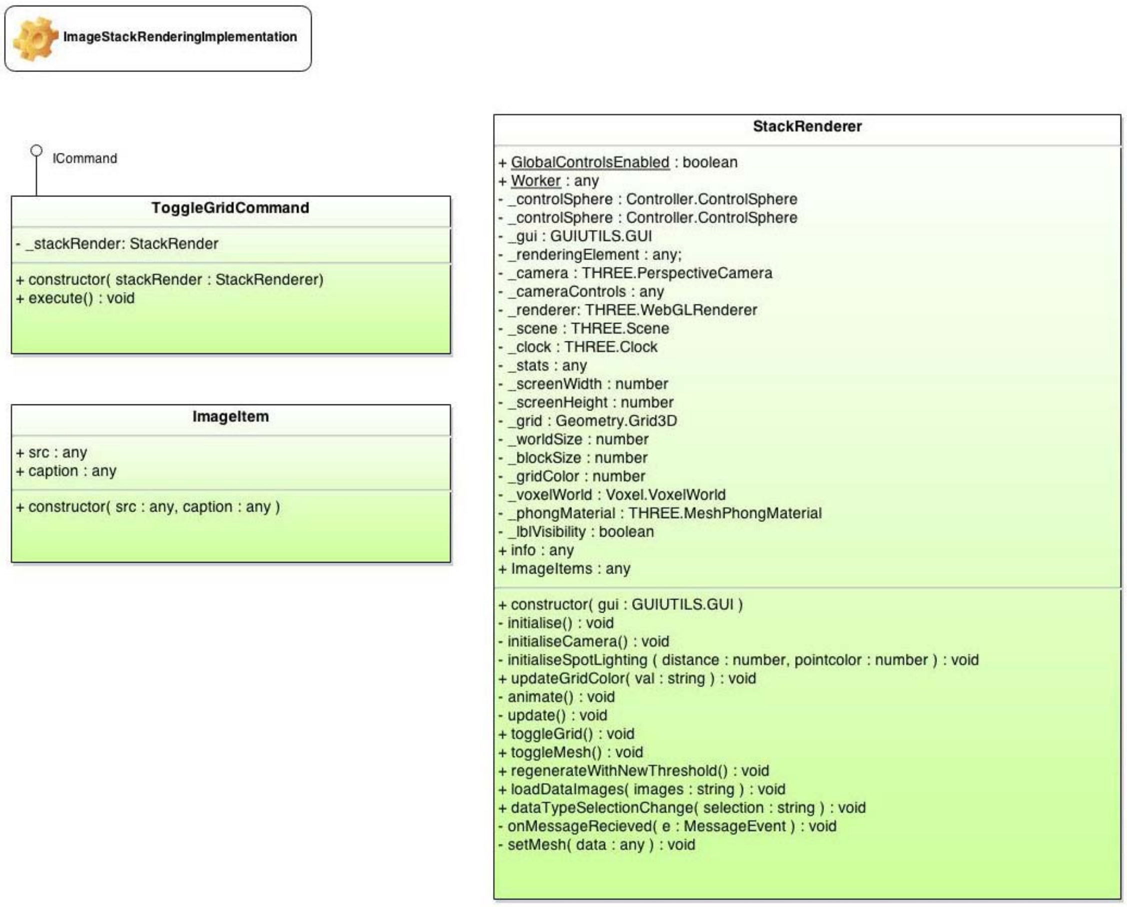
```
ICommand
ToggleVolumeVisibilityCommand
- _sculpt2 : Sculpt2
+ constructor( sculpt : Sculpt2 )
+ execute() : void
```



```
ICommand
ToggleControlVisibilityCommand
- _sculpt2 : Sculpt2
- _cont : ControllerSphere
+ constructor( sculpt : Sculpt2 )
+ execute() : void
```



```
Sculpt2
+ GlobalControlsEnabled:boolean;
+ Worker:any;
+ Clock:THREE.Clock;

- _controlSphere: Controller.ControlSphere;
- _controlSphereInner : Controller.ControlSphere;
- _gui:GUI;
- _renderingElement:any;
- _btmCanvasScan:any;
- _topCanvasScan:any;
- _camera:THREE.PerspectiveCamera;
- _cameraControls:any;
- _renderer:THREE.WebGLRenderer;
- _scene:THREE.Scene;

- _stats:any;
- _screenWidth:number;
- _screenHeight:number;
- _plane:THREE.Mesh;
- _grid:Geometry.Grid3D;
- _worldSize:number
- _blockSize:number
- _gridColor:number
- _voxelWorld:Voxel.VoxelWorld;
- _controllerSphereSegments:number;
- _controllerSphereRadius:number;
- _nodeSize:number;
- _nodeVelocity:THREE.Vector3;
- _nodeMass:number;
- _project:THREE.Projector;
- _offset:THREE.Vector3;
- _SELECTED:any;
- _INTERSECTED:any;
 - _springs:Array<Geometry.Spring>;
 _cursorTracker:number
- _cursorLvlTracker:number
- _cursorDebugger:THREE.Mesh;
- _demoSphereCenter1:THREE.Vector3
- _runDemo:boolean = false;
- _demoSphereRadius:number
- _demoSphereAdd:number
-  _phongMaterial:THREE.MeshPhongMaterial;
- _lblVisibility:boolean

- _horizontalImagesDivID: string = 'horizontal';
- _verticalImagesDivID: string = 'vertical';

- _arrayOfVisualRaylines: Array<THREE.Line>;
-  _arrayOfHorizontalSlices:Array<Imaging.IHorizontalImageSlice>;
- _arrayOfVerticalSlices:Array<Imaging.IVerticalImageSlice>;
- _canvasRender: Imaging.CanvasRender;
- info:any;
- _renderGridOnSliceImages:boolean = true;
-  _verticalSlice: number = 0;
-  _horizontalLines: Geometry.Collection<Geometry.ILine>;
-  _verticalLines : Geometry.Collection<Geometry.ILine>;

+ constructor()
- initialise():void
- initialiseCamera():void
- initialiseSpotLighting(distance:number, pointcolor:number):void
+updateGridColor(val:string):void
- animate() : void
- update() : void
- onNodeSelect( e : MouseEvent ) : void
- nodeDrag( e : MouseEvent ) : void
- nodeRelease ( e : MouseEvent ) : void
+ MoveCursor() : void
+ createHelperLabels( voxel : Voxel.VoxelState2 ) : void
+ toggleGrid() : void
+ toggleMesh() : void
+ toggleVolumeVisibility() : void
+ procedurallyGenerateSphere() : void
+ procedurallyGenerateSmallerInvertedSphere() : void
+ joinNodes() : void
+ connectNode(node : Geometry.Node, v1 : THREE.Vector3, v2 : THREE.Vector3 ) :
void
- onMessageRecieved( e.MessageEvent ) : void
+ clearOldData() : void
+ takeHorizontalImageSlice() : void
+ takeVerticalImageSlice() : void
+ drawAllSampledData(): void
```


ImageStackRenderingImplementation
ICommand
ToggleGridCommand
- _stackRender: StackRender
+ constructor(stackRender : StackRenderer)
+ execute() : void
ImageItem
+ src : any
+ caption : any
+ constructor(src : any, caption : any)
StackRenderer
+ GlobalControlsEnabled : boolean
+ Worker : any
- _controlSphere : Controller.ControlSphere
- _controlSphere : Controller.ControlSphere
- _gui : GUIUTILS.GUI
- _renderingElement : any;
- _camera : THREE.PerspectiveCamera
- _cameraControls : any
- _renderer: THREE.WebGLRenderer
- _scene : THREE.Scene
- _clock : THREE.Clock
- _stats : any
- _screenWidth : number
- _screenHeight : number
- _grid : Geometry.Grid3D
- _worldSize : number
- _blockSize : number
- _gridColor : number
- _voxelWorld : Voxel.VoxelWorld
- _phongMaterial : THREE.MeshPhongMaterial
- _lblVisibility : boolean
+ info : any
+ ImageItems : any
+ constructor(gui : GUIUTILS.GUI)
- initialise() : void
- initialiseCamera() : void
- initialiseSpotLighting (distance : number, pointcolor : number) : void
+ updateGridColor(val : string) : void
- animate() : void
- update() : void
+ toggleGrid() : void
+ toggleMesh() : void
+ regenerateWithNewThreshold() : void
+ loadDataImages(images : string) : void
+ dataTypeSelectionChange(selection : string) : void
- onMessageRecieved(e : MessageEvent) : void
- setMesh(data : any) : void

14.2 Libraries used in the development for this Thesis

Some of these licenses have moved to MIT licensing since (more liberal) but we may not be using the latest and will state the license of the used library.

Library	**Source**	**Purpose**	**License**
Three JS	Author: Richardo Cabello Now hosted on GitHub as community project.	A library that abstracts WebGL calls to a higher and more user friendly level.	MIT
JQuery and JQueryUI	http://jquery.com/	Description from the main site: “jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.”	MIT
Bootstrap (js and css)	http://getbootstrap.com/	Site description – “The most popular front-end framework for developing responsive, mobile first projects on the web.”	Apache License, Version 2.0
dat.gui.js	Google Data Arts Team https://github.com/dataarts/dat.gui	Out of the box controller gui/menu for JavaScript	Apache License, Version 2.0
dectector.js	alteredq -http://alteredqualia.com mr.doob - http://mrdoob.com/ Packaged as part of the Three.js download	Detects if WebGL is available in this browser and alerts the user to update their browser or fix it if it is not.	MIT
Knockout	http://knockoutjs.com/	Allows for associating DOM elements with JavaScript model data. I.e. updates to model data will be reflected in the UI	MIT
MCData.js	Multiple – see notice in file	Look up tables	NA
Orbitcontrols .js	qiao / https://github.com/qiao mrdoob / http://mrdoob.com alteredq / http://alteredqualia.com/ WestLangley / http://github.com/WestLangley erich666 / http://erichaines.com	Packaged as part of three.js download it provides a library for various camera controls	MIT
Pick-a-color.js	Lauren Sperber and Broadstreet Ads https://github.com/lauren/pick-a-color	Color selector	MIT

Qunit	https://qunitjs.com/	Test framework	MIT
stats.js	Mr Doob, this is packaged with THREE.JS	Display FPS in the GUI	MIT
threeoctree.js	Collinhover https://github.com/collinhover/threeoctree	Repository description – "(sparse + dynamic) 3D spatial representation structure for fast searches"	MIT
tinycolor	https://github.com/bgrins/TinyColor	"Fast, small color manipulation and conversion for JavaScript http://bgrins.github.com/TinyColor/"	MIT
Underscore.js	Jeremy Ashkenas, DocumentCloud and Investigative Reporters & Editors http://underscorejs.org	Provides functions such as map, filter and reduce	MIT

Works Cited

Anyuru, A. (2012). *Professional WebGL Programming : Developing 3D Graphics for the Web.* Somerset, NJ, USA: Wiley.

Autodesk. (2013, October 20). *Digital sculpting and digital painting software*. Retrieved from Autodesk: http://www.autodesk.com/products/mudbox/overview

Bidelman, E. (2010, July 26). *The Basics of Web Workers*. Retrieved from HTML5 Rocks: http://www.html5rocks.com/en/tutorials/workers/basics/

Blender Foundation. (2013, October 20). *Features*. Retrieved from Blender: http://www.blender.org/features-gallery/features/

Blender Foundation. (2013, July). *History*. Retrieved from Blender.org: http://www.blender.org/foundation/history/

Blender.org. (2013, February 20). *Blender 2.66: Dynamic Topology Sculpting*. Retrieved from Blender Wiki: http://wiki.blender.org/index.php/Dev:Ref/Release_Notes/2.66/Dynamic_Topology_Sculpting

Borland, D., & Taylor, R. (2007). Rainbow Color Map (Still) Considered Harmful. *Computer Graphics and Applications, 27*(2), 14-17.

Boulos, S., Edwards, D., Lacewell, D. J., Kniss, J., Kautz, J., Shirley, P., & Wald, I. (2007). Packet-based Whitted and Distribution Ray Tracing. *Proceedings of Graphics Interface 2007* (pp. 177-184). New York, NY, USA: ACM.

Bourke, P. (1994, May -). *Polygonising a scalar field*. Retrieved November 13, 2013, from Paul Bourke: http://paulbourke.net/geometry/polygonise/

Bridge, H. (2010, March 18). *Introducing the ANGLE Project*. Retrieved from The Chromium Blog: http://blog.chromium.org/2010/03/introducing-angle-project.html

Calhoun BFA, P. S., Kuszyk MD, B. S., Heath PhD, D. G., Carley BS, J. C., & Fishman MD, E. K. (1999). Three-dimensional volume rendering of Spiral CT data: Theory and Method. *RadioGraphics*, 745-764.

Collingridge, P. (2011, July 25). *Specific attraction: springs*. Retrieved from Petercollingridge.co.uk: http://www.petercollingridge.co.uk/pygame-physics-simulation/specific-attraction-springs

Crawfis, R., Xue, D., & Zhang, C. (2005). *8 - Volume Rendering Using Splatting In Visualisation Handbook.* (C. D. Hansen, & C. R. Johnson, Eds.) Burlington: Butterworth-Heinemann.

Cripe, B. E., & Gaskins, T. A. (1998). The DirectModel Toolkit: Meeting the 3D Graphics Needs of Technical Applications. *HEWLETT PACKARD JOURNAL 49*, 19-27.

Cui, J., Chow, Y. W., & Zhang, M. (2011). A Voxel-based Octree Construction Approach for Procedural. *IJCSNS International Journal of Computer Science and Network Security*, 160-168.

Dewaele, G., & Cani, M.-P. (2004). Interactive global and local deformations for virtual clay. *Computer Models, 66*(6), 352-369.

Echterhoff, J. (2014, April 29). *On the future of Web publishing in Unity*. Retrieved from Unity3D: http://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/

Flanagan, D. (2011). *JavaScript : The definitive guide.* 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly.

Forshaw, J., Stone, P., & Jordon, M. (2011, June 16). *WEBGL – MORE WEBGL SECURITY FLAWS.* Retrieved from Context- Information Security: http://www.contextis.com/research/blog/webgl-more-webgl-security-flaws/

Galyean, T. A., & Hughes, J. F. (1991). Sculpting: An Interactive Volumetric Modeling Technique. *Computer Graphics, 25*(4), 267-274.

Haines, E., & Akenine-Möller, T. (2002). *Real-Time Rendering.* Natick, MA, USA: AK Peters / CRC Press.

Hess, R. (2010). *Blender Foundations: The Essential Guide to Learning Blender 2.6.* Burlington, MA: Elsevier.

Hughes-Croucher, T., & Wilson, M. (2012). *Node : Up and Running.* O'Reilly.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 90-95.

Kaufman, A., & Mueller, K. (2003). Volume Visualization and Volume Graphics. *Kaufman, A., & Mueller, K. (2003). Volume visualization and volume graphics. Stony Brook University, Stony Brook, 10-15.*, 2.

Kaufman, A., Cohen, D., & Yagel, R. (1993). Volume graphics. *Computer , vol.26, no.7*, 51-64.

Khronos. (2013, October 15). *OpenGL ES 2.0 for the Web*. Retrieved from Khronos Group: http://www.khronos.org/webgl/

Khronos Group. (n.d.). *WebGL/Security.* Retrieved from Khronos Group: http://www.khronos.org/webgl/security/

Koten, J. (2013, June 10). *A Revolution in the Making*. Retrieved from The Wall Street Journal: http://online.wsj.com/news/articles/SB10001424127887324063304578522812684722382

Kurachi, N. (2011). *The magic of computer graphics.* Boca Raton, FL: CRC Press.

Lacroute, P. G. (1995). *Fast Volume Rendering Using A Shear-Warp Factorization Of The Viewing Transformation.* Stanford, CA: Stanford University.

Laine, S., & Karras, T. (2010). *Efficient Sparse Voxel Octrees -- Analysis, Extensions, and Implementation.* NVIDIA Corporation.

Lorensen, W. E., & Cline, H. E. (1987, July). Marching Cubes: A hige resolution 3D surface construction algorithm. *Computer Graphics, 21*(4), 163-169.

Magnusson, M., Lenz, R., & Per-Erik, D. (1991). Evaluation of methods for shaded surface display of CT volumes. *Computerized medical imaging and graphics 15.4*, 247-256.

Matsuda, K., & Lea, R. (2013). *WebGL Programming Guide - Interactive 3D graphics programming with WebGL.* New Jersey 07458: Pearson Education.

Meagher, D. (1982). Geometric modelling using octree encoding. *Computer Graphics and Image Processing, 19*(2), 129-147.

Microsoft. (2014, April). *TypeScript - Language Specification V1.0.* Retrieved from TypeScript Lang: http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf

Mozilla Developer Network. (2014, March 28). *HTTP access control (CORS)*. Retrieved from MDN Mozilla Developer Network: https://developer.mozilla.org/en/docs/HTTP/Access_control_CORS

Mozilla Developer Network. (2014, April 18). *Introduction to Object-Oriented JavaScript*. Retrieved from Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

Mozilla Developer Network. (2014, April 2). *Using web workers*. Retrieved from MDN Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/Guide/Performance/Using_web_workers

MSDN. (2013, November 12). *WebGL*. Retrieved from MSDN: http://msdn.microsoft.com/en-us/library/ie/bg182648(v=vs.85).aspx

Pavone, P., Luccichenti, G., & Cademartiri, F. (2001). From maximum intensity projection to volume rendering. *Seminars in Ultrasound, CT and MRI, Volume 22, Issue 5*, 413-419.

Pawasauskas, J. (1997, February 18). *Volume Visualization With Ray Casting*. Retrieved from Worcester Polytechnic Institute: http://web.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p1/ray-cast.htm

QuakeCon2011. (2012, August 3). *QuakeCon 2012 - John Carmack Keynote*. Retrieved from YouTube: https://www.youtube.com/watch?v=wt-iVFxgFWk

Rousset, D. (N.D). *Introduction to HTML5 Web Workers: The JavaScript Multi-threading Approach*. Retrieved from MSDN Microsoft Developer Network: http://msdn.microsoft.com/en-us/hh549259.aspx

Salama, C. R. (2006). Real-Time Volume Graphics: Introduction. *Eurographics 2006 Tutorial Notes T7* (pp. 1-22). The Eurographics Association.

Sato PhD, Y., Shiraga MD, N., Nakajima MD PhD, S., Tamura PhD, S., & Kikinis MD, R. (1998, November/December). Local Maximum Intensity Projection (LMIP): A new rendering method for vascular visualisation. *Journal of Computer Assisted Tomography, 22*(6), 912-917.

Schroeder, W. J., & Martin, K. M. (2005). *Overview of Visualisation in The Visualization Handbook.* (C. D. Hansen, & C. R. Johnson, Eds.) Burlingtion: Butterworth-Heinemann.

Sellers, G., Wright Jr, R. S., & Haemael, N. (2013). *OpenGL SuperBible (6th edition).* New Jersey: Addision Wesley.

Siegel, M. J. (2008). *Pediatric Body CT.* Philadelphia: Lippincott Williams & Wilkins, 2008.

Spiess, P. (2010, February 23). *Better know an algorithm 1: Marching Squares*. Retrieved from Phill Spiess (Blog): http://devblog.phillipspiess.com/2010/02/23/better-know-an-algorithm-1-marching-squares/

Stemkoski, L. (2013, August 23). *Marching Cubes*. Retrieved from stemkoski.github.io: http://stemkoski.github.io/Three.js/Marching-Cubes.html

Stytz, M. R., Frieder, G., & and Frieder, O. (1991). *Three--dimensional medical imaging: Algorithms and computer systems.* L.C. Smith College of Engineering and Computer Science - Former Departments, Centers, Institutes and Projects. Paper 6.

User: Jonathan @ stackoverflow.com. (2010, November 2). *procedurally generate a sphere mesh*. Retrieved from http://stackoverflow.com/: http://stackoverflow.com/a/4082020

Wang, S. W., & Kaufman, A. E. (1995). Volume Sculpting. *Proceedings of the 1995 Symposium on Interactive 3D Graphics* (pp. 151-ff). Monterey, California, USA: ACM.

wiki.blender.org. (2013, July 8). *Sculpt Mode*. Retrieved from wiki.blender.org: http://wiki.blender.org/index.php/Doc:2.6/Manual/Modeling/Meshes/Editing/Sculpt_Mode

WikiBooks.org. (2013, September 24). *Blender 3D: Noob to Pro/Box Modeling*. Retrieved from WikiBooks.org: Blender Noob to Pro: http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Box_Modeling

Wilhelms, J., & Van Gelder, A. (1992). Octrees for faster isosurface generation. *ACM Transactions on Graphics, 11*(3), 201-207.

Wunsche, B. (1999, April). The visualisation of 3d stress and strain tensor fields. *Proceedings of the 3rd New Zealand Computer Science Research Student Conference*, 109-116.