Durham E-Theses

Stereoscopic Sketchpad: 3D Digital Ink

CORLETT, SIMON,CHRISTIAN

How to cite:

CORLETT, SIMON,CHRISTIAN (2011) *Stereoscopic Sketchpad: 3D Digital Ink*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/3271/

Stereoscopic Sketchpad:
3D Digital Ink

Simon Corlett

A Thesis presented for the degree of

Master of Science

Durham
University

Department of Computer Sciences
University of Durham
England

September 2011

Stereoscopic Sketchpad:
Digital Ink

Simon Corlett

Submitted for the degree of Master of Science

September 2011

Abstract

Context - This project looked at the development of a stereoscopic 3D environment in which a user is able to draw freely in all three dimensions. The main focus was on the storage and manipulation of the 'digital ink' with which the user draws. For a drawing and sketching package to be effective it must not only have an easy to use user interface, it must be able to handle all input data quickly and efficiently so that the user is able to focus fully on their drawing.

Background - When it comes to sketching in three dimensions the majority of applications currently available rely on vector based drawing methods. This is primarily because the applications are designed to take a users two dimensional input and transform this into a three dimensional model. Having the sketch represented as vectors makes it simpler for the program to act upon its geometry and thus convert it to a model. There are a number of methods to achieve this aim including Gesture Based Modelling, Reconstruction and Blobby Inflation. Other vector based applications focus on the creation of curves allowing the user to draw within or on existing 3D models. They also allow the user to create wire frame type models. These stroke based applications bring the user closer to traditional sketching rather than the more structured modelling methods detailed [1].

While at present the field is inundated with vector based applications mainly focused upon sketch-based modelling there are significantly less voxel based applications. The majority of these applications focus on the deformation and sculpting of voxmaps, almost the opposite of drawing and sketching, and the creation of three dimensional voxmaps from standard two dimensional pixmaps. How to actually sketch freely within a scene represented by a voxmap has rarely been explored. This comes as a surprise when so many of the standard 2D drawing programs in use today are pixel based.

Method - As part of this project a simple three dimensional drawing program was designed and implemented using C and C++. This tool is known as Sketch3D and was created using a Model View Controller (MVC) architecture. Due to the modular nature of Sketch3Ds system architecture it is possible to plug a range of different data structures into the program to represent the ink in a variety of ways. A series of data structures have been implemented and were tested for efficiency. These structures were a simple list, a 3D array, and an octree. They have been tested for: the time it takes to insert or remove points from the structure; how easy it is to manipulate points once they are stored; and also how the number of points stored effects the draw and rendering times.

One of the key issues brought up by this project was devising a means by which a user is able to draw in three dimensions while using only two dimensional input devices. The method settled upon and implemented involves using the mouse or a digital pen to sketch as one would in a standard 2D drawing package but also linking the up and down keyboard keys to the current depth. This allows the user to move in and out of the scene as they draw. A couple of user interface tools were also developed to assist the user. A 3D cursor was implemented and also a toggle, which when on, highlights all of the points intersecting the depth plane on which the cursor currently resides. These tools allow the user to see exactly where they are drawing in relation to previously drawn lines.

Results - The tests conducted on the data structures clearly revealed that the octree was the most effective data structure. While not the most efficient in every area, it manages to avoid the major pitfalls of the other structures. The list was extremely quick to render and draw to the screen but suffered severely when it comes to finding and manipulating points already stored. In contrast the three dimensional array was able to erase or manipulate points effectively while the draw time rendered the structure effectively useless, taking huge amounts of time to draw each frame.

The focus of this research was on how a 3D sketching package would go about storing and accessing the digital ink. This is just a basis for further research in this area and many issues touched upon in this paper will require a more in depth analysis. The primary area of this future research would be the creation of an effective user interface and the introduction of regular sketching package features such as the saving and loading of images.

September 25, 2011

Declaration

The work in this thesis is based on research carried out at the University of Durham, the Department of Computer Sciences, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

The research undertaken as part of this project would not have been possible without the help of a few key people. First and foremost I would like to thank my supervisor, Dr Nick Holliman, for his assistance and support over the past year. His advice throughout has been invaluable. I would also like to thank Paul Gorley for his help formatting and compiling the template for this thesis, and Geng Sun for the use of his code in setting up a stereoscopic environment. Finally, thanks must go to my family for their unwavering support when the going got tough.

Contents

List of Figures

List of Tables

Glossary

6-DOF 6 Degrees of Freedom – the ability to move along three perpendicular axes combined with rotation about three perpendicular axes. 27, 30, 31

CAD Computer Aided Design – the use of computer technology for the process of design and design-documentation. 3, 18

digital ink a digital representation of real world ink drawn using a piece of sketching software. 2, 31–35, 39, 42, 52, 66–68

disparity the difference in coordinates of similar features within two stereo images. 4, 41

geometric primitive a simple geometric shape such as a cube, cylinder, sphere, cone or pyramid. 5

MVC Model, View, Controller – a modular software architecture which isolates the functional algorithms from the input and presentation permitting independent development, testing and maintenance of each. 30, 39, 67

NURBS Non-uniform rational B-spline – a mathematical model commonly used in computer graphics for generating and representing curves and surfaces. 23, 67

octant the leaf of an octree in which a list of vertices are stored, when the list reaches capacity the octree is forced to subdivide. 8, 9, 45, 46, 51–57, 59, 64, 65

octree a tree data structure in which each internal node has exactly eight children. 8, 9, 35, 42, 43, 45, 46, 50–57, 59, 61–68

OpenGL a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics.. 35, 39, 40, 43, 44, 46–49, 61, 67

surface an object composed of flat shapes, usually triangles, defined by vertices. 5, 6, 10, 18–24, 26–29, 31, 67

vertex a data structure which describes a point in two or three dimensional space. 16, 17, 30, 39, 43

volume a 3D data set made up of voxels in the form of a regular volumetric grid. 5, 6, 9–11, 27

voxel a volume element, representing a value on a regular grid in three dimensional space. 5–11, 23, 24, 27, 29, 34, 52, 53, 68

ZDP Zero-Disparity Plane – the plane in at which the two images within a stereo scene have a disparity equal to zero. 49

Chapter 1

Introduction

From early cave drawings in prehistoric times to the sketchbooks of Leonardo da Vinci and Edgar Degas, drawing and sketching has always been an inherent part of human society. The ability to record impressions and ideas quickly in an understandable and adaptable form is invaluable to the creative process.

Over time the tools and methods used to sketch have changed. In prehistoric times ochre was used to paint scenes of hunting and daily life upon the walls of the caves occupied as homes. By the 12th and 13th centuries AD, monks throughout Europe were using lead styli to draw lines for their writings and for the outlines of their illuminations while preparing manuscripts upon vellum or parchment. Soon lead was replaced by silver to make drawings and underdrawings as it did not blunt as easily. Then, when paper became generally available from the 14th century onwards, artists' drawings, both preparatory studies and finished works, became increasingly common [54].

By the 20th century drawing was still as important as ever. From architects plans to children's comics sketches could be found in every aspect of modern life. With the invention of the computer and the dawning of the digital age it was only a matter of time before these two fields to met. In 1963 Ivan Sutherland wrote a revolutionary program known as Sketchpad as part of his PhD thesis at the University of Cambridge. In 1988 he won the Turing Award as it was felt that Sketchpad helped change the way people interact with computers. Sketchpad demonstrated that computer graphics could be used for both artistic and technical purposes in addition to showing a novel method of human-computer interaction. It used the recently invented light pen to plot 2D points onto a computer.

The Sketchpad system stores information about drawings in two separate forms. One is a table of display spot coordinates designed to make display as rapid as possible; the other

is a file designed to contain the topology of the drawing. The topological file is set up in a specially designed ring structure. The ring structure was designed to permit rearrangement of the data storage structure for editing pictures with a minimum of file searching, and to permit rapid constraint satisfaction and display file generation. The ring structure was not intended to pack the required information into the smallest possible storage space. It was felt that faster running programs could be written in less time by including some redundancy in the ring structure. This was considered more important than the ability to store huge drawings. The particular form of the ring structure chosen has led to some of the most interesting features of the system simply because the changes required to keep the ring structure consistent led to useful facilities such as recursive merging [78].

Since Sutherland, drawing packages have become a regular feature on computers world-wide. A wide variety of tools allow users to create digital sketches or more precise drawings. Through the advances in computing these drawings can be stored for later, edited with ease and shared with friends and colleagues world wide at the click of a button.

The aim of this research is to attempt to emulate Sutherland and take drawing to the next level. It will look at the challenges behind creating a drawing package which allows a user to draw freely in three dimensions within a stereoscopic 3D environment. As with Sutherland's work there will be a focus upon how to store and represent the points drawn, in effect the digital ink. Sutherland's method of storing the data using a ring structure is a very good one as it means that when the program wants to recall a point it does not have to search through the whole list from the top down. However, as previously stated, the ring structure used was designed such that it was quick to put to use rather than have the ability to store huge drawings. In a program which is working with a three dimensional environment the data structure will need to be able to store a larger amount of data as the points will require three coordinates along with any information relating to attributes such as size and colour to be stored instead of two. It is also likely that the available drawing area will be larger than it would have been in two dimensions.

The paper will begin by identifying all previous work relating to the project. A drawing tool which allows a user to sketch freely within three dimensions within a stereoscopic 3D environment will then be designed and implemented. This software tool should be able to incorporate various means of storing the digital ink. Each of these solutions will be subjected to rigorous testing to determine which is most suitable for the task.

Chapter 2

Related Work

This chapter will look at a range of previous work directly related to the project. This will help to put the project into context and also to provide a clearer idea of how a solution could be produced. The main focus will be on the advantages of drawing in three rather than two dimensions and the various methods which currently exist to facilitate this. A summary of these methods can be seen in figure 2.6. First we review representation and rendering schemes for 3D data.

2.1   3D Verses 2D

The ability to make rough sketches of plans and ideas has always been vital to designers and artists. It provides them with the ability to express their ideas freely and record any spontaneous thoughts. The notebooks of Leonardo da Vinci demonstrate effectively how important sketching is to an artist and how much can be expressed through such an accessible form [10]. Two dimensional sketches do however have their drawbacks. It can be very difficult for a three dimensional object to be represented effectively in just two dimensions and it is easy for the human visual system to misinterpret what has been drawn. With this knowledge Bimber et al. make the assertion that 3D sketches hold an advantage over their 2D counterparts in that they have a higher information content. The fact that they provide depth information allows a viewer to fully interpret and reconstruct the outlined objects [7].

Modern day Computer Aided Design (CAD) systems focus purely on the creation of three dimensional models and almost completely skip the initial expressive and creative stage. In 1993 Baker pointed out that "it is difficult to find any software that supports the

truly creative areas of design activity, particularly when most software systems lack the ability to contain imprecise or 'fuzzy' information, which is often the hallmark of creative design work, certainly in it's early stages" [4]. It was for this very reason that Poletti laid out the need for a 3D sketching tool "to observe and create directly in 3D space, capturing the essence of the sketch and receiving instantaneous three dimensional feedback." [66]

2.2 Stereoscopic 3D Displays

Humans are able to see in stereo 3D by viewing two separate images, one with each eye. As the eyes are separated by about 50-75mm each eye sees a objects from a slightly different angle. This comes heavily into play when creating a computer display which appears three dimensional to a user. This means that if a viewer is presented with two overlapping 2D images which are slightly offset from each other they should be able to see the image in stereo 3D. The distance between coordinates of similar features within two stereo images is called the disparity between the two. The trick to creating a stereoscopic scene lies in drawing each of the viewer's eyes to the correct 2D image. On standard stereoscopic displays and within cinemas, this is achieved by polarising each image in a different direction. Specially designed glasses then filter each of these images to the correct eye. Alternatively, autostereoscopic displays are designed to guide the viewer's eyes without the need for specially designed glasses or any other devices.

There are three different types of autostereo displays: two-view displays; head-tracked displays (usually two-view); and multi-view displays with three or more views. Two view displays generate the two views for the left and right eyes in two viewing windows in space. These are primarily visible from a central viewing position and the user may have up to 20 or 30 mm of movement around the central viewing position before they lose the 3D effect [35]. This is shown below in figure 2.1.

The display divides the horizontal resolution of the underlying display device into two sets. One of the two visible images consists of every second column of pixels; the second image consists of the other columns. The two images are captured or generated so that one is appropriate for each of the viewer's eyes. When standing at the ideal distance and in the correct position, the viewer will perceive a stereoscopic image.

However, there are numerous practical problems: There is a 50 percent chance the viewer will be in the wrong position and see an incorrect, pseudoscopic image; the viewer must stay fairly still to remain in the correct viewing position; and moving much forward

September 25, 2011
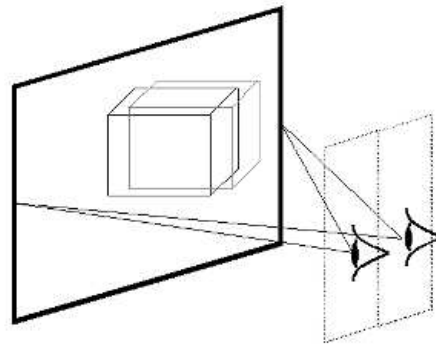
Figure 2.1: An image viewed upon a two view stereoscopic display [35]

of or back from the ideal distance greatly reduces the chance of seeing a correct image. These limitations necessitate using another auto-stereoscopic solution: either introducing head-tracking or increasing the number of views [22].

2.3   Volume Vs. Surface Graphics

In two dimensions the representation of digitally produced drawings can be split into two categories: vector and raster graphics. While vector graphics is the use of geometrical primitives to represent images, raster graphics is the representation of images as an array of pixels. This concept is extended when it comes to representing digital images in 3D and drawings are rendered using either surface graphics or volume graphics.

Like vector graphics, surface graphics represents the scene as a set of geometric primitives kept in a display list. In surface graphics, these primitives are transformed, mapped to screen coordinates, and converted by scan-conversion algorithms into a discrete set of pixels. Any change to the scene, viewing parameters, or shading parameters requires the image generation system to repeat this process. Like vector graphics, which does not support painting the interior of 2D objects, surface graphics generates merely the surfaces of 3D objects and does not support the representation of their interior.

Instead of a list of geometric objects maintained by surface graphics, volume graphics employs a 3D volume buffer as a medium for the representation and manipulation of 3D scenes. All objects are converted into one uniform meta-object, the voxel. Each voxel is atomic and represents the information about at most one object that resides in that voxel. [45]

The most important advantage that volume graphics provide is that of insensitivity to

environment and object complexity. With surface graphics, the number and elaborateness of the objects in the scene affects the time taken to render the scene since transformations of the objects on the display list are affected by their size and complexity. In volume graphics, the scene is already converted into a finite-size volume buffer.

Another advantage is the fact that viewpoint of the observer is not pertinent. In surface graphics, the geometrical data must be converted into surfaces after every change in viewpoint. In volume graphics, all viewpoints are pre-computed by the very nature of the stored data. This means that any viewpoint-independent characteristic of the voxel, for example it's density, can be stored along with it in the volume buffer. This cannot be done with surface graphics since the surfaces are polygonised and are not represented by the actual voxels themselves. [61]

The same appeal that drove the evolution of the computer graphics world from vector graphics to raster graphics, once the memory and processing power became available, is driving a variety of applications from a surface-based approach to a volume-based approach. Naturally, this trend first appeared in applications involving sampled or computed 3D data, such as 3D medical imaging and scientific visualisation, in which the datasets are in volumetric form. These diverse empirical applications of volume visualisation still provide a major driving force for advances in volume graphics.

2.4 Representation Techniques

One of the big challenges when dealing with the volumetric data discussed in the previous section is how to represent it effectively. This is due to the fact that if you were to store and then render every individual voxel represented within the scene this would be extremely slow for scenes of any considerable size. However within a sketching program the scene will never be completely filled, infact in the case of rough sketches only a fraction of the available points may contain a voxel. This means that the volumetric data which needs to be stored is only a small percentage of the entire scene. The other side of this argument is that enough points need to be stored to provide an image of an acceptable quality. As can be seen in figure 2.2 below when a limited number of points are displayed the visual appeal of the image is noticeably worse.

Another point to be considered when representing the data for a skething program is that the volumetric data will be constantly added to and edited as a user draws. Therefore a solution needs to be found which can not only store the data efficiently but also allows

September 25, 2011
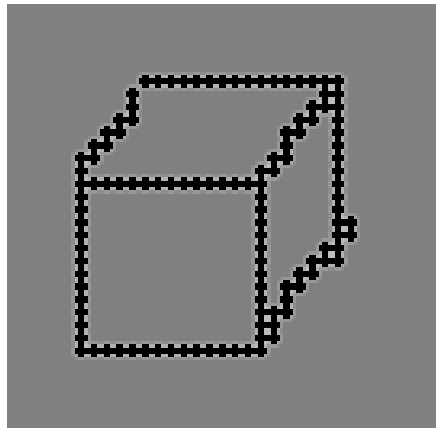
(a) A cube drawn upon an array with a length and a width of a quarter the number of pixels upon the display.
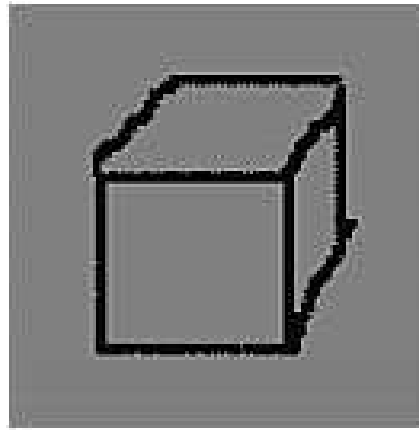
(b) A cube drawn upon an array which has dimensions equal to the number of pixels upon the display.

Figure 2.2: Subjective quality of the three dimensional array.

it be accessed, altered and rendered quickly. This section discusses a number of existing techniques for representing such data.

2.4.1 Exhaustive Enumeration (Cell-by-cell Encoding)

Exhaustive Enumeration, also known as cell-by-cell encoding, is the simplest form of compression for raster based data structures. However it is far more storage-intensive than other structures. Every single voxel is assigned a value, hence there is absolutely no compression when many like values are encountered. The result of exhaustive enumeration on a simple 2D raster structure can be seen in part a) of figure 2.3 [25].

2.4.2 Run-length Encoding

If a raster based data structure contains groups of cells with identical values, run length encoding can compress storage effectively. Instead of storing each individual cell, each component stores a value and a the number of cells in row with the same value. This can be seen on a two dimensional structure in part b) of figure 2.3. If there is only one cell the storage doubles, but for three or more cells it is possible there is a reduction. The longer and more frequent the consecutive values are, the greater the compression that will be achieved [25]. This technique is particularly useful for encoding monochrome images or binary images [14].

In 1996 Apple released a fast and simple form of lossless compression known as Pack-

Bits. A PackBits data stream consists of packets with a one-byte header followed by the data itself. The header is a signed byte and the data can be signed, unsigned, or packed in a form such as MacPaint pixels. PackBits packs the data only when there are three or more consecutive bytes with the same data; otherwise it just copies the data byte for byte.
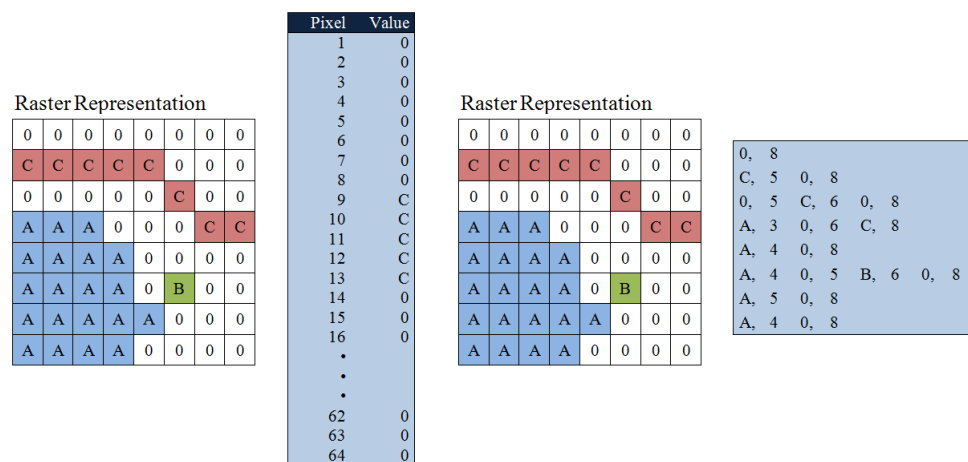
(a) Exhaustive enumeration.   (b) Run length encodinge.

Figure 2.3: The differing results of exhaustive enumeration and run length encoding on the same raster image [25]

2.4.3   Space Partitioning Data Structures

Another means of compressing rasterised data is to partition the structure in which it is stored into smaller sections containing less variation. Space partitioning systems are often hierarchical, meaning that a space (or a region of space) is divided into several sections, and then the same space partitioning system is recursively applied to each newly created section. The sections can be organised into a tree, called a space partitioning tree [70].

A common form of space partitioning tree is the octree, discussed in the PhD thesis of Hunter in 1978 [38] and a couple of years later in more detail by Meagher [57]. octrees are trees in which every node has a maximum of eight children and can be used to partition cuboidal regions. Each octree is subdivided into eight equally sized regions known as octants. If an octant contains only voxels of a certain value it will be left alone, however, if not it will be once again subdivided into eight more octants. This process continues until every region contains only like values. To reduce storage costs, only the complete list of leaf nodes is stored, i.e., as a linear octree. To use a linear representation, a locational code

is needed to identify the octants. A locational code is a code that contains information about the position and level of the octant in the tree [77]. A commonly used method of this is the Morton encoding [11].

A development of space partitioning is k-d trees examined in detail by Bentley in 1975 [5] as shown in figure 2.4. Samet tells us they are preferable when acting on an array of point data in more than three dimensions [71]. These work in a similar fashion to the octree in that the data structure is partitioned into smaller segments. However, in this case, instead of dividing the structure into equally sized sections the underlying space is decomposed into two halves as the points are inserted. The partition positions depend on the location of the points and the partition axes are cycled in the order x, y, z, x, y, z, etc [70].
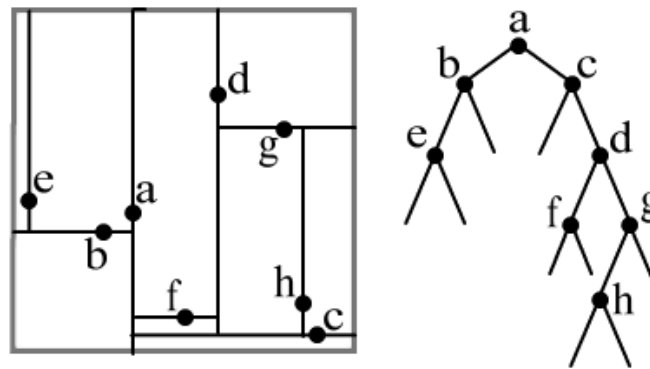
Figure 2.4: Bentley's k-d tree [5]

2.5 Volume Rendering

Once the geometrical data is represented efficiently the next step is to display it to a user. This section looks at various methods of drawing images of the data contained within the data structures. Although this is largely outside the scope of this project as the goal is to support sketching in a stereoscopic 3D environment, not full volume rendering, I feel it's important that these methods be discussed to aid future work in this direction.

2.5.1 Volume Ray Casting

In its basic form volume ray casting can be broken down into just a few simple steps as shown in figure 2.5. First, for each individual voxel of the final image, a ray of sight is

cast straight through the volume as shown in part 1. Part 2 then illustrates how samples are then taken at equidistant points along the ray. For each sampling point, the gradient is computed. These represent the orientation of local surfaces within the volume. The samples are then coloured and lighted, according to their surface orientation and the source of light in the scene as shown in part 3. After all sampling points have been shaded, they are composited with the other samples along the ray, resulting in the final colour value for the voxel that is currently being processed. As part 4 shows, the composition is derived directly from the rendering equation and is a merge of all the sampled colours. Computation starts with the sample farthest from the viewer and ends with the one nearest to them. This work flow direction ensures that masked parts of the volume do not affect the resulting voxel [49]. Volume ray casting provides a final image of a very high quality. However the trade off for this is the amount of time a single image takes to render. As every voxel is rendered separately it can be very slow although as modern day graphics cards are improved this is getting quicker as they are able to cope with parallel processes more effectively.
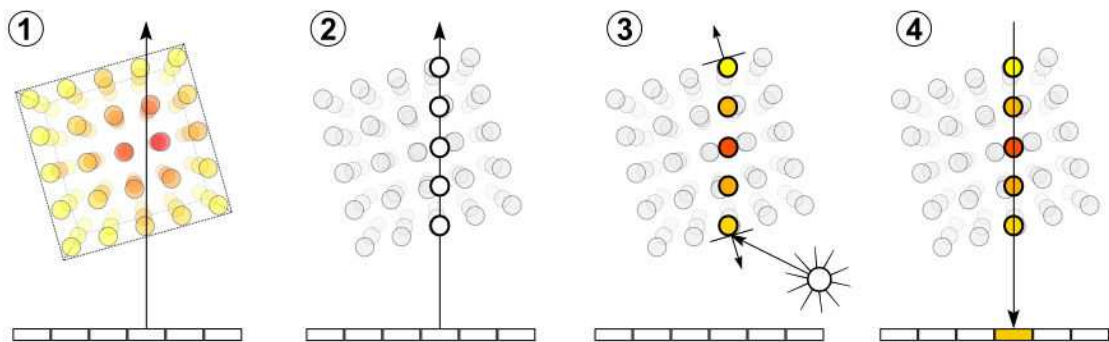
Figure 2.5: Volume Ray Casting [80]

2.5.2 Volume Ray Tracing

Ray casting can be improved upon in a number of ways, most notably by using ray tracing. Ray tracing entails ray casting but with the additional tracing of secondary rays for shadows and refractions. In the paper "Interactive Isosurface Ray Tracing of Large Octree Volumes" [47] Aaron Knoll et al. present a straightforward technique for ray tracing isosurfaces of large compressed structured volumes. The data is first converted into a lossless-compression octree representation that occupies a fraction of the original memory footprint. An isosurface is then dynamically rendered by tracing rays through a

min/max hierachy inside interior octree nodes, using an efficient ray intersection routine for piecewise trilinear interpolants in nonempty cells. By embedding the acceleration tree and scalar data in a single structure and employing optimised octree hash schemes, they achieved competitive frame rates on common multicore architectures, and are able to handle large and time-variant data in a completely in-core CPU algorithm [48].

2.5.3 Splatting

Splatting was developed to improve the speed of calculation of volume rendering techniques like ray casting, at the price of less accurate rendering. It is quite a complex technique but in simple terms it is similar to ray casting. Rays are cast through the volume and samples taken. This time however rather than just sampling the colour of the voxels which the ray has passed through, those around are also used but to a lesser extent as they get further away from the ray. These samples are then composited in the same way as in ray casting and projected onto the final image in what is called a gaussian splat. Projecting these splats provides a uniform screen image for homogeneous object regions, but leads to a blurry appearance of object edges [60].

2.5.4 Shear Warp

The shear-warp algorithm [64] is a very fast approach for evaluating the volume rendering integral. In contrast to ray-casting, no rays are cast back into the volume, but the volume itself is projected slice by slice onto the image plane. This projection uses bi-linear interpolation within two-dimensional slices, instead of the tri-linear interpolation used by ray-casting [24].

Using shear warp, the viewing transformation is transformed such that the nearest face of the volume becomes axis aligned with an off-screen image buffer with a fixed scale of voxels to pixels. The volume is then rendered into this buffer using the far more favourable memory alignment and fixed scaling and blending factors. Once all slices of the volume have been rendered, the buffer is then warped into the desired orientation and scaled in the displayed image [50]. This technique is relatively fast in software at the cost of less accurate sampling and potentially worse image quality compared to ray casting. There is memory overhead for storing multiple copies of the volume, for the ability to have near axis aligned volumes. This overhead can be reduced by using the compression techniques previously discussed.
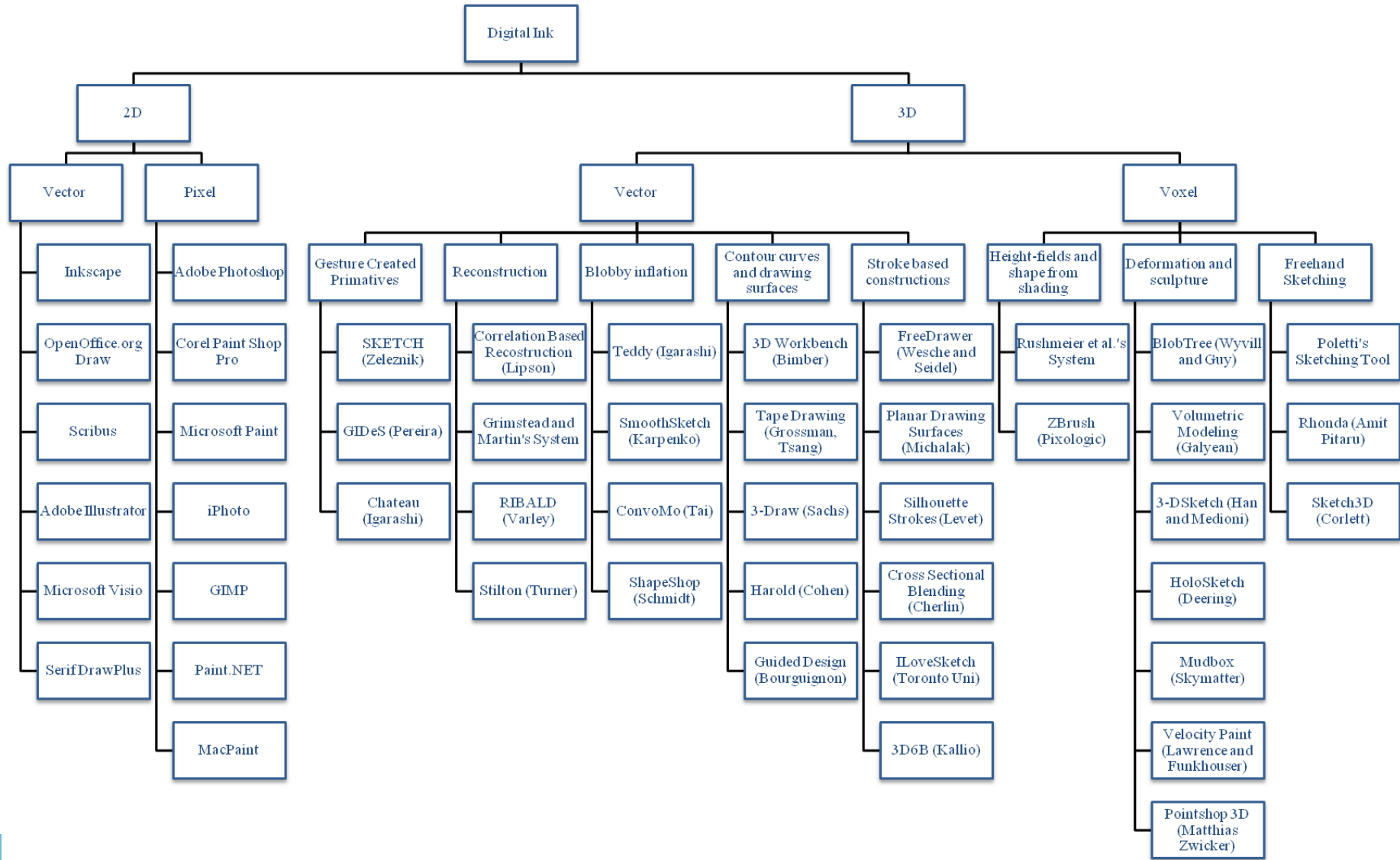
Figure 2.6: A taxonomy showing various applications for sketching in 2D and 3D

2.6   3D Sketching Methods

The various methods of sketching in two and three dimensions and the applications available for these can be broken down into various subsets. The taxonomy represented by figure 2.6 above shows a large selection of the applications within each subset. This section discusses each of the methods shown on the right hand side of the taxonomy. These are the methods which are currently available for sketching in 3D and this section will point out their strengths as well as exposing any flaws.

2.6.1   Vector Based Methods

When it comes to sketching in three dimensions the majority of applications currently available rely on vector based drawing methods. This is primarily because the applications are designed to take a users two dimensional input and transform this into a three dimensional model. Having the sketch represented as vectors makes it simpler for the program to act upon it's geometry and thus convert it to a model. There are a number of methods to achieve this aim including Gesture Based Modelling, Reconstruction and Blobby Inflation.

Other vector based applications focus on the creation of curves allowing the user to draw within or on existing 3D models. They also allow the user to create wire frame type models. These stroke based applications bring the user closer to traditional sketching rather than the more structured modelling methods detailed.

2.6.1.1   Gesture Based Modelling

An early approach to creating three dimensional drawings from sketched input was an interface which was purely based upon a series of predefined gestures. The aim of this type of application was to create fully fledged 3D models from a two dimensional input. The users input is interpreted as symbolic instructions which relate to a combination of 3D shapes defined by the software. This severely limits the drawings, or more accurately the models, which the user is able to create as they are limited to the primitives defined by the application.

The first example of this gesture based modelling is provided by the SKETCH application designed by Zeleznik, Herndon and Hughes [89]. Sequences of strokes are combined into gestures which define both the shape to be created, and details of its form. So, for

September 25, 2011

example, the user might draw three perpendicular lines, which would then be interpreted to define a box with sides matching the lengths of each stroke. Similar methods allow the user to create a variety of different shapes. Once these shapes have been created the user is able to resize and transform them as they please.

The original SKETCH system provided only a small set of basic shapes, allowing the gesture set to remain fairly small, and the gestures themselves iconic of the shapes they represent. A number of other researchers have attempted to build on the success of the SKETCH system, adding additional functionality. However as the number of gestures increases, the ability to recognise and differentiate them becomes more difficult.

A popular solution to this issue has been the use of an interactive contextual disambiguation system [23], or 'expectation list' [64]. A good example of such a system is the GIDeS modelling prototype developed by Pereira et al. [65]. GIDeS uses expectation lists to negotiate the meaning of a user's input interactively as part of the modelling process. When the user enters a gesture, a small contextual window as shown in figure 2.7 appears presenting icons describing possible interpretations of the gesture. The user can then select the appropriate interpretation, or correct their input if it has been misinterpreted. By allowing the user to help differentiate the meanings of ambiguous gestures the interface can accommodate a greater number of similar gestures that better approximate traditional drawing.
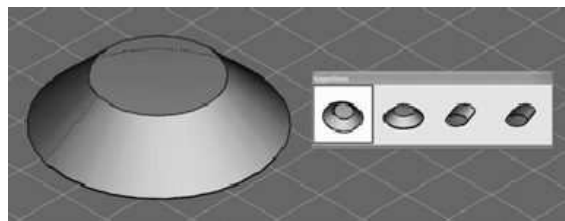
Figure 2.7: 3D Primitive Expectation List in the GIDeS system [65]

One drawback to expectation lists is that they take the user away from an environment in which they can sketch freely and turn the process into a call and response pattern of presenting a gesture and then approving a recognition. 'Suggestive interfaces' or 'mediated gesture system' [39] provide a possible solution. In 2007 Igarashi and Hughes designed an application called Chateau. As with expectation lists the user is presented with a number of suggestions after each operation. However, the user does not have to make an explicit selection after providing input, they are free to continue entering additional strokes. This means that the user can shift the system's focus at any time, allowing them to come

at a single operation from many different paths and incorporate components created at
different times.

2.6.1.2 Reconstruction

Interpreting a user's drawings as gestures provides a quick and easy way to generate 3D
simple content, but it also distances the user's drawn input from the resulting geometry.
In response a number of researchers have attempted to develop systems that interpret the
user's drawings more directly.

Although a 2D projection has no real depth, when we look at a drawing or picture
our brains can reconstruct the 3D shape of each object and describe the relative distances
between them. This reconstruction process is so fluid and natural that our brains make it
look simple, but the underlying task is surprisingly complex. Only by drawing upon clues
in the image, and applying basic knowledge about how the world works, are our brains
able to see past the literal 2D projection and construct a plausible 3D model [52]. Despite
its apparent effectiveness, we are all familiar with simple optical illusions which easily
thwart our perceptive systems. The shapes in figure 2.8 are a case in point. Researchers
are currently attempting to apply many of these same techniques in order to reconstruct
3D scenes from 2D images.
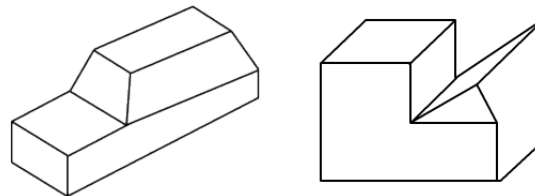
Figure 2.8: Two examples where it is unclear which edges should be parallel in 3D [83]

One method of resolving ambiguity is to apply previous knowledge, a common tech-
nique of machine learning. Lipson and Shpitalni [55] for example describe a modelling
system that can reconstruct a user's 2D wire-frame drawing by comparing the geometry
to models it has seen in the past. The program has stored data derived from 100,000
randomly generated 3D models and their 2D projections.

An alternative method of reconstructing line drawings focuses on optimising a number
of smaller elements in order to gain a global understanding of an image. In 1971 Huffman
and Clowes independently developed a formal method for reconstructing a specific class

of line drawings based on properties of individual lines and their intersections [37] [15]. Today this method is referred to as HuffmanClowes line labelling. The HuffmanClowes system deals with forms made up of only flat faces and containing vertices with no more than three incident edges.

The basis of the system is the fact that, no matter the overall shape of a model, given three incident edges there are only so many ways the edges can meet at a vertex that make physical sense. In turn, each line entering the junction can be assigned one of three labels: convex edges protrude out as in a mountain fold or the edge of a cube; concave edges sink in as in a valley fold; or occluding edges, convex edges along which only one of the two faces adjoining the edge in space is visible in the 2D projection. Huffman and Clowes produced a catalogue of all possible junction labelling for projections of this form.

The technique was first applied to a sketch-based system by Grimstead and Martin in 1995 [30]. In their system, the user provides a two dimensional line drawing, which is analysed to generate possible labellings. The user then has to select the proper configuration before a system of linear equations determines the z-coordinates of each vertex in the drawing. This basic design was taken a step further by Varley, Martin and Suzuki. [83] in their RIBALD modelling system. Following the basic reconstruction, RIBALD allows the user to use the reconstructed edges of the model as a framework onto which curved edges can be placed to create a more expressive model.

Some have suggested that reconstruction systems could be aided by providing additional contextual information. Turner, Chapman and Penn's Stilton system, for example, allows the user to draw their geometry into an existing 3D model [82]. The reconstruction system then uses clues from the pre-existing geometry such as the ground plane to aid in the labelling process.

Although these systems are effective at reconstructing certain classes of objects, their primary drawback is that a number of complex algorithms are needed to complete the process. Varley et al. discuss such a process in detail in the paper "Can Machines Interpret Line Drawings?" [83]. Systems are generally limited to reconstructing simple shapes with few edges, and even then the processing times can be so long users are unable to interact with them freely. Even under tight constraints, these systems must also deal with many ambiguous cases in which multiple interpretations of a line drawing are equally plausible.

2.6.1.3 Blobby Inflation

The final method we will look at of taking a users 2D input and creating a three dimensional model is known as blobby inflation. While the previously mentioned approaches tend to provide a user with a high degree of control, some researchers have explored how a limited set of modelling capabilities might provide a more satisfying, if less capable interface.

Developed by Igarashi, Matsuoka, and Tanaka, Teddy is a 3D sketch based modelling system designed for children. It is capable of generating 3D models from a user's simple two dimensional drawing [40]. At the centre of the system is an ingenious inflation method that converts the user's input into a 3D shape.

To create a model, the user begins by drawing a simple, 2D outline. The user's stroke is collected as a closed polyline loop, and then analysed by the system to find a central chordal axis or 'spine', a single branching line that passes through the middle of the closed shape. Vertices of the spine are elevated from the plane of the initial stroke based on their distance from the stroke, and used to form a tessellated mesh dome that is mirrored to the other side and sewn together to create a symmetric, watertight model topologically equivalent to a sphere. Figure 2.9 shows examples of input strokes and the corresponding 3D models.

(a) snake	(b) snail	(c) cherry	(d) muscular arm
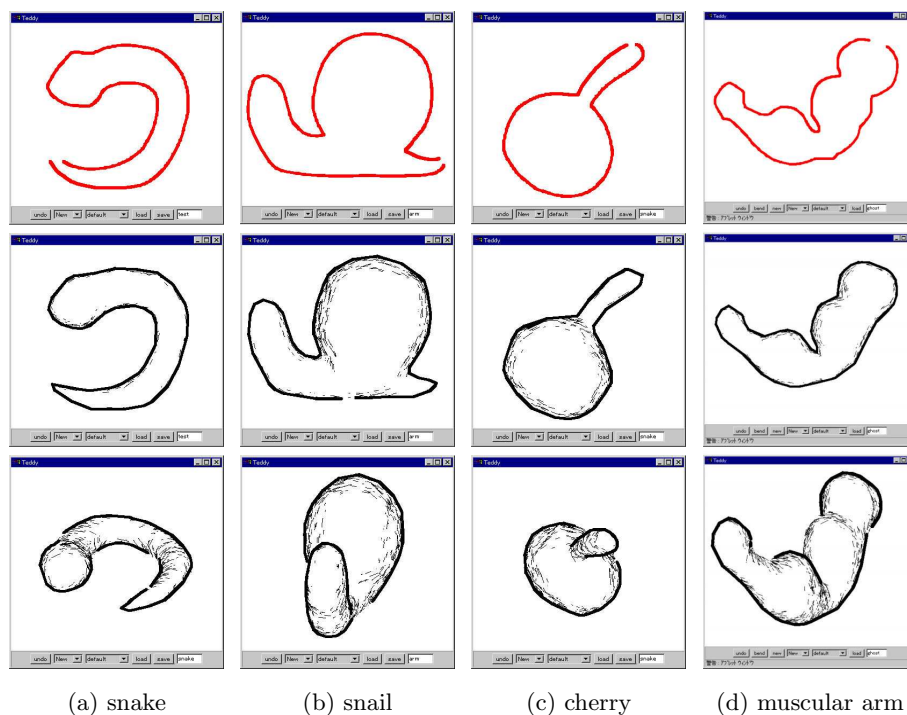
Figure 2.9: Examples of creating models using Teddy (top: input stroke, middle: result of creation, bottom: rotated view). [40]

Although a wide variety of shapes can be created using this method, models are limited to single pieces, and do not contain sharp features like corners or edges. Despite these limitations, the whole process is simple and straightforward, and requires no interpretation or negotiation with the user, meaning that models can be created extremely quickly, and with a minimum of interface complexity.

The simple interface provided by Teddy proved to be inspirational to a number of other researchers. SmoothSketch developed by Karpenko, Hughes and Raskar in 2002 is an inflation-based modeller built around variational implicit surfaces [44]. Owada et al. worked with Igarashi from the Teddy system to develop and application which allows for the creation of objects with complex topologies and internal structures [63]. A similar approach using an implicit-modelling representation called convolution surfaces is presented by Tai, Zhang and Fong's ConvoMo [79]. Here a user adjustable cross section is convolved along the chordal axis to create the model shape, allowing greater flexibility and the creation of 'semi-sharp' rather than blobby features. Finally, ShapeShop, developed by Schmidt et al. attempts to expand inflational modelling to a more fully featured application [73]. An example of blobby inflation using ShapeShop is shown below in figure 2.10.
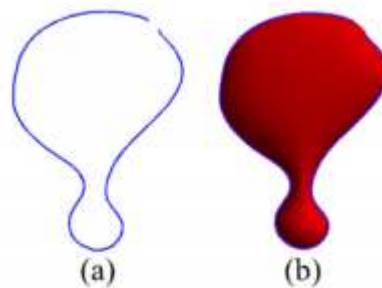
Figure 2.10: ShapeShop converts the 2D sketch shown in (a) into the 3D surface (b) [73]

The inflation based systems provide a prime example of how even simple interfaces can provide effective and even powerful means of creating 3D models quickly and more intuitively. However, the models created always look slightly rough and childish meaning this technique is not necessarily suitable for CAD or more serious modelling applications.

2.6.1.4   Contour Curves and Drawing Surfaces

Where the previous methods focus on created models from the sketched input some research has been directed at how best to use 2D drawing input in a 3D environment. These

vector based methods include both utilising 2D strokes in 3D, and allowing the user to draw fully three dimensional space curves.

When working with traditional 2D input, a common technique has been the use of drawing planes positioned in 3D space. Some projects have taken this approach quite literally. Bimber et al. for example describe an immersive modelling system where in the user stands above a 3D workbench display wearing special glasses [6]. The user is provided with both a stylus, as well as a translucent plexiglass sketchpad, both of which are tracked in 3D. The pad allows the user to sketch input in space, and provides a surface for other 2D inputs like gestures and handwritten notes. Sachs, Roberts and Stoops' 3-draw system takes a similar tact, affixing the model to a three dimensionally tracked physical palette. This allows the user to not only draw on a solid surface, but also to move and rotate the scene by physically manipulating the palette in space [69].

A less physical approach can be found in work by Grossman et al [31]. Automotive design is one industry in which contour and profile curve modelling is used heavily. In order to create characteristic smooth flowing lines automotive designers traditionally employ a unique drawing technique known as tape drawing in which long strips of photographic tape are applied to large vertical work surfaces, creating an image similar to a 2D wire frame projection of a vehicle design. In their paper, Grossman et al. explored how this technique could be extended to creating 3D wire-frame models. In their system, the sensation of tape drawing is simulated by two 3D pointing devices directed against a large screen. As the user draws out virtual strips of tape, the lines and curves they form are projected onto planar work surfaces within the 3D environment. By positioning the planes the user can construct a 3D wire-frame model from these planar curves. Based on Grossman et al.'s interface designs, Tsang et al. developed a similar tape drawing system using a more standard digitising tablet interface [81].

The problem with these methods is that they require very specialised hardware and are therefore not available to standard users, in their 1999 paper Cohen et al. developed a method of drawing 3D space curves from a standard 2D interface [17]. The researcher's approach was to allow the user to draw the curve from a single perspective, and then draw the shadow cast by the curve onto a nearby plane. The system then combines these two strokes to create a single space curve by projecting the first curve onto a surface drawn out from the shadow perpendicular to plane.

The idea of drawing in space has also been explored as a method of altering or anno-

tating existing models or interacting in 3D environments. Perhaps the first example of this approach was inspired by the 1955 classic children's book Harold and the Purple Crayon by Johnson [42]. Taking this imaginative tale as a cue, researchers at Brown University developed Harold, a prototype program designed to facilitate creative exploration and storytelling for young children [16]. In their application, users explore a virtual landscape in which they can draw objects in mid air. As the user draws, the system automatically creates an invisible planar surface parallel to the viewing plane called a billboard, and projects the marks onto its surface. Because the artwork only defines one side of an object, as the user moves around the environment the boards turn to face them from any angle. This can be seen in figure 2.11. As the viewpoint changes the trees turn to face the viewer while the hammock remains linking the two objects.

(a)   (b)
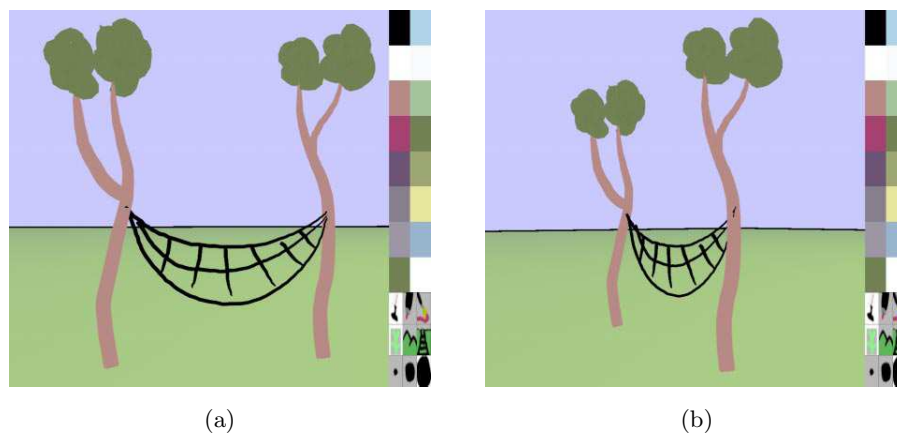
Figure 2.11: Two tree billboards drawn in Harold from the viewpoint in (a); (b) shows the objects from a new viewpoint. [16]

A similar tack was taken by Bourguignon et al. to allow what the authors call 'guided design' [9]. Here users might begin, for example, with an exiting 3D model of a dressmaker's dummy, and draw their clothing designs over the form in three dimensions. To generate models from silhouette strokes, the system converts the user's drawing input into transparent troughs or partial tube shaped 3D surfaces onto which the visible line is projected. The trough is shaped by the curvature of the stroke and the visible portion is located at its apex. As the user's view changes, the visibility of the stroke attenuates, giving the illusion of a 3D surface contour. Because these strokes actually have 3D surfaces associated with them the system can perform limited occlusion with other elements.

Together, these projects demonstrate that it is possible to create and situate curves in three dimensions. They also provide an example of how general 3D space curves can

September 25, 2011

be defined by the same system. However, indications from the researchers' publications suggest that the creation of completely general 3D space curves can be difficult and even frustrating for the user. Although some of this difficulty can be blamed on deficiencies in the interfaces themselves or the limitations of 2D display devices, remarks by the authors suggest that to some degree users simply find it taxing to fully consider the 3D structure of the curves. Grossman et al. for example noted that the majority of modelling time by test subjects using their system was not spent creating curves, but carefully considering what curves to create in order to best define the desired model [32].

2.6.1.5 Stroke Based Constructions

As the previous section mentioned, the creation of completely general 3D space curves can be difficult and even frustrating for users. Therefore a number of programs have been developed with the intuitive creation of curves solely in mind. One example of this sort of system is FreeDrawer [85]. Built around an immersive 3D workbench and two 3D pointers, this system allows a designer to draw and edit networks of spline curves. Loops within this network can then be filled with surfaces to create a model. A more powerful example of this sort of sketch-based curve and surface editing is provided by Michalik et al. [58]. To create a model the user draws strokes with a digitising tablet or mouse into the 3D environment, which are then projected onto planar drawing surfaces. These strokes then enter a constraint solving system that generates a B-spline surface approximating the shape suggested by the curves. Because curves are used as constraints rather than scaffolding, the variety of shapes that can be created by this system is more diverse.

Systems of this sort provide a much more intuitive method of quickly generating parametric surfaces. However, as one might expect the complexity of the fitting process can be quite high. For their part, Michalik et al. employ a number of techniques to cope with the explosive growth of the problem, but admit that as the number of constraints increases, performance is adversely effected.

It may at first seem counterintuitive to forgo additional functionality to arrive at a more functional system. However it is important to remember that the underlying techniques of traditional sketching are not based on precision or quality of output, but on the speed and ease with which that output can be created by the artist. We have already seen how seemingly simple systems based on the inflation techniques make this trade-off, forgoing user control of the 3D aspect of the geometry in exchange for rapid development from

silhouette strokes alone.

Other researchers have taken note of this as well. Starting from the same basic interface as Igarashi et al.'s Teddy, Levet et al. for example developed a more expressive interface by allowing the user slightly more control [53]. The Levet et al. system, rather than working with a silhouette shape alone, requires two inputs from the user: a silhouette stroke, and a profile curve. Profile curves replace the standard rounded cross sectional profile used in Teddy, allowing not only rounded shapes to be created, but also forms with an arbitrary cross section. An example of this can be seen in figure 2.12.
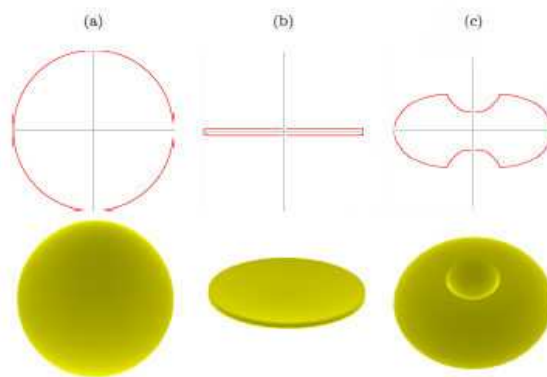
Figure 2.12: The shape created by the silhouette curve in (a) is effected in different ways by the profile curves in (b) and (c) [53]

This basic idea has been expanded on by a number of researchers. An approach that has shown some recent promise is the use of procedural modelling methods, especially those based on generalised sweeps and extrusions. A good example is provided by Cherlin et al. [13]. Starting from a traditional drawing technique in which the artist draws tight spirals to feel out a 3D shape, the authors developed a construction method they call rotational and cross sectional blending surfaces in which the user defines a closed cross sectional shape, and a pair of contour curves. A parametric surface is then generated by sweeping the cross section along the path defined by the two curves, while dynamically scaling the cross section based on the curve's relative separation. Once a number of modelling components have been constructed, the user can further distort their shapes using a stroke based deformation system, and position them in 3D space to construct a complete model.

Techniques such as these described by Levet et al. and Cherlin et al. have a number of advantages. First, as Cherlin et al. note, because each surfaces is generated from parametric strokes, the surfaces themselves have a parametric definition, allowing them to

be evaluated at arbitrary levels of precision, and providing a ready-made coordinate system for the application of surface techniques like textures. Second, from the user's perspective, models are created directly from input strokes, making the construction process highly intuitive. Finally, like the inflation based systems, because the algorithms used to generate the surfaces are straightforward and mechanical, they can easily run at interactive rates.

A team of computer scientists at the University of Toronto have developed a 3D curve sketching system called ILoveSketch [1]. The tool makes a few novel contributions to the field: automatic view rotation to improve curve sketchability; an axis widget for sketch surface selection; and implicitly inferred changes between sketch techniques. Although this system was created for use by professional designers, the same team have recently released a new version called EverybodyLovesSketch aimed at a far broader audience [2].

The main focus of the tool is the creation of NURBS curves and it's usability with gestures. The interface is designed to be like a 'virtual sketchbook' allowing users to use various gestures to turn the page or throw away a particular sketch. Along with the ability to manually rotate the scene, ILoveSketch auto-rotates in an attempt to mirror how an actual artist works. The software was tested by a professional designer who felt the auto-rotation was unnecessary but generally enjoyed being able to work in three dimensions over the standard two. As with Poletti's system, discussed in section 2.6.2.3, ILoveSketch was created for use without stereoscopic screens so lines were shown as varying shades of grey depending on there depth within the scene. It was noted however that this often led to sketches becoming cluttered as they became more complex. It seems likely that stereo may resolve this as it would give the user the ability to perceive the sketch as they would real world objects.

NURBS allow a user to create perfect curves and are extremely useful when it comes to three dimensional modelling. However, as Hollister points out once you start using them for sketching "you are forced into the limitations and idiosyncrasies of the program and its underlying mathematical geometry technique" [36]. In contrast voxel based drawing should allow a user to sketch freely with no restrictions.

2.6.2 Voxel Based Methods

While at present the field is inundated with vector based applications mainly focused upon sketch-based modelling there are significantly less voxel based applications. The majority of these applications focus on the deformation and sculpting of voxmaps, almost

the opposite of drawing and sketching, and the creation of three dimensional voxmaps from standard two dimensional pixmap. How to actually sketch freely within a scene represented by a voxmap has not really been explored. This comes as a surprise when so many of the standard 2D drawing programs in use today are pixel based.

The signicant advantage of voxels is the richness of this representation and the very regular structure which makes it easy to manipulate. However, despite this, voxel based structures are often disregarded in favour of vectors as the scene size and resolution can become so large that voxels often do not even t in the computers memory. In addition to storage, the rendering of such data is also extremely costly, even for previsualization [20].

2.6.2.1   Height-fields and Shape from Shading

As with some of the vector based methods discussed earlier a number of voxel based applications have been developed which attempt to create three dimensional models from 2D sketches and images. Many researchers have recognised that the depth and surface normal information described by an artist's shading offers a potential source of information that could be used in reconstructing the 3D shape of an object from a 2D shaded image.

In the past, perturbations of the lighting model have been used to add surface details to 3D models in a process called 'bump maps' developed by Blinn [8], however these effects are only an illusion. In 1990 Williams took this process one step further into displacement mapping, which rather than disturbing the surface normals, actually changes the geometry of a surface based on a displacement map [86]. However, this method still relies on underlying geometry.

A first step towards the use of shading in modelling is the use of height-fields or digital elevation models (DEM) as shown in figure 2.13. In the same way that a grayscale raster image assigns an intensity value to each pixel within a composition, a height-field image assigns a value to each pixel corresponding to the distance between the viewer and the surface of an object. The result is a grayscale image that can then be interpreted by a modelling system to generate a 3D surface – think, for example, of a topographical map with darkly coloured valleys, bright peaks, and a smooth gradation in between. Because this height field data is represented as a 2D raster graphic, it can be edited using off the shelf image manipulation programs [86]. By applying familiar painting tools such as smearing brushes and lasso selection the user can manipulate the height-field data in 2D, and then convert the model back to a 3D surface representation.
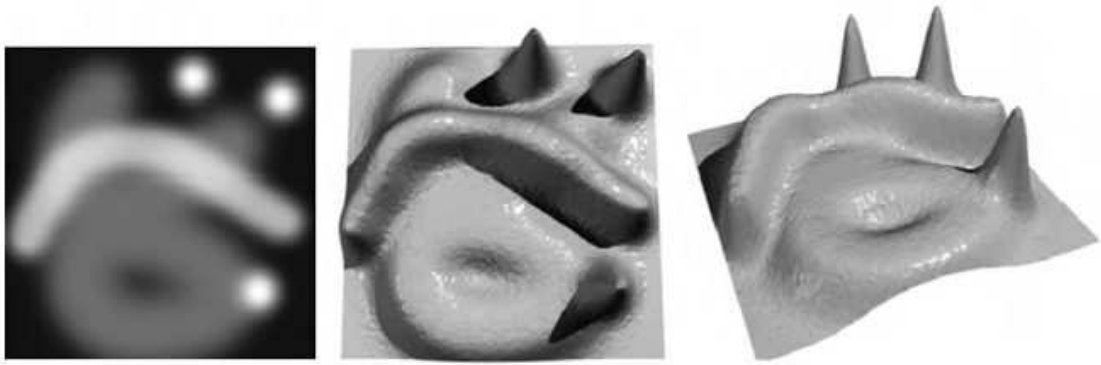
Figure 2.13: An example of height-field modelling [18]

However, as Williams notes, working directly with height-fields has some difficulties. First, most users are unfamiliar with interpreting a grayscale image as height rather than light information. Furthermore, Rushmeier et al. point out that the value scale of a height-field image is always oriented towards the viewer [68]. Interpreted as a shaded image this corresponds to a light source shining directly from the viewer's position, a direction that tends to wash out many small details.

Where as height-fields work with depth information directly, another technique called 'shape from shading' attempts to reconstruct this information from a more traditional shaded image. However, this reconstruction process is particularly difficult. Because the shading is a property not only of the position and shape of the object but also the location of the light source, even under ideal conditions it is necessary to know or estimate additional information about the scene such as each light's position and intensity in order to make a successful reconstruction. Furthermore, ambiguities and mistakes in artist's renderings create additional problems. Thus, systems that successfully apply shape from shading are those that can narrow these variables to some extent.

Rushmeier et al. for example tackle this problem by generating a shaded image from an existing 3D model. The user can edit this shaded image, then re-enter the changes to the system where edits are reinterpreted as changes to the 3D shape. The key to the reintegration process is that the original image was based on a rendering created by the system, meaning that the exact and unambiguous size and orientation of the model, and the positions of any and all light sources are known quantities.

As Kerautret et al. point out, this method is appropriate for small scale edits and repairs, but is not well suited for creating new geometry from scratch [46]. In response, Kerautret and colleagues describe a system in which the user provides several 2D shaded

September 25, 2011

sketches, each with its light source in a different location. These sketches are then combined and interpreted by the reconstruction algorithm to generate a single $2\frac{1}{2}$D model. That is, a model with depth extending from a single direction, like the shape of a sheet laid over an object placed on a table.

This sort of $2\frac{1}{2}$D or 3D depth painting interface to modelling has also found its way into the commercial sphere. First introduced in 1999, ZBrush developed by Pixologic is a sort of hybrid paint and modelling application that uses a unique pixel representation to store both standard pixel information (colour and alpha values), as well as modelling information (depth, texture, and material properties) [67]. Using ZBrush, the artist can not only paint an image, but also push and pull the canvas surface in and out. This allows the artist to apply lighting effects and other traditionally 3D modelling techniques to the creation of 2D artwork.

Although applications like ZBrush make effective use of this sort of modelling interaction, as we can see from the efforts of researchers like Rushemier et al. and Kerautret et al., as the focus of a modelling interface this method of creating 3D geometry has some serious drawbacks. Working directly with height-field data is effective for generating minor details, but for general modelling its visual interpretation is both unintuitive and uninformative to the artist. While an artist may be familiar with working with shading information, this method of interaction over complicates matters as any ambiguities and mistakes in the drawing, which are highly likely, will create unforeseen problems.

2.6.2.2 Deformation and Sculpture

Whereas height-fields and shape from shading techniques manipulate a model from a single direction, this idea can be extended to working more directly in three dimensions in a technique called 'deformation'. Here, the surface of a 3D object is interactively pushed in, puffed out, pulled, smudged, smoothed, gouged, or otherwise deformed to create the model's features.

Deformations are not a new technology in the 3D modelling arena, but it is their resemblance to physical artistic techniques that relates them to the field of sketch based modelling. In many ways, deformation systems allow users to 'sculpt' the surface of a model by applying deformation operations.

The most general form of deformation operations are global deformations, and can be thought of much like an image processing filter; applying a single change to the whole

model. An example of a system using global deformations is provided by Wyvill and Guy, who's application uses spatial warping functions and their hierarchical arrangement into BlobTree structures to create models [88].

A more popular approach to deformation as a modelling tool has been the use of local deformations. These are operations that effect only a small portion of the model, and can often be equated to activities like sculpting or carving. An early example of this method is presented by Galyean and Hughes [28]. The authors' system uses a voxel based volume representation coupled with a carving tool controlled by a force feedback 3D pointing device to add and remove material from a virtual sculpture.

A major limiting factor for these voxel-carving applications is the fact that as the model grows bigger, there is a cubic growth in the memory requirements of the underlying voxel structures. This ultimately places a low ceiling on the level of detail that can be expressed by the user. One possible alternative to is the use of an adaptive voxel grid or 'adaptively sampled distance field' (ADF), a technique proposed by Frisken et al. [27].

Implicit modelling offers still another alternative representation for volumetric modelling in which the model is represented by implicit functions. Andreas and Niels Jørgen, describe a volume sculpting system based around the level-set implicit surface representation, and provide a number of references on volumetric modelling [3]. Han and Medioni's 3DSketch system uses an equipotential surface representation to allow users to quickly trace and then refine clay-like models using a 3D stylus [34].

Because of the resemblance to 3D artistic techniques like sculpture, a number of systems in this category make use of 6 Degrees of Freedom (6-DOF) pointing devices and other spatially tracked input sources as a means of modelling. Ferley et al. for example demonstrate a system in which gobs of material can be deposited in space using a 'toothpaste' tool directed in such a manner [26]. Deering's HoloSketch system uses a 6-DOF wand device along with a head tracking and 3D display system to allow a user to sweep out shapes in mid air [21]. Schkolne et al.'s surface drawing project pushes this manual creation idea one step further [72]. A user wearing a special data glove creates ribbon like surfaces by sweeping the gloved hand through space over a workbench display.

Clay like deformation can also be applied to more standard polygonal model representations. The commercial application Mudbox, developed by Skymatter, for example allows users to create basic shapes like blocks and spheres or import existing models and then subdivide and deform their surfaces using a variety of 'brush' tools [59]. Resulting

models are polygon meshes that can easily be imported into other software for texturing or animation.

(a) Painted Base Model   (b) Painted Intermediate Model   (c) Result
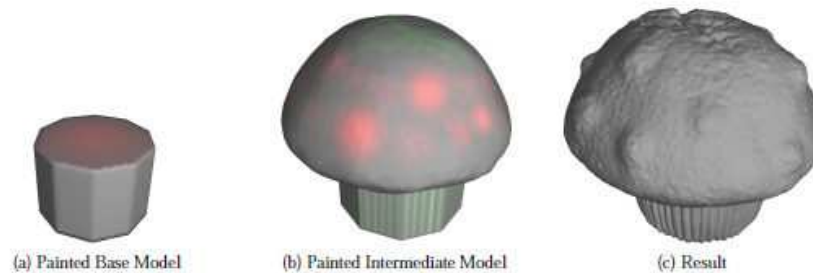
Figure 2.14: Velocity Paint: Red paint adds to the model while green paint subtracts from it [51]

A more novel approach to deformation called 'velocity paint' is described by Lawrence and Funkhouser [51]. In this system, rather than deforming a model directly, a user paints the surface of a model signifying areas he or she would like to be distorted. Each colour of paint signifies a different distortion operation like growing, shrinking, etc as can be seen above in figure 2.14. Once the user has applied paint, the model is simulated and the distortions gradually take effect as an interactive animation.

Finally, along with paint-like methods, several researchers have investigated sketch based deformation interfaces that use strokes as their control mechanism. A representative paper in this area, authored by Singh and Fiume, describes a sketch-based deformation method called 'wires' in which parametric curves are bound to an arbitrary model to act as manipulation handles [75].

Deformations provide artists and designers with powerful tools to affect the look of their models. However, just as a sculptor begins with a block of stone or a lump of clay, for the most part, deformation operations must be applied to an existing model. It can be difficult to create an entirely new model using these techniques. Deformation can instead be integrated into other modelling techniques, where it functions as one of a number of tools at the artist's disposal. Fully 3D sculpting systems offer another approach where matter can be created and manipulated manually. While this approach more closely fits with working in a 3D environment, such systems require additional space and expensive display and input equipment. 3D interfaces can also be more physically taxing on the user, who must manipulate and hold their hands and arms in space for extended periods of time [21]. This is discussed in more depth in section 2.7.

2.6.2.3 Freehand Sketching

While the voxel based methods of deformation and sculpture and height-fields move away from sketching freely in three dimensions a couple of attempts have been made to achieve this goal. In 1995 Helen Poletti designed a three dimensional sketching tool as part of her MA in Computing and Design at Middlesex University [66]. The project was inspired by her desire to create directly in 3D space, capturing the essence of the sketch and receiving instantaneous three dimensional feedback. She argued that, although sketching is a very important part of the design process, much creativity and enthusiasm is dissipated during the transformation from the sketch to a 3D model. Therefore, a three dimensional sketching tool would allow users to keep the truly creative areas of design, while allowing them to view their design in 3D space.

(A)   (B)   (C)
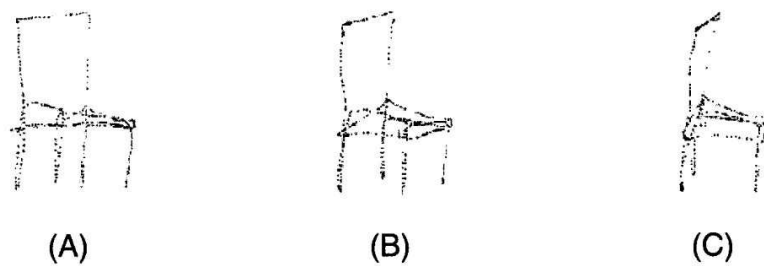
Figure 2.15: (a) An original sketch of a chair created using Poletti's system, shown (b) after one rotation, and (c) after second rotation [66]

The input for the program was gathered using a 3D digitiser. This works by plotting the coordinates from the surface of a 3D object placed upon a digitising tablet. In effect it is similar to present day pen and tablets. A continuous stream of data was recorded as the pen is moved over the tablet. This was then displayed on a computer monitor with a trackball provided to rotate the image (view rotation was only possible when the user was not drawing). As stereoscopic displays were not available Poletti thought about displaying the sketch as a red/green anaglyph. However, it was decided that this would restrict the user. As an alternative voxels at greater depth in the scene were shown as lighter grey. The screen itself acted as the drawing plane.

While Poletti's 3D sketching tool was a success with the limited equipment she had available there are still significant issues with her system. With no cursor there was no actual reference to the previous position of the pen, this meant that sketching had to be carried out swiftly, exposing and challenging both memory and mental imaging

capabilities.

In early 2009 Sketch3D was created as part of my BSc in Computer Sciences at Durham University [19]. As with Poletti's work, the aim of the project was to attempt to create an environment which allowed a user to draw freely in three dimensions using a 2D input. It also focused on finding the most efficient means of doing this. While an environment where a user could sketch in a stereoscopic environment was developed, at present Sketch3D limits this to the ability to sketch on the x and y plane at varying depths within the scene. It is possible to change this depth by using keys upon the user's keyboard. A number of alternative solutions to allow a user to draw freely within the scene were proposed, but further research is required to ascertain the most user friendly and effective technique.

Sketch 3D was implemented using a Model, View, Controller (MVC) architecture with varying data structures to store the plotted points within the Model class. A series of tests were run upon each Model to ascertain the most efficient. The results of these tests concluded that a Model based upon a vertex array would be the most efficient for small amounts of data but would lose its advantage over other structures, such as a 3D array and a series of 2D arrays, as the amount of data stored increased.

2.7 Three Dimensional Input

When it comes to working with a system based within a three dimensional environment one of the key requirements for that system must be to provide an intuitive user interface and means to interact with it. There have been a number of attempts to make this interaction possible including use of 6 Degrees of Freedom (6-DOF) pointing devices and other spatially tracked input sources.

2.7.1 'Flying' Mice

A 'flying' mouse is one designed so that it can be moved and rotated in the air for 3D object manipulation. Rather than being sat flat on a desk, where only one plane of movement is available, a user is able to move the mouse in any possible direction. Advantages to the flying mouse include the fact that it is easy to pick up and learn how to use and that it is relatively fast in comparison to static desktop devices. However, a number of disadvantages are inherent due to the way in which people can move. There is a limited movement range and a lack of coordination as peoples joints are only able to stretch a certain distance and

rotate at certain angles. After a time people will also suffer from fatigue from use as it is tiring keeping limbs extended in the air for long periods of time [90].

An extension to the idea of a flying mouse is the 'wand' used by Deering with his HoloSketch system [21]. The 3D mouse is augmented by an offset digitiser rod protruding from it; effectively making the mouse a six-axis wand, or a 'one-fingered data glove'. The end of the rod functions as the direct manipulation cursor for most of the HoloSketch system. The 3D mouse has three top buttons and one side button. Similar to this idea is the data glove used in Schkolne et al.'s surface drawing project [72]. As the user sweeps their gloved hand through the air above a workbench display, the gloves position and motion is continuously tracked and ribbon like surfaces drawn in relation to it's movements.

2.7.2 Desktop Devices

There are also a number of 6-DOF input devices which just sit flat and stationary on a user's desk and usually take the form of a 'joystick' device. The user can move the joystick anyway they want the cursor to move within a three dimensional space.

These devices provide a number of advantages in comparison to flying mice. These advantages include reduced fatigue, since the user's arm can be rested on the desktop while they work. They also provide increased coordination which leads to a smoother and steadier cursor movement. The major disadvantage of such a device however is the fact that it is very difficult for a user to control the rate at which they move in a particular direction through 3D space [90]. This prevents users with little to no experience from being able to pick up the device and sketch freely.

2.8 Digital Ink Data Formats

As the everyday use of digital ink increases it is vital that users are able to save, load and share their drawings. Various formats, each with its own benefits and drawbacks, have been developed to make this possible. While the formats discussed in this section are focused on two dimensional ink it is possible that they will need to be extended to cater for three dimensional ink in future. This would be a topic for future research.

2.8.1 Jot

Jot defines a format for the storage and interchange of electronic ink between software applications. It is the joint work of Slate, Lotus, GO, Microsoft, Apple, General Magic and others. The goal of Jot is to provide a simple and convenient format for digital ink exchange. It is designed with the intent to maintain fidelity to the original ink to ensure it can be shared and understood across many environments [29].

The Jot specification describes the intended uses of Jot as well as some of it's benefits. Applications of Jot include: Sharing signatures and annotations between mobile, pen-based computers and a central database; sharing electronic mail between hand-held devices and desktop systems; and taking and sharing notes throughout an organisation. As Jot is a binary format it is light-weight and includes lossless compression to help reduce the space required for ink storage. Jot also supports a wide variety of ink properties which enable it maintain a complete likeness to the ink as it was originally drawn. These include multiple strokes of ink combined into single objects, bounds, scale, offset, colour with opacity, pen tips, timing information, height of the pen over the digitiser, stylus tip force, buttons on the stylus and X and Y angle of the stylus. Applications can choose to recognise or ignore properties as required. In addition to these properties, there are reserved record types to allow for expansion [87].

2.8.2 UNIPEN

UNIPEN provides a document based representation of digital ink. It incorporates the features of several institutions' internal ink data formats, including IBM, Apple, Microsoft, Slate (Jot), HP, AT&T, NICI, GO and CIC. In contrast to Jot the UNIPEN format is mainly for technical and scientific use. It contains a lot of data, such as coordinates and annotations, to suit the needs of people testing handwriting recognition algorithms on huge amounts of data. Therefore it requires a lot of memory and a powerful processor. The format provides no provisions for aesthetically altering properties such as ink width or colour [33].

UNIPEN is very focused on handwriting recognition requirements, with features to support labelling of ink data, but is not optimised for data storage or real time data transmission. Neither is it designed to handle ink manipulation applications involving colours, pen tip, image rotation, rescaling, etc. While Jot is a proprietary format that avoids any abstract characterisation of ink. [84].

2.8.3 InkML

Ink Markup Language (InkML) is an XML format under development for pen-based applications that store, manipulate or exchange digital ink. It is intended to unify various ink representations in a common modular format [12]. XML representation of digital ink will be larger than a compressed binary formats such as UNIPEN and Jot, however using this representation allows a broader range of applications to handle the ink data [84].

Ink ML provides similar features to both UNIPEN and Jot. It supports a complete and accurate representation of digital ink. In addition to the pen position over time, InkML allows recording of information about device characteristics and detailed dynamic behaviour to support applications such as handwriting recognition. It can record pen tilt, pressure, accuracy and dynamic distortion as well as properties directly related to the ink such as width and colour information [87]. Each stroke in InkML has a number of attributes attached. These are known as 'channels' and can be altered to change various aspect of the stroke. InkML does contain a Z channel but this relates to the distance the pen is above the tablet rather than the depth of drawing.

2.8.4 ISF & the Microsoft Ink SDK

Microsoft has developed it's own SDK for creating programs which are able to handle digital ink. They refer to these as 'ink enabled programs' and a pair of managed code assemblies supports their creation in a managed language such as C# or Visualbasic.NET. Ink created by an ink enabled program is it's own portable graphic format known as Ink Serialised Format (ISF). Data in ISF can be serialised and saved to disk in either binary format or XML [41].
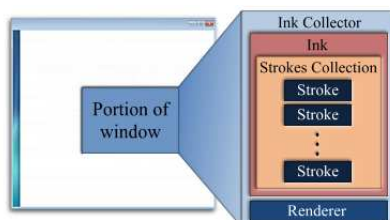
Figure 2.16: Containment relationship of Microsoft Ink [41]

Ink, Stroke and Strokes Collection are key classes for the representation of ink. The Ink class is the fundamental data structure used to represent ink captured by the Tablet PC. An Ink object is a container for Stroke objects, while a Stroke object is essentially an

ordered collection of packets that is captured in a single pen-down, pen-move and pen-up sequence. A packet is the set of data that the tablet device sends at each sample point, such as x and y coordinates, pen pressure, pen angle, and so on. The Ink class typically exposes its Stroke objects through another collection class called Strokes Collection. The Strokes Collection class is actually a collection of references to Stroke objects [87]. All of these classes are contained within an Ink Collector object which also contains a renderer to draw ink as the pen is moved. The relationship between these classes can be seen in figure 2.16 above.

2.9 Chapter Summary

When it comes to sketching in three dimensions the majority of applications currently available rely on vector based drawing methods. This is primarily because the applications are designed to take a users two dimensional input and transform this into a three dimensional model. Having the sketch represented as vectors makes it simpler for the program to act upon its geometry and thus convert it to a model. There are a number of methods to achieve this aim including Gesture Based Modelling, Reconstruction and Blobby Inflation. Other vector based applications focus on the creation of curves allowing the user to draw within or on existing 3D models. They also allow the user to create wire frame type models. These stroke based applications bring the user closer to traditional sketching rather than the more structured modelling methods detailed [1].

While at present the field is inundated with vector based applications mainly focused upon sketch-based modelling there are significantly less voxel based applications. The majority of these applications focus on the deformation and sculpting of voxmaps, almost the opposite of drawing and sketching, and the creation of three dimensional voxmaps from standard two dimensional pixmaps. How to actually sketch freely within a scene represented by a voxmap has rarely been explored. This comes as a surprise when so many of the standard 2D drawing programs in use today are pixel based.

The big challenge when dealing with a voxmap is how to represent the data efficiently. While there are a number of techniques to achieve this, as discussed in section 2.4, the most effective seems to be the development of a space partitioning data structure. Various formats have already been developed to represent two dimensional digital ink, and while it may be possible to adapt these formats to also support three dimensional ink it is important to first develop a means to represent and interact with it on a more basic level.

September 25, 2011

Chapter 3

Design & Implementation

This chapter will look at the design of the software created as part of this project, henceforth referred to as Sketch3D. It will also focus on the issues encountered during the implementation of the software. These include the task of devising a simple means of drawing in three dimensions while using a 2D input device such as a digital pen; a discussion of various aspects of the basic user interface created such as a 3D cursor; and finally a description and analysis of how the problems posed by the relationship between the on screen coordinate system and the OpenGL coordinate system were tackled.

First we lay out the requirements which had to be met when designing the software before moving on to discuss some of the technical decisions which had to made in regards to actually sketching in three dimensions and putting together the system architecture for the software. This includes an in depth look at the octree data structure which is used to represent the digital ink.

As detailed in section 2.6.2.3 a simple sketching interface was developed as part of my BSc. However while this code was used as a base to work from, the design and implementation laid out in this chapter introduces significant differences and new features. To all intensive purposes it is a substantial new implementation.

3.1 Requirements

The means of sketching in a 3D environment must meet certain integral requirements: the user must be able begin to draw at any point within the scene; the user must be able to easily alter the depth at which they are drawing, aiding them to draw at any point; and finally the user must also be able to draw points continuously in any feasible direction

35

within the scene. Following the discussion and analysis of existing 3D input devices in chapter 2 the following requirements have been added: the input device must sit flat upon the user's desk so as not to incorporate any device which will quickly fatigue a user; and the means of input must make the system accessible to users of every experience level. The final requirements restrict the input devices to hardware commonly used with most home computers, a mouse, a keyboard and a pen and tablet.

In regards to the back end of the software, the data structure used must store the data efficiently but also allow it be accessed, altered and rendered quickly. This goes hand in hand with the integral requirments which must be met in order to successfully sketch in a 3D environment.

3.2 Sketching in Three Dimensions

As discussed in section 2.7 an important factor when working in a three dimensional environment is the successful implementation of intuitive user interface and means to interact with it. Section 3.5 deals with the user interface but firstly we will look at solutions to actually sketch within this environment.

While a number of different methods were considered to meet the requirements laid out in the previous section, only a couple of alternatives were implemented in order for the focus of the project to remain on the data structure behind the software. This section will now look at the various advantages and pitfalls of the methods implemented and also those considered.

3.2.1 Function Keys Assigned to Depth

The initial method implemented within Sketch3D involves simply assigning a series of keyboard keys to represent different depths within the scene. For example the F1 key on the keypad sets the plane the user is drawing upon to a plane near to the camera while in contrast the F10 key enables the user to draw on a plane a long way back from the camera.

While basic, this is still quite an effective method. It is very simple to operate and users will always know exactly what depth within the scene they are drawing at. However, the obvious problem with this technique is that it still does not provide a method to draw across these two dimensional planes and thus fails to meet the requirements as it does not

allow the user to sketch freely within the scene.

3.2.2 Arrow Keys Related to Continuous Depth

The more advanced method implemented is to use the up and down arrow keys upon the keypad to alter the depth at which a user is drawing. If the up arrow is pressed the depth is moved one unit, equal to the size of a single pixel, further back into the scene. If the up arrow is held down then the depth continues to gradually move further back into the scene. In the same way the down arrow will alter the depth and bring the level the user is drawing at back towards them. This seems to be a more effective method as it allows the user to draw continuously across various depths instead of limiting the user to a set of predifined planes to draw upon.

This seems to be a very effective method of allowing a user to draw freely within the scene and it also meets the requirements laid out previously. In contrast to the method involving function keys they are not limited to a particular set of preset planes. The downfall of this method is that, even in stereoscopic 3D, it can be hard for a user to judge the depth at which they are currently drawing and to re-find a depth at which they were drawing previously. However, a number of solutions to these issues were created such as the implementation of a three dimensional cursor and also an indication on which points drawn within the scene lie at the current depth. More information on these solutions can be found in section 3.5.

3.2.3 Other Options Considered

Before implementing the above methods a number of other solutions where considered. However, in order for the project to remain focused upon the data structure behind the software these options were rejected as they would have taken up valuable research time. If this project is revisited from a usability angle then more research into these techniques would be necessary.

3.2.3.1 Button to Alter Axis

The first method which was considered involves using a different button upon the input device to alter the axis which is being drawn upon. For example while pressing and holding the left mouse button as usual the user will draw upon the x and y axis. However, if the user instead uses the right mouse button the program will draw upon the y and z axis.

September 25, 2011

The means to draw upon the y and z axis grants the user the ability to sketch across the 2D planes allowed by just using the keyboard function keys to alter the depth. This method is still limited however as it only allows drawing across the scene and from front to back in the scene. It does not allow for free movement around the scene such as diagonally. Due to this flaw this solution fails to meet the requirements and would not be as effective as the use of the arrow keys to control depth.

3.2.3.2 Pressure Sensitive Pen

The next method looked into completely solves the issue of being able to sketch freely within the stereoscopic environment. It involves making full use of the API for the digital pen which can be used with Sketch3D. The pen is actually pressure sensitive. This means that it would be possible for the depth within the scene at which the user is drawing to be altered by the amount of pressure being put upon the pen.

While seeming to solve some of the issues posed by other solutions, this method does have its pitfalls. It would be difficult for inexperienced users to control the speed at which they move backwards into the scene and also to maintain a particular depth while drawing. However, if a user was able to cope with this solution and control the depth effectively then this method could possibly create more accurate and artistic pictures than the arrow keys method suggested.

3.2.3.3 Rotational Geometry

Perhaps the most advanced method looked into to allow the user to sketch freely involves giving the user the ability to rotate the scene within which they are drawing. The user would then be able to draw upon the two dimensional plane currently lying across the physical display. If the user was able to not only rotate around the x, y axis but also the y, z axis this would enable them to sketch at any point around the scene.

This rotation could be performed in a number of ways. Either by using pre-assigned keys upon the keyboard or by dragging the scene around using the mouse or pen. The disadvantage of using the input device itself to alter the scene is that the user would then be unable to draw as the scene was in motion. A system similar to this has already been designed and implemented within the ILoveSketch system which was discussed in section 2.6.1.5. The software allows users to work on concept models using a pen and tablet. It is able to recognise the type of curve which a user has just drawn and, in turn, rotates

the scene automatically in much the same way as a person sketching with pen and paper would rotate the page. The software also provides a set of preset gestures which, when performed, rotate and also zoom the scene. This means that the user us able to switch from modelling to manipulating the scene just by performing a predefined gesture [1].

3.3 System Architecture

The Sketch3D tool was created using a Model, View, Controller (MVC) architecture. MVC is a design pattern that simplifies application development and maintenance. It achieves this by splitting the application into three logical components; the Model, the View and the Controller [56]. The Model contains a data structure which handles all of the data relating to the digital ink drawn within the scene. The View takes the data contained within the Model and visualises and plots it upon the monitor for the user to see. Finally we have the Controller. This is responsible for handling all user input into the program and using this to update the data within the Model. These components are initiated and run during the main program loop. This loop is the OpenGL rendering loop which runs as long as the program is open and operating. Figure 3.1 below shows the interaction between the components of this architecture.
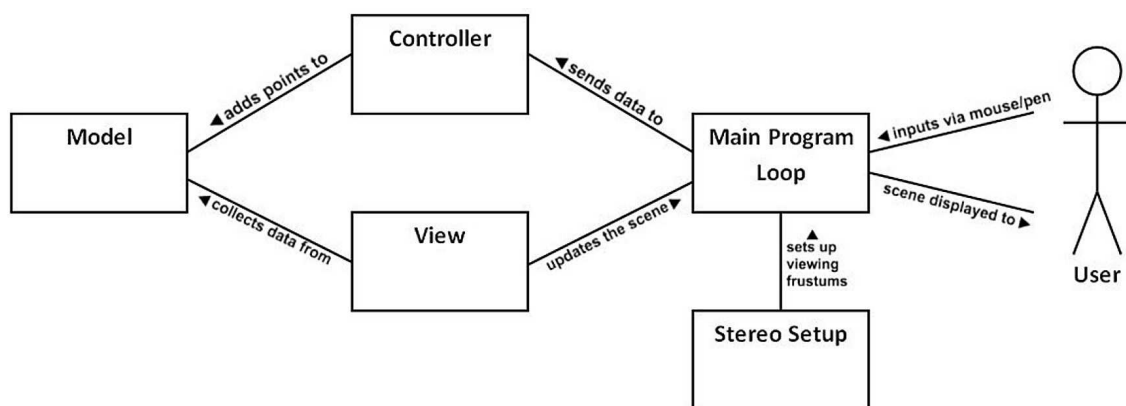
Figure 3.1: UML diagram showing the high level architecture of Sketch3D

3.3.1 Model

The Model stores a data structure containing every vertex in which the mouse has been clicked or the pen has touched upon within the stereoscopic environment. Each point is defined as an individual structure with a number of separate attributes. As this is a

position within a three dimensional space the point stored within the data structure will have three attributes representing it's current location: the x coordinate, the y coordinate, and also the z coordinate which represents its depth within the scene. In addition to this the point also contains information relating to it's current colour and also to it's current width.

The Model is completely independent from the rest of the program. While different classes may access it, the Model itself relies on no other classes. This means that it is simple to remove the Model and replace it with another containing a different data structure. The Model is also responsible for storing a number of variables separate to the data structure but integral to the running of the program. These include the current size of the window and the current position of the cursor.

3.3.2 View

The View takes the data contained within the Models data structure to actually plot the points on the screen for the user to see and therefore interact with. As long as the main loop of the program is running the View component will continue to update this drawing every time the data within the Model is updated.

The View class works using the standard drawing library packaged with OpenGL. The function glBegin(GL_POINTS) is called to tell the software that it needs to plot the following points upon the screen as three dimensional points. A loop is then used to iterate through all of the points stored in the Models data structure. These points are displayed within the stereoscopic environment at the positions corresponding to the values of the coordinates stored within each point structure. The visual aspects of each point such as the size and colour are also read and the point drawn will altered to reflect these attributes. Once all points have been drawn correctly the function glEnd() is called to indicate to the program that the end of the structure has been reached. This process is repeated every frame within the OpenGL rendering loop.

3.3.3 Controller

The Controller handles all user input into the program. It is responsible for updating the data contained within the Model; either adding, removing and editing points in the underlying data structure or updating the current cursor position. The Controller makes use of the OpenGL utility toolkit, GLUT. Using standard calls from this library it is able

September 25, 2011

to retrieve the cursors current coordinates in relation to the top left hand corner of the screen. It is also able to retrieve the present nature of the input device. For example if the pen is touching the tablet, if a mouse or keyboard button is being pressed and whether the input device is currently in motion.

When the user draws the pen across the tablet or moves the mouse while holding the left mouse button the Controller will recognise this movement and add every point the cursor crosses to the data structure stored in the Model. The points are only added to the Model when the user first clicks a point and then when the cursor is in motion. This prevents the same point from being continuously added to the Model if the user clicks and holds the cursor stationary. Sketch3D also provides controls to remove points from the data structure and to alter the current depth of the cursor. These actions will be discussed further in the remaining sections of this chapter.

3.3.4 Stereo Setup

The Stereo Setup is actually not a single component but a group of classes separate from the main program. These classes are responsible for setting up the viewing frustums to enable three dimensional viewing on a stereoscopic screen. The code takes the regular image of the scene and produces a stereoscopic pair with a high image quality. This is achieved by using a Real-Eye configuration and a newly defined mapping algorithm. The camera separation is set equal to the nominal human eye interpupillary distance and the perceived depth on the display is identical to the scene depth without any distortion. The mapping algorithm maps the scene depth to a predefined range on the display to avoid excessive perceived depth [76].

Figure 3.2, below, shows an example of a stereoscopic pair produced by Sketch 3D using these techniques. The left and right images have been overlaid upon each over to make the disparity between the two clear. As this is outside of the scope of this project the source code for solely these classes was provided for me by Geng Sun, a Computer Science PhD student at the University of Durham. These classes were called by the main program which was written by myself for this project.

Figure 3.2: A stereoscopic pair produced by Sketch3D overlaid upon each other.

3.4 Data Representation

The main focus of this project is how the digital ink is represented within a data structure. Due to the modular nature of Sketch3D's system architecture it is possible to plug a range of different data structures into the program to represent the ink in a variety of ways. The design of the software means that it is possible to alter the Model class so that it contains a different data structure without having to edit either the View or Controller classes. This project looks at a couple of data structures which were previously tested in a number of areas during my 3rd year computer science project [19] and then will move on to focus on space partitioning data structures and primarily the octree.

3.4.1 Abstract Data Type

Before looking the structures to be used an abstract data type must be defined. This lays out the key requirements of the data structures which need to be implemented for Sketch3D to perform correctly and effectively. This abstract data type is represented graphically in figure 3.3 below.

The data contained within the structure will be the points which have been drawn by the user. In turn each point will be made up of an x coordinate, a y coordinate and also a z coordinate corresponding to depth. There may also be other information attached to the point such as it's current colour or size.

Initially the data structure will only be required to have three related functions: an 'add' function which takes a point as input and stores it in an accessible manner within
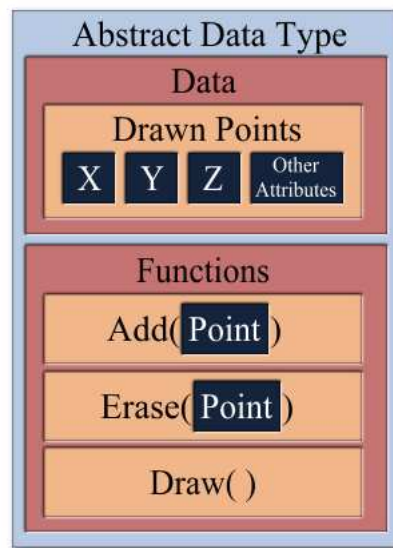
Figure 3.3: A graphical representation of the abstract data type to be implented.

the structures set of data; an erase function which also takes a point as input and, if the input point is already stored, removes the matching point from the structures data set; and finally a draw function which takes no input but when called gathers all of the stored points and provides them as output to the View class which can then draw them to the screen.

3.4.2   Previous Structures

During the initial research conducted as part of my 3rd year computer science project [19] a number of alternative data structures where looked at. A couple of these basic structures were a simple list type structure and a three dimensional array. As these structures are also implementations of the abstract data type defined in section 3.4.1 they will be tested to gather comparative data for the results obtained by the tests conducted upon the octree structure.

3.4.2.1   List

The first data structure to be implemented was the list like structure. Although this is technically a one dimensional vertex array, a structure built into the OpenGL library, it is referred to as a list within this project to avoid confusion with the 3D array introduced in section 3.4.2.2. The vertex array essentially acts a simple list with the x, y, and z coordinates stored one after the other as shown below in figure 3.4.
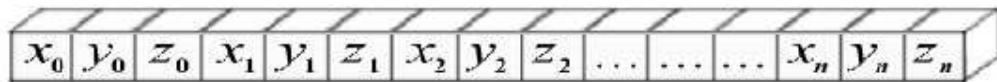
Figure 3.4: Voxel coordinates stored within a vertex array

In the implementation of the list structure the Model class also holds a variable which acts as a counter to keep track of where in the array the next empty block of memory can be found. This counter is incremented after each value is added to the array meaning it will always point to the next available space. The x, y, and z values are stored within the array as GLfloats. The OpenGL float data type was selected because, as the OpenGL Red Book points out, "implementations of OpenGL have leeway in selecting which data types to use to represent OpenGL data types. If you resolutely use the OpenGL defined data types throughout your application you will avoid mismatched types when porting your code between different implementations." [74].

While erasing points is possible using a list it is incredibly impractical. When a point is selected for erasure it is necessary to search the entire list comparing it to every point to see of the point is present and if so then erase it.

3.4.2.2   3D Array

The other data structure implemented and plugged into the Sketch 3D architecture which we will look at here was a 3D array. Each block of memory stored within the array represents a pixel within the stereoscopic environment. As can be seen in figure 3.5 below, the array covers the entirety of the points within the scene in the x, y and z directions. When the array is initialised each block of memory initially contains an integer which is set to 0. When the user draws upon a point and the Controller adds it to the Model, the value of the integer stored within the corresponding block of memory is updated to 1. The View class then iterates over the array and, if the block contains a 1, plots the point in the stereoscopic environment.

In contrast to the list structure it is not only possible but also very simple to erase points from the 3D array. It is just a case of inverting the method used to add points to the structure. Instead of converting the integer contained within a memory block from a 0 to a 1 it will be changed from a 1 to a 0.
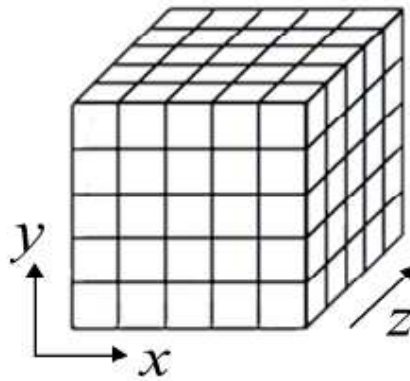
Figure 3.5: Voxel coordinates stored within a 3D array

3.4.3 Octree Structure

The main data structure implemented as part of this project is an octree. This is a form of space partitioning data structure represented as a tree in which each internal node has eight children. The initial scene in which a user can draw acts as the root of the tree. As points are added to the scene it is broken down recursively into eight equally sized smaller segments, as shown below in figure 3.6, such that each segment contains no more than a preset number of points at one time.
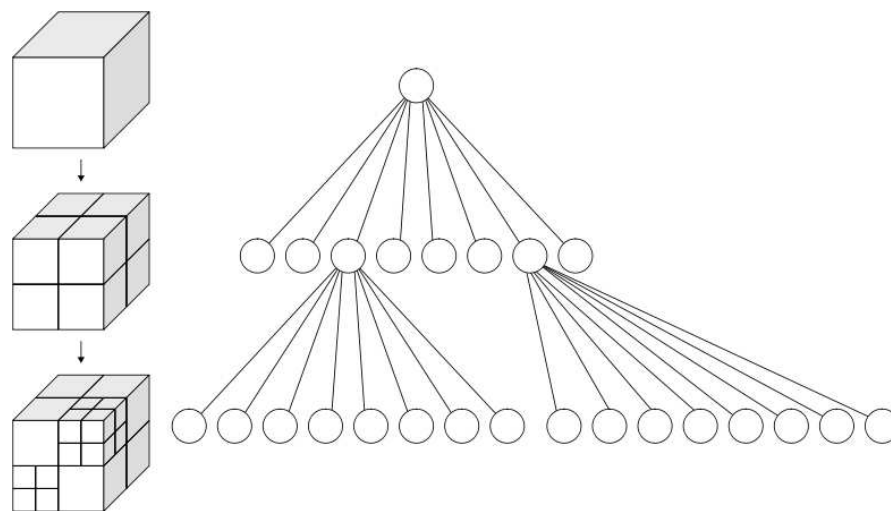
Figure 3.6: Recursive subdivision of a cube into octants and the corresponding octree.

The leaves of an octree are commonly referred to as octants. In the case of Sketch3D each octant contains a list of points within the scene, if the number of points in the section of the scene exceeds a preset limit then the octant will once again subdivide into eight more segments. The preset limit and therefore the size of the list contained within

each octant will effect how many times the scene will be subdivided which is expected to impact insertion and deletion times of points and also the rendering time of the tree. This is investigated later in the project.

When an octant is subdivided each child is given a reference by it's parent. For example the bottom right-hand front corner is 000 while the bottom left-hand front corner is 010 and so on up to the top left hand back corner which has reference 111. As each parent knows the relative position of each child it is possible to find which octant covers a particular area of the scene. This means that when adding a point to the scene it is possible to traverse the octree, find the correct octant and add the point to the list contained within. The same applies for erasing, the location of the point trying to be erased is recorded. The octree is then traversed to find the correct octant. The recorded point is then compared to all of the points within the octant to see if it is indeed present in the scene. If it is the relevant point is removed from the list. If, after the erasure of points, all of the octants within a segment are empty and therefore redundant the segment will be undivided and the parent will once again become an octant itself. This eradicates unnecessary nodes within the tree and hence keeps traversal time to a minimum.

3.5 User Interface

A good user interface is vital for any program working in a stereoscopic 3D environment. Compared to a standard 2D user interface the extra spatial dimension throws up a lot of issues. Not only does the user need to be able to navigate the scene with ease, as discussed in the previous chapter, but they also need to be able to keep track of their current location. This section will explore the solutions developed to combat this problem. It will also address a couple of other tools added to Sketch3D to try and give it the feel of a standard drawing program.

3.5.1 3D Cursor

The first issue tackled was the fact that it was very difficult for a user to tell what depth they were currently drawing at within the scene. This was largely due to the fact that however the current depth was altered the standard Windows operating system cursor remained on the same level as the screen plane. This situation was remedied by first calling the OpenGL utility toolkit command glutSetCursor(GLUT_CURSOR_NONE) which

removes the standard operating system cursor from the program entirely. In it's place a small three dimensional pyramid was drawn to act as Sketch3D's new cursor. This pyramid is redrawn with every pass of the OpenGL rendering loop meaning that it is possible to have it, not only follow the x and y position of the removed cursor, but also to move in and out of the scene in relation to the current depth.

3.5.2   Depth Toggle

Once the 3D cursor had been implemented it became clear that, while it successfully gave the user a much better idea of whereabouts in the scene they were currently drawing, it was not always crystal clear. This raised issues when it came to trying to draw a new point on the exact same level as previous points. A second user interface tool was implemented to help cope with this issue. This new functionality means that all of the points intersecting the depth plane on which the cursor currently resides are rendered as larger red points. A screenshot of this tool in use is shown in figure 3.7 below.
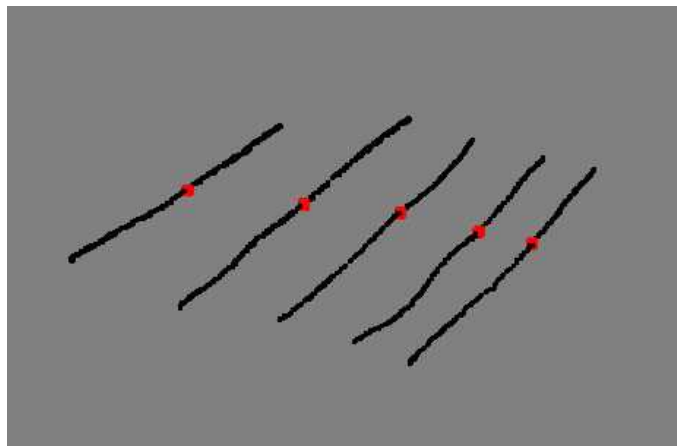
Figure 3.7: A screenshot taken from Sketch3D showing the indication of the current drawing depth.

It was quickly discovered however that this tool became a nuisance when a user wished to just draw at one depth. Every point drawn would be large and red. To get around this issue a function key, F12, was assigned to toggle the depth indication on and off. This means that when drawing across a variety of depths a user is able to see where they are in relation to previously drawn points and when drawing at a single depth a users artistic skills are not hampered by the production of large, bright red lines.

3.5.3 Ink Attributes

Although not the main focus of the research it was decided that a small number of features should be added to Sketch3D the feel of a standard drawing package. To this effect it is possible for the user to alter attributes related to the ink with which they are drawing. As each ink point drawn is a self contained structure this was simple to implement. Along with the locational coordinates contained within each point a couple of other variables are also stored. One containing the colour of the point and the other it's size. As the View class draws each point to screen these values are read and the point is displayed accordingly. At present there is no graphical user interface attached to Sketch3D so once again keyboard keys were assigned to specific values so the user can switch between them at will. As a finishing touch the top end of the pyramid used as a 3D cursor changes colour to match the colour in which the user is currently drawing.

3.6 Scaling of OpenGL Coordinates

In the course of implementing Sketch3D a major hurdle encountered was mapping the position of the cursor on the screen to it's actual location within the three dimensional scene. This section will describe this problem in more detail and explain how the equation to perform the mapping was derived.

The coordinate system used by OpenGL is not the number of pixels displayed upon the screen. This means that the position of the cursor on the screen does not match that of its position within the OpenGL scene. When creating a standard program in 2D this did is not a major issue and a simple scaling can be calculated. The simplest way to demonstrate this is to set up the scene such that the camera is at a distance of 10 OpenGL points (GLints) with a viewing angle of 90°. OpenGL scales via the y axis. So if the user was working with a window 640 pixels in width and 480 pixels in height as shown in figure 3.8, the scaling can be calculated using simple trigonometry where x is the value to be found. This gives us $\tan 45° = \frac{x}{10}$ therefore $x = 10$. The height of the scene, in GLints, can be written as $2x$ and thus equals 20. The height in pixels is 480. It follows that $\frac{480}{20} = 24$ meaning that the scaling from pixels to GLints is $24 : 1$.
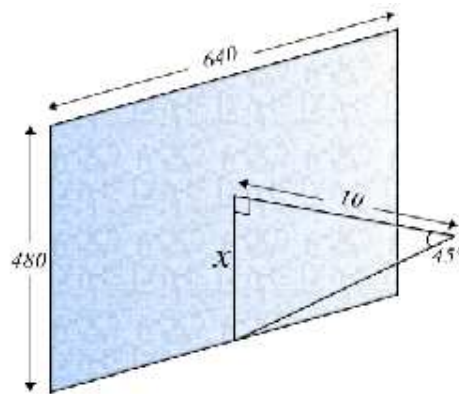
Figure 3.8: Scaling of coordinates in OpenGL.

We can write this scaling as equation 3.1 where z is the distance from the camera to the screen.

$$(z * \tan\theta) * 2 \tag{3.1}$$

While this method of scaling works fine in two dimensions it becomes ineffective once depth is added to the environment and it becomes stereoscopic. First and foremost this is due to the introduction of a second camera into the scene. To create the stereoscopic environment one camera is required for the right eye and one for the left. To allow for this, the viewer position is taken at the point exactly midway between the cameras. In a stereoscopic space the Zero-Disparity Plane (ZDP) is the point at which objects within the scene will appear to be on the display plane, the physical display, when the images are viewed. This means that the ZDP is the effective position of the display in scene space, called the virtual display [43]. The relationship between the virtual display and the physical display also affects the scaling from the cursors actual position to its position within the scene as they are not necessarily identical in size.

This means there are effectively two scalings to be done; the scaling from OpenGL to the physical display and also from both of these to the virtual display. Using x as the current x coordinate of the cursor in pixels, W as the width of the physical display, W' as the width of the virtual display, z' as the distance from the camera to where the user is drawing within the scene, and θ as the viewing angle from the point exactly midway between the cameras.

First we map the x coordinate to the physical display. This gives us equation 3.2.

$$\frac{x}{W} \tag{3.2}$$

We then need to use equation 3.1 to scale this from GLints to pixels. This is shown in equation 3.3

$$\frac{x}{W} * (z' * \tan\theta) * 2 \tag{3.3}$$

Finally we need to be sure to map both the physical display and the scaling to the virtual display. This leaves us with equation 3.4 which calculates the current x coordinate of the cursor within the scene.

$$\frac{x}{(W/W')} * \frac{(z' * \tan\theta) * 2}{W'} \tag{3.4}$$

Similarly substituting the initial value of the y coordinate for x the actual y coordinate of the cursor is calculated. The scaling of the coordinates takes place within the Controller class of Sketch3D. It is these revised coordinates which are then added to the data structure stored within the Model class.

3.7 Subjective Quality of Solution

Once each version of Sketch 3D had been implemented the actual visual appeal of each solution was looked at subjectively. It was found that it is possible to alter the appearance of the 3D array by altering the dimensions of the array. Earlier in this paper we saw figure 3.9 below, this clearly shows that the smaller the size of the array the blockier the image appears.

While an image drawn with a smaller array has the advantage that its draw time would, in theory, be proportionally smaller it's visual appeal is noticeably worse. In comparison, as the list structure and the octree store floating point numbers, they have a far higher precision when it comes to interpreting the exact point a user has just drawn upon. This means it is able to draw a higher quality image.

It is possible to make the images produced using a list or an octree even smoother. This would be done by using the glBegin(GL_LINE_STRIP) function in place of glBe-

September 25, 2011
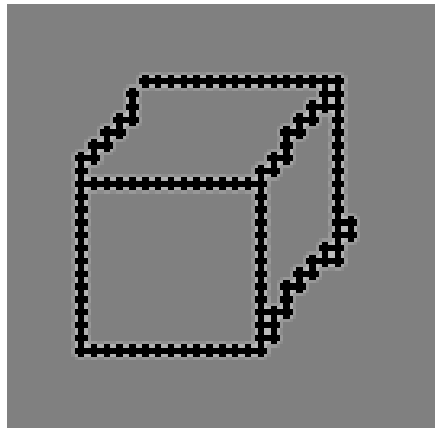
(a) A cube drawn upon an array with a length and a width of a quarter the number of pixels upon the display.
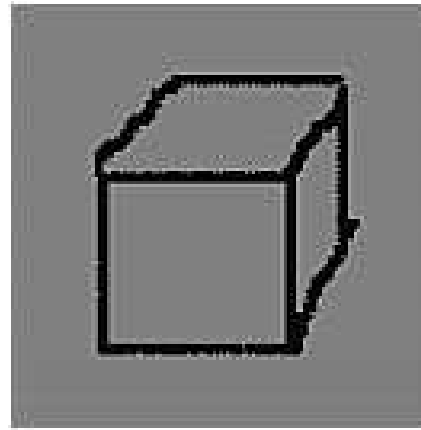
(b) A cube drawn upon an array which has dimensions equal to the number of pixels upon the display.

Figure 3.9: Subjective quality of the three dimensional array.

gin(GL_POINTS) within the View class. It would then be necessary to indicate the end of a line every time the user releases the mouse button or removes the pen from the tablet. Within the list structure a null point could simply be added to indicate the end of a line and the start of a new one. However the octree would be a little more complex. One possible solution would be to time stamp each point. This would allow the View class to tell the order in which the points were drawn regardless of the octant in which they are stored. In parallel to this a boolean value would be added to a point if it is the first or last within a series allowing Sketch3D to find the beginning and end of a line. At present it is not possible to use this technique within the 3D array. Due to the structure of the array the program is unable to tell where one line ends and another begins.

Chapter 4

Testing & Results

Once data structures which met the criteria laid out by the abstract class had been identified it was then necessary to undertake non-functional testing upon each structure, specifically load testing. The aim of the testing was to ascertain how well the structures fulfilled the task of storing and editing the digital ink created by Sketch3D.

This chapter presents the results of the performance tests undertaken upon the data structures detailed earlier in this paper. A simple list, a 3D array and finally the octree were tested in regard to three main areas. The insertion time is the time it takes to insert a single point into the structure; the draw time is the amount of time it takes for the View class to iterate through the structure and plot the points within; and finally the erase time is the time it takes to locate and remove a single point from the structure.

Before these tests were performed a stable test environment was set up and the data structures were closely analysed to produce a set of expected results. The octree structure was also tested separately to ascertain the optimum maximum number of points within an octant. This was done via a form of white box testing known as mutation testing [62]. The number of points at which the octree subdivides was altered to see how this affects the insertion, draw and erase times of 1 millions points. In an environment measuring 1050 x 1680 x 150 it was felt that 1 million points represented an average drawing. While this is only a small proportion of the available points, as section 2.4 explained, in the case of rough sketches only a fraction of the available points may contain a voxel.

4.1   Test Environment

In order to obtain reliable results for the tests upon the performance times it was necessary to ensure a stable test environment was created. This was to ensure that there were no unwanted external factors influencing the results and the environment was identical for every test. The first task was to ensure that when the tests were run all unnecessary threads running upon the test computer were closed. The open threads were monitored throughout each test to ensure that this remained the case. Next it was vital to ensure that editable variables within Sketch 3D itself were constant for each version. Anti-aliasing was turned on, the size of the points drawn was set to 3 GLints, and finally the virtual display was set to a height of 1050, a width of 1680 and a depth of 150 voxels.

4.2   Analysis of Data Structures

Before the tests were run upon each separate version of the model, the underlying data structures were analysed and expected results produced. This meant that when it came to testing the structures for real it was possible to see if they were handling the data as expected or if an unforeseen factor was having an effect upon the results.

4.2.1   Insertion Time

Each data structure works in a slightly different way when it comes to the insertion of points. The list structure locates the next free block of memory and inserts the x, y, and z coordinates of the point one after the other. The 3D array simply locates the correct block of memory based upon the x, y, and z coordinates and alters it's value accordingly. Finally, the octree must use the x, y and z coordinates to ascertain which region of the scene it is located within. It will then attempt to add this point to the relevant octant. If the octant is not yet full the point will be added to the end of the list. However, if the octant has reached it's maximum size the octree must subdivide and the point will be added to the list contained within the correct newly created octant.

The method of inserting points into the list and the 3D array data structure should take a constant period of time, the number of points being added to the structure should not affect the way in which each points is added. Therefore the insertion time for a single point into these data structures will be $O(1)$. In the case of the octree it is expected that the insertion time will gradually increase as the number of points already contained within

the structure increases. The larger the depth of the octree the more nodes the insertion algorithm has to pass over to find the correct octant. As this operation requires time proportional to the height of the tree it is therefore expected that the insertion time of the octree will increase at a logarithmic rate. In mathematical terms it's running time is expected to be $O(\log n)$ where n represents the number of nodes within the tree.

4.2.2 Erase Time

The time taken to locate and remove points from each structure should vary considerably. For the 3D array the erase time should be equal to it's insertion time. This is due to the fact that it would follow exactly the same method as insertion. The coordinates will be used to identify the correct block of memory and it will be altered it accordingly. Therefore, the update time will also have a running time of $O(1)$.

It is in this area where a major pitfall of the list structure is discovered. As discussed earlier, new points inserted into the array are added to the end of the list. This leads to the ordering of the points within the array to be random. Therefore, to delete or alter a point from the list the entire list will have to be checked to see if that point actually exists within the array. Consequently the erase time will increase linearly as the size of the list increases. This gives the erase time of the list structure a running time of $O(t)$ where t represents the total number of points within the structure. While the same situation will occur within each octant of the octree, the fact that when these octant lists reach a certain size the octree will subdivide should alleviate the rapid increase expected in the list structures erase time. As the number of points within the octree grows the number of octants will increase rapidly. At first this will also cause the depth of the octree to increase rapidly. However this increase of depth will soon slow as the number of octants at each level becomes larger. As the erase time of the octree is dependant on it's depth it therefore expected that it will increase at a logarithmic rate, $O(\log n + k)$ where n is once again the number of nodes within the tree and k is the number of points within an octant.

4.2.3 Draw Time

The amount of time it takes to draw each data structure is expected to differ. As points are added to to the list structure, its length and size will increase. In turn this will increase the draw time of the data as the View class has to iterate over every point. The draw time will continue to increase as points are added until every point within the scene has been

September 25, 2011

added. This gives it a linear running time of $O(t)$ where t is the total number of points within the list. In contrast to this the draw time for the 3D array is directly related to the overall size of the structure, which we will refer to as S. Every single point within the scene will be iterated over for every frame giving it a running time of $O(S)$, regardless of the number of points added to it the draw time will remain equal to the size of the array. When the array is initialised it already contains data for all the points within the scene. As a user sketches the amount of data the array contains is never increased just altered.

The draw time of the octree is expected to start off similar to that of the list but increase with a run time of $O(q \log n)$ where q represents the total number of octants and n represents the number of points within the tree. As such it should gradually take an increasing amount of time in comparison to the list. While the algorithm which draws the list has to iterate over every drawn point, the octree algorithm has to do the same for but must traverse the tree in between each octant. As the size of the tree grows more and more time will be spent in this traversal phase.

4.3 Further Analysis of Octree

Before testing the data structures on their performance it is important to make sure that the most effective form of each structure is being tested. This is simple for the basic structures of the list and the 3D array as they are unchangeable. However, the octree contains variables which may alter the performance of it's insertion, draw and erase times. The most significant of these is the size an octant has to reach before it subdivides creating further octants.

Altering the maximum octant size would have a different effect on each of the performance tests. The aim of these further tests will be to find the most effective octant size to optimise the insertion, draw and erase times. While the octant size does not directly effect the insertion time it is possible that as the maximum octant size is increased the insertion time will actually speed up. This is due to the fact that if the octant size is larger the depth of the octree will be smaller for the same amount of points. This means that the insertion algorithm will have to traverse a smaller number of nodes to find the relevant octant. This is also true for the draw time algorithm which has to traverse the entire tree. The larger the maximum octant size the fewer nodes and therefore less time spent in traversal. However, in contrast to this the erase time is expected to increase as the maximum octant size grows. While there will be more nodes to traverse the time this

takes is expected to be marginal in comparison to the increased amount of time it will take to locate the point to be erased within each octant.

4.4 Performance of Data Structures

Once the test conditions were in place a high resolution timer from the Windows API was used to measure how long it took for the relevant algorithm to execute. This time was taken in seconds and recorded to an accuracy of five decimal places. Each time was measured three times and the average of these times calculated. This section will give an overview of the results in tabular and graphical form and explain how each test was carried out. For a full breakdown of all the results collected refer to Appendix A at the end of this paper.

4.4.1 Finding the Optimum Octant Size

Before performance tests were carried out upon each structure the optimum number of points at which an octant is subdivided was found. Firstly the time it takes to add one million points to the structure was measured. The location of each point was randomly generated so as to not manipulate the way in which the octree breaks down. Initially the maximum octant size was only a single point, this was then increased until it was equal to the number of points being inserted. This effectively turned the octree into a list structure as it did not need to subdivide at any point. The next task was to measure the draw time of an octree containing one million points. Once again this was done with varying maximum octant sizes, ranging from one to one million. Finally the erase time of was measured with the same varied octant sizes. An octree containing a million points was initialised and the erase algorithm then passed over this identifying and removing each point in a randomly selected order. The results of each of these performance tests can be seen below in table 4.1.

Table 4.1: The average performance times in seconds of 1 million points in an octree which subdivides at an increasing number of points per octant.

	Maximum points per octant:					
	1	**200,000**	**400,000**	**600,000**	**800,000**	**1,000,000**
Insertion Time	8.82016	6.43512	5.19871	3.31675	2.43649	1.53974
Draw Time	8.26584	5.93845	4.69685	3.08273	2.27991	1.44725
Erase Time	5.28961	28.63467	49.67341	62.63597	71.16459	77.78646

These results are represented graphically in part (a) of figure 4.1. It can clearly be seen that, as expected, the erase time grew rapidly as the maximum octant size was increased while the insertion and draw times both gradually dropped. It is interesting to note from this graph that when the octree structure contains a single node, effectively making it a list, the erase time becomes many times slower than the insertion and draw times.

To find the optimum number of points at which to limit the octants maximum size, the area of the graph where the performance times of each algorithm are closest was looked at. This area is shown on a larger scale in part (b) of figure 4.1. As highlighted upon the graph the optimum limit for octant size lies within the region of 23,110 and 27,212 points. The median point of this region, 25,161 points, was taken as the optimum maximum size of an octant within the octree and used within the performance tests detailed in section 4.4.2. This optimum will deliver the lowest draw times and insertion times without being compromised by the erase time.
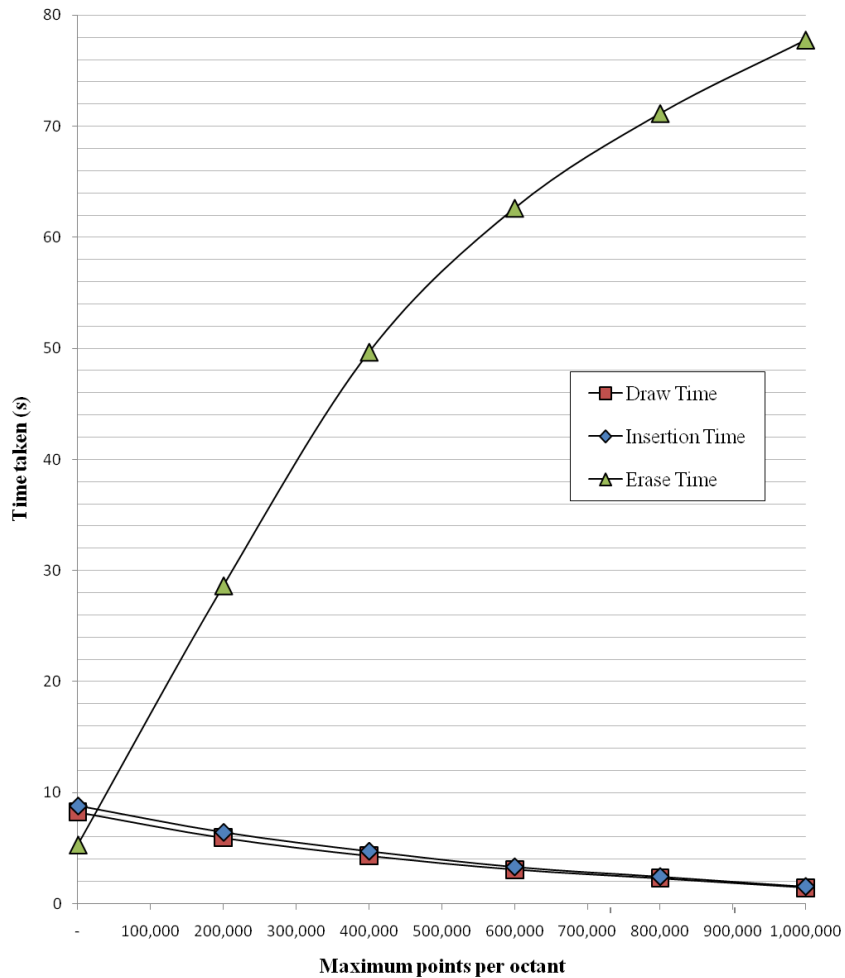
(a) The average performance times in seconds of 1 million points in an octree which subdivides at an increasing number of points per octant.
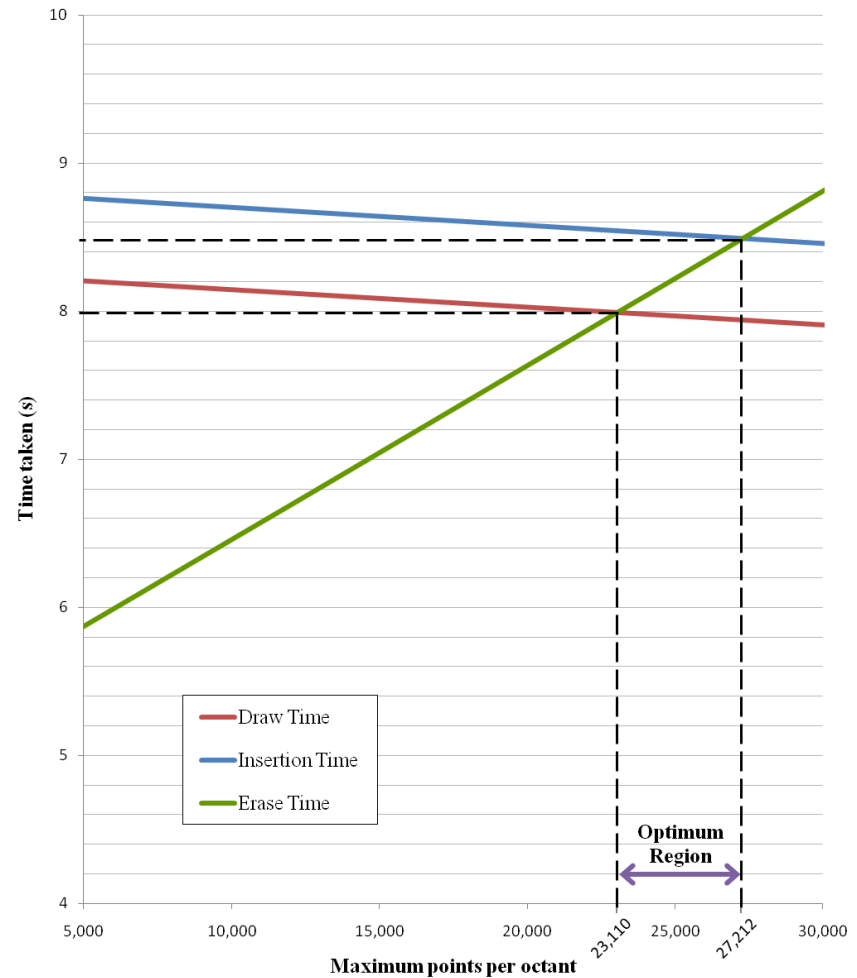
(b) The section of the graph in part (a) which contains the optimum region for the maximum number of points per octant.

Figure 4.1: Finding the optimum number of points at which an octree subdivides.

4.4.2 Data Structure Performance Times

Once the optimum octant size had been identified the performance of the three data structures were rigorously tested. The three areas focused upon were the algorithm which inserts points into each structure; the time it takes to traverse each structure and draw the points contained within; and finally the time it takes the erase algorithm to locate a point within each structure and remove it. Each test was performed upon an increasing number of points. This was to see how the structures coped as a user draws an ever increasing number of points into the Sketch3D system. The maximum number of points tested was three million as it was felt that this would represent a drawing of a reasonable size rather than just a rough sketch.

4.4.2.1 Insertion Time

The first test to be performed was a measure of the insertion time. The timer was begun and then varying amounts of points were inserted into each data structure. As soon as all of the points had been successfully inserted the timer was halted. The average insertion times for each structure are detailed below in table 4.2. As with the previous tests upon the octree the locations of the points were all randomly selected every time the test was run. This was to make sure the location of the points did not skew the results in any way.

Table 4.2: The average time in seconds taken to insert an increasing amount of points into each structure.

	Number of points inserted:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
List	0.82371	1.57284	2.41040	3.22752	4.03714	4.87132
3D Array	1.26768	2.64243	3.95831	5.29168	6.65121	8.07275
Octree	0.87267	1.87349	2.83680	3.79861	4.74975	5.70062

These results are represented graphically in figure 4.2 below. It can be seen that the insertion time for list structure and the 3D array increases at a constant rate as more points are inserted. The octree appears to differ slightly, the performance time of the algorithm initially increases quickly before gradually slowing to a similar rate of increase as the list. These results are subject to a detailed evaluation in the following chapter.
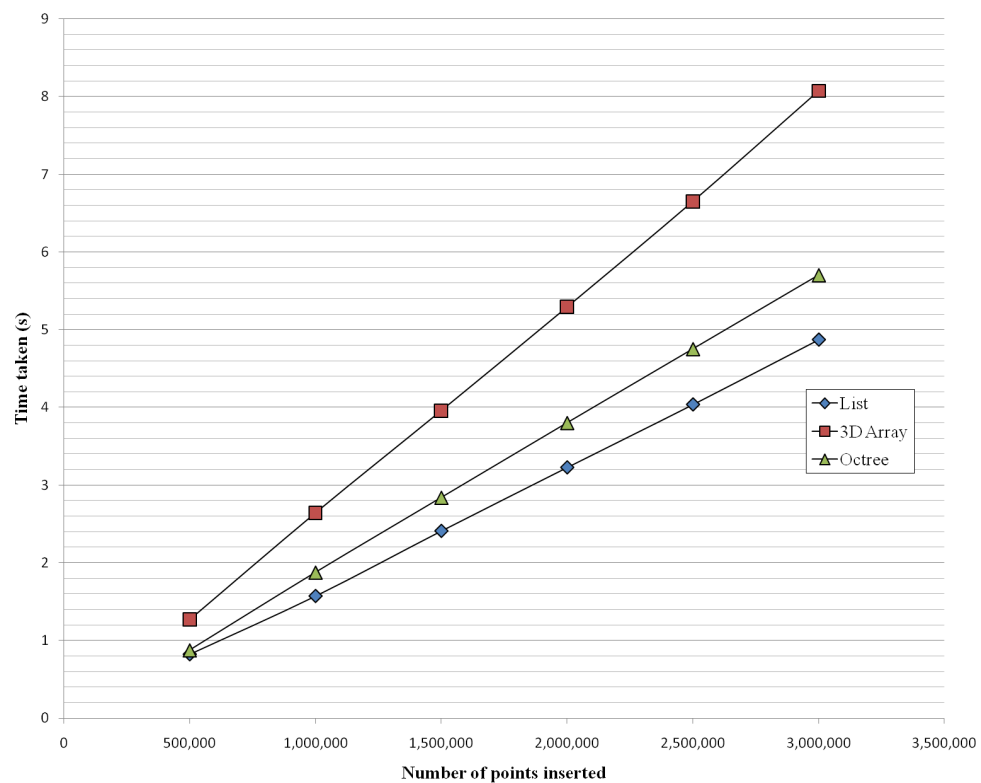
Figure 4.2: The average time in seconds taken to insert an increasing amount of points into each structure.

4.4.2.2 Erase Time

The erase algorithm takes a point as input, searches the data structure to see if this point exists, and if so removes it. To test it's performance each structure was initialised containing a varying amount of randomly located points. The timer was begun and this same series of points was fed into the erase algorithm. Once all points had been removed the timer was halted. The average times for each structure are shown in table 4.3.

Table 4.3: The average time in seconds taken to erase an increasing amount of points from each structure.

	Number of points erased:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
List	3.84325	14.36450	24.78577	35.97821	46.87256	57.38721
3D Array	1.26843	2.64199	3.95807	5.29265	6.64890	8.07272
Octree	1.92367	8.55216	16.18470	23.78211	30.50923	37.15962

Figure 4.3, shown below, clearly shows that while the erase time for the list and the
3D array increases at a constant rate, the rate of growth for the list structure is for larger
than that of the 3D array. The octree's erase time increases quickly at first as the number
of points to be found and removed increases. However, as the number of points tops 1.5
million the rate with which the algorithms speed is increasing begins to slow.
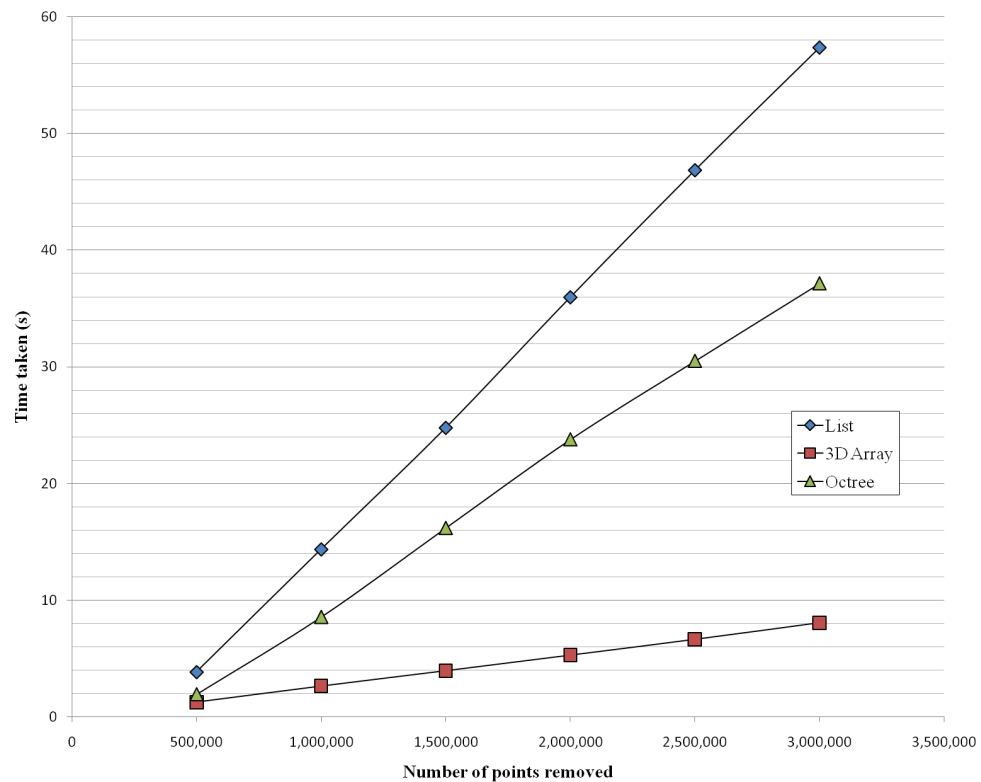
Figure 4.3: The average time in seconds taken to erase an increasing amount of points
from each structure.

4.4.2.3   Draw Time

Finally the performance of the Sketch3D's draw time was tested. This was done by
initialising the program with the data structure already filled by a varied number of
randomly located points. The timer was then initialised and left to run for a total of
500 iterations of the OpenGL rendering loop and in turn the algorithm which collects the
points within the structure and draws them to the screen. Once the final iteration had
been completed and the final scene processed and drawn the timer was stopped and the
result recorded. Table 4.4 below shows the average draw times for each data structure.

It can be seen from these results and also from the graph in figure 4.4 that the draw

Table 4.4: The average time in seconds taken to draw an increasing amount of points from each structure.

	Number of points drawn:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
List	1.27435	6.56421	11.87354	17.14380	22.48577	27.70285
3D Array	320.25542	320.18635	320.34008	320.33182	320.29967	320.37802
Octree	1.82367	8.25216	13.88443	19.90738	25.92764	32.36942

time of the 3D array remain constant regardless of the number of points stored. Meanwhile the draw time of the list structure grows at an increasing rate as the number of points stored increases. Similarly to this the octree's draw time grows very gradually at an increasing rate as the number of points stored increases. The reasons behind these rates of growth are explored in depth in the following chapter.
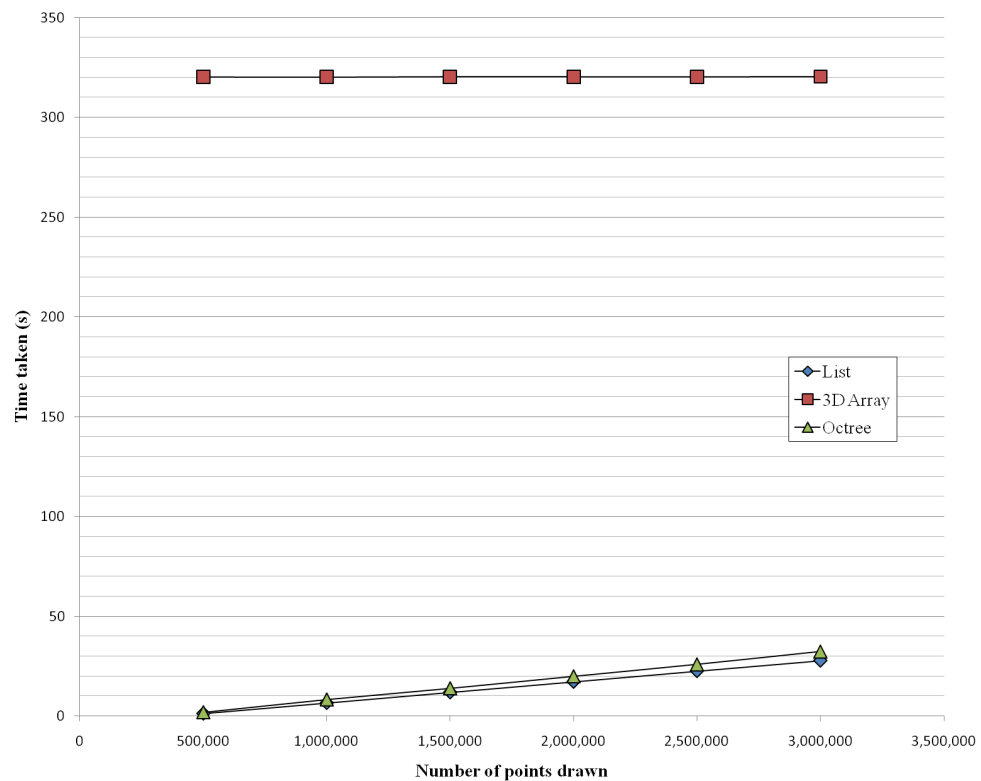
Figure 4.4: The average time in seconds taken to draw an increasing amount of points from each structure.

Chapter 5

Evaluation

This chapter discusses the strengths and weaknesses of the outcomes produced and provide an in depth evaluation of the results presented in Chapter 4. The results obtained over the course of the project were largely as expected following the initial analysis. There were however a few outcomes which differed from this. This section will try to explain the reasons for these differing results and also how this relates to the overall efficiency of the software. It will also try to use the collated results to ascertain which of the data structures is the most effective and should therefore be used as a basis for Sketch3D as the software is developed going forward.

5.1 Insertion Times

Using the obtained results we can see that the insertion time for each data structure was exactly as predicted during the analysis in the previous chapter. The time it takes to insert a point into the list and 3D array increases at a constant rate regardless of how many points are already stored within the structure, giving the insertion algorithms a running time of $O(1)$. As the number of points added increases the total time taken to add the points also increases at a constant rate demonstrating that the insertion time for each individual point remains unchanged. The insertion time into the 3D array is larger than the time it takes for the algorithm to insert a point into the list structure. This must be related to the method the C programming language uses to access each type of structure as there is nothing in the algorithms used which would indicate a reason for such a result.

While the octree's insertion time is similar to that of the list structure the runtime of

it's insertion algorithm is marginally worse. The amount of time it takes to insert points into the list is unaffected by the number of points already stored in the structure while the octree's insertion algorithm increases gradually as the number of points increases. This will be due to number of nodes which make up the octree structure. As more points are added the number of nodes and octants will grow. Unlike the the algorithm for the list structure the octree must first locate the relevant octant before inserting a point. As the number of nodes increases the time it takes to traverse the tree to the relevant octant also increases at a rate of $O(\log n)$. However, as this rate of growth is so small it means the list structure is only marginally more efficient for all quantities of points tested.

5.2   Erase Times

Comparing the data recorded relating to the data structures erase times we can see that once again the prior analysis was correct. The 3D array proves very effective as it's erase times are virtually identical to it's insertion times. This is down to the fact the algorithm to add and remove points from the array is identical. In both scenarios it is just a case of locating the correct point based on it's coordinate values and then altering it's value to a zero or one accordingly giving it an erase time of $O(1)$

The erase time for the list also increases at a constant rate. However, in a reversal of efficiency from the insertion time, it increases at a much faster rate than that of the 3D array. This is due to the fact that while the 3D array can easily locate the point to be removed using it's coordinates, the algorithm contained within the list structure has to compare the point with all other stored points in turn until it locates the correct one. This particular result was to be expected as the running time of a search algorithm on an unsorted array such as the list is always $O(t)$ where t is the size of the list.

The performance of the erase algorithm upon the octree differs significantly from the other two structures. The results show that, as expected during the initial analysis, it's growth is $O(\log n + k)$ where n is once again the number of nodes within the tree and k is the number of points within an octant. Initially the structure grows in the same manner as the list as it is also simply locating a point in an unsorted array. However once the maximum octant size is exceeded the algorithm will be searching within an array of the same size each time. The only difference will be the size of the octree and the number of nodes which have to be traversed to identify the correct octant. As has been seen various times already this increase in nodes to traverse only adds marginal amounts of time to

each algorithms run time.

5.3   Draw Times

Comparing the results collected to the analysis of the draw time for each data structure it is clear that something unexpected is going on. The 3D array acts as expected. The draw time remains constant even as the number of points stored within also increased. As stated earlier, this is because the algorithm used within the View class iterates over every point contained within the three dimensional scene rather than just those points which have been drawn by a user. This gives it a running time of $O(S)$ where S is the total size of the array. As such, the total amount of data added to the array does nothing to impact the amount of time it takes to iterate over and draw the array. In terms of stability a drawing algorithm which is unaffected by the amount of points contained would be perfect for a piece of software such as Sketch3D, however while the 3D array offers this stability the trade off is that each frame is drawn so slowly that it renders the program virtually unusable.

The draw time of the list like structure was very impressive. As expected it there was an increase as the number of points stored increased, giving it a linear running time of $O(t)$ where t is the total number of points within the list.

The algorithm to draw the octree structure also performs as the prior analysis suggested. The draw time grows at a gradually increasing rate. This gradual increase will be due to the increasing size and hence number of octants within the octree. The draw algorithm will take $\log(n)$ time to reach each octant where n is the number of nodes within the tree, this occurs q times giving it a run time of $O(q \log n)$ where q represents the total number of octants. This rate of growth is only marginally higher than that of the list structure and as the increase advances at such a slow rate it is unlikely it would ever reach the painfully slow draw times of the 3D array.

5.4   Comparison of Data Structures

From this evaluation we can see that each data structure has it's strengths and weaknesses. The 3D array offers a good insertion time coupled with a very impressive erase time. Unfortunately these positives are outweighed by the arrays very poor performance when it comes to actually collecting and displaying the results. In comparison the list structure

has an even lower insertion speed and a very effective draw time for the various quantity of points which were subject to testing. However, as with the 3D array there is a pitfall and in this case it's the lists erase time. It not only begins at a slower rate than the other two structures but then increases rapidly.

Finally we have the octree structure. While the erase time is not quite as impressive as that of the 3D array it is still a great amount more efficient than the list structure. It's insertion speed is only marginally slower than the list's algorithm and, although it does not increase at a constant rate, it grows so slowly so as to be negligible. The last area to consider is the draw time and the results clearly show that the octree provides a far quicker solution than both the 3D array and is only slightly slower than the list.

From this evaluation it seems we can state with confidence that the octree data structure is the most efficient of those tested. It is well suited to act as an effective storage structure for the points used to represent digital ink within Sketch3D. It is able to insert, erase and draw points quickly and efficiently.

Chapter 6

Conclusion

Over the course of this research a number of main points have become clear. Although the scope of this project covered a variety of issues these are just the basis for further research in this area. Many issues touched upon in this paper will require a more in depth analysis. This section will review the main topics discussed within this paper and also detail the further research which needs to be conducted.

6.1   Summary of Paper

The paper begins with a thorough review of all the work already conducted in this area. It was found that, while various pieces of software have been developed to facilitate converting three dimensional sketches to models or to manipulate NURBS curves and draw surface based graphics, very little has been done towards developing a freehand 3D sketching tool since Poletti in 1995 [66]. This is surprising when freehand two dimensional raster based drawing packages are so numerous and popular.

A system, known as Sketch3D, was developed and implemented in C++ using OpenGL. This tool is based upon a MVC architecture and allows a user to draw freely in a three dimensional environment using the keyboard and mouse in unison. Sketch3D also contains a small number of features designed to improve the user experience. These include a three dimensional cursor, a toggle which indicates the users current depth and also the ability to alter various attributes of the ink such as size and colour.

A series of test were then run upon Sketch3D to ascertain the most efficient way to store the digital ink within the software. The storage structures investigated were a simple list structure, a 3D array and an octree. They were tested on the amount of time it took

to add and erase points from the structure and finally the time it took to collect all the points stored and draw them to the screen. It was found that while the list and 3D array excelled in certain areas the octree was the all round most effective structure.

6.2   Areas of Further Research

The research conducted as part of this project was heavily focussed upon the storage of digital ink within a voxel based drawing program. Now that an efficient solution has been identified there are a number of other areas which would need research to create a fully functioning three dimensional drawing package.

The most significant of these areas would be the creation of an effective three dimensional user interface. The features presented within this paper, while highly effective, are only very basic. The sketch package would require a fully functioning graphical user interface (GUI) which allowed the user to not only sketch freely but also make effective use of features such as changing line colour and thickness. As more features are added to the software the GUI would need to be able to support these updates.

Along with an effective user interface there are still certain areas relating to the back end of the software which require further work. Certain features which are commonly found in drawing packages will rely heavily on interactions with the structure storing the digital ink. The ability to cut and paste points would be an interesting challenge as how the user selects the three dimensional region to cut would have to be carefully considered. Another important feature which Sketch3D is currently lacking is the ability to save and load images once they have been drawn. As storing the data for the entire scene in it's raw form, as in the 3D array, would lead to an impractically large file it is hoped that the octree representation would alleviate this and create a more manageable file.

The saving and loading of images would tie into a further look at various ways to represent digital ink more effectively than shown in this paper via the further compression of voxel structures. This could possibly be achieved by implementing run length encoding on a 3D array or storing all of the points from an octree in a separate list so they can be iterated over and drawn quickly.

Bibliography

[1] S.H. Bae, R. Balakrishnan, and K. Singh. ILoveSketch: as-natural-as-possible sketching system for creating 3D curve models. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 151–160. ACM New York, NY, USA, 2008.

[2] S.H. Bae, R. Balakrishnan, and K. Singh. EverybodyLovesSketch: 3D sketching for a broader audience. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 59–68. ACM, 2009.

[3] J.A. Bærentzen and N.J. Christensen. Volume sculpting using the level-set method. In *International Conference on Shape Modeling and Applications*, pages 175–182, 2002.

[4] R. Baker. *Designing the future: the computer transformation of reality.* Thames and Hudson, 1993.

[5] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):517, 1975.

[6] O. Bimber, L.M. Encarnacáo, and D. Schmalstieg. Augmented reality with back-projection systems using transflective surfaces. In *Computer Graphics Forum*, volume 19, pages 161–168. Citeseer, 2000.

[7] O. Bimber, LM Encarnacao, and A. Stork. A multi-layered architecture for sketch-based interaction within virtual environments. *Computers and Graphics*, 24(6):851–867, 2000.

[8] J.F. Blinn. Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292. ACM New York, NY, USA, 1978.

[9] D. Bourguignon, M.P. Cani, and G. Drettakis. Drawing for illustration and annotation in 3D. In *Computer Graphics Forum*, volume 20, pages 114–123. Citeseer, 2001.

[10] S. Bramly. *Leonardo: the artist and the man.* Penguin Group USA, 1994.

[11] P.M. Campbell, K.D. Devine, J.E. Flaherty, L.G. Gervasio, and J.D. Teresco. Dynamic octree load balancing using space-filling curves. *Williams College Department of Computer Science Technical Report CS-03-01*, 2003.

[12] Y-M. Chee, K. Franke, M. Froumentin, S. Madhvanath, J-A. Magaa, G. Russell, G. Seni, C. Tremblay, S.M. Watt, and L. Yaeger. Ink Markup Language (InkML). Working draft, W3C, October 2006.

[13] J.J. Cherlin, F. Samavati, M.C. Sousa, and J.A. Jorge. Sketch-based modeling with few strokes. In *Proceedings of the 21st spring conference on Computer graphics*, page 145. ACM, 2005.

[14] N.R. Chrisman. *Exploring geographic information systems.* John Wiley & Sons Inc, 2001.

[15] MB Clowes. On seeing things. *Artificial Intelligence*, 2(1):79–116, 1971.

[16] J.M. Cohen, J.F. Hughes, and R.C. Zeleznik. Harold: A world made of drawings. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 83–90. ACM New York, NY, USA, 2000.

[17] J.M. Cohen, L. Markosian, R.C. Zeleznik, J.F. Hughes, and R. Barzel. An interface for sketching 3D curves. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 17–21. ACM New York, NY, USA, 1999.

[18] M.T. Cook and A. Agah. A survey of sketch-based 3-D modeling techniques. *Interacting with Computers*, 2009.

[19] S.C. Corlett. Steroscopic Sketchpad, Dissertation for BSc from University of Durham. 2009.

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

[21] M.F. Deering. The HoloSketch VR sketching system. 1996.

[22] N.A. Dodgson, JR Moore, and SR Lang. Multi-view autostereoscopic 3D display. In *International Broadcasting Convention*, volume 99. Citeseer, 1999.

[23] L. Eggli, B.D. Brüderlin, and G. Elber. Sketching as a solid modeling tool. In *Proceedings of the third ACM symposium on Solid modeling and applications*, pages 313–322. ACM New York, NY, USA, 1995.

[24] K. Engel, M. Hadwiger, J.M. Kniss, A.E. Lefohn, C.R. Salama, and D. Weiskopf. Real-time volume graphics. In *ACM Siggraph 2004 Course Notes*, pages 29–es. ACM, 2004.

[25] F. Escobar, A.P.G. Hunter, A.P.I. Bishop, and A. Zerger. Introduction to GIS. *Department of Geomatics, The University of Melbourne, USA*, 2008.

[26] E. Ferley, M.P. Cani, and J.D. Gascuel. Practical volumetric sculpting. *The Visual Computer*, 16(8):469–480, 2000.

[27] S.F. Frisken, R.N. Perry, A.P. Rockwood, and T.R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[28] T.A. Galyean and J.F. Hughes. Sculpting: An interactive volumetric modeling technique. *ACM SIGGRAPH Computer Graphics*, 25(4):267–274, 1991.

[29] D. Gerrety. JOT - A Specification for an Ink Storage and Interchange Format). Technical report, Slate Corporation, March 1996.

[30] IJ Grimstead and RR Martin. Creating solid models from single 2D sketches. In *Proceedings of the third ACM symposium on Solid modeling and applications*, pages 323–337. ACM New York, NY, USA, 1995.

[31] T. Grossman, R. Balakrishnan, G. Kurtenbach, G. Fitzmaurice, A. Khan, and B. Buxton. Interaction techniques for 3D modeling on large displays. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, page 23. ACM, 2001.

[32] T. Grossman, R. Balakrishnan, G. Kurtenbach, G. Fitzmaurice, A. Khan, and B. Buxton. Creating principal 3D curves with digital tape drawing. In *Proceedings of the*

SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves, pages 121–128. ACM New York, NY, USA, 2002.

[33] I. Guyon, L. Schomaker, R. Plamondon, M. Liberman, and S. Janet. UNIPEN project of on-line data exchange and recognizer benchmarks. In *International Conference on Pattern Recognition*, pages 29–29. Citeseer, 1994.

[34] S. Han and G. Medioni. 3DSketch: modeling by digitizing with a smart 3D pen. In *Proceedings of the fifth ACM international conference on Multimedia*, pages 41–49. ACM New York, NY, USA, 1997.

[35] N.S. Holliman. Auto-stereoscopic 3D Display Designs, 2007. https://www.dur.ac.uk/n.s.holliman/3dDisplayTypes.html.

[36] Stephen M. Hollister. The Dirty Little Secrets of NURBS, 2001. Copyright of New Wave Systems, Inc., http://http://www.pilot3d.com/NurbSecrets.htm.

[37] DA Huffman. Impossible Objects as Nonsense Sentences. *Machine Intelligence*, page 295, 1971.

[38] G.M. Hunter. *Efficient computation and data structures for graphics.* PhD thesis, Princeton University Princeton, NJ, USA, 1978.

[39] T. Igarashi and J.F. Hughes. A suggestive interface for 3D drawing. In *ACM SIG-GRAPH 2007 courses*, page 20. ACM, 2007.

[40] T. Igarashi, S. Matsuoka, and H. Tanaka. Teddy: a sketching interface for 3D freeform design. In *ACM SIGGRAPH 2007 courses*, page 21. ACM, 2007.

[41] R. Jarrett and P. Su. Building Tablet PC Applications. 2002.

[42] C. Johnson. *Harold and the purple crayon.* HarperCollins, 1981.

[43] G. Jones, D. Lee, N. Holliman, and D. Ezra. Controlling perceived depth in stereo-scopic images. In *Stereoscopic Displays and Virtual Reality Systems VIII, Proceedings of SPIE*, volume 4297, pages 42–53. Citeseer, 2001.

[44] O. Karpenko, J.F. Hughes, and R. Raskar. Free-from sketching with variational implicit surfaces. In *Eurographics 2002.* Citeseer, 2002.

[45] A. Kaufman, D. Cohen, and R. Yagel. Volume graphics. *Computer*, 26(7):51–64, 1993.

[46] B. Kerautret, X. Granier, and A. Braquelaire. Intuitive shape modeling by shading design. *Lecture notes in computer science*, 3638:163, 2005.

[47] A. Knoll, I. Wald, S. Parker, and C. Hansen. Interactive isosurface ray tracing of large octree volumes. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 115–124. IEEE, 2006.

[48] A.M. Knoll. *Ray tracing implicit surfaces for interactive visualization.* PhD thesis, The University of Utah, 2009.

[49] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. *IEEE Visualization, 2003. VIS 2003*, pages 287–292, 2003.

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

[51] J. Lawrence and T. Funkhouser. A painting interface for interactive surface deformations. *Graphical Models*, 66(6):418–438, 2004.

[52] S. Lehar. Gestalt isomorphism and the primacy of subjective conscious experience: A Gestalt Bubble model. *Behavioral and Brain Sciences*, 26(04):375–408, 2003.

[53] F. Levet, X. Granier, and C. Schlick. 3d sketching with profile curves. *Lecture Notes in Computer Science*, 4073:114, 2006.

[54] J. Leymarie, G. Monnier, and B. Rose. History of an art: Drawing, 1979.

[55] H. Lipson and M. Shpitalni. Correlation-based reconstruction of a 3D object from a single freehand sketch. In *AAAI spring symposium on sketch understanding*, pages 99–104, 2002.

[56] K. McArthur. *Pro PHP: Patterns, Frameworks, Testing and More.* APress, 2008.

[57] D. Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. *Electrical and Systems Engineering IPL-TR-80-111, Rensselaer Polytechnic Institute, Troy, NY*, 1980.

[58] P. Michalik, D.H. Kim, and B.D. Bruderlin. Sketch-and constraint-based design of B-spline surfaces. In *Proceedings of the seventh ACM symposium on Solid modeling and applications*, page 304. ACM, 2002.

[59] D.E.L.A.F. Mike and M. Bridgette. Digital sculpting with mudbox: essential tools and techniques for artists. *Documentaliste Sciences de l'information*, 46(3), 2009.

[60] K. Mueller, T. Möller, and R. Crawfis. Splatting without the blur. In *Proceedings of the conference on Visualization'99: celebrating ten years*, pages 363–370. IEEE Computer Society Press Los Alamitos, CA, USA, 1999.

[61] A. Oakman. Volume Graphics: The road to interactive medical imaging? *SURPRISE 96 Journal*, 2(2), 1996.

[62] A.J. Offutt. A practical system for mutation testing: help for the common programmer. In *Test Conference, 1994. Proceedings., International*, pages 824–830. IEEE, 2002.

[63] S. Owada, F. Nielsen, K. Nakazawa, and T. Igarashi. A sketching interface for modeling the internal structures of 3d shapes. In *ACM SIGGRAPH 2007 courses*, page 38. ACM, 2007.

[64] J. Pereira, J. Jorge, V. Branco, and F.N. Ferreira. Towards calligraphic interfaces: sketching 3D scenes with gestures and context icons. *Proceedings of WSCG00*, 2000.

[65] J.P. Pereira, V.A. Branco, J.A. Jorge, N.F. Silva, T.D. Cardoso, and F.N. Ferreira. Cascading recognizers for ambiguous calligraphic interaction. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, volume 174, 2004.

[66] H. Poletti. A 3D sketching tool: An MA computing in design research project. *ACM SIGGRAPH Computer Graphics*, 29(3):27, 1995.

[67] W. Robson. Essential ZBrush, Pap/DVD edition. *Wordware Game And Graphics Library*, page 754, 2008.

[68] H. Rushmeier, J. Gomes, L. Balmelli, F. Bernardini, and G. Taubin. Image-based object editing. In *Fourth International Conference on*. Citeseer, 2003.

[69] E. Sachs, A. Roberts, and D. Stoops. 3-Draw: a tool for designing 3D shapes. *IEEE Computer Graphics and Applications*, 11(6):18–26, 1991.

[70] H. Samet. *Foundations of multidimensional and metric data structures.* Morgan Kaufmann, 2006.

[71] H. Samet. Spatial data structures. In *International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2007 courses: San Diego, California.* Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA, 2007.

[72] S. Schkolne, M. Pruett, and P. Schröder. Surface drawing: creating organic 3D shapes with the hand and tangible tools. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 261–268. ACM New York, NY, USA, 2001.

[73] R. Schmidt, B. Wyvill, MC Sousa, and JA Jorge. Shapeshop: Sketch-based solid modeling with blobtrees. In *ACM SIGGRAPH 2006 Courses*, page 14. ACM, 2006.

[74] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL programming guide: the official guide to learning OpenGL, version 2.1.* Addison-Wesley, 2007.

[75] K. Singh and E. Fiume. Wires: a geometric deformation technique. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 405–414. ACM New York, NY, USA, 1998.

[76] G. Sun and N. Holliman. Evaluating methods for controlling depth perception in stereoscopic cinematography. In *Proceedings of SPIE*, volume 7237, page 72370I, 2009.

[77] H. Sundar, R.S. Sampath, S.S. Adavani, C. Davatzikos, and G. Biros. Low-constant parallel algorithms for finite element simulations using linear octrees. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing.* ACM New York, NY, USA, 2007.

[78] I. Sutherland. Sketchpad. In *A man-machine graphical communication system. In the AFIPS Spring Joint Computer Conference*, volume 23, pages 329–346, 1963.

[79] C.L. Tai, H. Zhang, and J.C.K. Fong. Prototype modeling from sketched silhouettes based on convolution surfaces. In *Computer Graphics Forum*, volume 23, pages 71–83. Citeseer, 2004.

[80] Thetawave. Volume_ray_casting.png, January 2006. Licensed under the terms of the GNU Free Documentation License. http://upload.wikimedia.org/wikipedia/commons/e/ec/volume_ray_casting.

[81] S. Tsang, R. Balakrishnan, K. Singh, and A. Ranjan. A suggestive interface for image guided 3D sketching. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 591–598. ACM New York, NY, USA, 2004.

[82] A. Turner, D. Chapman, and A. Penn. Sketching space. *Computers & Graphics*, 24(6):869–879, 2000.

[83] PAC Varley, RR Martin, and H. Suzuki. Can machines interpret line drawings? In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*. Citeseer, 2004.

[84] S.M. Watt. New aspects of InkML for pen-based computing. In *International Conference on Document Analysis and Recognition. IEEE Computer Society*, pages 457–460, 2007.

[85] G. Wesche and H.P. Seidel. FreeDrawer: a free-form sketching system on the responsive workbench. In *Proceedings of the ACM symposium on Virtual reality software and technology*, page 174. ACM, 2001.

[86] L. Williams. 3D paint. In *Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 225–233. ACM New York, NY, USA, 1990.

[87] X. Wu. *Achieving interoperability of pen computing with heterogeneous devices and digital ink formats.* PhD thesis, The University of Western Ontario, 2004.

[88] B. Wyvill, E. Galin, A. Guy, M. Tigges, B. Wyvill, L.L. Fedenczuk, B. Wyvill, R. Zonenschein, J. Gomes, L. Velho, et al. The Blob Tree, Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *Proc. SIBGRAPI, IMPA, Rio De Janero, Brazil*, 3, 1998.

[89] R.C. Zeleznik, K.P. Herndon, and J.F. Hughes. SKETCH: An interface for sketching 3D scenes. In *ACM SIGGRAPH 2006 Courses*, page 9. ACM, 2006.

[90] S. Zhai. User performance in relation to 3D input device design. *ACM Siggraph Computer Graphics*, 32(4):50–54, 1998.

Appendix A

Complete Results

A.1 Optimum Octant Size

A.1.1 Octant Insertion Times

Table A.1: Effect of the maximum points per octant on Octree insertion time (s).

	Number of points inserted:					
	1	200,000	400,000	600,000	800,000	1,000,000
1st Test	8.91532	6.43806	5.08399	3.30941	2.60159	1.55832
2nd Test	8.84750	6.36983	5.17419	3.36275	2.32242	1.52629
3rd Test	8.69766	6.49747	5.33795	3.27809	2.38546	1.53461
Average	8.82016	6.43512	5.19871	3.31675	2.43649	1.53974

A.1.2 Octant Draw Times

Table A.2: Effect of the maximum points per octant on Octree draw time (s).

	Number of points drawn:					
	1	200,000	400,000	600,000	800,000	1,000,000
1st Test	7.93083	5.93017	4.71307	2.94769	2.34552	1.44610
2nd Test	8.41399	6.11643	4.67735	3.25137	2.25103	1.23245
3rd Test	8.45270	5.76875	4.70013	3.04913	2.24318	1.66320
Average	8.26584	5.93845	4.69685	3.08273	2.27991	1.44725

A.1.3   Octant Erase Times

Table A.3: Effect of the maximum points per octant on Octree erase time (s).

	Number of points erased:					
	1	**200,000**	**400,000**	**600,000**	**800,000**	**1,000,000**
1st Test	5.01903	28.63562	49.81039	62.56910	70.89023	77.79268
2nd Test	5.48902	28.57234	49.62146	62.58567	71.39938	77.61348
3rd Test	5.36078	28.69605	49.58838	62.75314	71.20416	77.95322
Average	5.28961	28.63467	49.67341	62.63597	71.16459	77.78646

A.2   Insertion Times

A.2.1   List Structure

Table A.4: Time taken to insert points into the list structure (s).

	Number of points inserted:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
1st Test	0.82352	1.57415	2.41028	3.22687	4.03715	4.86778
2nd Test	0.82401	1.57288	2.40935	3.22717	4.03684	4.87065
3rd Test	0.82360	1.57149	2.41157	3.22852	4.03743	4.87553
Average	0.82371	1.57284	2.41040	3.22752	4.03714	4.87132

A.2.2   3D Array

Table A.5: Time taken to insert points into the 3D array (s).

	Number of points inserted:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
1st Test	1.26751	2.64199	3.97154	5.29193	6.64846	8.07264
2nd Test	1.26806	2.64271	3.95162	5.29168	6.65187	8.07098
3rd Test	1.26747	2.64259	3.95177	5.29143	6.65330	8.07463
Average	1.26768	2.64243	3.95831	5.29168	6.65121	8.07275

September 25, 2011

A.2.3   Octree

Table A.6: Time taken to insert points into the octree (s).

	Number of points inserted:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
1st Test	0.86541	1.83490	2.83428	3.74206	4.74861	5.75410
2nd Test	0.87524	1.89822	2.84058	3.84867	4.75999	5.65745
3rd Test	0.87736	1.88735	2.83554	3.80510	4.74065	5.69031
Average	0.87267	1.87349	2.83680	3.79861	4.74975	5.70062

A.3   Erase Times

A.3.1   List Structure

Table A.7: Time taken to remove points from the list structure (s).

	Number of points erased:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
1st Test	3.68536	14.29574	24.92028	35.82240	46.65879	57.86590
2nd Test	3.89082	14.48002	24.72435	36.26854	46.92376	57.38746
3rd Test	3.95357	14.31774	24.71268	35.82240	46.92376	56.90827
Average	3.84325	14.36450	24.78577	35.97821	46.87256	57.38721

A.3.2   3D Array

Table A.8: Time taken to remove points from the 3D array (s).

	Number of points erased:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
1st Test	1.26765	2.63752	3.93590	5.29366	6.64725	8.07076
2nd Test	1.26806	2.64885	4.00374	5.29243	6.64810	8.06068
3rd Test	1.26958	2.63960	3.93457	5.29186	6.65135	8.08672
Average	1.26843	2.64199	3.95807	5.29265	6.64890	8.07272

A.3.3   Octree

Table A.9: Time taken to remove points from the octree (s).

	Number of points erased:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
1st Test	1.68266	8.65893	16.85292	23.55372	31.67038	37.15324
2nd Test	1.84310	8.75207	15.54210	23.75664	29.86720	37.79087
3rd Test	2.24525	8.65893	16.15908	24.03597	29.99011	36.53475
Average	1.92367	8.55216	16.18470	23.78211	30.50923	37.15962

A.4   Draw Times

A.4.1   List Structure

Table A.10: Time taken to draw points from the list structure (s).

	Number of points drawn:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
1st Test	1.27121	6.59542	11.87153	17.12215	22.42495	27.73458
2nd Test	1.27925	6.57854	11.87268	17.15541	22.56396	27.69419
3rd Test	1.27259	6.51867	11.87641	17.15384	22.46840	27.67978
Average	1.27435	6.56421	11.87354	17.14380	22.48577	27.70285

A.4.2   3D Array

Table A.11: Time taken to draw points from the 3D array (s).

	Number of points drawn:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
1st Test	320.18742	320.28745	320.25679	320.35185	320.25509	320.39120
2nd Test	320.36794	320.20412	320.39868	320.29741	320.33528	320.38238
3rd Test	320.21097	320.06748	320.36477	320.34620	320.30864	320.36048
Average	320.25542	320.18635	320.34008	320.33182	320.29967	320.37802

September 25, 2011

A.4.3 Octree

Table A.12: Time taken to draw points from the octree (s).

	Number of points drawn:					
	500,000	**1,000,000**	**1,500,000**	**2,000,000**	**2,500,000**	**3,000,000**
1st Test	1.82534	8.24964	13.89597	19.94251	25.97121	32.37413
2nd Test	1.82235	8.25271	13.90247	19.89553	25.89740	32.35928
3rd Test	1.82332	8.25413	13.85485	19.88410	25.91431	32.37485
Average	1.82367	8.25216	13.88443	19.90738	25.92764	32.36942