

**Mark Streatfield**  
a1490378



**A Study and Implementation of the  
Three Dimensional Digitisation of Objects**



**Innovations Project**



5 Dalmeny Road  
Worcester Park  
Surrey  
KT4 8UU  
England



0 44 7771566553



markstreatfield@blueyonder.co.uk



www.markstreatfield.pwp.blueyonder.co.uk

## Table of Contents

<b>1 Abstract</b>	<b>1</b>
<b>2 Introduction</b>	<b>2</b>
<b>3 Three Dimensional Digitisation</b>	<b>4</b>
3.1 The Modelling Process	
3.2 The Language of Modelling	
3.3 Spatial Descriptions	
3.4 Alternative Methods of Digitisation	
<b>4 Hardware Based Acquisition Methods</b>	<b>7</b>
4.1 Magnetic Resonance Imaging	
4.2 Computerized Axial Tomography	
4.3 Laser Scanners	
4.4 Touch Probe	
4.5 Profilers	
4.6 Structured Light Scanner	
4.7 Laser Trackers, Theodolites and Total Stations	
4.8 Other Considerations	
<b>5 Software Based Acquisition Methods</b>	<b>14</b>
5.1 The "Teddy" Sketching Interface	
5.2 Solid Models from Two Dimensional Sketches	
5.3 Commercial Software Applications	
5.4 Other Considerations	
<b>6 Photogrammetry</b>	<b>19</b>
6.1 Stereoscopic Vision in Humans	
6.2 The Digital Equivalent	
6.3 Approaches for Feature Matching in Stereoscopic Images	
6.4 Examples of Stereoscopic Photogrammetry in Use	
<b>7 Simple Implementation</b>	<b>22</b>
7.1 Creating Three Dimensional Geometry in Theory	
7.2 Creating Three Dimensional Geometry in Code	
7.3 Evaluating the Results	
7.4 Extensions and Improvements	
<b>8 Conclusion</b>	<b>28</b>
<b>9 Acknowledgements</b>	<b>29</b>
<b>10 References</b>	<b>30</b>
<b>Appendix A Verbal Description of a Vase</b>	<b>32</b>
<b>Appendix B Teddy : A Sketching Interface for 3D Freeform Design</b>	<b>33</b>
<b>Appendix C Examples of Stereoscopic Images</b>	<b>34</b>
<b>Appendix D Results</b>	<b>35</b>
<b>Appendix E User Manual</b>	<b>36</b>
<b>Appendix F Makefile Code Listing</b>	<b>37</b>
<b>Appendix G Digit.c Code Listing</b>	<b>38</b>

## Section 1 - Abstract

This document presents research into the three dimensional digitisation of physical objects. We also present a possible implementation of such a system. The research was been divided into two separate categories: digitisation through a hardware based procedure (such as a laser scan) and digitisation through a more software orientated approach. After examining the various techniques available to the digital artist, a possible approach is described for simple geometry reconstruction in the digital domain using stereo image pairs and a calculation of disparity. While this is arguably not the most effective method of three-dimensional digitisation, improvements and extensions are suggested before conclusions are drawn.

Although examples and explanations will naturally gravitate towards the film/special effects industry (as this is the author's current area of study) the concept of digitisation will be linked to other areas where computer graphics are of use. Furthermore, while the term *computer generated imagery* is generally associated with such examples as *Toy Story*, *Shrek* and blockbuster special effects, it is used here in its more accurate meaning - any image that has been created using a computer, for any purpose.

## Section 2 - Introduction

The use of computer-generated imagery has become commonplace in almost every area of our day-to-day lives. Whilst the most obvious uses are in the entertainment industry (creating films, television programmes, computer games and virtual reality applications) it stretches to far more diverse environments such as archaeology and architecture, dentistry and education, fashion and forensics, medicine and even children's toys [www.simple3d.com]. Although the convenience of these images is unparalleled, they are often the result of a substantial amount of labour, with lengthy production times and costs, especially when the majority of these images are only suitable for one application. This is a result of the complex processes that are performed during the production of such imagery and can be best demonstrated in the production pipeline (figure 2.1 [Kerlow 2000]).

Examining this pipeline makes it apparent how expensive (both in terms of monetary value and man hours) production is, particularly when we consider that each element of the pipeline is, in itself, an extensive chain of events. Take for example the modelling of characters, this could include creating a skeleton, scripting expressions, skinning, weighting, dynamics set-up, and deformation management.

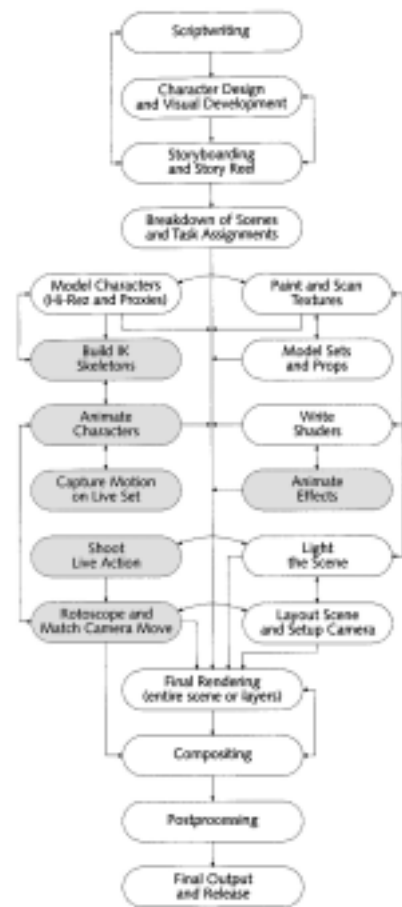
While the pipeline cited here is more typical of that found in the production of computer animation - although not limited to use in a particular field - only six of the tasks (highlighted in grey) are specific to animation. The remainder can be considered common to most areas of CGI production, and it is not unreasonable to assume that all computer generated images will pass through at least some of the stages shown here, regardless of how varied their final application may be.

This is validated by the inescapable fact that computers are only capable of performing the operations that the user programs them to perform - "I suppose computers can do anything nowadays" [Comninos 1997]. To the digital artist this means that no component of a computer-generated image is *free*. Unlike the live action photographer, the digital artist has to recreate every single detail of their scene within the computer for it to be complete. The computer is unable to *fill in the gaps* if the artist *forgets* to include something in his/her scene because the computer has no knowledge or thought process. Therefore, to make the creation of computer images as smooth and easy as possible, with the least possible scope for error, most production companies follow a pipeline in one form or another.

With the popularity of CG exceeding all expectations in all areas of its use and with external backers financing many CGI productions there has been a strong need to reduce the time spent in the production pipeline (and so cost) by simplifying or automating each stage as much as possible. (The need to reduce the cost of CGI in film production may have to be tackled afresh in the UK with the news that the Inland Revenue has closed a tax relief loophole for UK based film production [www.inlandrevenue.gov.uk]).

This has been achieved most notably in the last decade with considerable advances in commercially available software and hardware. *Off the shelf* packages such as Alias|Wavefront Maya [www.aliaswavefront.com], SideEffects Houdini [www.sidefx.com] and Autodesk AutoCAD [www.autodesk.co.uk] automate and combine many aspects of the pipeline into one user friendly tool. They also allow a large team to work concurrently on a single project, meaning that the pipeline no longer has to be a sequential chain of events, without any disruption to workflow. Software development has also meant that it is possible to replicate many natural phenomena (such as fluid dynamics, wind-tunnel tests, cloth simulations, fur/hair and stress tests) with relative ease.

Hardware development has also helped accelerate the production process. Advances in the level of processor power available to production companies (also at a lower cost) has allowed complex simulations



**figure 2.1** the production pipeline for a typical computer animation project

to be achieved in a shorter time period while new data acquisition equipment such as motion capture [Menache 1999] simplifies animation. These advancements have reduced production times and costs while not compromising the quality of work produced.

While there are relatively few - if any - pipeline processes that have not seen some improvement through software and hardware development (whether in cost, time, ease of use etc), some are still a long way off from having a complete solution. One such area that is still an extremely time consuming process is that of modelling (a process that featured twice in our example pipeline). Modelling (or rather the "spatial description and placement of imaginary three-dimensional objects, environments and scenes with a computer system" [Kerlow 2000]) is an unavoidable event in the production of any computer-generated image, and it is on this subject that we will now concentrate.

### Section 3 - Three-Dimensional Digitisation

Modelling (or digitisation) is required for any object to be visible in a computer-generated image, and is a process that frequently takes weeks or months to complete. While it is of relatively little consequence if the object does not move, or if the object is not rendered with a complex *Renderman* shader, it is vital (for any digital image) that the digital version of the object matches the object it is intended to represent. If the artist is unable to convince the viewer that they can identify what they see, then the image is prone to failure. Many computer artists (myself included) will argue that flaws or shortcomings in object geometry can be overcome with a clever use of lighting and cinematography, however this is not a suitable *excuse* for every field that uses CGI, especially when accurate high resolution models are required. It remains that if the shape of the object is inaccurate at any level it will propagate through to the final image with various possible ramifications. (How would you feel, if when about to fly off on holiday, you discovered the wing of your aircraft was not manufactured correctly because the designer using the CAD system could not produce a micron accurate model?)

#### Section 3.1 - The Modelling Process

A simple object, such as a ceramic vase, is to a human just that, a simple ceramic vase, its form and shape taken for granted. To a computer however, the terms *simple*, *ceramic* and *vase* are meaningless, instead the vase is represented by a series of *simple* lines, points and surfaces (which can be textured to look *ceramic*,) all of which the computer only understands as a series of numbers. This means that for the digital artist to be able to construct a model in a piece of software he/she has to *think* in a similar manner to the computer, mentally deconstructing and abstracting the object being modelled into a series of lines, points, surfaces and numbers.

However, the process is unfortunately not that simple. After deconstructing the object, the artist has to be able to use the tools provided by the software package (generally a combination of over a dozen) to reconstruct the lines, points and surfaces in a virtual environment that provides no tactile response. A common modelling technique is to start with a simple geometric primitive (such as a cone, sphere, cube, torus or cylinder) and work this object into the desired shape by performing a number of *modelling operations* on the data. These modelling operations could include extrudes, revolves, lofts, sub-divisions, Boolean operations ... and so the list continues.

This process of mental deconstruction and abstraction followed by the virtual reconstruction of the object is what makes modelling such a time consuming part of the production pipeline. This is complicated by the fact that many organic structures, for example trees, plants and clouds, cannot be modelled in this manner, forcing the artist to use other, more challenging modelling techniques such as L-Systems and other fractal based approaches [Prusinkiewicz *et al* 1990].

#### Section 3.2 - The Language of Modelling

As has already been suggested, one of the fundamental drawbacks of modelling objects in the digital domain is the language that is used in software to describe object geometry. As humans, we have developed an elaborate vocabulary for describing objects in the physical world. From an early age we learn different methods of describing objects to both ourselves and other people, basing our description not only on visual stimulus, but also stimuli from touch, sound, smell and even any tastes that may accompany the object. Unfortunately, we are unable to transport this vocabulary easily to a computer, the computer has no awareness of the material world, (or the world at all for that matter) and it would be impractical to think that we could *teach* the computer about such things. Instead, a computer is programmed to treat objects as graphical representations of mathematic formula (numbers) and as a result, we are limited to the vocabulary that has been bestowed upon us by the computer scientists. Table 3.1 shows a brief comparison of the contrasting vocabulary available when describing physical and virtual objects.

Physical Object Adjectives	Virtual Object Adjectives
size (large, medium, small, tiny ...)	polygon surface
colour (red, green, blue, black, white ...)	NURBS (non uniform rational b-spline) surface
texture (smooth, rough, undulating ...)	sub-division surfaces
outline (smooth, jagged, fuzzy ...)	isoparms
material (velvet, plastic, metal ...)	vertices
weight (heavy, light, overweight ...)	surface patches
form (cube like, spherical ...)	edges
reflection/refraction properties (opaque, mirror ...)	faces
"looks like ..." (a table, a chair, a house ...)	hulls
sex (male, female ...)	control points
strength (hard, soft, flexible ...)	modelling operations (lofts, extrudes, revolves ...)
density (solid, hollow ...)	geometric primitives
other ("kinda round with a bumpy bit here ...")	voxels

table 3.1 describing physical and virtual objects

While it is clear that some of these descriptions relate to the texture of a physical object (texture as used in computer graphics terms) they are still intrinsic to the way we describe the object - as humans we are not troubled by the mathematics behind the surface, only what the surface looks like.

This is further complicated when we consider that different areas of society can potentially have a different vocabulary for describing objects, and a different interpretation of their meaning. For example, someone from a medical background could describe a house as simply as having "four walls and a roof" while an architect would use words from a specific range of his vocabulary that he has learnt while training, undoubtedly providing a much more detailed description. The fact remains however, that neither are of any use to a computer. Isaac Kerlow [2000], director of digital talent and new technology at *The Walt Disney Company*, provides a nice example of the difficulties that language produces when used to describe objects or complete scenes and is included in appendix A for reference.

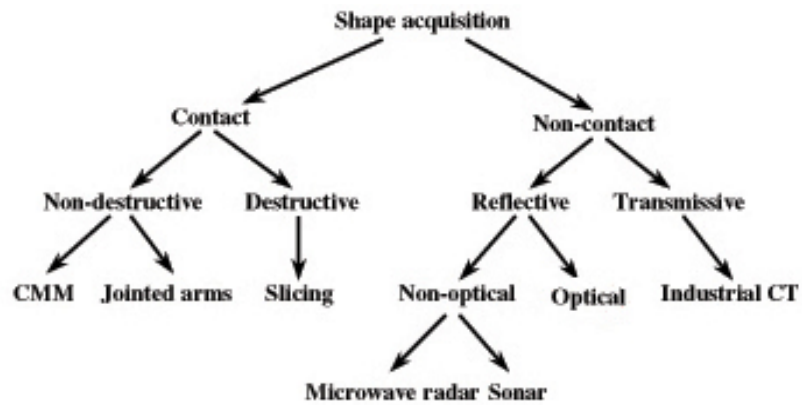
### Section 3.3 - Spatial Descriptions

Another fundamental restriction of digitising objects or complete scenes is the restraints that are placed upon the artist by the space in which they are working. When humans describe physical objects, the objects exist in our immediate three-dimensional world (or sometimes our vivid imaginations). As humans, we are able to navigate around the object, absorbing its form from many differing perspectives. We are able to physically interact with the object, using our sense of touch to tell us about its form (this is also the primary method of *sight* for the visually impaired).

The digital artist is faced with the challenge of creating objects based on the two dimensional version of what he/she can see on the computer monitor. A three dimensional shape has to be translated to two dimensional, horizontal and vertical mouse movement before being displayed on a two dimensional monitor. Although all modelling packages allow the artist to navigate around the virtual scene in three-dimensional space and in near real time, it is still presented in a two dimensional format, and the artist is unable to directly interact with its components. The hardware developments discussed earlier have brought with them new input devices such as tablets, space balls and track/roller balls, however these devices still limit the articulation of the artist.

### Section 3.4 - Alternative Methods of Digitisation

As a result of the restrictions described here, new methods have been developed to try and make the digitisation of objects and scenes easier, more efficient and cheaper. Some of the techniques that we are about to describe have already been used in the production of computer generated imagery and so have already proved their worth, while others are more limited in their application and not so popular. To be able to effectively discuss the range of techniques available to the artist, we need to find some way of categorising their implementation or use.



**figure 3.1** taxonomy of active vision techniques

Although Brian Curless [2000] suggests a taxonomy of automatic and semi-automatic techniques for acquiring three-dimensional data (figure 3.1) this was found to be of relatively little use in this instance. Firstly, the taxonomy is based solely on the acquisition of data through hardware-based techniques and does not make any allowances for innovation in software. Secondly, Curless later goes on to extend the taxonomy of optical active vision techniques which if followed would limit the scope of the research presented here. While some of the categories Curless uses (*contact* and *destructive*) will feature in the research, we have opted to present the details of a broad range of the most popular techniques categorised as either hardware or software based.



## Section 4 - Hardware Based Acquisition Methods

What follows is a discussion of the most common techniques that are available for the digitisation of three-dimensional objects that are predominantly hardware based. While it is noted that all hardware evidently requires some software to control it, these techniques all rely on a specific piece of hardware to be used for the capture (data acquisition) process. This is not intended to be a comparison of techniques in regard to suitability for a particular task, or recommending one technique over another, but rather a survey of some of the techniques available, their implementation and their strengths and weakness in general.

### Section 4.1 - Magnetic Resonance Imaging

Magnetic resonance imaging (MRI) is typically associated with the medical field and the diagnosis of many different conditions, however its use is not exclusive to this field and was considered by Industrial Light+Magic when commissioned to create special effects for *Hollow Man* [Gritz *et al* 2001]. The first MRI scan was performed on a human in 1977 by Dr. Raymond Damadian, Dr. Larry Minkoff and Dr. Michael Goldsmith, using their *Indomitable* machine, and allows the user to build a 2D or 3D map of body tissue types and integrate them together to form a complex 2D image or 3D model.

The MRI scanner consists of a horizontal tube with a magnet (the bore) running from the front to the back of the machine in line with the patient, while the part of the body to be scanned is placed at the isocenter of the magnetic field. The magnets used in an MRI scanner are typically between 0.5 and 2.0 Tesla (or 5000 to 20000 gauss) - the Earth's magnetic field is only 0.5 gauss - however researchers are experimenting with magnets up to 60 Tesla. In practice three different types of magnet can be used:

1) Resistive Magnet	coils of wire are wrapped around the bore magnet through which a current is passed generating a magnetic field that can be turned on and off. These magnets are cheap to make but expensive to run (using 50kW of energy) because of the resistance in the wire. Anything above 0.3 Tesla is too costly to justify
2) Permanent Magnet	this magnet costs nothing to maintain as it is always <i>on</i> but they are very heavy weighing many tons, this makes them difficult to construct
3) Super conducting Magnet	these are similar to resistive magnets but they are bathed in liquid helium at -452.4 degrees with the patient protected by a vacuum similar to that used in a thermos flask. At this temperature there is no electrical resistance, reducing the electrical requirement

**table 4.1** bore magnets used in MRI scanners

The bore magnet creates a uniform or homogeneous field that is very stable, allowing high quality images to be acquired. Aside from the bore, MRI scanners also use three smaller magnets called gradient magnets. These are lower in strength (ranging from 18 to 27 millitesla or 180 to 270 gauss) and create a variable magnetic field.

When the MRI machine is switched on, the magnetic field causes the protons in the hydrogen atoms within the body to align themselves with either the head or feet of the patient. The human body consists of billions of atoms, the nuclei of which spin (or gyrate) on an axis, however hydrogen atoms are particularly useful because they consist of a single proton and a large magnetic moment. This means that a proton that is aligned with the head will be cancelled out by a proton pointing towards the feet - however a few protons in every million will not be cancelled out in this way and remain unaligned.

Using a radio wave, the machine then applies a resonance frequency pulse (also known as the Larmour frequency, see equation 4.1 [www.hslmc.cam.ac.uk 01]) to every hydrogen atom within the body. The hydrogen protons absorb this energy and, as a result, gyrate in a different direction. The resonance frequency causes the one or two unaligned protons per million to gyrate at a particular frequency and in a particular direction. The tissue type being examined and the magnetic strength of the bore magnet determine the resonance frequency.

$$f = gB_o$$

$f$  = the frequency of precession  
 $g$  = gyro magnetic ratio (for hydrogen this is 42.6 MHz/Tesla)  
 $B_o$  = strength of the external magnetic field

**equation 4.1** Larmour frequency

As the resonance frequency is applied, the gradient magnets are turned on and off rapidly in a specific manner, altering the main magnetic field at a very local level. Changing the magnetic field causes abnormal and normal tissues to react differently, and it is this response that enables medical practitioners to diagnose tissue abnormalities. When the resonance frequency is turned off, the hydrogen atoms return to their natural alignment and release the stored energy as a signal that is detected by the machine and returned to the central computer. The computer is then able to deconstruct the signal using the Fourier transform (equation 4.2 [Peters *et al* 1997]), and create an image of the subject being scanned. While explaining the Larmour equation and the Fourier transform is far beyond the scope of this research, it is provided to try and aid comprehension of the implementation of MRI and its complexities.

$$F(Kx, Ky) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) e^{-i2\pi(k_x x, k_y y)} dx, dy$$

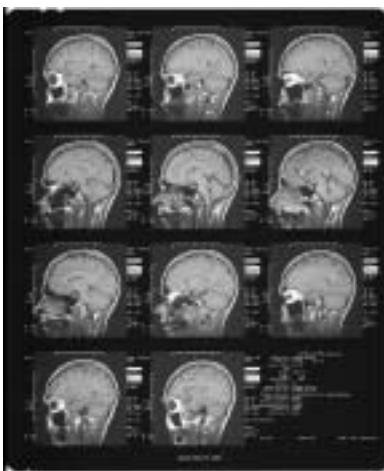
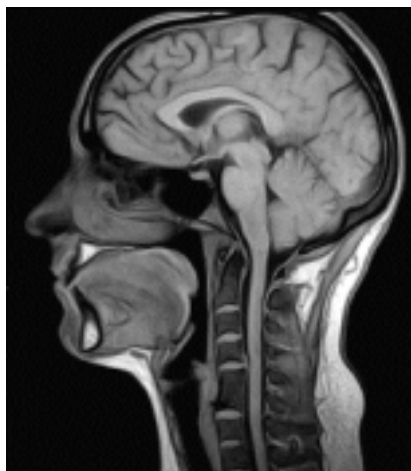
**equation 4.2a** Fourier Transform

$$f(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(Kx, Ky) e^{i2\pi(K_x x, k_y y)} dkx, dky$$

**equation 4.2b** Inverse Fourier Transform

Each image produced from the MRI scan can then be integrated by further software transformations to produce a useable three-dimensional model. By manipulating the gradient magnets we are able to take a *slice* of any part of the body (to as thin as a few millimetres in thickness) in the axial (x/z), sagittal (y/z), and coronal (x/y) plane, or any degree in-between, without moving the subject. This is obviously an extremely useful property of the scanner as it inevitably aids the capture process [www.howstuffworks.com 01]. Examples of the images captured using a magnetic image resonance scanner can be seen in figure 4.1a [focus.aps.org], figure 4.1b [biblip.vet.cornell.edu] and figure 4.1c [neurovia.umn.edu].

This technique is a useful non-contact and non-destructive method for constructing models of the internal structure of humans, animals and other such organic objects [www.hslmc.cam.ac.uk 02], however there are limitations. Due to the large magnetic field generated, people with pacemakers or orthopaedic hardware are unsuitable for scanning, and the machines are very noisy and can cause claustrophobia among some people being scanned. Further, to be able to scan an object, the user has to be able to fit it inside the machine, a potential problem for large or irregular subjects. While the clarity and detail of the images produced is an obvious advantage, the major drawback however is cost, units can cost millions of pounds to purchase, run and maintain, and as a result are only found in medical establishments, restricting their possible use and benefits.

**figure 4.1a** various sagittal sections of a human head**figure 4.1b** a close up of a sagittal human head slice**figure 4.1c** a three dimensional model created from 27 slices of a human brain

## Section 4.2 - Computerized Axial Tomography

Computer axial tomography (or CAT scan) is another technique predominantly associated with digitisation in the medical field, and is a variation of the MRI scan, however x-rays are used in place of magnets. The machine consists of a moveable ring around the subject that supports an array of x-ray detectors opposite an x-ray emitting tube. A motor inside the machine causes this ring to rotate around the subject, scanning a narrow slice of the subject with each full revolution. These slices can then be reconstructed to form a three dimensional image of the body.

An x-ray is a form of electromagnetic energy, and as such, x-ray photons (not to be confused with the *protons* affected in MRI) are essentially the same as visible light but with a much higher energy level, and so wavelength. The higher energy level of x-rays allows them to pass through soft body tissue, leaving a *shadow* of any dense objects on the x-ray detectors. X-ray photons are produced in the emitting tube by the movement of the electrons in an atom. These electrons move around the atom nucleus on different orbitals or energy levels and as an electron drops to a lower energy level, energy is released as a photon, and the energy level of the released photon depends on the energy drop of the electron.

A larger atom is likely to absorb the x-ray photon because it has a greater energy difference between the orbitals while smaller atoms are less likely to absorb the photons for the opposite reason. If we consider a human subject, the tissue surrounding the skeleton is generally made of smaller atoms, while the bones, or rather their calcium content, consist of larger atoms. This means that when an x-ray is fired at a human subject, the ray is absorbed by the bones but is able to pass through the surrounding tissue to be collected by the x-ray detectors, allowing us to generate an image of the human body [www.howstuffworks.com 02]. A three dimensional model can then be created by integrating a number of these images together.

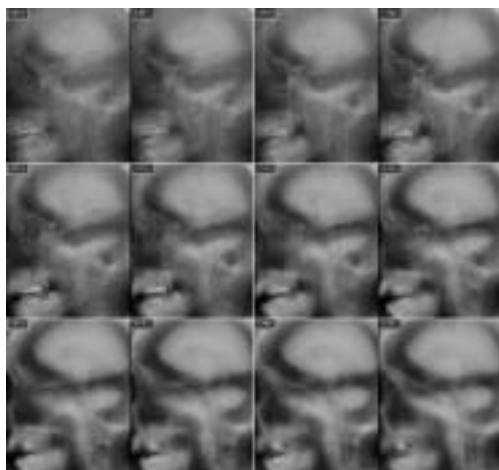


figure 4.2 an example of a CAT scan

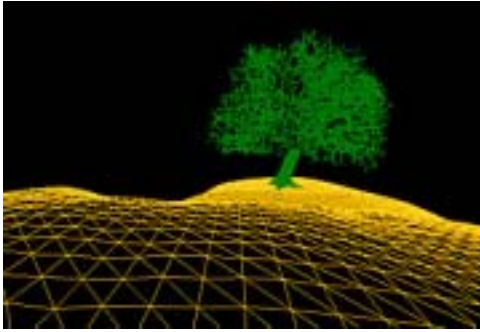
This technique of digitisation has strong limitations in that it is only able to distinguish between small and large density objects, producing arguably ambiguous results (as shown in figure 4.2 [www.aeic.alaska.edu]). This makes it unsuitable for use where a high-resolution model is required. It also has similar problems to the MRI scanner in that the user is limited to what he/she can fit inside the machine to scan, while the cost of installing a scanner is another prohibiting factor. What is of greater concern to the people who use these machines however is the possibility of radiation sickness caused by prolonged exposure to the x-ray radiation. The x-ray is a form of ionising radiation; this means that when an x-ray hits an atom it can cause it to throw off electrons to create an ion, an electrically charged atom. The free electrons then collide with other atoms to create even more ions and so forth. The electrical charge of an ion can lead to an unnatural chemical reaction inside tissue cells which can cause a break

in a DNA chain, and possibly cancerous DNA mutations [www.howstuffworks.com 02]. This means that while a non-contact scanning device, it could be considered as semi-destructive and although other objects aside from organic structures can be scanned it has proved to be unsuitable for use outside of the medical field.

### Section 4.3 - Laser Scanners

The previous two examples of data acquisition have only truly been of benefit to the medical research and diagnosis field, and so we introduce now a technique that has a far wider range of applications. Laser scanners have already proved their value in the general computer graphics field and have been in use for a number of years. One British based company (AG Electro-Optics [www.ageo.co.uk]) that manufactures and distributes such scanners state that their MENSi 3D laser scanning technology has been used in:

- aerospace to obtain dimensional information regarding a Boeing 777
- automotive industry by Renault for crash testing
- architecture to create a virtual model of Liverpool City centre
- museums to record Egyptian archaeological sites
- oil and gas industries to create a model of the Shell Petroleum Platform in Nigeria
- industrial plants for refitting pipe work
- film production for *What Dreams May Come*, scanning sections of Glacier National Park in Montana (figure 4.3 [www.ageo.co.uk]).

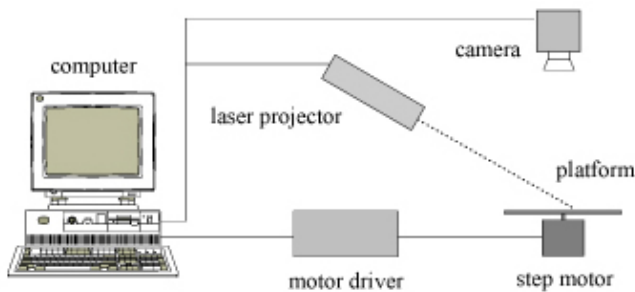


**figure 4.3a** wire frame model created from the laser scan

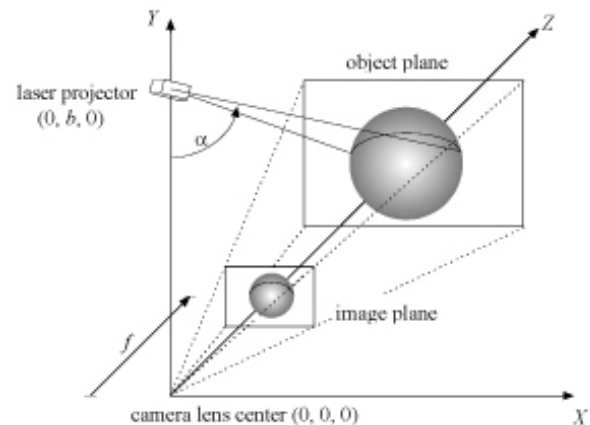


**figure 4.3b** final shot from *What Dreams May Come* fully modelled, textured and lit

Laser scanning is an active stereoscopic technique that is based on the triangulation of the laser source and a detector. During the scanning process, a laser beam (which can be in the form of a single point, a single line, or multiple lines) is projected onto the surface of the subject, which is distorted by the surface relief. The image of the beam on the surface is then captured by a camera that is offset from the position of the laser (hence the term stereoscopic). The  $x$ ,  $y$  and  $z$  coordinates of a point on the surface can then be determined through the use of simple trigonometry (figure 4.4, figure 4.5 and equation 4.3 [Xu *et al* 1998]) [www.imapctstudiostc.com] [www.simple3d.com]. The number of points (and so  $x$ ,  $y$  and  $z$  coordinates) calculated by the scanner depends of the complexity of the mesh that is required, however most scanners are capable of producing 1000 points a second. The resultant *point cloud* can then be converted into a polygonal mesh using a triangulation algorithm, such as Delaunay [Delaunay *et al* 1995]. It is important to note the difference between the two uses of the term *triangulation* in this process. The first use of the term refers to the triangular properties of the set-up used by a laser scanning system while the latter use of the term refers to the formation of triangular faces connecting vertices in a geometric polygonal surface



**figure 4.4** a typical laser scanner system set-up



**figure 4.5** triangulation of a three dimensional coordinate

$$[X, Y, Z] = \frac{b}{j + f \cdot \cot \alpha} [i, j, f]$$

$[X, Y, Z]$  = 3D coordinate

$b$  = distance in  $y$  of light source from origin

$f$  = focal length of camera

$i, j$  = position of point on image plane

$\alpha$  = projection angle

**equation 4.3** triangulation to obtain a three dimension coordinate based on the laser scanner set-up shown in figure 4.4 and figure 4.5

By changing the laser detector/camera we can obtain different information about the surface of the subject. If a black and white camera is used then only geometry data is captured, however if a colour device is also used it is possible to obtain coloured texture information that can later be used to texture the model ready for rendering [www.imapctstudiostc.com], an obvious advantage of this technique.

Most scanners use a Class 1 Laser which is eye safe and so pose no safety risk to the operator and subject, additionally, most software and hardware that is used during the capture process contains built in safety measures for added security. A further benefit of the laser scanner is that they can capture a wide variety of shapes and sizes. Scanners range from a portable handheld version where an operator is required to move the scanner around the subject to static scanners that can capture large objects using a computer controlled turntable to rotate the subject so that it can be captured from multiple angles. However, this diversity in size is also represented with diversity in price with scanners ranging from 30000 to 300000+ US dollars. It is also completely non-contact and non-destructive; the subject is fully preserved during the scan making it safe for delicate or valuable objects (such as Michelangelo's *David* [Levoy *et al* 2000]).

The laser scanner has only two downfalls. The first is that because the point cloud is based on the reflection of the laser beam from the object's surface onto the detector the scanner does not work well on extremely dark or dull surfaces, as well as surfaces that are highly reflective. This can easily be overcome however by either painting the subject a lighter colour, or by dusting it with a light powder (talcum powder) - however this now makes it a non-contact based destructive process. The second inhibiting feature is a fundamental property of any light (or in this case laser) beam. The laser beam can only travel in a straight line, and consequently only surfaces within the line of sight are captured. This means that either the scanner or the subject have to be moved if a complete scan is to take place.

#### Section 4.4 - Touch Probe

After the non-contact laser scanner we move onto an example of a contact (although if used correctly still non-destructive) digitisation device. The touch probe (or joint arm coordinate measurement machine) physically touches the object being captured to acquire the three dimensional data. This immediately suggests a disadvantage of this type of system because it means that an operator has to manipulate the touch probe to touch every point on the object that we would like three-dimensional data for. This is undesirable for most situations, but especially when larger subjects are being considered. Having said this, they are more intuitive to use than some other techniques, and allow the user to reach both inside and behind the subject without necessarily having to reposition the arm entirely. Touch probe systems are also significantly cheaper than other solutions, ranging from 1200 to 3000+ US dollars. An example of a commercially available touch probe device is shown in figure 4.6 [www.axila.com].



figure 4.6 Axila's Arm 2000 touch probe

The device consists of a number of limbs or arms that are connected by flexible joints (similar to an elaborate angle-poise lamp). The seven degrees of freedom allow the user to move the tip of the probe to the location to be scanned on the object. The computer is then able to acquire three-dimensional data for that point based on the configuration of its joints and limbs [www.simple3d.com]. They range from small portable desktop probes to large industrial devices that can measure a 12 foot (3.66 metre) sphere, however they are not very accurate (0.002in - 0.004in per 12ft arm which is equivalent to 0.05mm - 0.1mm per 3.66 metre arm) and may need to be relocated several times during the capture of an extremely large subject [www.qualitydigest.com].

Although they are suitable for use in most computer graphics fields, they are of most benefit in the manufacturing industry for checking the production of individual parts, the structure of fitted components and reverse engineering. Their reduced cost does make them more appealing to small production companies in all areas of computer graphics however.

#### Section 4.5 - Profilers

The process of a profile capture - which is again non-contact and non-destructive - is similar to the use of a chroma-key (commonly known as blue/green screen) in film production and could actually be considered a compositing technique that can take place in either software or hardware. The object to be scanned is placed on a computer controlled turntable in front of a mono-coloured background (normally this is either blue or green), which itself is in front of a static camera. The computer rotates the turntable and after a predefined number of degrees of rotation, the object is photographed in its current position. A special piece of computer hardware is then used to calculate the profile of the object by scanning the image for a

transition from the background colour to a different colour (which is considered to be the object) within specified tolerances. The coordinates of a point on the profile and the angle of rotation of the turntable allow the computer to calculate three dimensional coordinates that enable us to make a three dimensional model of the object.

This technique is relatively inexpensive, especially as we can use software to remove the background colour from the image instead of a specialised piece of hardware. It has the added benefit that the non-background colours in the profile image can be used to create a texture map to be used with the computer-generated model. Having said this, profilers are not generally used for digitising objects because they are unable to detect and capture concave parts of an object's surface [www.simple3d.com]. While this can be overcome if the technique is combined with the use of a laser scanner for the concave sections, would it not be more efficient and cost effective to use the laser scanner for the whole capture?

Furthermore, removing the background colour from the image is not an easy task. If we look to Apple *Shake* [www.apple.com] (a high end compositing package) for its solution to *keying* we find that there are number of possible algorithms that can be used - *chromaKey*, *photonPrimate* and *CFCKeylight*. Each of these solutions is suitable for specific images and is constantly being hand tweaked by the compositor to produce the best profile. There are other factors to consider however such as a *spill* which occurs when the background colour is reflected on the subject which could *confuse* the profiler producing inaccurate results. While such problems can be reduced through proper set up (good lighting, choosing the correct background colour) this all takes time, and unfortunately money. Essentially the profile machine would have to be reconfigured from the ground up for each object being scanned and by requiring a very specific scan environment it makes the whole process too time consuming.

#### Section 4.6 - Structured Light Scanner

Digitisation using a structured light scan is very similar to geometry reconstruction using a laser scanner (as discussed earlier). However, instead of a laser beam being directed at the subject, the user projects a specific light pattern (typically a series of stripes) at the surface of the object, which is then recorded using a camera. The image captured by the camera shows how the pattern is distorted by the topology of the surface and allows the computer to triangulate the data to calculate three-dimensional points, as with the laser scan.

A specific example of a structured light scanner is a moiré fringe projection. The surface of the subject is illuminated through a periodic grating while the resultant image is captured at an angle through another grating. The interference pattern that results from the two reference gratings form a *moiré fringe contour pattern* which shows bands of dark and light stripes. The patterns can be analysed in software to produce accurate descriptions of the changes of depth (and so shape) within the object (figure 4.7) [Curless 2000].

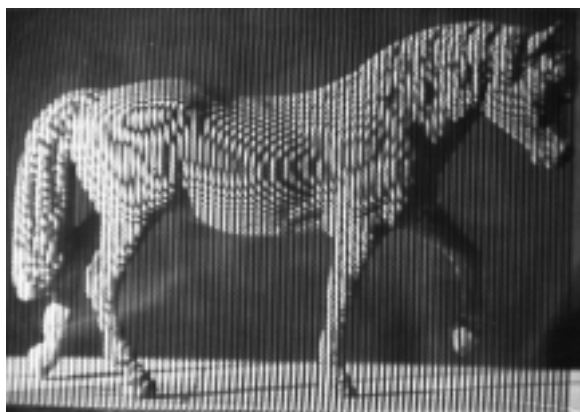


figure 4.7a projected pattern through first grating



figure 4.7b pattern through second grating

This technique has further similarities to the laser scanner in that it is non-contact and non-destructive. Similarly, a structured light scanner is only capable of capturing surface geometry that is within the line of sight of the light projector (as has already been stated, the light beam is only able to travel in a straight line). This means that either the subject or the scanner has to be moved/rotated if a complete mesh is required. The structured light scanner is also perfectly safe because it uses either a simple halogen white light source or a projector connected to a computer (similar to the ones found in many academic institutions) as the light

source [www.simple3d.com]. While this type of scanner is accurate to within 10 microns, there are a number of factors that can complicate the scanning process. If the projector is too far away from the subject, the projected patterns become too sparse and as a result we are unable to capture the surface topology accurately. Likewise, if the moiré fringe pattern becomes too dense (whether as a result of the surface topology or the proximity of the scanner) the computer is unable to resolve a three-dimensional mesh from the data collected. Finally, because the computer has to project and interpret the recorded pattern on the mesh, it makes this process computationally expensive and so slower than its laser scan counterpart [Curless 2000].

#### Section 4.7 - Laser Trackers, Theodolites and Total Stations

The final hardware based three-dimensional digitisers to be looked at are laser trackers, total stations and theodolites. These three have been grouped together because while they are all capable of providing the user with the relative (and sometimes even absolute) distance between points, they are not particularly suitable for constructing large, complex or complete geometry meshes. They have been included in this research because for some computer graphics purposes this information is sufficient. They also have other similar properties, they are semi-contact, arguably semi-destructive, require more operators or user interaction than the other techniques discussed so far, and they do not actively scan the subject.

A laser tracker is a portable coordinate measurement machine that can accurately measure an object from one to one hundred metres in size. They use a laser interferometer (this is a measuring instrument that uses interference patterns to make accurate measurements of waves of light - also called a refractometer) to measure distance and angular encoders to determine the orientation of the centre of a spherical mirror retroreflector (SMR) with respect to the tracker. These measurements enable the user to obtain positional coordinates. The SMR must be moved to different locations on or around the subject to capture more than a single point of data [www.simple3d.com]. Laser trackers can be accurate to 0.001 inches (or 25.4 microns) when within yards of the SMR [www.qualitydigest.com].

Because the target is handheld and has to be interactively placed on the target it makes this method laborious for large subjects and requires that two operators are present. The semi-contact nature of this method also means that the laser tracker is not suitable for delicate or valuable items. It is however suitable for architectural or design purposes and is often used by surveyors on construction sites (as is the case with theodolites and total stations).

A theodolite is a combination of two telescopes that are aimed at a single target in space. Digital encoders register the relative angular positions of the two theodolite heads and targets, this can then be fed into a computer that can use simple trigonometry to determine the target positions, usually at a rate of one point per minute. Both the targets and the telescopes have to be aimed and positioned interactively throughout the capture process, again requiring that two operators are present. The placement of targets also makes this a semi-contact semi-destructive method that has all the disadvantages of a laser tracker.

A variation of the theodolite is a total station. These are a single station theodolite that uses the visual sighting of interactively placed reflective targets to obtain three-dimensional data. A total station uses a digital encoder to measure the azimuth and elevation angles while the distance to the target is measured with an electronic distance meter [www.simple3d.com].

#### Section 4.8 - Other Considerations

While we have now covered the majority of the three-dimensional digitisers already in use in the computer graphics industry, we mention here (by name only) the few remaining techniques that could be considered when looking to capture a physical object. These are: microwave scanners (that are based on the principal of laser and structured light scanners, but using microwaves), sonar radar/scanners (that are again based on the principal of laser and structured light scanners, but using sound waves), active slicing (a contact and destructive method where the subject is physically sliced into sections before being scanned [www.nlm.nih.gov]), and BRDF or *bi-directional reflection distribution function* (a four dimensional function that examines the incoming and outgoing light direction for a single point on a surface to obtain information about its relief).



## Section 5 - Software Based Acquisition Methods

In this section, we examine three techniques of geometry reconstruction that are predominantly software based. While hardware may be required in one form or another, (a computer to run the software on) the actual data acquisition is performed using a software algorithm. The first two techniques that we focus on are not commercially available, but the result of research by a group of individuals from universities and other institutions. As such, the information included here is based on the publication of their successes and not any specific practical implementation/demonstration.

The third technique that we examine is a brief discussion of the software that is commercially available. The fact that they are commercially available means that their implementation and reconstruction techniques are commercially sensitive and so unavailable for documentation. As a result, we are left with a comparison of the product specification, and while this could be considered a divergence from the intention of this research, it is still included to help put the previous two techniques in context.

Again, what follows is not intended to be a comparison of techniques in regard to suitability for a particular task, or recommending one technique over another, but rather a survey of some of the techniques available.

### Section 5.1 - The "Teddy" Sketching Interface

In their paper [Igarashi *et al* 1999], Igarashi, Matsuoka and Tanaka present a method of digitisation based on the interactive two dimensional free form strokes of the user which could be seen as an extension of the *SKETCH* project [Zelevnik *et al* 1996]. This is an innovative and refreshing approach to geometry digitisation as although the user is still required to *construct* the geometry by hand, the user only has to present a two dimensional image of the object. The two dimensional silhouette of the subject is *inflated* by the software to a three dimensional mesh. The algorithm has been implemented in Java and so can be done in real time on a standard PC without the need for high end or specialised equipment. Igarashi, Matsuoka and Tanaka describe *Teddy's* ease of use as being of benefit for "rapid prototyping in the early stages of design, educational/recreational use for non-professionals and children and real-time communication."

Possibly the most interesting part of this implementation (especially within the context that we are dealing) is the *inflation* process where the user's freeform strokes are used to construct a three dimensional polygonal mesh. After the user has drawn a freeform stroke using the program's graphical user interface, the first step is to create a "closed planar polygon by connecting the start-point and end-point of the stroke" and determining "the spine or axes of the polygon using the *chordal axis*." During this step, the algorithm makes all edges of the closed polygon a predefined unit length whilst also checking to make sure the polygon is not self-intersecting. The algorithm then performs constrained Delaunay triangulation (mentioned earlier [Delaunay *et al* 1995]) on the polygon to create internal edges. The triangles are then divided into three categories: terminal triangle (with two external edges), sleeve triangle (with one external edge) and junction triangle (with no external edges). The chordal axis is then obtained by "connecting the midpoints of the internal edges" after *pruning* "insignificant branches and the retriangulation of the mesh" to create *fan triangles*.

After the pruning stage has taken place the "pruned spine is obtained by connecting the midpoints of the remaining sleeve and junction triangles' internal edges." Once this is complete, the algorithm subdivides the sleeve and junction triangles to enable them to be made into a three dimensional mesh, or *elevated*, by creating a "complete 2D triangular mesh between the spine and the perimeter of the initial polygon." Finally, the algorithm raises each vertex of the spine "proportionally to the average distance between the vertex and the external vertices that are directly connected to the vertex." The internal edge of each fan triangle created by the pruning process (excluding spine edges) is "converted to a quarter oval and the system constructs an appropriate polygonal mesh by sewing together the neighbouring elevated edges." The elevated mesh is finally mirrored to create a "closed and symmetric" mesh before being refined using "mesh refinement algorithms." This process is demonstrated graphically in appendix B.

The user is then able to edit the mesh that they have created by performing a number of modelling operations on the object similar to those described earlier in section 3.1 - "each editing operation modifies the mesh to conform to the shape specified by the user's input strokes." Although we are returned to the problem of the language of modelling and the deconstruction and abstraction of an object to such



operations, it is presented in a more informal, intuitive and less complex manner than most other modelling packages.

Once the modelling process is complete, the user is able to save his/her mesh to a *.obj* or object file that contains all of the vertex and face data for use in a more advanced program such as *Maya*. After using this tool it is possible to see just how easy it is to create reasonably complex shapes, however the geometric mesh that is produced is very *stylised* and not suitable for the complex objects that are typically required by the computer graphics industry.

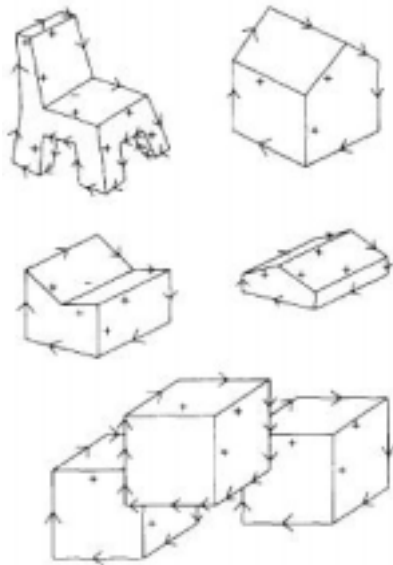
## Section 5.2 - Solid Models from Two Dimensional Sketches

Grimstead and Martin [Grimstead *et al* 1995] present another software based reconstruction technique that is particularly suitable for the early stages of CAD design, describing a system that "can interactively input a 2D line drawing of a 3D object drawn using pen and tablet, and interpret it as a 3D object in real time as the designer sketches." This technique would allow the digitisation of imaginary or conceptual objects as long as the user was able to draw them free hand using pen and ink, or in this case, pen and *tablet*. Here we shall describe the overall process of creating the *solid models* before looking at the *labelling* stage in more detail.

The first stage of the process involves the user drawing their sketch on a tablet that is connected to the computer (although presumably the user could draw the sketch in pen/pencil and import it into the program through the use of a traditional flatbed scanner). The algorithm then analyses the lines of the drawing, and "2D tidying is carried out; for instance, lines are connected together, tested for parallelism and so on." The tool assumes that the user is sketching an "opaque, trihedral, planar solid," and by doing so simplifies the reconstruction of the surface. This would suggest an initial disadvantage of this system as the user would like free range over the sketches they produce, however Grimstead and Martin go on to state that this is not an inherent limitation of their implementation, but was merely imposed to keep the initial system as simple as possible.

After the drawing is tidied, it is possible to start obtaining three dimensional data from it. The tool starts by adding line labels to the drawing (based on the Huffman-Clowes labelling systems to be examined later) and dividing the drawing into regions. This gives the "basic topology of the visible part of the object." At this stage, the depths of the vertices and face equations are still unknown so the drawing is converted into a *precursor solid model*. The tool goes on to examine the line labels and based on the interpretation, "any hidden faces that intersect visible faces in the drawing are added to the precursor solid model as incomplete faces."

From this initial solid model, "a linear system is produced that constrains the vertices as lying on the intersections of their parent faces. Further constraints are then produced from the artefacts in the drawing (such as skewed symmetry and parallelism), and are added to the linear system." The algorithm continues by validating the sketch as possible for reconstruction by recalculating the position of the vertices by intersecting faces. In the final stage of the algorithm, any incomplete hidden faces are completed (or made as complete as possible), producing the final topology of the object and so the final solid model. The solid model is then "presented to the user for verification and further editing."



**figure 5.1** line labelling using the Huffman-Clowes label set

Line labelling (used by Grimstead and Martin, figure 5.1 [Grimstead *et al* 1995]) was one of the first tasks studied in computer vision and allows the computer program to "evaluate the significance of each line in an image [Russell *et al* 2002]. Huffman [Huffman 1971] and Clowes [Clowes 1970] independently developed a line labelling technique for scenes that contained opaque trihedral solids - making it suitable for use in this implementation - using four different labels to indicate the relative position of the lines in space.

The labels used in the Huffman-Clowes line labelling system indicate "that a line is convex or concave with respect to the viewer, or occluding something to the left or right of the line." In the system, a '-' represents a concave edge, a '+' represents a convex edge, while an arrow ('>' or '<') represents an occluding edge. The arrow used is directional, "such that the solid object is lying to the right of the line and empty space or hidden detail is visible to the left of the line [Grimstead *et al* 1995]. Grimstead and Martin then refine this line labelling system using Waltz's constraint based system that removes inconsistent line labels [Waltz 1975].

It is worth noting at this stage that if there is any ambiguity in the sketch then the Huffman-Clowes labelling system will produce several alternatives for the possible geometry of the object. If this is the case, the first possibility is transformed into a solid model and presented to the user. If the model is incorrect, the user informs the tool and it will use the next line labelling possibility until all are exhausted or the correct interpretation is found. While this suggests that this method is not particularly accurate, by keeping all possible solutions in memory this is no longer a problem.

It is easy to see how this tool is of use when reconstructing surfaces that have three planar surfaces meeting at each vertex, however its success with more complex surfaces is questionable. Although Grimstead and Martin state that it is not an inherent property of their system that the objects are trihedral, it would appear that any more complex surfaces would not be able to use the Huffman-Clowes line labelling system.

### Section 5.3 - Commercial Software Applications

As we stated at the beginning of this section, information regarding the implementation of digitisation in commercially available software applications is unavailable due to its commercially sensitive nature. Therefore, we will try and compare the specification of the products as provided by the manufacturer in the hope that this will enable us to gain some useful information regarding their advantages, disadvantages and possible execution.

The first package we shall look at is MetaCreations *Canoma* [www.metacreations.com]. *Canoma* allows the user to "create photorealistic 3D models from scanned or digital photographs without extensive 3D skills." The example in figure 5.2 [www.metacreations.com] shows the creation of a textured three-dimensional model created from six base photographs using *Canoma*.

While the package places a strong emphasis on its suitability for internet based productions (*Canoma* is "web ready") it claims to be useful for a wide range of applications, including online catalogue customers, travel, tourism and



**figure 5.2** screenshot from MetaCreations *Canoma*

real estate, graphic design, advertising, architects, urban planners, property developers, interior designers and game developers.

Unfortunately the *Canoma* specification provides no examples or descriptions of the quality of the three dimensional data it produces, however it is able to produce "distortion free textures" and export the geometry in a number of standard file formats (including *dxl*, *obj* and *vrml2*) for use in "traditional 3D modelling and animation software."

Next, we consider RealViz *Image Modeler* (figure 5.3) [www.realviz.com]. *Image Modeler* allows the user to reconstruct photorealistic scenes, while also allowing for the integration of existing three-dimensional models into still photographs. *Image Modeler* can also provide the user with an accurate measurement of distances in the scene. It claims to be the "solution to measure and create 3D scenes using photographs." As with *Canoma*, textures can be created on a per face basis or unfolded from the whole object and geometry can be exported to a favoured modelling package for further modification. *Photo Modeler* claims to be of use in architecture, archaeology and cultural heritage, industrial design, video games, virtual reality, web 3D formats, animation and special effects industries.

Again, *Image Modeler* does not go into detail about how it actually reconstructs the data for the user, however the application documentation does say that the program uses "advanced algorithms." The user must import at least two images into the program before calibrating them to find and place common features (about nine to 20 in total.) The program can then automatically calibrate the position of the cameras, including their rotation, focal length and lens distortion. The user is then able to model the scene using standard polygonal modelling tools [www.realviz.com].

The last commercial application that shall be considered in EOS System's *PhotoModeler* (figure 5.4) [www.photomodeler.com]. According to their website, *PhotoModeler* is the "professionals choice for 3D modelling and measurement throughout many industry sectors that include accident reconstruction, archaeology and anthropology, architecture and preservation, film, video and animation, forensics and plant and mechanical engineering. This is obviously similar to the previous programs that have been featured, however one interesting feature that isn't included with the other packages is modelling using NURBS curves and surfaces, which have the potential to create surfaces and shapes that are more organic.

This program has a similar implementation to the previous two; the user takes two or more overlapping photographs from different angles that are then loaded into the program for analysis. The user then has to mark features on the photographs and use referencing functions to instruct the program to match points across the images. The program is then able to process this data to create three dimensional *point data* to produce an accurate three-dimensional model. *PhotoModeler* also boasts further programmability using *dynamic data exchange* (DDE) to control various functions with the output of other Windows applications. Other features include silhouette solids modelling, edge modelling, cylinder modelling and sub pixel target marking for precision point measurements.

Unlike the previous packages, *PhotoModeler* does give some details of the techniques it uses for geometry reconstruction, stating that it uses *photogrammetry* techniques to recover the data (which forms part of their core technology). Using the marked and reference points at distinct features on the images, *PhotoModeler* uses photogrammetry to calculate the position of the camera for each photo. The tool then calculates the intersection of light rays from each photo's position out into space. Multiple images (and so light ray intersections) allow the reconstruction of an entire object or scene from just a few photographs. EOS Systems have also undertaken a number of accuracy studies for their software, the results of which are freely available for viewing on their website.

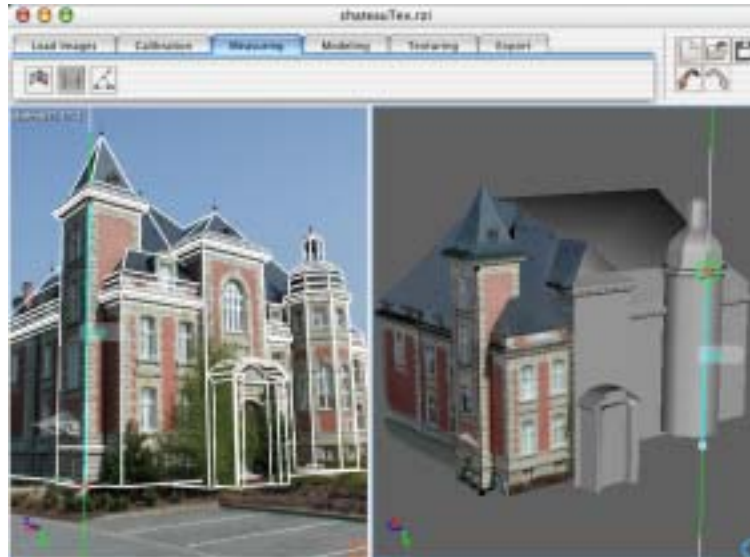


figure 5.3 screenshot from RealViz Image Modeller

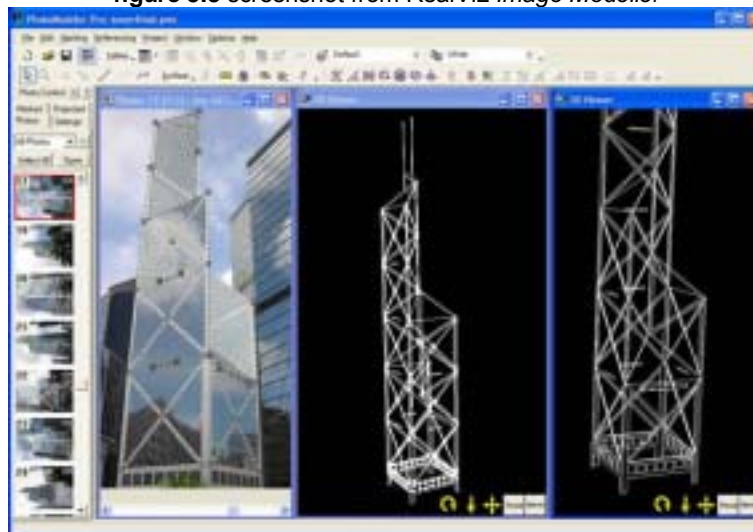


figure 5.4 screenshot from EOS System's PhotoModeler

## Section 5.4 - Other Considerations

We mention here two further techniques that could be considered for use when implementing three-dimensional digitisation in software, however they are mentioned by name only and we shall not go into detail. The first is *depth through defocus* where depth coordinates are obtained based on the level of focus in an image. If we know the position, or distance from the camera, of a point in the image that is in focus, then we can calculate depth for other points based on the amount by which these points are out of focus. The second technique uses *optical flow* algorithms. Two or more images are taken of a scene or object and software tracks the change in position of corresponding points in the two images. The change in position of the two points is stored as a vector that shows the direction and magnitude of movement. Motion parallax principles can then be used to obtain depth data. This technique is also used in motion tracking or match move programs.

## Section 6 - Photogrammetry

When examining the commercial applications available for three-dimensional digitisation, the term photogrammetry was raised to describe the techniques these applications used to create their data - most notably *PhotoModeler*. This technique has not been included in the initial body of research because it is the basis of our implementation of geometry reconstruction, and so warrants being discussed separately. Photogrammetry is defined by one of its users [US EPA] as the "science or art of obtaining reliable measurements or information from photographs or other sensing systems." It could be argued that a number of the techniques discussed so far are also a form of photogrammetry; laser and structured light scanners calculate coordinate information from the image of the reflected laser/light beam. Having said this, we shall focus on the use of photogrammetry when applied to photographs that have not been subject to external treatment. Although there are a number of variations for the construction of geometry using photogrammetry, this research (and subsequent implementation) will concentrate on digitisation using stereoscopic image pairs. This involves using a minimum of two, slightly offset, photographs of an object to obtain depth information.

### Section 6.1 - Stereoscopic Vision in Humans

This method of digitisation finds its roots in the way that humans are able to perceive depth. Our eyes provide our brain and sensory system with a stereoscopic view of the world, enabling us to judge distances to and between objects around us. It is even possible to demonstrate this process taking place, if one eye is covered we lose our sense of depth and struggle with rudimentary tasks such as picking up a cup on a table.

Our eyes naturally fixate on the point that we are looking at ( $f_1$  in figure 6.1), this is called the *fixation point* and appears on the fovea of both the left and right eye. However, we do not just see the fixation point, but objects in front of ( $f_2$  in figure 6.1) and behind it, and this causes *retinal disparity* which allows us to perceive depth. Objects in front of and behind the fixation point will appear on a different place on the fovea to the fixation point. The difference between the two positions is called retinal disparity and can be used to determine which objects are closer than others. Closer objects create a negative retinal disparity (also called *crossed disparity*) while objects further away than the fixation point cause positive retinal disparity (also called *uncrossed disparity*). The fixation point obviously has zero retinal disparity. This is shown in figure 6.1 [William *et al* 2004] and is accompanied by an equation for calculating retinal disparity in equation 6.1 [William *et al* 2004].

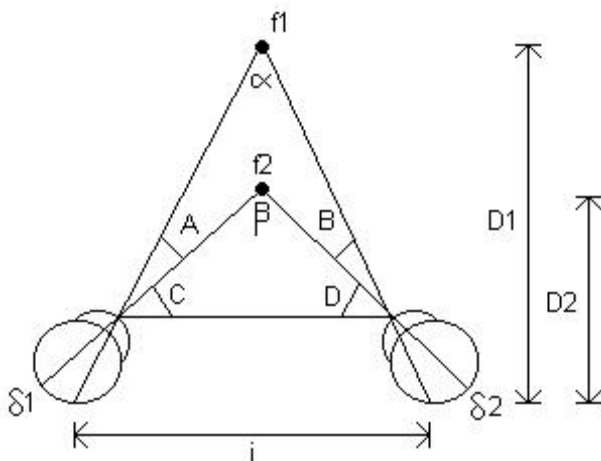


figure 6.1 retinal disparity in humans

$$\alpha + A + C + B + D = 180$$

$$\beta + C + D = 180$$

$$\alpha - \beta = A + (-B) = \delta_1 + \delta_2 = \text{disparity}$$

equation 6.1 calculating retinal disparity

### Section 6.2 - The Digital Equivalent

We can apply the principle of disparity to images taken with a stereoscopic camera with relative ease as stereoscopic cameras have a fixation point of  $\infty$ . This means that disparity is now based on the change in the x-axis of a point between the two images, and the *motion parallax* of the point informs us of its relative depth. The motion parallax principle states that objects in the distance appear to show little movement as the viewer/camera moves from left to right in space. Closer objects however show a large translation as the viewer/camera moves from left to right. This can be demonstrated by a simple train ride. When you look out

of the window, the tracks and fence close to the railway line move quickly, flashing past the window, while the trees and fields in the background move slowly, taking a long time to disappear from view. Therefore, when we compare the x position of a point in the left image to the x position of the corresponding point in the right image we are able to gain a relative depth measurement for that point. We can do this for every point in the image and create a *disparity map* which is similar to a *depth map*, where distance from the camera is encoded as a value of grey.

While on the surface this seems like a simple process, the time it takes to perform depends on the stereoscopic technique being used. If a passive approach is taken, the software or computer algorithm is responsible for finding corresponding points in the two images, needless to say, this is a computationally expensive procedure and can slow this technique down considerably. However, this can be avoided, or at least simplified if we take an active approach, with the user specifying the corresponding points.

### Section 6.3 - Approaches for Feature Matching in Stereoscopic Images

In the implementation to be discussed shortly, we take an active stereoscopic approach, however, to aid comprehension and understanding of the passive stereoscopic technique we shall briefly look at the possible algorithms that are available for finding corresponding points across two or more images. This is a large subject area in itself and is the subject of a large amount of research into computer vision and artificial intelligence. Consequently, only short explanations, if any, will be provided because again, they are beyond the scope of this research.

In their Biological Cybernetics paper, Porr *et al* [1998] describe a method where they exploit the similarities between the analysis of visual disparity and the analysis of the azimuth of a sound source. They state, using [Konishi *et al* 1986] that the direction of sound is determined from the temporal delay between the left and right ear signals. Porr *et al* use two resonators "implemented in the form of casual (electronic) filters to determine the disparity as local temporal phase differences between the left and right filter responses." Put simply, their system takes the luminance signal of the scan-line (from both the left and right image) and pass it through a resonator. The phase difference (of the oscillating waves) is "directly equivalent to the disparity" while the waveform allows them to find the corresponding points.

Two further techniques are based on complex mathematical formula, the first is called *cross-correlation* and the second, the *sum of squared differences*. These produce a large number of corresponding points and so a dense disparity map is produced. Both of these techniques also allow us to maintain a *confidence map*. This means that any points that suffer from possible ambiguity can be recorded and used later to ensure that the correct three-dimensional model is created. The equations used to calculate cross-correlation and the sum of squared differences are shown in equation 6.2 and equation 6.3 [www.umd.edu].

$$SSD = \sum_{[i,j] \in R} (f(i,j) - g(i,j))^2$$

**equation 6.2** sum of squared differences equation

$$C_{fg} = \sum_{[i,j] \in R} f(i,j)g(i,j)$$

**equation 6.3** cross correlation equation

$f(i, j)$  = point in left image coordinate

$g(i, j)$  = point in right image coordinate

$f(i, j)$  = point in left image coordinate

$g(i, j)$  = point in right image coordinate

Another technique is feature based, and must be adapted for each application. Lines, edges or corners between the two images are compared, possibly using the equation in equation 6.3, to enable the algorithm to find the corresponding points in the images. The corresponding points are identified through the orientation, contrast, midpoint locations and lengths of the lines and edges.

$$s = \frac{1}{w_l(l_l - l_r)^2 + w_\theta(\theta_l - \theta_r)^2 + w_m[(x_l - x_r)^2 + (y_l - y_r)^2] + w_c(c_l - c_r)^2}$$

$l_l$  and  $l_r$  = line lengths  
 $\theta_l$  and  $\theta_r$  = line orientations  
 $(x_l, y_l)$  and  $(x_r, y_r)$  = line midpoints  
 $c_l$  and  $c_r$  = average contrast along lines  
 $w_l w_\theta w_m w_c$  = weights controlling influence  
as line similarity increases, so does  $s$   
**equation 6.3** line comparison equation

None of these techniques are infallible and can be impaired through poor quality images (whether they be low in resolution, distorted through noise, or damaged with dust and scratches) and are likely to fail if the images contain regular repetitive patterns (such as repeated dots) or if they are too bland.

#### Section 6.4 - Examples of Stereoscopic Photogrammetry in Use

One contact based possibility for using stereoscopic images to produce geometry is to use two digital cameras to capture an array of light emitting diodes (LEDs) embedded in the handle of a probe. The computer is able to use the locations of the LEDs to determine their position in space with an accuracy of 0.003 - 0.004 inches per 10 feet. This will enable the user to obtain relative distances and scales of objects in the scene. This can be extended by placing a *yardstick* in the scene to enable the user to obtain a more absolute measurement. By placing the yardstick in the image, the computer is able to compare the relative position between the target and yardstick [www.qualitydigest.com].

An alternative approach is to indicate the same three objects or points in the two images along with their known dimensions. This will allow the user to obtain the depth information for other three dimensional points in the image. This method of geometry reconstruction is good if time is limited because we only have to take two photographs of the scene, however it can take a large amount of time to process the images, especially if a complex mesh is required. The quality of the resultant mesh is also determined by the quality of the camera lens and the resolution of the images - a higher resolution image will allow us to create a higher resolution mesh [www.simple3d.com].



## Section 7 - Simple Implementation

In this section we describe and discuss a simple implementation for creating three dimensional data using photogrammetry and stereo image pairs. This implementation has been undertaken using the C programming language and uses the OpenGL libraries to display graphics on screen. The source code compiles successfully under a Linux operating system although should also work with little or no modification under a Windows environment with appropriate OpenGL support. The images used by the program can be obtained from any source (digital or otherwise) but must be stereoscopic, taken from two slightly offset cameras if the technique is to work. Potentially any image size and format would be compatible with this implementation, however for the benefits of simplicity we have limited compatibility to bitmap images (.bmp) and size to 360 by 288 pixels (half PAL resolution). Larger and smaller source images can of course be used if scaled appropriately beforehand. Examples of the images used to test the implementation can be found in appendix C along with a full code listing in appendix F and appendix G.

Once the program is started the images are displayed side by side on screen along with a three dimensional view of the scene (including a grid and coloured axis for reference and orientation) as shown in figure 7.1. Before geometry reconstruction can begin, the images must be calibrated by following the prompts on the user's shell which identify the bottom left and top right corner of each image. While not absolutely necessary for this implementation as the image size is strictly regulated, it does have several benefits. Firstly, if the user is running the program with images smaller than 360 by 288, but surrounded by an empty border, the program can identify the correct image boundaries. Secondly, it also allows the program to confirm that subsequent operations are also within the image boundaries, avoiding potentially erroneous data and geometry. Once image calibration is complete, the user is able to insert three-dimensional geometry and navigate around the scene.

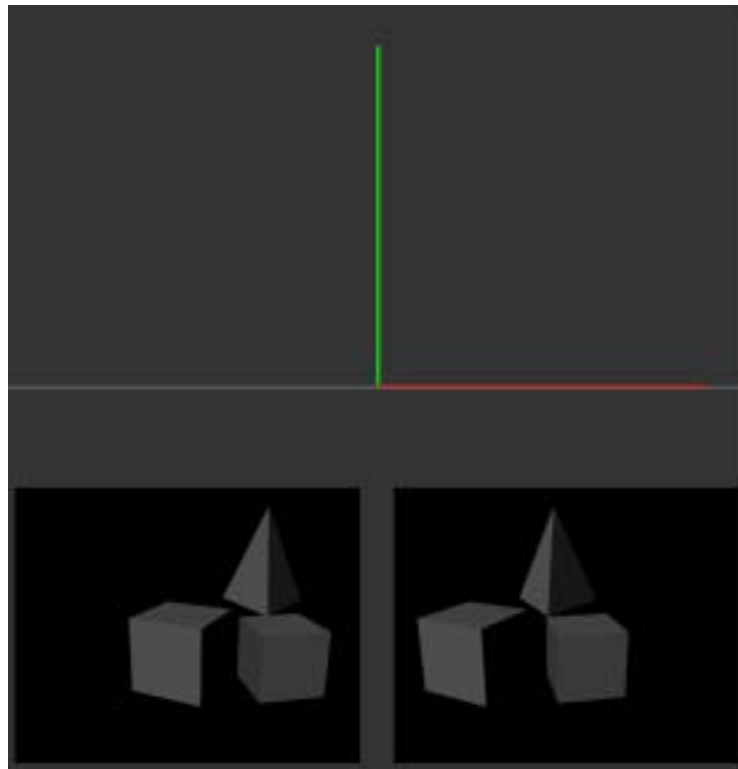


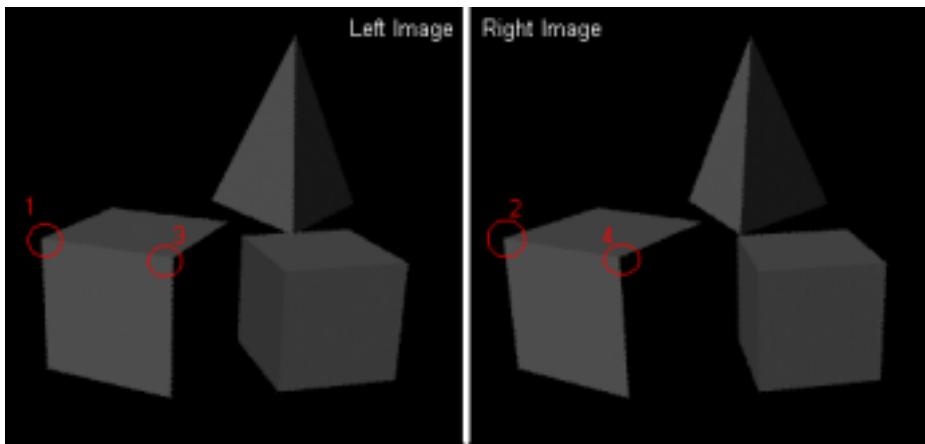
figure 7.1 screenshot of implementation interface

### Section 7.1 - Creating Three Dimensional Geometry in Theory

In this implementation photogrammetry is used to calculate the three dimensional data, although this technique has already been covered in some detail we now discuss how this has been applied to the current implementation. The user can instruct the program to insert a new line or edge into the scene by pressing 'I' on the keyboard. This changes the execution of the program and stops the refresh of the OpenGL display and instead the user is prompted to identify four key points in the two images (figure 7.2a) by clicking the left

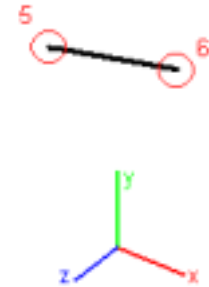


mouse button. The position of the mouse click within the window is recorded for each point and used to calculate the three-dimensional coordinates of the edge.



**figure 7.2a** inserting an edge into the three dimensional scene

**Three Dimensional Line**



**figure 7.2b** the line in 3D

This process is best explained when considered with the sample data shown in table 7.1, which was obtained when the program was executed using the images shown in figure C.4a and figure C.4b. In stage 1 the user is prompted to identify point 1 through which we obtain the x and y coordinates of that point relative to the program window (which has its origin at the top left hand corner). Using the data obtained from the calibration of the two images, these coordinates are transformed to be relative to the image (which for point 1 is the left image) with the origin at the bottom left corner. In stage 2, this process is repeated for point 2, x and y coordinates of the point relative to the window are obtained and transformed using the data from the calibration process to be relative to the right image. The same is done for points 3 and 4 in stages 3 and 4 respectively.

At the end of stage 4, we have two possible x and y coordinates for each point in the final line, for example, vertex 5 on the three dimensional line can have the x and y coordinate of either point 1 from the left image, or, point 2 from the right image. To overcome this we chose a reference or base image, which in this implementation is always the left image of the pair. This means that we use the x and y coordinates of each vertex in the left image as the x and y coordinates of the corresponding vertices in the three-dimensional scene. Therefore, after stage 1 we also have the x and y coordinate of point 5, and after stage 3 we have the x and y coordinate of point 6.

		Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6
Point 1	x	126 → 119	119	119	119	calculate disparity	119
	y	644 → 151	151	151	151		151
Point 2	x	-	427 → 25	25	25		25
	y	-	645 → 150	150	150		150
Point 3	x	-	-	201 → 194	194		194
	y	-	-	655 → 140	140		140
Point 4	x	-	-	-	493 → 91		91
	y	-	-	-	655 → 140		140
Point 5	x	119	119	119	119		119
	y	151	151	151	151		151
Point 6	z	-	-	-	-		94
	x	-	-	194	194		194
	y	-	-	140	140		140
	z	-	-	-	-		103

**table 7.1** sample data obtained using figure C.4a and figure C.4b

We are now able to calculate the disparity, and so depth, of each vertex in the final scene. As has been discussed in section 6, this is a simple subtraction of the x coordinates from the left and right images and is summarised in equation 7.1. This technique can be applied to each pair of vertices that occur in both the left and right images to recreate the entire scene. In appendix D the result of using the program with the images in figure C.4a and figure C.4b is shown from top, side, front and perspective views.

point 5	x	point 1 x	119	119
	y	point 1 y	151	151
	z	point 1 x - point 2 x	119 -25	94

equation 7.1 calculating the disparity or depth of a point

## Section 7.2 - Creating Three-Dimensional Geometry in Code

After examining the theoretical implementation of the program, we can start to look at how this has been implemented in code. The main requirement of the program is to store a number of pieces of data (which is unknown at compilation time) which store the coordinates necessary to recreate edges in the scene. This has been achieved by using a self-referential data structure to create a singly linked list, each node of which stores the necessary data. Although this code has been adapted for the purposes described here, it is based on that described in [Deitel *et al* 1992]. The data structure used for each node of the linked list is given in listing 7.1 and can be found in context with the complete source code in appendix G. Based on the description given in section 7.1, the data structure holds the x and y coordinate of the start and end vertices of the line relative to the left image, and the x coordinate of the start and end vertices of the line relative to the right image. The structure also stores the depth (or z coordinate) of the start and end vertices.

```

1.  struct line_node{
2.      int start_left_x;
3.      int start_left_y;
4.      int start_right_x;
5.      float start_depth;
6.      int end_left_x;
7.      int end_left_y;
8.      int end_right_x;
9.      float end_depth;
10.     struct line_node *next_line_pointer;
11. };
...
12. typedef struct line_node line_list_node;
12. typedef line_list_node *line_node_pointer;
...
14. line_node_pointer line_list_start = NULL;
...
```

listing 7.1 self referential data structure

The linked list is constructed and maintained using functions adapted from [Deitel *et al* 1992], however data is only inserted when the user prompts the program that he/she would like to create a new three-dimensional line. When the user wishes to insert a new line a complex series of "if" statements ensure that the correct data is obtained from the user, a sample of which is shown in listing 7.2. This series of statements is spread across two functions. The first is in the display function (display\_func in appendix G), which controls the contents of the display buffer in OpenGL. When the user wishes to insert a new line, the display function suspends the continuous redisplay of the display buffer, instead prompting the user to identify a specific point on the image. The second series of "if" statements in the mouse function (mouse\_func in appendix G) identify the change in state of the mouse required by the prompt (i.e. the mouse click) and ensure that the data that is collected is stored in the correct variables.

```

1.  ...
2.  if (add_line == TRUE)
3.  {
4.      glutSwapBuffers();
5.      if (start_left_x_position == FALSE && start_left_y_position == FALSE &&
          start_right_x_position == FALSE && end_left_x_position == FALSE &&
          end_left_y_position == FALSE && end_right_x_position == FALSE)
6.      {
7.          printf("Digit : add line ...\n");
8.          printf("Digit : click on start point of first vertex in left image ...\n");
9.      }
10.     else if ...
        ...
        ...
11. if (down == GLUT_UP && add_line == TRUE && start_left_x_position == FALSE &&
    start_left_y_position == FALSE && start_right_x_position == FALSE &&
    end_left_x_position == FALSE && end_left_y_position == FALSE &&
    end_right_x_position == FALSE)
12. {
13.     printf("        ... mouse position x = %d y = %d\n", x, y);
14.     start_left_x_position = x;
15.     start_left_y_position = y;
16.     glutSwapBuffers();
17.     glutPostRedisplay();
18. }
19. else if ...

```

listing 7.2 prompting the user

When all of the necessary data has been collected, the display function once again allows the continual refresh of the display buffer after a new node has been inserted into the linked list, and it is at this stage that the depth is calculated for the new line. Initially the x and y coordinates are transformed from being relative to the OpenGL window to being relative to the source image and written into a new node of the linked list (lines 1 and 2, 7 and 8 in listing 7.3). Then, in lines 5 and 10 of the code, the depth of the two points is calculated by subtracting to x coordinate of the vertex in the right image from the x coordinate of the vertex in the left image.

```

1.  new_node->start_left_x = start_left_x_position - left_image_left_position;
2.  new_node->start_left_y = left_image_bottom_position - start_left_y_position;
3.  new_node->start_right_x = start_right_x_position - right_image_left_position;
4.
5.  new_node->start_depth = new_node->start_left_x - new_node->start_right_x;
6.
7.  new_node->end_left_x = end_left_x_position - left_image_left_position;
8.  new_node->end_left_y = left_image_bottom_position - end_left_y_position;
9.  new_node->end_right_x = end_right_x_position - right_image_left_position;
10.
11. new_node->end_depth = new_node->end_left_x - new_node->end_right_x;

```

listing 7.3 calculating the depth

Once the data is stored in the new node, and the node inserted into the list, the line(s) are displayed on the screen on every redisplay of the display buffer. This is done by traversing the linked list from head to tail and drawing a line in OpenGL (using GL\_LINES) from the start point to the end point using the data stored in each node. When displaying the lines we transform the line coordinates for the final time to be around the origin of the world space. Initially, the world origin in three-dimensional space will be the equivalent to the bottom left hand corner of the reference image, with the geometry in the positive x/y space. To make the geometry easier to view, we place the centre of the reference image at the origin; this is done by subtracting a horizontal and vertical adjustment value from the coordinates. We also multiply each point by a scale value; this means that larger scenes or objects can be displayed more easily in a smaller viewing volume, or smaller objects can be enlarged to be more visible. The user is also able to artificially inflate or deflate the depth of the object by altering the depth factor, this means that if disparity is low across the two images, resulting in a small depth even for large objects, it can be corrected manually and is shown in lines 8 and 11 in listing 7.4.

```

1.  if (current_node != NULL)
2.  {
3.      while(current_node != NULL)
4.      {
5.          glBegin(GL_LINES);
6.          glVertex3f((current_node->start_left_x - horizontal_adjustment) * scale,
7.                  (current_node->start_left_y - vertical_adjustment) * scale,
8.                  current_node->start_depth * depth_factor * scale);
9.          glVertex3f((current_node->end_left_x - horizontal_adjustment) * scale,
10.                 (current_node->end_left_y - vertical_adjustment) * scale,
11.                 current_node->end_depth * depth_factor * scale);
12.      glEnd();
13.      current_node = current_node->next_line_pointer;
14.  }
15.  }

```

listing 7.4 displaying lines

This group of procedures is repeated for each new line that is added to the three dimensional scene, and has been proved to be very efficient, with program execution being relatively fast and simple. Having said this, it is recognised that the "if" statements are rather large and could prove a potential bottle neck if the program is pushed to its limits.

### Section 7.3 - Evaluating the Results

Appendix D shows the results of using the program with the two images show in figure C.4a and figure C.4b from the top, side, front and perspective viewpoints. Whilst these images are not particularly clear, and it is slightly difficult to gain a good perception of the three-dimensional space that the lines occupy, they do show that the program has been able to generate objects that contain a depth that is approximately similar to the depth of the source object. This result is more obvious when the user is free to navigate through the scene manually, however we can see the potential that this technique has.

Having said, this approach does have some limitations, one of which became very apparent during testing. When the program was tested using the images in figure C.1, figure C.2 and figure C.3 some three-dimensional data was obtained, however it was marginal and in the OpenGL scene the objects appeared to be wafer thin. While this was improved artificially by increasing the depth factor, it proved insufficient. The cause of this relates to the resolution of the two images and the accuracy of the program as a whole.

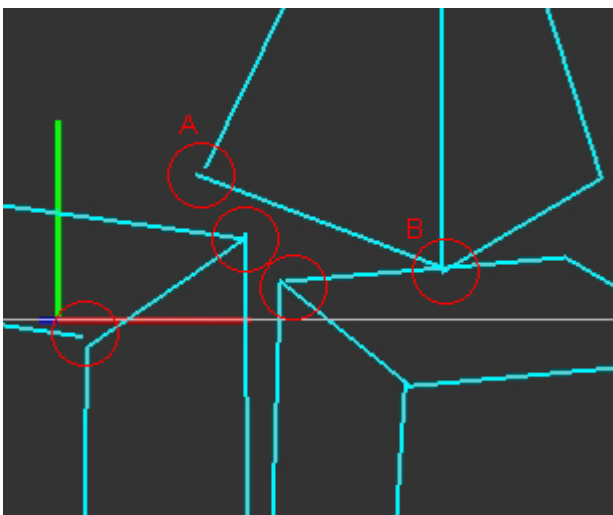


figure 7.3 errors caused by the implementation

If the resolution of the two images is poor, than there will only be a few pixels difference between the x position of a point in the left image, and the x position of the corresponding point in the right image. This means that the depth will only be a few pixels, which in turn is only a few units in OpenGL. This can be solved in part by increasing the image resolution and by artificially inflating the depth. The second cause of this problem is the accuracy of the program. The selection of points in the two images is only pixel accurate, and as a result no floating point data will be created. This pixel inaccuracy also means that we witness the effects shown in figure 7.3. This means that lines and vertices that would physically occupy the same space can intersect (as in example B) or not meet at all (as in example A). This can be easily rectified by creating a *snapping* function that identifies if selected points should be in the same position as other points already selected and responds accordingly.

Another disadvantage of this implementation is that it proves an arduous task to create more than a handful of lines inside of the program. Each line that is created requires the user to accurately click on four separate points. This means that to create 20 lines the program would require 80 individual mouse clicks. This obviously proves time consuming and so would not be effective for large complex scenes. This is

unfortunately an inherent property of the active stereoscopic technique that we have used in this implementation.

#### Section 7.4 - Extensions and Improvements

There are a number of possible extensions and improvements that could be considered for use in conjunction with the implementation described here. The first, that could be of most benefit to the user and the objects/scenes created would be to allow the program to create curves as opposed to straight lines and edges. As the implementation stands currently, the user would have to combine many tiny line segments to create a complete curve. Curves could be created by identifying a number of control points and then interpolating a curve through them.

Another feature that could be added to increase the speed with which scenes and objects could be created is to use known object topology. Given a set number of vertices and a description of the object, we would be able to allow the computer to generate the remaining vertices automatically. For example, by instructing the computer that four specific vertices form a cube, the program would be able to use pre-coded algorithms to deduce the location of the remaining points. We started to use this concept in the implementation here, however it was still being developed when this report was being written. Instead of selecting two points in the left and right image that defines a single line, the user can identify eight points across the two images that identify four vertices of the cube to be reconstructed. This is demonstrated in figure 7.4, and in our current implementation we were able to reconstruct lines AB, BC, DB, AE, ED, CG and DG, however accuracy is lacking somewhat and requires further development.

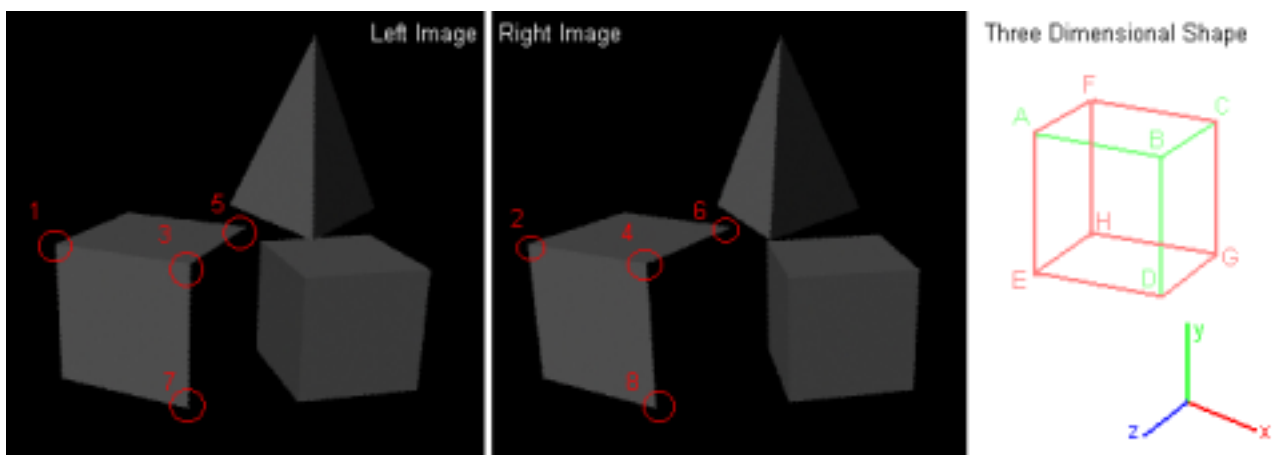


figure 7.4 creating regular shapes

Two further, however extremely complex extensions, would be to change this implementation from an active stereoscopic method to a passive method, implementing one of the procedures stated in section 6 to match corresponding points across the two images. This would also increase computation time but would have to be weighed up against the automated approach that this provides. The second, would be to use more than two images, integrating all of the data collected together to form a complete model of the objects/scene. This would also allow surfaces that were previously occluded, to be identified and digitised.

## Section 8 - Conclusion

In this report we have examined a number of the existing techniques that can and are used to construct three dimensional objects and scenes in the digital domain. After categorising the techniques as either hardware or software based, we have discovered a number of common features among them. Hardware based processes typically scan the subject, relying on a physical model of the object existing. These techniques can either be contact (touch probe) or non-contact (laser scanner) based, and can be destructive (arguably a CAT scanner) or non-destructive (MRI scanner). They also range in price and their abilities to reproduce accurate data. Software based techniques on the other hand reconstruct geometry based on one or more images of the object, therefore meaning that drawn conceptual objects and scenes can be digitised. We have also identified that while some methods are very specific in their application, such as MRI scanners, others are broader in their abilities, such as the commercial software applications.

After completion of the research, an implementation was considered that allows the user to recreate three-dimensional data from stereoscopic image pairs. This technique proved successful and showed potential, especially when the extensions and improvements are also considered. While the implementation proved to work best with the images identified earlier that had a higher resolution and larger disparity, we have shown that the remaining images will also work with the implementation of some of the extensions suggested.

Over the course of this research and subsequent documentation we have shown how the new methods of digitisation have reduced the time that projects spend in the production pipeline, and how they allow three-dimensional data to be captured more efficiently and cost effectively. These methods and techniques have proved innovative in their area of use, serving their purpose by simplifying the digitisation process for the artist, and in many cases they remove the limitations of traditional modelling discussed earlier. On a more personal note, this project has been personally innovative. The author has a strong interest in procedural techniques in all areas of the computer industry, whether they are procedural layout tools, animation scripts, or texture shaders. It has been interesting to closely examine all of these techniques as well as attempt to implement one of them.

However, to finally conclude, we are reminded of the debate that is currently raging between animators and motion capture supporters. With animation being as time consuming and difficult as modelling itself, motion capture was designed and created (just as the digitisation techniques described here have been) to speed up the animation process, creating accurate lifelike animations of humans in real time. Animators believe that motion capture is the end of their art form - often referring to it as *evil* - and that they will no longer be required (a true validation of the success of this process). Consequently they do not overly support the use of motion capture in computer animation, especially when they are required to *touch up* mistakes in the capture data. Its supporters however, argue that motion capture is the most efficient way to create animation for today's purposes, especially when production times and costs are considered. While the debate is far more detailed than we have summarised here, we are left to wonder if the same debate will apply to these digitisation techniques. The techniques described here apparently render the traditional modeller useless, leaving them only required for the few tasks that these approaches are unsuitable for. Therefore, will these traditional modellers take the stance of the animator and disagree with and try to rebuke these new digitisation techniques, or will they embrace the new technology and work with it for the benefit of the industry as a whole?

## **Section 9 - Acknowledgements**

Ian Stephenson for tutorials during the winter term of 2003 and for the use of his stereoscopic camera and scanning of 35mm negative film, and Anargyros (Ari) Sarafopoulos for tutorials and implementation advice during the spring term of 2004.

*Java, Renderman, AutoCAD, Maya, Houdini, Canoma, Image Modeler* and *PhotoModeler* are trademarks and used respectfully without permission

All images and diagrams (with the exception of my own) have been used respectfully without permission of the owner, where possible the image source has been provided in section 10.

## Section 10 - References

- [[www.aeic.alaska.edu](http://www.aeic.alaska.edu)] Title: *UAF Seismology Laboratory* Author/Editor: AEIC URL: [www.aeic.alaska.edu/Input/martin/physics212/cat.html](http://www.aeic.alaska.edu/Input/martin/physics212/cat.html) [last accessed 09/03/04]
- [[www.ageo.co.uk](http://www.ageo.co.uk)] Title: *AG Electro-Optics Ltd* Author/Editor: AG Electro-Optics URL: [www.ageo.co.uk](http://www.ageo.co.uk) [last accessed 09/03/04]
- [[www.aliaswavefront.com](http://www.aliaswavefront.com)] Title: *Alias* Author/Editor: Alias Systems URL: [www.aliaswavefront.com](http://www.aliaswavefront.com) [last accessed 09/03/04]
- [[www.apple.com](http://www.apple.com)] Title: *Apple* Author/Editor: Apple URL: [www.apple.com](http://www.apple.com) [last accessed 09/03/04]
- [[www.autodesk.co.uk](http://www.autodesk.co.uk)] Title: *Autodesk - Autodesk Revit 6* Author/Editor: Autodesk Inc URL: [www.autodesk.co.uk](http://www.autodesk.co.uk) [last accessed 09/03/04]
- [[www.axila.com](http://www.axila.com)] Title: *Axila* Author/Editor: Axila URL: [www.axila.com](http://www.axila.com) [last accessed 09/03/04]
- [[biblip.vet.cornell.edu](http://biblip.vet.cornell.edu)] Title: Unknown Author/Editor Unknown URL: [biblip.vet.cornell.edu/godfrey/mri.html](http://biblip.vet.cornell.edu/godfrey/mri.html) [last accessed: website no longer available]
- [Clowes 1970] CLOWES, M. B. 1970 *On Seeing Things* Artificial Intelligence 2:79-116
- [Comninos 1997] COMNINOS, P. 1997 *Computer Animation Systems*. Found under *Computer Controlled Studio Housekeeping* Page 54
- [Curless 2000] CURLESS, B., 2000 SIGGRAPH Course 3D *Photography*
- [Delaunay et al 1995] DELAUNAY, L. A. P., et al. 1995 *Triangulation in Three Dimensions* IEEE Computer Graphics and Applications
- [Deitel et al 1992] Deitel et al. 2001 *C How to Program 3rd Edition*
- [[www.inlandrevenue.gov.uk](http://www.inlandrevenue.gov.uk)] Title: *Inland Revenue* Author/Editor: Her Majesty's Government URL: [www.inlandrevenue.gov.uk](http://www.inlandrevenue.gov.uk) [last accessed 09/03/04]
- [[focus.aps.org](http://focus.aps.org)] Title: *Physical Review Focus* Author/Editor: American Physical Society URL: [focus.aps.org/stroy/v4/st13](http://focus.aps.org/stroy/v4/st13) [last accessed 09/03/04]
- [Grimstead et al 1995] GRIMSTEAD, I. J., et al. 1995 *Creating Solid Models from Single 2D Sketches* SIGGRAPH 1995
- [Gritz et al 2001] GRITZ, L., et al. 2001 SIGGRAPH Course 48 *Advanced Renderman 3 : render Harder*
- [[www.howstuffworks.com](http://www.howstuffworks.com) 01] Title: *How Stuff Works "How MRI Works"* Author/Editor: GOULD, T. A. URL: [electronics.howstuffworks.com/mri.htm](http://electronics.howstuffworks.com/mri.htm) [last accessed 09/03/04]
- [[www.howstuffworks.com](http://www.howstuffworks.com) 02] Title: *How Stuff Works "How CAT Scans work"* Author/Editor: HARRIS, T. URL: [science.howstuffworks.com/cat-scan.htm](http://science.howstuffworks.com/cat-scan.htm) [last accessed 09/03/04]
- [[www.hslmc.cam.ac.uk](http://www.hslmc.cam.ac.uk) 01] Title: *Magnetic Resonance Imaging (MRI) at the Herchel Smith Laboratory* Author/Editor: University of Cambridge URL: [www.hslmc.cam.ac.uk/index\\_hires.html](http://www.hslmc.cam.ac.uk/index_hires.html) [last accessed 09/03/04]
- [[www.hslmc.cam.ac.uk](http://www.hslmc.cam.ac.uk) 02] Title: *Seeing the Invisible Human* Author/Editor: EPSRC URL: [www.hslmc.cam.ac.uk/epsrc/index.html](http://www.hslmc.cam.ac.uk/epsrc/index.html) [last accessed 09/03/04]
- [Huffman 1971] HUFFMAN, D. A. 1971 *Impossible Objects as Nonsense Sentences* pages 295-323 Machine Intelligence 6 New York American Elsevier
- [Igarashi et al 1999] IGARASHI, T., et al. 1999 *Teddy : A Sketching Interface for 3D Freeform Design* SIGGRAPH 1999
- [[www.imapctstudiostc.com](http://www.imapctstudiostc.com)] Title: *...: Impact Studios LLC ...* Author/Editor: Impact Studios LLC URL: [www.impactstudiostv.com/laser\\_scan\\_site/htm/technology.htm](http://www.impactstudiostv.com/laser_scan_site/htm/technology.htm) [last accessed 09/03/04]
- [Kerlow 2000] KERLOW, I. V., 2000 *The Art of 3-D Computer Animation and Imaging 2nd Edition* Published by John Wiley & Sons, Inc. Page 54.
- [Konishi et al 1986] KONISHI, M., et al. 1986 *Neural Map of Interaural Phase Difference in the Owl's Brainstem* Proc Natl Acad Sci USA 83:8400-8404
- [Levoy et al 2000] LEVOY, M., et al. 2000 *The Digital Michelangelo Project : 3D Scanning of Large Statues* SIGGRAPH 2000
- [Menache 1999] MENACHE, A., *Understanding Motion Capture for Computer Animation and Video Games* Published by Morgan Kaufmann
- [[www.metacreation.com](http://www.metacreation.com)] Title: *MetaCreations* Author/Editor: MetaCreations URL: [www.metacreation.com](http://www.metacreation.com) [last accessed 09/03/04]
- [[neurovia.umn.edu](http://neurovia.umn.edu)] Title: *NeuroImaging Visualisation and Data Analysis* Author/Editor: NeuroVia Lab URL: [neurovia.umn.edu/home/kelly/IsoParcel\\_demo/isolate\\_parcellate.html](http://neurovia.umn.edu/home/kelly/IsoParcel_demo/isolate_parcellate.html) [last accessed 09/03/04]
- [[www.nlm.nih.gov](http://www.nlm.nih.gov)] Title: *US National Library of Medicine* Author/Editor: National Library of Medicine URL: [www.nlm.nih.gov/research/visible/visible\\_human.html](http://www.nlm.nih.gov/research/visible/visible_human.html) [accessed 09/03/04]
- [Peters et al 1997] PETERS, T. M., et al. 1997 *The Fourier Transform in Biomedical Engineering* Published by Birkhäuser



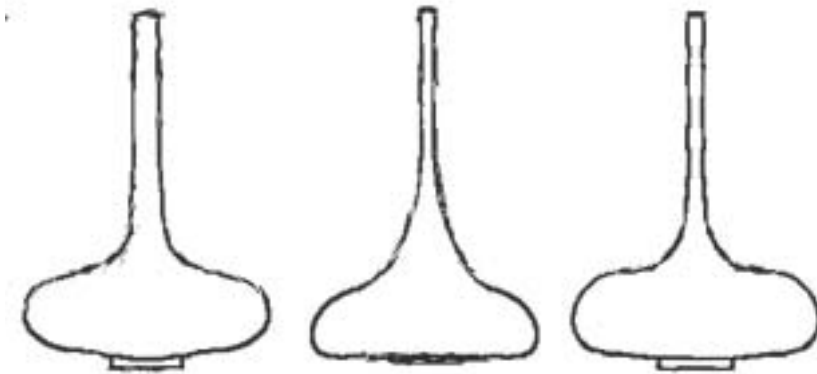
- [**www.photomodeler.com**] Title: *PhotoModeler Photogrammetry Software* Author/Editor: EOS Systems URL: [www.photomodeler.com](http://www.photomodeler.com) [last accessed 09/03/04]
- [**Porr et al 1998**] PORR, B., et al. 1990 *How to "hear" Visual Disparities : Real-Time Stereoscopic Spatial Depth Analysis Using Temporal Resonance* Biological Cybernetics Springer-Verlag
- [**Prusinkiewicz et al 1990**] PRUSINKIEWICZ, A. et al, 1990 *The Algorithmic Beauty of Plants* Published by Springer-Verlag
- [**www.qualitydigest.com**] Title: *Quality Digest Magazine* Author/Editor: Various URL: [www.qualitydigest.com](http://www.qualitydigest.com) [last accessed 09/03/04]
- [**Russell et al 2002**] RUSSELL, S., et al. 2002 *Artificial Intelligence: a Modern Approach 2nd Edition*
- [**www.realviz.com**] Title: *RealViz - Image Processing Software and Solutions for Content Creation* Author/Editor: RealViz URL: [www.realviz.com](http://www.realviz.com) [last accessed 09/03/04]
- [**Shreiner et al 2003**] Shreiner, D., et al. 2003 *OpenGL Programming Guide 3rd Edition*
- [**www.sidefx.com**] Title: *Side Effects Software Inc* Author/Editor: Side Effects Software URL: [www.sidefx.com](http://www.sidefx.com) [last accessed 09/03/04]
- [**www.simple3d.com**] Title: *Simple 3D - 3D Scanners, Digitisers, and Software for 3D Models and Measurements* Author/Editor: Unknown URL: [www.simple3d.com](http://www.simple3d.com) [last accessed 09/03/04]
- [**www.umd.edu**] Title: *University of Maryland* Author/Editor: University of Maryland URL: [www.umd.edu](http://www.umd.edu) [last accessed 09/03/04]
- [**US EPA**] Title: *Environmental Protection Agency* Author/Editor: US EPA URL: [www.epa.gov](http://www.epa.gov) [last accessed 09/03/04]
- [**Waltz 1975**] WALTZ, D. L. 1975 *Understanding Line Drawings of Scenes with Shadows* in P. Winston, editor, *The Psychology of Computer Vision*, McGraw-Hill New York
- [**William et al 2004**] William et al. 2004 *Seeing 3D from 2D Images* Source Unknown
- [**Xu et al 1998**] XU, B., et al 1998 *Evaluation of Fabric Smoothness Appearance Using a Laser Profilometer* Textile Research Journal 68(12) 900-906 1998
- [**Zelevnik et al 1996**] ZELEVNIK, R. C., et al. 1996 *SKETCH : An Interface for Sketching 3D Scenes* SIGGRAPH 1996

## Appendix A - Verbal Description of a Vase

The text that follows is taken from [Kerlow 2000] and constitutes the verbal description of a flower vase.

- the vase has a very long neck and a short, round base
- the neck is about five times the height of the base, the width of the base is about twice its own height
- the cylindrical neck grows out of the base slowly
- at the point where the neck touches the base it has a width that equals the height of the base
- as it moves upward the neck gets narrow, and as it passes the first fifth of its height the neck reaches a thin, delicate width that remains constant until the end of the neck
- a small section of the oval shape the constitutes the base of the vase is sliced off so that the bottom of the vase is flat
- the resulting sharp edge at the bottom of the base is rounded off just a little bit
- halfway between the edge of the base and its centre, a thin slice of a short cylinder is attached to the base

Kerlow proposes that this description could be interpreted in many different ways, and should the reader be asked to draw the vase, and number of possible results could be obtained, as shown in figure A.1.



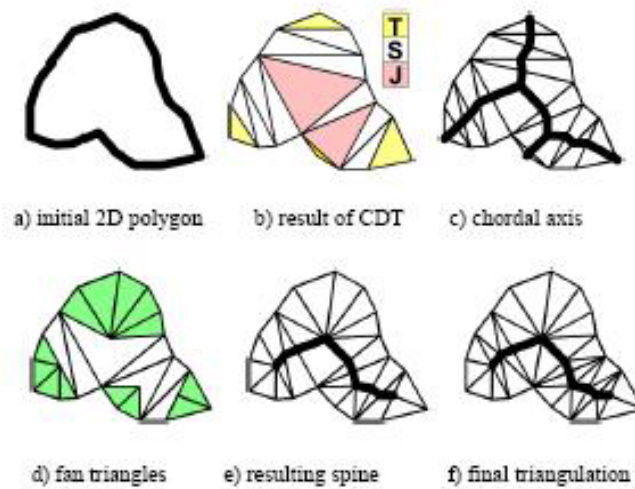
**figure A.1a** different interpretations of the description of a vase based on [Kerlow 2000]



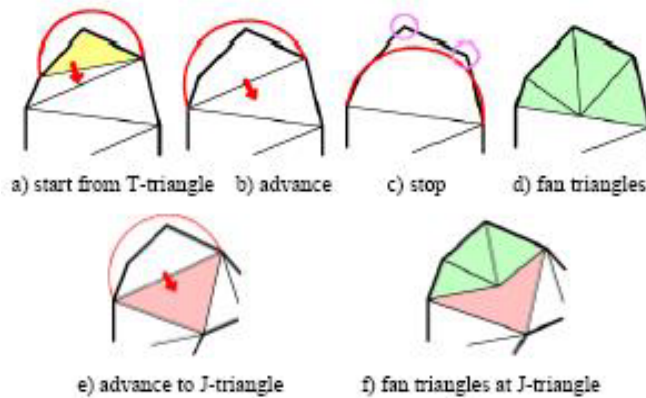
**figure A.1b** my drawing of the vase

## Appendix B - Teddy : A Sketching Interface for 3D Freeform Design

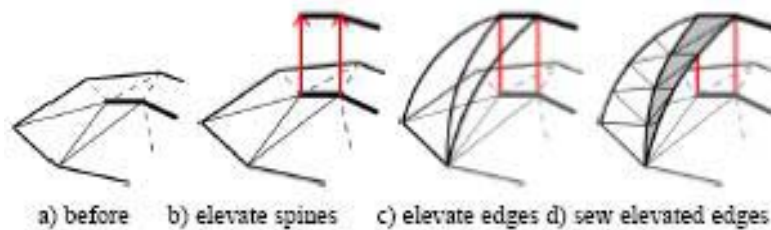
The diagrams below are taken from [Igarashi *et al* 1999] and shown the creation of a three dimensional mesh from a single two dimensional freeform stroke provided by the user



**figure B.1** finding the chordal axis and spine of the stroke



**figure B.2** pruning the closed planar polygon



**figure B.3** polygonal mesh construction

## Appendix C - Examples of Stereoscopic Images



**figure C.1a** left image taken by myself using a camera provided by Ian Stephenson, lacks sufficient resolution and disparity to best demonstrate the tool



**figure C.1b** right image taken by myself using a camera provided by Ian Stephenson, lacks sufficient resolution and disparity to best demonstrate the tool



**figure C.2a** left image provided by Ian Stephenson, lacks sufficient resolution and disparity to best demonstrate the tool



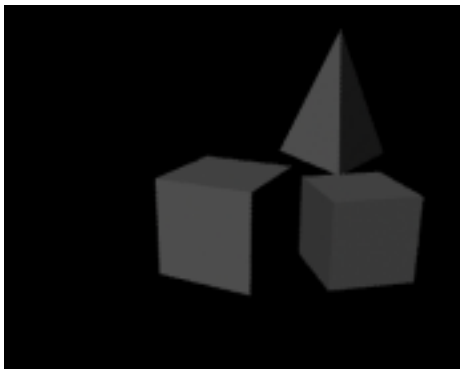
**figure C.2b** right image provided by Ian Stephenson, lacks sufficient resolution and disparity to best demonstrate the tool



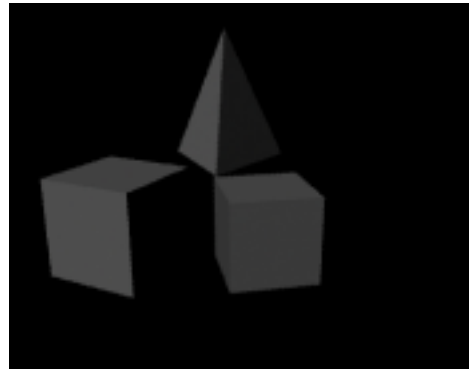
**figure C.3a** left image provided by Ian Stephenson, lacks sufficient resolution and disparity to best demonstrate the tool



**figure C.3b** right image provided by Ian Stephenson, lacks sufficient resolution and disparity to best demonstrate the tool



**figure C.4a** left image created in *Maya* provides greater disparity to demonstrate the implementation of the tool



**figure C.4b** right image created in *Maya* provides greater disparity to demonstrate the implementation of the tool

## Appendix D - Results

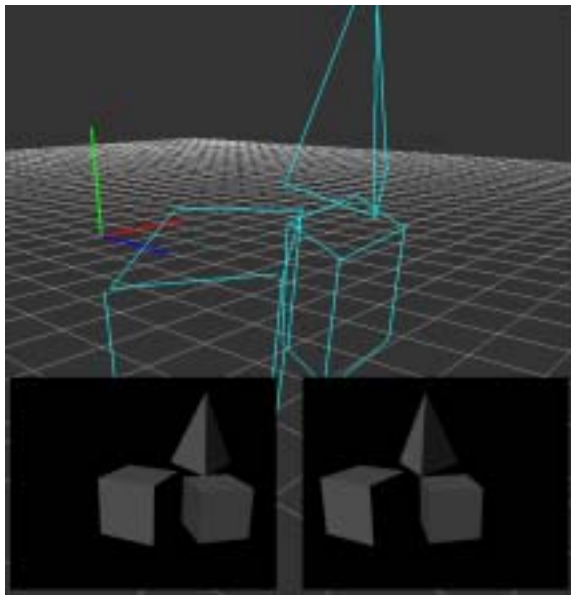


figure D.1 perspective view

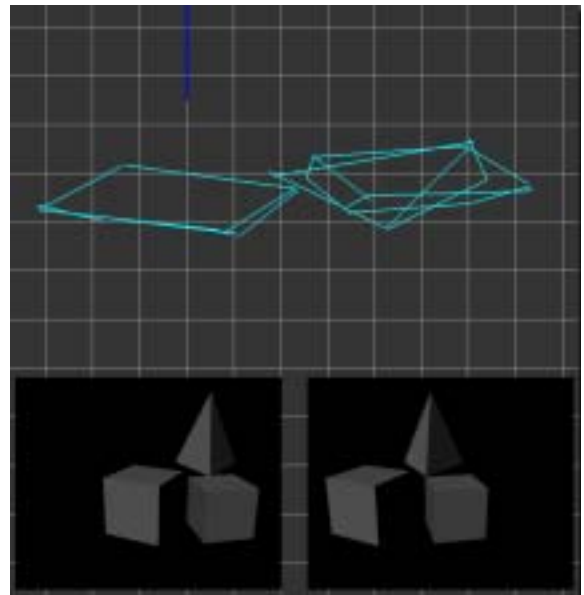


figure D.2 top view

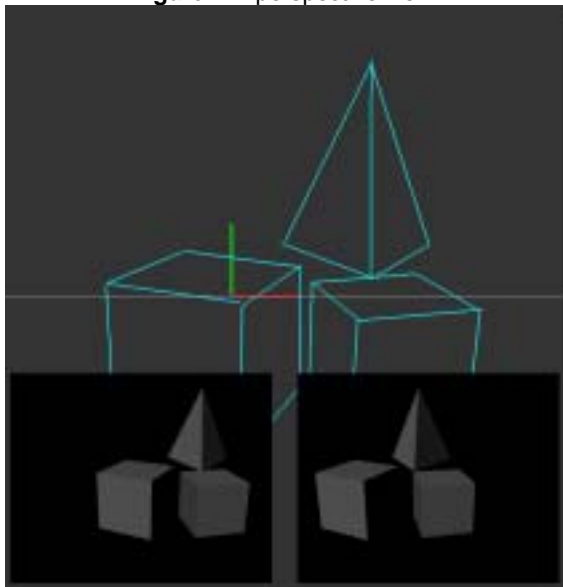


figure D.3 front view

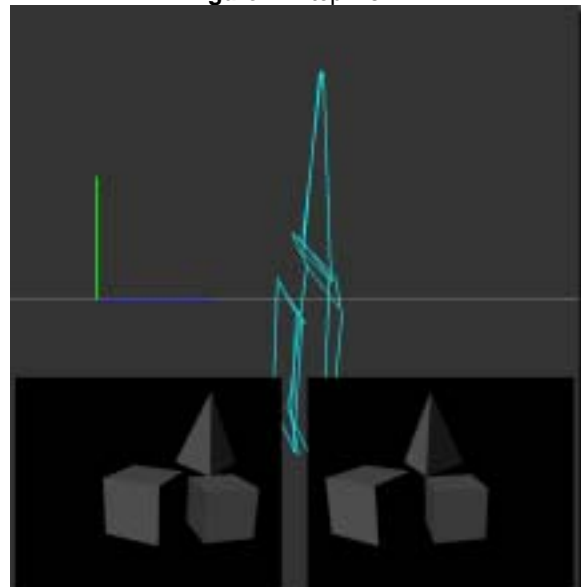


figure D.4 side view

## Appendix E - User Manual

Included here is a brief user manual that explains how to execute and use the implementation described in this report.

The source code (included in the file digit.c and in appendix G) can be compiled to run on a Linux machine with appropriate OpenGL libraries using the command

```
$ make
```

Once compiled, the program can be executed with the following

```
$ ./digit <left_image> <right_image>
```

Where <left\_image> and <right\_image> are replaced with two valid bitmap files which contain the left and right images of the stereoscopic pair. The user should then follow the prompts on the shell, clicking on the points as instructed, to calibrate the two images ready for use.

Once calibration is complete, the user will be able to navigate around the scene using the following key controls.

Pan	+ve	x	axis	right arrow key
	-ve	x	axis	left arrow key
	+ve	y	axis	up arrow key
	-ve	y	axis	down arrow key
	+ve	z	axis	page up key
	-ve	z	axis	page down key
Rotate	+ve	x	axis	a
	-ve	x	axis	z
	+ve	y	axis	v
	-ve	y	axis	b

**table E.1** user controls

To add a new line/edge to the scene, press 'l' and follow the prompts on the shell. To add a new cube to the scene (this section of code is incomplete as described earlier in section 7.4) press 'c' and follow the prompts on the shell.

The remaining key controls can be used to artificially alter the depth of the objects and as well as changing the scale of the entire scene to make it easier to view.

d	increment depth
f	decrement depth
,	increment scene scale
.	decrement scene scale

**table E.2** user controls

There are a few remaining controls that implement simple tasks and are documented here

o	print camera position coordinates and orientation
r	recalibrate the bottom left and top right points of the images
h	display a help dialog on the shell
-	zoom out of the scene - obsolete and no longer used
=	zoom into the scene - obsolete and no longer used

**table E.3** user controls

**Appendix F - Makefile Code Listing**

```

# =====
#       Author       : Mark Streatfield
#       Contact      : a1490378@bournemouth.ac.uk
# =====
#       File        : makefile
#       Version     : 1_01
#       Date Created : 25-02-2004
# =====
#       About       : this makefile can be used to compile the source file "digit.c"
# =====

# =====
#       Macro Definitions
# =====

INCLUDE_DIRECTORY = -I/OpenGL/include -I/usr/include
GCC_FLAGS = -g -Wall
OPEN_GL_AND_X_LIBS = -L /usr/X11/lib -L/usr/X11R6/lib/ -lX11 -lXext -lXmu -lXt -lXi -lSM -lICE \
                    -lglut -lGL -lGLU

# =====
#       Targets and Rules
# =====

digit : digit.o
        gcc -o digit digit.o $(GCC_FLAGS) $(INCLUDE_DIRECTORY) $(OPEN_GL_AND_X_LIBS)

digit.o : digit.c
        gcc -c digit.c $(GCC_FLAGS) $(INCLUDE_DIRECTORY)

clean :
        rm -f *.o; rm -f digit

# =====
#       End of File
# =====

```

## Appendix G- Digit.c Code Listing

```

/* ===== */
/*      Author      : Mark Streatfield                               */
/*      Contact     : a1490378@bournemouth.ac.uk                     */
/* ===== */
/*      File        : digit.c                                         */
/*      Version     : 1_04                                           */
/*      Date Created : 25-02-2004                                     */
/* ===== */
/*      About       : this code specifies a Linux based OpenGL program for the creation of 3D */
/*                   geometry from stereo image pairs. This form of photogrammetry uses two */
/*                   images of a scene that are slightly offset to calculate a relative      */
/*                   depth for vertices in the image and so produce three dimensional data.  */
/*      Notes       : this code compiles under the ANSI C standard using the GNU C compiler */
/*                   and the supplied makefile. It has only been compiled and tested to run */
/*                   on machines running Red Hat Linux Enterprise Edition Version 3, but    */
/*                   no code has been identified as Linux specific and so should also     */
/*                   execute under a Windows operating system that has OpenGL support.    */
/*      Usage       : once compiled this program can be executed as follows                */
/*                   ./digit <left_input_image> <right_input_image>          */
/*                   where <left_input_image> and <right_input_image> are replaced with    */
/*                   valid paths to bitmap images                                     */
/*      References  : the code included in this source file has been created by myself. The */
/*                   code relating to the construction and maintenance of the linked lists */
/*                   is based on that described in [Deitel, et al 1992] although has in    */
/*                   part been adapted for my own uses. Other sources have been consulted */
/*                   for research, most notably [Shreiner, et al 2003], however to my    */
/*                   knowledge no code has been directly used or copied                */
/*      Known Bugs  : none, please post all bugs to a1490378@bournemouth.ac.uk          */
/* ===== */

/* ===== */
/*      Includes                                          */
/* ===== */

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>

/* ===== */
/*      Constant Definitions                               */
/* ===== */

#define IMAGE_WIDTH 360
#define IMAGE_HEIGHT 288
#define FALSE -1000
#define TRUE -2000

/* ===== */
/*      Self-Defined Data Type Definitions                */
/* ===== */

/* in this data structure we store the start and end vertices of a line as taken from both the */
/* left and right images, we ignore the y value of the right image to simplify the task        */
/* adapted from [Deitel, et al 1992] */

struct line_node{
    /* a linked list node to hold data for lines/edges in the program */
    int start_left_x;
    int start_left_y;
    int start_right_x;
    float start_depth;
    int end_left_x;
    int end_left_y;
    int end_right_x;
    float end_depth;
    struct line_node *next_line_pointer; /* self-referential pointer */
};

/* in this data structure we store the coordinates of four vertices that are used to form the */
/* corner of a cube, to reduce memory usage we do not store the right image coordinates      */
/* adapted from [Deitel, et al 1992] */

struct square_node{
    /* a linked list node to hold data for cubes in the program */

```



```

    int point_a_x;
    int point_a_y;
    float point_a_z;
    int point_b_x;
    int point_b_y;
    float point_b_z;
    int point_c_x;
    int point_c_y;
    float point_c_z;
    int point_d_x;
    int point_d_y;
    float point_d_z;
    struct square_node *next_square_pointer;          /* self-referential pointer */
};

/* adapted from [Deitel, et al 1992] */
typedef struct line_node line_list_node;
typedef line_list_node *line_node_pointer;

typedef struct square_node square_list_node;
typedef square_list_node * square_node_pointer;

/* ===== */
/*      Variable Declarations      */
/* ===== */

line_node_pointer line_list_start = NULL;
square_node_pointer square_list_start = NULL;

/* these variables are used as boundaries for the left and right images so we can make sure that */
/* the lines/edges/cubes created by the program fall within the image and are not erroneous */

int left_image_bottom_position = FALSE;
int left_image_top_position = FALSE;
int left_image_left_position = FALSE;
int left_image_right_position = FALSE;
int right_image_bottom_position = FALSE;
int right_image_top_position = FALSE;
int right_image_left_position = FALSE;
int right_image_right_position = FALSE;

/* these variables are used to temporarily store the coordinates of a line/edge before being */
/* stored in the linked list */

int start_left_x_position = FALSE;
int start_left_y_position = FALSE;
int start_right_x_position = FALSE;
int end_left_x_position = FALSE;
int end_left_y_position = FALSE;
int end_right_x_position = FALSE;

/* these variables are used to temporarily store the coordinates of four vertices on a cube */
/* before being stored in the linked list */

int left_image_point_a_x = FALSE;
int left_image_point_a_y = FALSE;
int left_image_point_b_x = FALSE;
int left_image_point_b_y = FALSE;
int left_image_point_c_x = FALSE;
int left_image_point_c_y = FALSE;
int left_image_point_d_x = FALSE;
int left_image_point_d_y = FALSE;
int right_image_point_a_x = FALSE;
int right_image_point_b_x = FALSE;
int right_image_point_c_x = FALSE;
int right_image_point_d_x = FALSE;

/* these are used as state variables to allow the program to determine the current state of */
/* execution and ensure the appropriate function calls are made */

int calibration = FALSE;
int add_line = FALSE;
int add_cube = FALSE;

/* variables defined here are used to store the current position of the virtual camera */

float translate_x;

```

```

float translate_y;
float translate_z;
float rotate_x;
float rotate_y;
float zoom = 1.0;
float depth_factor = 1.0;
float scale = 1.0;

/* two pointers to a location in memory to store the bitmap image data */

GLubyte *left_image_array = NULL;
GLubyte *right_image_array = NULL;

/* OpenGL display list indexes */

GLuint axis_display_list_index;
GLuint grid_display_list_index;

/* ===== */
/*      Function Name : print_help                                */
/*      Function Args : none                                      */
/*      Function Task : prints help information to the shell prompt */
/* ===== */

void print_help(void)
{
printf("      KEY              ACTION              KEY              ACTION\n");
printf("-----\n");

printf("      Esc              terminate             c              add cube\n");
printf("      Left Arrow      pan left              l              add line/edge\n");
printf("      Right Arrow     pan right             o              camera orientation\n");
printf("      Up Arrow        pan up               r              recalibrate camera\n");
printf("      Down Arrow      pan down             d              depth factor up\n");
printf("      Page Up         zoom in              f              depth factor down\n");
printf("      Page Down       zoom out             .              scale up\n");
printf("      a               rotate x             ,              scale down\n");
printf("      z               rotate x             -              zoom out\n");
printf("      v               rotate y             =              zoom in\n");
printf("      b               rotate y             h              display this help\n");
printf("      dialog\n");
printf("-----\n");
printf("      \n\n");
}

/* ===== */
/*      Function Name : delete_line_list                        */
/*      Function Args : pointer to the start of the linked list containing edge data */
/*      Function Task : deletes the entire linked list that contains the data for the lines/ */
/*                      edges in the scene                      */
/* ===== */

void delete_line_list(line_node_pointer *list_start_pointer)
{
/* adapted from [Deitel, et al 1992] */
line_node_pointer previous_node = NULL;
line_node_pointer current_node = NULL;

if ((*list_start_pointer) != NULL)
{
previous_node = current_node;

while(*list_start_pointer != NULL)
{
current_node = *list_start_pointer;

if (current_node->next_line_pointer != NULL)
*list_start_pointer = current_node->next_line_pointer;
else
*list_start_pointer = NULL;

free(current_node);
}
*list_start_pointer = NULL;
}
}

```

```

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : delete_square_list      */
/*      Function Args : pointer to the start of the linked list containing cube data      */
/*      Function Task : deletes the entire linked list that contains the data for the cubes      */
/*                      in the scene      */
/* ===== */

void delete_square_list(square_node_pointer *list_start_pointer)
{
/* adapted from [Deitel, et al 1992] */
square_node_pointer previous_node = NULL;
square_node_pointer current_node = NULL;

if ((*list_start_pointer) != NULL)
{
    previous_node = current_node;

    while(*list_start_pointer != NULL)
    {
        current_node = *list_start_pointer;

        if (current_node->next_square_pointer != NULL)
            *list_start_pointer = current_node->next_square_pointer;
        else
            *list_start_pointer = NULL;

        free(current_node);
    }
    *list_start_pointer = NULL;
}
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : print_line_list      */
/*      Function Args : the start of the linked list containing line data      */
/*      Function Task : prints the data held in the linked list containing line/edge data      */
/* ===== */

void print_line_list(line_node_pointer current_node)
{
/* adapted from [Deitel, et al 1992] */
if(current_node == NULL)
    printf("Error : no line list data to display\n");
else
{
    printf("Digit : line list data ...\n");
    while (current_node != NULL)
    {
        printf("    ... NEW LINE START ...\n");
        printf("    ... start vertex left  image x = %d\n", current_node->start_left_x);
        printf("    ... start vertex left  image y = %d\n", current_node->start_left_y);
        printf("    ... start vertex right image x = %d\n", current_node->start_right_x);
        printf("    ... start vertex      depth  = %f\n", current_node->start_depth);
        printf("    ... end  vertex left  image x = %d\n", current_node->end_left_x);
        printf("    ... end  vertex left  image y = %d\n", current_node->end_left_y);
        printf("    ... end  vertex right image x = %d\n", current_node->end_right_x);
        printf("    ... end  vertex      depth  = %f\n", current_node->end_depth);
        printf("    ... CURRENT LINE END ...\n");

        current_node = current_node->next_line_pointer;
    }
    printf("Digit : ... end of line list data\n");
}
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : print_square_list      */
/*      Function Args : the start of the linked list containing cube data      */
/*      Function Task : prints the data held in the linked list containing cube data      */
/* ===== */

```

```

void print_square_list(square_node_pointer current_node)
{
/* adapted from [Deitel, et al 1992] */
if (current_node == NULL)
    printf("Error : no cube list data to display\n");
else
{
    printf("Digit : cube list data ...\n");
    while (current_node != NULL)
    {
        printf("    ... NEW CUBE START ...\n");
        printf("    ... vertex a coordinate x = %d\n", current_node->point_a_x);
        printf("    ... vertex a coordinate y = %d\n", current_node->point_a_y);
        printf("    ... vertex a coordinate z = %f\n", current_node->point_a_z);
        printf("    ... vertex b coordinate x = %d\n", current_node->point_b_x);
        printf("    ... vertex b coordinate y = %d\n", current_node->point_b_y);
        printf("    ... vertex b coordinate z = %f\n", current_node->point_b_z);
        printf("    ... vertex c coordinate x = %d\n", current_node->point_c_x);
        printf("    ... vertex c coordinate y = %d\n", current_node->point_c_y);
        printf("    ... vertex c coordinate z = %f\n", current_node->point_c_z);
        printf("    ... vertex d coordinate x = %d\n", current_node->point_d_x);
        printf("    ... vertex d coordinate y = %d\n", current_node->point_d_y);
        printf("    ... vertex d coordinate z = %f\n", current_node->point_d_z);
        printf("    ... CURRENT CUBE END ...\n");

        current_node = current_node->next_square_pointer;
    }
    printf("Digit : ... end of cube list data\n");
}
}

/* ===== */
/*      End of Function */
/* ===== */
/*      Function Name : exit_func */
/*      Function Args : none */
/*      Function Task : executed whenever an "exit()" call is made to ensure clean */
/*                      terminateion of program */
/* ===== */

void exit_func(void)
{
if (left_image_array != NULL); /* free the memory associated with the left image */
    free(left_image_array);

if (right_image_array != NULL); /* free the memory associated with the right image */
    free(right_image_array);

#ifdef DEBUG
    print_line_list(line_list_start);
    print_square_list(square_list_start);
#endif

delete_line_list(&line_list_start); /* free the memory associated with the edge/line linked list */
delete_square_list(&square_list_start); /* free the memory associated with the cube linked list */

#ifdef DEBUG
    print_line_list(line_list_start);
    print_square_list(square_list_start);
#endif

printf("Digit : terminating, good bye\n\n");
}

/* ===== */
/*      End of Function */
/* ===== */
/*      Function Name : load_images */
/*      Function Args : pointer to a character string containing the file names for the left */
/*                      and right image to be loaded */
/*      Function Task : reads the byte data of the images from disk to memory to be displayed */
/*                      in OpenGL */
/* ===== */

int load_images(char *left_filename, char *right_filename)
{
FILE *left_input_file = NULL;

```

```

FILE *right_input_file = NULL;

long left_image_bytes_per_line;
long left_image_actual_bytes_per_line;
short left_image_byte_alignment;
long right_image_bytes_per_line;
long right_image_actual_bytes_per_line;
short right_image_byte_alignment;

unsigned short int left_image_type;
unsigned short int right_image_type;
unsigned short int left_image_planes;
unsigned short int right_image_planes;
unsigned short int left_image_bits_per_pixel;
unsigned short int right_image_bits_per_pixel;
unsigned int left_image_compression;
unsigned int right_image_compression;
int left_image_width;
int right_image_width;
int left_image_height;
int right_image_height;
int left_image_size;
int right_image_size;
int i;

/* because the program, at this stage, is very specific as to the images it will accept - only */
/* bitmaps that are 360x288 pixels - a lot of checking is required to ensure that the image files */
/* conform to these criteria. As a result more code is used than strictly necessary but provides */
/* a "bolt and braces" approach to avoid errors */

left_input_file = fopen(left_filename, "rb"); /* open the left image file for reading */
right_input_file = fopen(right_filename, "rb"); /* open the right image file for reading */

if (!left_input_file) /* check success and terminate on failure */
{
    printf("Error : unable to open left image source\n");
    return 0;
}
else if (!right_input_file)
{
    fclose(left_input_file);
    printf("Error : unable to open right image source\n");
    return 0;
}

/* read in the file identifier from the image file and confirm it is a bitmap image */
fread(&left_image_type, 2, 1, left_input_file);
fread(&right_image_type, 2, 1, right_input_file);

if (left_image_type != 19778)
{
    printf("Error : incorrect source type for left image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}
else if (right_image_type != 19778)
{
    printf("Error : incorrect source type for right image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}

/* more through the bitmap header to where the image width and height is stored */
fseek(left_input_file, 16, SEEK_CUR);
fseek(right_input_file, 16, SEEK_CUR);

/* read in the width and height and confirm the images are of the correct size */
fread(&left_image_width, 4, 1, left_input_file);
fread(&left_image_height, 4, 1, left_input_file);
fread(&right_image_width, 4, 1, right_input_file);
fread(&right_image_height, 4, 1, right_input_file);

if (left_image_width != IMAGE_WIDTH || left_image_height != IMAGE_HEIGHT)
{
    printf("Error : incorrect image size in left image\n");
}

```

```

        fclose(left_input_file);
        fclose(right_input_file);
        return 0;
    }
else if (right_image_width != IMAGE_WIDTH || right_image_height != IMAGE_HEIGHT)
{
    printf("Error : incorrect image size in right image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}

/* calculate the size of the images based on their height, width and rgb values */
left_image_size = left_image_width * left_image_height * 3;
right_image_size = right_image_width * right_image_height * 3;

/* read in the number of planes in the bitmap image and ensure they equal 1 */
fread(&left_image_planes, 2, 1, left_input_file);
fread(&right_image_planes, 2, 1, right_input_file);

if (left_image_planes != 1)
{
    printf("Error : incorrect image format in left image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}
else if (right_image_planes != 1)
{
    printf("Error : incorrect image format in right image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}

/* ensure the images have 24 bits per pixel */
fread(&left_image_bits_per_pixel, 2, 1, left_input_file);
fread(&right_image_bits_per_pixel, 2, 1, right_input_file);

if (left_image_bits_per_pixel != 24)
{
    printf("Error : incorrect image formate in left image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}
else if (right_image_bits_per_pixel != 24)
{
    printf("Error : incorrect image format in right image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}

/* confirm that there is no compression on the images */
fread(&left_image_compression, 4, 1, left_input_file);
fread(&right_image_compression, 4, 1, right_input_file);

if (left_image_compression != 0)
{
    printf("Error : incorrect image formate in left image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}
else if (right_image_compression != 0)
{
    printf("Error : incorrect image format in right image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}

/* move to the file pointer to point to a total of 54 bytes within the file where the rgb byte */
/* data is stored */
fseek(left_input_file, 20, SEEK_CUR);
fseek(right_input_file, 20, SEEK_CUR);

```

```

/* calculate the number of junk bytes, or byte alignment, stored after the rgb data on each scan */
/* line of the image, if this is not done the image will be corrupt */
left_image_bytes_per_line = (left_image_width * left_image_bits_per_pixel) / 8;
right_image_bytes_per_line = (right_image_width * right_image_bits_per_pixel) / 8;

left_image_actual_bytes_per_line = ((left_image_bytes_per_line + 3) / 4) * 4;
right_image_actual_bytes_per_line = ((right_image_bytes_per_line + 3) / 4) * 4;

left_image_byte_alignment = left_image_actual_bytes_per_line - left_image_bytes_per_line;
right_image_byte_alignment = right_image_actual_bytes_per_line - right_image_bytes_per_line;

/* allocate memory for the left and right images and test success */
left_image_array = (GLubyte *) malloc(left_image_width * left_image_height * 3);
right_image_array = (GLubyte *) malloc(right_image_width * right_image_height * 3);

if (left_image_array == NULL)
{
    printf("Error : memory allocation failure in left image\n");
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}
else if (right_image_array == NULL)
{
    printf("Error : memory allocation failure in right image\n");
    free(left_image_array);
    fclose(left_input_file);
    fclose(right_input_file);
    return 0;
}

/* read in the byte data from the left image file */
for (i = 0; i < left_image_height; i++)
{
    fread((left_image_array + i * left_image_width * 3), left_image_width * 3, 1,
left_input_file);
    fseek(left_input_file, left_image_byte_alignment, SEEK_CUR);
}

/* read in the byte data from the right image file */
for (i = 0; i < right_image_height; i++)
{
    fread((right_image_array + i * right_image_width * 3), right_image_width * 3, 1,
right_input_file);
    fseek(right_input_file, right_image_byte_alignment, SEEK_CUR);
}

fclose(left_input_file); /* close both input files */
fclose(right_input_file);

return 1;
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : axis_display_list      */
/*      Function Args : none      */
/*      Function Task : creates a display list for a virtual axis that is displayed in OpenGL */
/* ===== */

void axis_display_list(void)
{
    axis_display_list_index = glGenLists(1);

    glNewList(axis_display_list_index, GL_COMPILE);
        glLineWidth(3.0);

        glColor3f(1.0, 0.0, 0.0);
        glBegin(GL_LINES);
            glVertex3f(0.0, 0.0, 0.0);
            glVertex3f(5.0, 0.0, 0.0);
        glEnd();

        glColor3f(0.0, 1.0, 0.0);
        glBegin(GL_LINES);

```

```

        glVertex3f(0.0, 0.0, 0.0);
        glVertex3f(0.0, 5.0, 0.0);
    glEnd();

    glColor3f(0.0, 0.0, 1.0);
    glBegin(GL_LINES);
        glVertex3f(0.0, 0.0, 0.0);
        glVertex3f(0.0, 0.0, 5.0);
    glEnd();

    glLineWidth(1.0);
glEndList();
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : grid_display_list      */
/*      Function Args : none      */
/*      Function Task : creates a display list for a virtual grid that is displayed in OpenGL */
/* ===== */

void grid_display_list(void)
{
    int i;

    grid_display_list_index = glGenLists(1);

    glNewList(grid_display_list_index, GL_COMPILE);
        glColor3f(0.715, 0.715, 0.715);

        for (i = 0; i <= 50; i++)
        {
            glBegin(GL_LINES);
                glVertex3f(-50.0, 0.0, (i * 2)-50);
                glVertex3f(50.0, 0.0, (i * 2)-50);
            glEnd();
        }

        for (i = 0; i <= 50; i++)
        {
            glBegin(GL_LINES);
                glVertex3f((i * 2)-50, 0.0, -50.0);
                glVertex3f((i * 2)-50, 0.0, 50.0);
            glEnd();
        }
    glEndList();
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : insert_line      */
/*      Function Args : none      */
/*      Function Task : adds a new node to the line/edge linked list      */
/* ===== */

void insert_line(void)
{
    /* adapted from [Deitel, et al 1992] */
    line_node_pointer new_node = NULL;
    line_node_pointer previous_node = NULL;
    line_node_pointer current_node = NULL;

    new_node = malloc(sizeof(line_list_node));

    if (new_node == NULL)
    {
        printf("Error : memory allocation failure in new line\n");
        exit(0);
    }

    new_node->start_left_x = start_left_x_position - left_image_left_position;
    new_node->start_left_y = left_image_bottom_position - start_left_y_position;
    new_node->start_right_x = start_right_x_position - right_image_left_position;

    new_node->start_depth = new_node->start_left_x - new_node->start_right_x;

```



```

new_node->end_left_x = end_left_x_position - left_image_left_position;
new_node->end_left_y = left_image_bottom_position - end_left_y_position;
new_node->end_right_x = end_right_x_position - right_image_left_position;

new_node->end_depth = new_node->end_left_x - new_node->end_right_x;

new_node->next_line_pointer = NULL;
previous_node = NULL;
current_node = line_list_start;

while(current_node != NULL)
{
    previous_node = current_node;
    current_node = current_node->next_line_pointer;
}

if (previous_node == NULL)
{
    new_node->next_line_pointer = line_list_start;
    line_list_start = new_node;
}
else
{
    previous_node->next_line_pointer = new_node;
    new_node->next_line_pointer = current_node;
}

printf("Digit : inserting new line based on following data ...\n");
printf("    ... start vertex left  image x = %d\n", new_node->start_left_x);
printf("    ... start vertex left  image y = %d\n", new_node->start_left_y);
printf("    ... start vertex right image x = %d\n", new_node->start_right_x);
printf("    ... start vertex      depth  = %f\n", new_node->start_depth);
printf("    ... end   vertex left  image x = %d\n", new_node->end_left_x);
printf("    ... end   vertex left  image y = %d\n", new_node->end_left_y);
printf("    ... end   vertex right image x = %d\n", new_node->end_right_x);
printf("    ... end   vertex      depth  = %f\n", new_node->end_depth);
printf("Digit : ... insert complete\n");
}

/* ===== */
/*      End of Function                                */
/* ===== */
/*      Function Name : insert_square                  */
/*      Function Args : none                           */
/*      Function Task : adds a new node to the cube linked list */
/* ===== */

void insert_square(void)
{
    /* adapted from [Deitel, et al 1992] */
    square_node_pointer new_node = NULL;
    square_node_pointer previous_node = NULL;
    square_node_pointer current_node = NULL;

    new_node = malloc(sizeof(square_list_node));

    if (new_node == NULL)
    {
        printf("Error : memory allocation failure in new square\n");
        exit(0);
    }

    new_node->point_a_x = left_image_point_a_x - left_image_left_position;
    new_node->point_a_y = left_image_bottom_position - left_image_point_a_y;
    new_node->point_a_z = new_node->point_a_x - (right_image_point_a_x - right_image_right_position);

    new_node->point_b_x = left_image_point_b_x - left_image_left_position;
    new_node->point_b_y = left_image_bottom_position - left_image_point_b_y;
    new_node->point_b_z = new_node->point_b_x - (right_image_point_b_x - right_image_right_position);

    new_node->point_c_x = left_image_point_c_x - left_image_left_position;
    new_node->point_c_y = left_image_bottom_position - left_image_point_c_y;
    new_node->point_c_z = new_node->point_c_x - (right_image_point_c_x - right_image_right_position);

    new_node->point_d_x = left_image_point_d_x - left_image_left_position;
    new_node->point_d_y = left_image_bottom_position - left_image_point_d_y;

```

```

new_node->point_d_z = new_node->point_d_x - (right_image_point_d_x - right_image_right_position);

new_node->next_square_pointer = NULL;
previous_node = NULL;
current_node = square_list_start;

while(current_node != NULL)
{
    previous_node = current_node;
    current_node = current_node->next_square_pointer;
}

if (previous_node == NULL)
{
    new_node->next_square_pointer = square_list_start;
    square_list_start = new_node;
}
else
{
    previous_node->next_square_pointer = new_node;
    new_node->next_square_pointer = current_node;
}

printf("Digit : inserting new cube based on following data ...\n");
printf("    ... vertex a coordinate x = %d\n", new_node->point_a_x);
printf("    ... vertex a coordinate y = %d\n", new_node->point_a_y);
printf("    ... vertex a coordinate z = %f\n", new_node->point_a_z);
printf("    ... vertex b coordinate x = %d\n", new_node->point_b_x);
printf("    ... vertex b coordinate y = %d\n", new_node->point_b_y);
printf("    ... vertex b coordinate z = %f\n", new_node->point_b_z);
printf("    ... vertex c coordinate x = %d\n", new_node->point_c_x);
printf("    ... vertex c coordinate y = %d\n", new_node->point_c_y);
printf("    ... vertex c coordinate z = %f\n", new_node->point_c_z);
printf("    ... vertex d coordinate x = %d\n", new_node->point_d_x);
printf("    ... vertex d coordinate y = %d\n", new_node->point_d_y);
printf("    ... vertex d coordinate z = %f\n", new_node->point_d_z);
printf("Digit : ... insert complete\n");
}

/* ===== */
/*      End of Function                                */
/* ===== */
/*      Function Name : display_object                  */
/*      Function Args : the start of the two linked lists used in the code                      */
/*      Function Task : completely traverse the two linked lists, interpreting the data and      */
/*                      displaying it in OpenGL                                                */
/* ===== */

void display_object(line_node_pointer current_node, square_node_pointer current_square_node)
{
    int horizontal_adjustment;
    int vertical_adjustment;

    int cube_depth;
    int cube_width;
    int cube_breadth;

    horizontal_adjustment = (left_image_right_position - left_image_left_position) * 0.5;
    vertical_adjustment = (left_image_bottom_position - left_image_top_position) * 0.5;

    glColor3f(0.0, 0.965, 1.0);

    glLineWidth(2.0);

    /* traverse through the complete linked list that contains the line/edge data and display the */
    /* lines on screen. The horizontal and vertical adjustment values are used to move the center of */
    /* the scene to the origin, instead of having the bottom left corner centered at the origin    */
    /* ===== */
    if (current_node != NULL)
    {
        while(current_node != NULL)
        {
            glBegin(GL_LINES);
            glVertex3f((current_node->start_left_x - horizontal_adjustment) * scale,
                      (current_node->start_left_y - vertical_adjustment) * scale,
                      current_node->start_depth * depth_factor * scale);
            glVertex3f((current_node->end_left_x - horizontal_adjustment) * scale,
                      (current_node->end_left_y - vertical_adjustment) * scale,

```

```

        current_node->end_depth * depth_factor * scale);

    glEnd();

    current_node = current_node->next_line_pointer;
}

/* traverse through the complete linked list that contains the cube data and display the cube on */
/* screen. The lines connecting the four vertices that were picked by the user are displayed in */
/* red while the green lines represent those that the code reconstructs to show the complete cube */
/* This code is still under construction and as such is not complete. */
if (current_square_node != NULL)
{
    while(current_square_node != NULL)
    {
        glColor3f(0.580, 1.0, 0.478);

        cube_depth = current_square_node->point_b_y - current_square_node->point_d_y;
        cube_width = current_square_node->point_a_x - current_square_node->point_b_x;
        cube_breadth = current_square_node->point_c_z - current_square_node->point_b_z;

        glBegin(GL_LINES); /* draw line AB */
        glVertex3f((current_square_node->point_a_x - horizontal_adjustment) * scale,
                    (current_square_node->point_a_y - vertical_adjustment) *
                    scale,
                    current_square_node->point_a_z * depth_factor * scale);
        glVertex3f((current_square_node->point_b_x - horizontal_adjustment) * scale,
                    (current_square_node->point_b_y - vertical_adjustment) *
                    scale,
                    current_square_node->point_b_z * depth_factor * scale);
        glEnd();

        glBegin(GL_LINES); /* draw line BC */
        glVertex3f((current_square_node->point_b_x - horizontal_adjustment) * scale,
                    (current_square_node->point_b_y - vertical_adjustment) *
                    scale,
                    current_square_node->point_b_z * depth_factor * scale);
        glVertex3f((current_square_node->point_c_x - horizontal_adjustment) * scale,
                    (current_square_node->point_c_y - vertical_adjustment) *
                    scale,
                    current_square_node->point_c_z * depth_factor * scale);
        glEnd();

        glBegin(GL_LINES); /* draw line BD */
        glVertex3f((current_square_node->point_b_x - horizontal_adjustment) * scale,
                    (current_square_node->point_b_y - vertical_adjustment) *
                    scale,
                    current_square_node->point_b_z * depth_factor * scale);
        glVertex3f((current_square_node->point_d_x - horizontal_adjustment) * scale,
                    (current_square_node->point_d_y - vertical_adjustment) *
                    scale,
                    current_square_node->point_d_z * depth_factor * scale);
        glEnd();

        glColor3f(1.0, 0.318, 0.318);

        /* code beyond this point has not been completed fully, it has been included */
        /* because when executed it is possible to see the potential of this method */
        glBegin(GL_LINES); /* draw line DE */
        glVertex3f((current_square_node->point_d_x - horizontal_adjustment) * scale,
                    (current_square_node->point_d_y - vertical_adjustment) *
                    scale,
                    current_square_node->point_d_z * depth_factor * scale);
        glVertex3f((current_square_node->point_a_x - horizontal_adjustment) * scale,
                    ((current_square_node->point_a_y - cube_depth) -
                    vertical_adjustment) * scale,
                    current_square_node->point_a_z * depth_factor * scale);
        glEnd();

        glBegin(GL_LINES); /* draw line AE */
        glVertex3f((current_square_node->point_a_x - horizontal_adjustment) * scale,
                    (current_square_node->point_a_y - vertical_adjustment) *
                    scale,
                    current_square_node->point_a_z * depth_factor * scale);
        glVertex3f((current_square_node->point_a_x - horizontal_adjustment) * scale,
                    ((current_square_node->point_a_y - cube_depth) -
                    vertical_adjustment) * scale,

```

```

        current_square_node->point_a_z * depth_factor * scale);
glEnd();

glBegin(GL_LINES); /* draw line DG */
    glVertex3f((current_square_node->point_d_x - horizontal_adjustment) * scale,
               (current_square_node->point_d_y - vertical_adjustment) *
               scale,
               current_square_node->point_d_z * depth_factor * scale);
    glVertex3f((current_square_node->point_c_x - horizontal_adjustment) * scale,
               ((current_square_node->point_c_y - cube_depth) -
               vertical_adjustment) * scale,
               current_square_node->point_c_z * depth_factor * scale);
glEnd();

glBegin(GL_LINES); /* draw line CG */
    glVertex3f((current_square_node->point_c_x - horizontal_adjustment) * scale,
               (current_square_node->point_c_y - vertical_adjustment) *
               scale,
               current_square_node->point_c_z * depth_factor * scale);
    glVertex3f((current_square_node->point_c_x - horizontal_adjustment) * scale,
               ((current_square_node->point_c_y - cube_depth) -
               vertical_adjustment) * scale,
               current_square_node->point_c_z * depth_factor * scale);
glEnd();

glBegin(GL_LINES); /* draw line AF */
glEnd();

glBegin(GL_LINES); /* draw line CF */
glEnd();

glBegin(GL_LINES); /* draw line FH */
glEnd();

glBegin(GL_LINES); /* draw line EH */
glEnd();

glBegin(GL_LINES); /* draw line GH */
glEnd();

current_square_node = current_square_node->next_square_pointer;
}

glLineWidth(1.0);
}

/* ===== */
/*      End of Function */
/* ===== */
/*      Function Name : display_func */
/*      Function Args : none */
/*      Function Task : this is the registered display function for OpenGL and writes to the */
/*                      display buffer */
/* ===== */

void display_func(void)
{
    glEnable(GL_DEPTH_TEST);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, 12.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    glPushMatrix();

    /* the following calls to "glRotate" "glTranslate" and "glScale" move the scene relative to the */
    /* virtual camera, giving the appearance that the camera has or is moving */
    glRotatef(rotate_x, 1.0, 0.0, 0.0 );
    glRotatef(rotate_y, 0.0, 1.0, 0.0 );

    glTranslatef(translate_x, translate_y, translate_z);

    glScalef(zoom, zoom, zoom);

```

```

glCallList(axis_display_list_index); /* draw the axis using the display list */
glCallList(grid_display_list_index); /* draw the grid using the display list */

display_object(line_list_start, square_list_start);

glPopMatrix();

glDisable(GL_DEPTH_TEST);

glRasterPos2f(-5.5,-5.5); /* display the left image on screen */
glDrawPixels(IMAGE_WIDTH, IMAGE_HEIGHT, GL_BGR, GL_UNSIGNED_BYTE, left_image_array);
glRasterPos2f(0.25,-5.5); /* display the right image on screen */
glDrawPixels(IMAGE_WIDTH, IMAGE_HEIGHT, GL_BGR, GL_UNSIGNED_BYTE, right_image_array);

/* the execution of the program is determined by the current state, and the user is prompted
/* accordingly. If the state is not "add line", "add cube" or "calibrate" then this section is
/* bypassed, otherwise the execution steps through this "if" statement on each redisplay

/* if the images are being calibrated ...
if (calibration == FALSE)
{
    glutSwapBuffers();

    /* determine the current stage in the calibration process and prompt the user for the
    /* point to be calibrated
    if (left_image_bottom_position == FALSE && left_image_top_position == FALSE &&
        left_image_left_position == FALSE && left_image_right_position == FALSE &&
        right_image_bottom_position == FALSE && right_image_top_position == FALSE &&
        right_image_left_position == FALSE && right_image_right_position == FALSE)
    {
        printf("Digit : calibrate camera ...\n");
        printf("Digit : click on bottom left pixel of left image ...\n");
    }
    else if (left_image_bottom_position != FALSE && left_image_top_position == FALSE &&
        left_image_left_position != FALSE && left_image_right_position == FALSE &&
        right_image_bottom_position == FALSE && right_image_top_position == FALSE &&
        right_image_left_position == FALSE && right_image_right_position == FALSE)
        printf("Digit : click on top right pixel of left image ...\n");
    else if (left_image_bottom_position != FALSE && left_image_top_position != FALSE &&
        left_image_left_position != FALSE && left_image_right_position != FALSE &&
        right_image_bottom_position == FALSE && right_image_top_position == FALSE &&
        right_image_left_position == FALSE && right_image_right_position == FALSE)
        printf("Digit : click on bottom left pixel of right image ...\n");
    else if (left_image_bottom_position != FALSE && left_image_top_position != FALSE &&
        left_image_left_position != FALSE && left_image_right_position != FALSE &&
        right_image_bottom_position != FALSE && right_image_top_position == FALSE &&
        right_image_left_position != FALSE && right_image_right_position == FALSE)
        printf("Digit : click on top right pixel of right image ...\n");
}

/* if a new line/edge is being added ...
else if (add_line == TRUE)
{
    glutSwapBuffers();

    /* determine the current stage in the addition of a new line/edge and prompt the user for
    /* the next vertex in either the left or right image
    if (start_left_x_position == FALSE && start_left_y_position == FALSE &&
        start_right_x_position == FALSE && end_left_x_position == FALSE &&
        end_left_y_position == FALSE && end_right_x_position == FALSE)
    {
        printf("Digit : add line ...\n");
        printf("Digit : click on start point of first vertex in left image ...\n");
    }
    else if (start_left_x_position != FALSE && start_left_y_position != FALSE &&
        start_right_x_position == FALSE && end_left_x_position == FALSE &&
        end_left_y_position == FALSE && end_right_x_position == FALSE)
        printf("Digit : click on end point of first vertex in right image ...\n");
    else if (start_left_x_position != FALSE && start_left_y_position != FALSE &&
        start_right_x_position != FALSE && end_left_x_position == FALSE &&
        end_left_y_position == FALSE && end_right_x_position == FALSE)
        printf("Digit : click on start point of second vertex in left image ...\n");
    else if (start_left_x_position != FALSE && start_left_y_position != FALSE &&
        start_right_x_position != FALSE && end_left_x_position != FALSE &&
        end_left_y_position != FALSE && end_right_x_position == FALSE)
        printf("Digit : click on end point of second vertex in right image ...\n");
}

```

```

/* if a new cube is being added ... */
else if (add_cube == TRUE)
{
    glutSwapBuffers();

    /* determine the current stage in the addition of a new cube and prompt the user for the */
    /* next vertex in either the right or left image */
    if (left_image_point_a_x == FALSE && left_image_point_a_y == FALSE &&
        left_image_point_b_x == FALSE && left_image_point_b_y == FALSE &&
        left_image_point_c_x == FALSE && left_image_point_c_y == FALSE &&
        left_image_point_d_x == FALSE && left_image_point_d_y == FALSE &&
        right_image_point_a_x == FALSE && right_image_point_b_x == FALSE &&
        right_image_point_c_x == FALSE && right_image_point_d_x == FALSE)
    {
        printf("Digit : add cube ...\n");
        printf("Digit : click on point A in left image ...\n");
    }
    else if (left_image_point_a_x != FALSE && left_image_point_a_y != FALSE &&
        left_image_point_b_x == FALSE && left_image_point_b_y == FALSE &&
        left_image_point_c_x == FALSE && left_image_point_c_y == FALSE &&
        left_image_point_d_x == FALSE && left_image_point_d_y == FALSE &&
        right_image_point_a_x == FALSE && right_image_point_b_x == FALSE &&
        right_image_point_c_x == FALSE && right_image_point_d_x == FALSE)
        printf("Digit : click on point A in right image ...\n");
    else if (left_image_point_a_x != FALSE && left_image_point_a_y != FALSE &&
        left_image_point_b_x == FALSE && left_image_point_b_y == FALSE &&
        left_image_point_c_x == FALSE && left_image_point_c_y == FALSE &&
        left_image_point_d_x == FALSE && left_image_point_d_y == FALSE &&
        right_image_point_a_x != FALSE && right_image_point_b_x == FALSE &&
        right_image_point_c_x == FALSE && right_image_point_d_x == FALSE)
        printf("Digit : click on point B in left image ...\n");
    else if (left_image_point_a_x != FALSE && left_image_point_a_y != FALSE &&
        left_image_point_b_x != FALSE && left_image_point_b_y != FALSE &&
        left_image_point_c_x == FALSE && left_image_point_c_y == FALSE &&
        left_image_point_d_x == FALSE && left_image_point_d_y == FALSE &&
        right_image_point_a_x != FALSE && right_image_point_b_x == FALSE &&
        right_image_point_c_x == FALSE && right_image_point_d_x == FALSE)
        printf("Digit : click on point B in right image ...\n");
    else if (left_image_point_a_x != FALSE && left_image_point_a_y != FALSE &&
        left_image_point_b_x != FALSE && left_image_point_b_y != FALSE &&
        left_image_point_c_x == FALSE && left_image_point_c_y == FALSE &&
        left_image_point_d_x == FALSE && left_image_point_d_y == FALSE &&
        right_image_point_a_x != FALSE && right_image_point_b_x != FALSE &&
        right_image_point_c_x == FALSE && right_image_point_d_x == FALSE)
        printf("Digit : click on point C in left image ...\n");
    else if (left_image_point_a_x != FALSE && left_image_point_a_y != FALSE &&
        left_image_point_b_x != FALSE && left_image_point_b_y != FALSE &&
        left_image_point_c_x != FALSE && left_image_point_c_y != FALSE &&
        left_image_point_d_x == FALSE && left_image_point_d_y == FALSE &&
        right_image_point_a_x != FALSE && right_image_point_b_x != FALSE &&
        right_image_point_c_x == FALSE && right_image_point_d_x == FALSE)
        printf("Digit : click on point C in right image ...\n");
    else if (left_image_point_a_x != FALSE && left_image_point_a_y != FALSE &&
        left_image_point_b_x != FALSE && left_image_point_b_y != FALSE &&
        left_image_point_c_x != FALSE && left_image_point_c_y != FALSE &&
        left_image_point_d_x != FALSE && left_image_point_d_y != FALSE &&
        right_image_point_a_x != FALSE && right_image_point_b_x != FALSE &&
        right_image_point_c_x != FALSE && right_image_point_d_x == FALSE)
        printf("Digit : click on point D in left image ...\n");
    else if (left_image_point_a_x != FALSE && left_image_point_a_y != FALSE &&
        left_image_point_b_x != FALSE && left_image_point_b_y != FALSE &&
        left_image_point_c_x != FALSE && left_image_point_c_y != FALSE &&
        left_image_point_d_x != FALSE && left_image_point_d_y != FALSE &&
        right_image_point_a_x != FALSE && right_image_point_b_x != FALSE &&
        right_image_point_c_x != FALSE && right_image_point_d_x == FALSE)
        printf("Digit : click on point D in right image ...\n");
}

/* if no new line/edge or cube addition, or image calibration is taking place, redisplay */
else
{
    glutSwapBuffers();
    glutPostRedisplay();
}

/* ===== */
/* End of Function */

```

```

/* ===== */
/*      Function Name : mouse_func                                */
/*      Function Args : button state (up or down) and the screen coordinates when pressed */
/*      Function Task : this is the registered mouse button callback function in OpenGL and */
/*                      reacts to a change in state of the mouse buttons                    */
/* ===== */

void mouse_func(int button, int down, int x, int y)
{
/* the program only requires the position of a mouse click when calibrating the images, adding */
/* the vertex of a line/edge, or when adding the vertex of a new cube. This "if" statement */
/* controls the collection of mouse coordinates depending on the current state of execution */

/* calibration of bottom left corner of left image */
if (down == GLUT_UP && left_image_bottom_position == FALSE &&
    left_image_top_position == FALSE && left_image_left_position == FALSE &&
    left_image_right_position == FALSE && right_image_bottom_position == FALSE &&
    right_image_top_position == FALSE && right_image_left_position == FALSE &&
    right_image_right_position == FALSE)
{
    printf("        ... mouse position x = %d y = %d\n", x, y);
    left_image_bottom_position = y;
    left_image_left_position = x;
    glutSwapBuffers();
    glutPostRedisplay();
}

/* calibration of top right corner of left image */
else if (down == GLUT_UP && left_image_bottom_position != FALSE &&
    left_image_top_position == FALSE && left_image_left_position != FALSE &&
    left_image_right_position == FALSE && right_image_bottom_position == FALSE &&
    right_image_top_position == FALSE && right_image_left_position == FALSE &&
    right_image_right_position == FALSE)
{
    printf("        ... mouse position x = %d y = %d\n", x, y);
    left_image_top_position = y;
    left_image_right_position = x;
    glutSwapBuffers();
    glutPostRedisplay();
}

/* calibration of bottom left corner of right image */
else if (down == GLUT_UP && left_image_bottom_position != FALSE &&
    left_image_top_position != FALSE && left_image_left_position != FALSE &&
    left_image_right_position != FALSE && right_image_bottom_position == FALSE &&
    right_image_top_position == FALSE && right_image_left_position == FALSE &&
    right_image_right_position == FALSE)
{
    printf("        ... mouse position x = %d y = %d\n", x, y);
    right_image_bottom_position = y;
    right_image_left_position = x;
    glutSwapBuffers();
    glutPostRedisplay();
}

/* calibration of top right corner of right image */
else if (down == GLUT_UP && left_image_bottom_position != FALSE &&
    left_image_top_position != FALSE && left_image_left_position != FALSE &&
    left_image_right_position != FALSE && right_image_bottom_position != FALSE &&
    right_image_top_position == FALSE && right_image_left_position != FALSE &&
    right_image_right_position == FALSE)
{
    printf("        ... mouse position x = %d y = %d\n", x, y);
    right_image_top_position = y;
    right_image_right_position = x;
    glutSwapBuffers();
    glutPostRedisplay();
    printf("Digit : ... calibration complete\n");
    calibration = TRUE; /* calibration of images is complete so change state */
}

/* add the start point in the left image of a new line/edge */
else if (down == GLUT_UP && add_line == TRUE && start_left_x_position == FALSE &&
    start_left_y_position == FALSE && start_right_x_position == FALSE &&
    end_left_x_position == FALSE && end_left_y_position == FALSE &&
    end_right_x_position == FALSE)
{
    printf("        ... mouse position x = %d y = %d\n", x, y);
    start_left_x_position = x;
    start_left_y_position = y;
    glutSwapBuffers();
}

```

```

        glutPostRedisplay();
    }
    /* add the start point in the right image of a new line/edge */
    else if (down == GLUT_UP && add_line == TRUE && start_left_x_position != FALSE &&
        start_left_y_position != FALSE && start_right_x_position == FALSE &&
        end_left_x_position == FALSE && end_left_y_position == FALSE &&
        end_right_x_position == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        start_right_x_position = x;
        glutSwapBuffers();
        glutPostRedisplay();
    }
    /* add the end point in the left image of a new line/edge */
    else if (down == GLUT_UP && add_line == TRUE && start_left_x_position != FALSE &&
        start_left_y_position != FALSE && start_right_x_position != FALSE &&
        end_left_x_position == FALSE && end_left_y_position == FALSE &&
        end_right_x_position == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        end_left_x_position = x;
        end_left_y_position = y;
        glutSwapBuffers();
        glutPostRedisplay();
    }
    /* add the end point in the right image of a new line/edge */
    else if (down == GLUT_UP && add_line == TRUE && start_left_x_position != FALSE &&
        start_left_y_position != FALSE && start_right_x_position != FALSE &&
        end_left_x_position != FALSE && end_left_y_position != FALSE &&
        end_right_x_position == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        end_right_x_position = x;

        /* all points are collected, ensure they are within the calibration tolerances */
        if (start_left_x_position >= left_image_left_position &&
            start_left_x_position <= left_image_right_position &&
            start_left_y_position >= left_image_top_position &&
            start_left_y_position <= left_image_bottom_position &&
            start_right_x_position >= right_image_left_position &&
            start_right_x_position <= right_image_right_position &&
            end_left_x_position >= left_image_left_position &&
            end_left_x_position <= left_image_right_position &&
            end_left_y_position >= left_image_top_position &&
            end_left_y_position <= left_image_bottom_position &&
            end_right_x_position >= right_image_left_position &&
            end_right_x_position <= right_image_right_position)

            insert_line(); /* insert the line/edge data into the linked list for display */
        else
            printf("Error : measurements fell outside image calibration\n");

        start_left_x_position = FALSE; /* reset the temporary values that held the data */
        start_left_y_position = FALSE;
        start_right_x_position = FALSE;
        end_left_x_position = FALSE;
        end_left_y_position = FALSE;
        end_right_x_position = FALSE;

        add_line = FALSE; /* all data collected so change execution state */
        glutSwapBuffers();
        glutPostRedisplay();
    }
    /* add point A in the left image of a new cube */
    else if (down == GLUT_UP && add_cube == TRUE && left_image_point_a_x == FALSE &&
        left_image_point_a_y == FALSE && left_image_point_b_x == FALSE &&
        left_image_point_b_y == FALSE && left_image_point_c_x == FALSE &&
        left_image_point_c_y == FALSE && left_image_point_d_x == FALSE &&
        left_image_point_d_y == FALSE && right_image_point_a_x == FALSE &&
        right_image_point_b_x == FALSE && right_image_point_c_x == FALSE &&
        right_image_point_d_x == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        left_image_point_a_x = x;
        left_image_point_a_y = y;
        glutSwapBuffers();
        glutPostRedisplay();
    }

```



```

    }
    /* add point A in the right image of a new cube */
    else if (down == GLUT_UP && add_cube == TRUE && left_image_point_a_x != FALSE &&
        left_image_point_a_y != FALSE && left_image_point_b_x == FALSE &&
        left_image_point_b_y == FALSE && left_image_point_c_x == FALSE &&
        left_image_point_c_y == FALSE && left_image_point_d_x == FALSE &&
        left_image_point_d_y == FALSE && right_image_point_a_x == FALSE &&
        right_image_point_b_x == FALSE && right_image_point_c_x == FALSE &&
        right_image_point_d_x == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        right_image_point_a_x = x;
        glutSwapBuffers();
        glutPostRedisplay();
    }
    /* add point B in the left image of a new cube */
    else if (down == GLUT_UP && add_cube == TRUE && left_image_point_a_x != FALSE &&
        left_image_point_a_y != FALSE && left_image_point_b_x == FALSE &&
        left_image_point_b_y == FALSE && left_image_point_c_x == FALSE &&
        left_image_point_c_y == FALSE && left_image_point_d_x == FALSE &&
        left_image_point_d_y == FALSE && right_image_point_a_x != FALSE &&
        right_image_point_b_x == FALSE && right_image_point_c_x == FALSE &&
        right_image_point_d_x == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        left_image_point_b_x = x;
        left_image_point_b_y = y;
        glutSwapBuffers();
        glutPostRedisplay();
    }
    /* add point B in the right image of a new cube */
    else if (down == GLUT_UP && add_cube == TRUE && left_image_point_a_x != FALSE &&
        left_image_point_a_y != FALSE && left_image_point_b_x != FALSE &&
        left_image_point_b_y != FALSE && left_image_point_c_x == FALSE &&
        left_image_point_c_y == FALSE && left_image_point_d_x == FALSE &&
        left_image_point_d_y == FALSE && right_image_point_a_x != FALSE &&
        right_image_point_b_x == FALSE && right_image_point_c_x == FALSE &&
        right_image_point_d_x == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        right_image_point_b_x = x;
        glutSwapBuffers();
        glutPostRedisplay();
    }
    /* add point C in the left image of a new cube */
    else if (down == GLUT_UP && add_cube == TRUE && left_image_point_a_x != FALSE &&
        left_image_point_a_y != FALSE && left_image_point_b_x != FALSE &&
        left_image_point_b_y != FALSE && left_image_point_c_x == FALSE &&
        left_image_point_c_y == FALSE && left_image_point_d_x == FALSE &&
        left_image_point_d_y == FALSE && right_image_point_a_x != FALSE &&
        right_image_point_b_x != FALSE && right_image_point_c_x == FALSE &&
        right_image_point_d_x == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        left_image_point_c_x = x;
        left_image_point_c_y = y;
        glutSwapBuffers();
        glutPostRedisplay();
    }
    /* add point C in the right image of a new cube */
    else if (down == GLUT_UP && add_cube == TRUE && left_image_point_a_x != FALSE &&
        left_image_point_a_y != FALSE && left_image_point_b_x != FALSE &&
        left_image_point_b_y != FALSE && left_image_point_c_x != FALSE &&
        left_image_point_c_y != FALSE && left_image_point_d_x == FALSE &&
        left_image_point_d_y == FALSE && right_image_point_a_x != FALSE &&
        right_image_point_b_x != FALSE && right_image_point_c_x == FALSE &&
        right_image_point_d_x == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        right_image_point_c_x = x;
        glutSwapBuffers();
        glutPostRedisplay();
    }
    /* add point D in the left image of a new cube */
    else if (down == GLUT_UP && add_cube == TRUE && left_image_point_a_x != FALSE &&
        left_image_point_a_y != FALSE && left_image_point_b_x != FALSE &&
        left_image_point_b_y != FALSE && left_image_point_c_x != FALSE &&

```

```

        left_image_point_c_y != FALSE && left_image_point_d_x == FALSE &&
        left_image_point_d_y == FALSE && right_image_point_a_x != FALSE &&
        right_image_point_b_x != FALSE && right_image_point_c_x != FALSE &&
        right_image_point_d_x == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        left_image_point_d_x = x;
        left_image_point_d_y = y;
        glutSwapBuffers();
        glutPostRedisplay();
    }
/* add point D in the right image of a new cube */
else if (down == GLUT_UP && add_cube == TRUE && left_image_point_a_x != FALSE &&
        left_image_point_a_y != FALSE && left_image_point_b_x != FALSE &&
        left_image_point_b_y != FALSE && left_image_point_c_x != FALSE &&
        left_image_point_c_y != FALSE && left_image_point_d_x != FALSE &&
        left_image_point_d_y != FALSE && right_image_point_a_x != FALSE &&
        right_image_point_b_x != FALSE && right_image_point_c_x != FALSE &&
        right_image_point_d_x == FALSE)
    {
        printf("        ... mouse position x = %d y = %d\n", x, y);
        right_image_point_d_x = x;

/* all points are collected, ensure they are within the calibration tolerances */
if (left_image_point_a_x >= left_image_left_position &&
    left_image_point_a_x <= left_image_right_position &&
    right_image_point_a_x >= right_image_left_position &&
    right_image_point_a_x <= right_image_right_position &&
    left_image_point_b_x >= left_image_left_position &&
    left_image_point_b_x <= left_image_right_position &&
    right_image_point_b_x >= right_image_left_position &&
    right_image_point_b_x <= right_image_right_position &&
    left_image_point_c_x >= left_image_left_position &&
    left_image_point_c_x <= left_image_right_position &&
    right_image_point_c_x >= right_image_left_position &&
    right_image_point_c_x <= right_image_right_position &&
    left_image_point_d_x >= left_image_left_position &&
    left_image_point_d_x <= left_image_right_position &&
    right_image_point_d_x >= right_image_left_position &&
    right_image_point_d_x <= right_image_right_position &&
    left_image_point_a_y >= left_image_top_position &&
    left_image_point_a_y <= left_image_bottom_position &&
    left_image_point_b_y >= left_image_top_position &&
    left_image_point_b_y <= left_image_bottom_position &&
    left_image_point_c_y >= left_image_top_position &&
    left_image_point_c_y <= left_image_bottom_position &&
    left_image_point_d_y >= left_image_top_position &&
    left_image_point_d_y <= left_image_bottom_position)

        insert_square(); /* insert the data into the linked list for displaye */
    else
        printf("Error : measurements fell outside image calibration\n");

    left_image_point_a_x = FALSE; /* reset the temporary variables that held the data */
    left_image_point_a_y = FALSE;
    left_image_point_b_x = FALSE;
    left_image_point_b_y = FALSE;
    left_image_point_c_x = FALSE;
    left_image_point_c_y = FALSE;
    left_image_point_d_x = FALSE;
    left_image_point_d_y = FALSE;
    right_image_point_a_x = FALSE;
    right_image_point_b_x = FALSE;
    right_image_point_c_x = FALSE;
    right_image_point_d_x = FALSE;

    add_cube = FALSE; /* all data is collected so change the execution state */
    glutSwapBuffers();
    glutPostRedisplay();
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : keyboard_func */
/*      Function Args : key pressed and the screen coordinates of the mouse when pressed */

```

```

/*      Function Task : this is the registered keyboard callback function in OpenGL and reacts */
/*      to a key press on the keyboard */
/*      ===== */

void keyboard_func(unsigned char ch, int x, int y)
{
/* these keys are used to control the execution of the program through user interaction */
switch(ch)
{
    case 27 : exit(0); /* terminate the program */
        break;

    case 'a' : if (calibration == TRUE)
        rotate_x = rotate_x + 1 ; /* rotate the camera around x */
        break;

    case 'b' : if (calibration == TRUE)
        rotate_y = rotate_y - 1; /* rotate the camera around y */
        break;

    case 'c' : if (calibration == TRUE)
        add_cube = TRUE; /* add a cube geometry set */
        break;

    case 'd' : depth_factor = depth_factor + 0.1; /* increase the depth factor */
        printf("Digit : new depth factor %f\n", depth_factor);
        break;

    case 'f' : depth_factor = depth_factor - 0.1; /* decrease the depth factor */
        printf("Digit : new depth factor %f\n", depth_factor);
        break;

    case 'h' : print_help(); /* display the help dialog on the shell */
        break;

    case 'l' : if (calibration == TRUE)
        add_line = TRUE; /* add a line/edge geometry set */
        break;

    /* print the current camera orientation */
    case 'o' : printf("Digit : current camera values ...\n");
        printf("    ... camera x position = %f\n", translate_x);
        printf("    ... camera y position = %f\n", translate_y);
        printf("    ... camera z position = %f\n", translate_z);
        printf("    ... camera x rotation = %f\n", rotate_x);
        printf("    ... camera y rotation = %f\n", rotate_y);
        printf("    ... camera zoom      = %f\n", zoom);
        printf("Digit : ... end of camera values\n");
        break;

    case 'q' : exit(0); /* terminate the program */
        break;

    case 'r' : calibration = FALSE; /* recalibrate the two images */
        left_image_bottom_position = FALSE;
        left_image_top_position = FALSE;
        left_image_left_position = FALSE;
        left_image_right_position = FALSE;
        right_image_bottom_position = FALSE;
        right_image_top_position = FALSE;
        right_image_left_position = FALSE;
        right_image_right_position = FALSE;
        break;

    case 'v' : if (calibration == TRUE)
        rotate_y = rotate_y + 1; /* rotate the camera around y */
        break;

    case 'z' : if (calibration == TRUE)
        rotate_x = rotate_x - 1; /* rotate the camera around x */
        break;

    case '=' : if (calibration == TRUE)
        zoom = zoom + 0.01; /* zoom into the scene */
        break;
}
}

```

```

    case '-' : if (calibration == TRUE)
        zoom = zoom - 0.01; /* zoom out of the scene */
        break;

    case '.' : scale = scale + 0.1; /* scale the geometry up */
        break;

    case ',' : scale = scale - 0.1; /* scale the geometry down */
        break;
    }
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : special_func      */
/*      Function Args : key pressed and the screen coordinates of the mouse when pressed */
/*      Function Task : this is the registered special callback function in OpenGL and reacts */
/*                     to a special key press on the keyboard (such as function keys etc) */
/* ===== */

void special_func(int key, int x, int y)
{
    /* these special keys are used to pan the virtual camera through the scene */

    if (calibration == TRUE)
    {
        switch(key)
        {
            case GLUT_KEY_LEFT : translate_x = translate_x + 0.1; /* pan left */
                break;

            case GLUT_KEY_RIGHT : translate_x = translate_x - 0.1; /* pan right */
                break;

            case GLUT_KEY_UP : translate_y = translate_y - 0.1; /* pan up */
                break;

            case GLUT_KEY_DOWN : translate_y = translate_y + 0.1; /* pan down */
                break;

            case GLUT_KEY_PAGE_UP : translate_z = translate_z + 0.1; /* zoom in */
                break;

            case GLUT_KEY_PAGE_DOWN : translate_z = translate_z - 0.1; /* zoom out */
                break;
        }
    }
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : reshape_func      */
/*      Function Args : none */
/*      Function Task : this is the registered reshape callback function in OpenGL and reacts */
/*                     to a change in size of the OpenGL window as a result of user inter- */
/*                     action */
/* ===== */

void reshape_func()
{
    /* to avoid unnecessary confusion with calls to glRasterPos window resizing has been "disabled" */
    /* for this program, while it is not a specific OpenGL call that disables the resize, when the */
    /* attempts to resize the window, it is simply reset to the default values */

    glutReshapeWindow(770, 800);
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      Function Name : initialise_func      */
/*      Function Args : none */
/*      Function Task : this function initialises the OpenGL display environment and prepares */
/*                     the program for writing to the display buffer */
/* ===== */

```

```

/* ===== */

void initialise_func(void)
{
    glShadeModel(GL_SMOOTH);
    glClearColor(0.2f, 0.2f, 0.2f, 1.0f);

    axis_display_list(); /* create the display lists */
    grid_display_list();

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_NORMALIZE);

    glMatrixMode(GL_PROJECTION); /* set the projection matrix */
    glLoadIdentity();
    gluPerspective(50, 1, 0.1, 400);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/* ===== */
/*      End of Function */
/* ===== */
/*      Function Name : main */
/*      Function Args : the number of and an array of the command line arguments */
/*      Function Task : main function of the program than registers all callback functions and */
/*                      places the program in the "main loop" */
/* ===== */

int main(int argc, char *argv[])
{
    char *left_image_filename;
    char *right_image_filename;

    int flag_counter;

    /* upon program start the command line arguments are parsed to ensure the correct number have */
    /* been received, and program execution continues accordingly */

    if (argc == 1)
    {
        /* incorrect command line arguments, terminate execution */
        printf("Usage : digit <left_image_file> <right_image_file>\n");
        printf("      : controls include - \n");
        print_help();
        return 0;
    }
    if (argc == 2)
    {
        /* incorrect command line arguments, terminate execution */
        printf("Error : only one image source specified [%s]\n", argv[1]);
        printf("Usage : digit <left_image_file> <right_image_file>\n");
        printf("      : controls include - \n");
        print_help();
        return 0;
    }
    if (argc == 3)
    {
        /* correct command line arguments, continue program execution */
        printf("Digit : starting, hello\n");
        left_image_filename = argv[1];
        right_image_filename = argv[2];
        printf("Digit : left image source = %s\n", left_image_filename);
        printf("Digit : right image source = %s\n", right_image_filename);
    }
    else
    {
        /* incorrect command line arguments, terminate execution */
        printf("Error : invalid flags ");

        for (flag_counter = 3; flag_counter < argc; flag_counter++)
            printf("[%s] ", argv[flag_counter]);

        printf("\n");
        printf("Usage : digit <left_image_file> <right_image_file>\n");
        printf("      : controls include - \n");
    }
}

```

```

    print_help();
    return 0;
}

atexit(exit_func); /* registration of a function to be called upon execution if exit() */

/* here we load the bitmap images specified by the user in the command line arguments */

if (load_images(left_image_filename, right_image_filename) == 0)
{
    /* images loaded unsuccessfully, so terminate execution */
    printf("Error : unable to load images\n");
    return 0;
}
else
    /* images loaded OK, continue with execution */
    printf("Digit : images loaded successfully\n");

glutInit(&argc, argv);

/* create the OpenGL window and set-up the display mode */

glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB | GLUT_DOUBLE);
glutInitWindowSize(770, 800);
glutInitWindowPosition(255, 0);
glutCreateWindow("Mark Streatfield | Digit");

/* OpenGL callback function registration for the display function, mouse button function,
/* keyboard function, special keyboard function and window reshape function */

glutDisplayFunc(display_func);
glutMouseFunc(mouse_func);
glutKeyboardFunc(keyboard_func);
glutSpecialFunc(special_func);
glutReshapeFunc(reshape_func);

initialise_func();

glutMainLoop();

return 0;
}

/* ===== */
/*      End of Function      */
/* ===== */
/*      End of File      */
/* ===== */

```

