INTERACTIVE VISUALISATION USING 3D GRAPHICS

AN ARCHAEOLOGICAL CASE STUDY

Submitted to the University of Cape Town in partial fulfilment of the requirements for the Degree of Master of Science in Engineering

by
Ross Rozendaal

Department of Geomatics
February 2000

Declaration

I hereby declare that this thesis is my original work and has not been submitted in any form to another university.

Ross Rozendaal

Abstract

The methods of displaying data from archaeological surveys are of considerable importance in representing realistic impressions of archaeological sites that few people are able to visit. In many cases, further study of a site is not possible at the location of the site. This would require that the surveyed data of the site be displayed in such a way as to be accurate and realistic as well as including interactive tools, enabling further studies.

Traditional displays of archaeological data have been either in textual form or in the conventional hardcopy form of maps and drawings. With the advent of computers and computer graphics alternative methods of displaying the data have become possible. 3D graphics have become an important method of displaying archaeological data.

In 1995 and 1996 the Department of Geomatics at the University of Cape Town participated in the survey of the 3.6 million year old hominid footprints in Tanzania. The survey was required for the documentation and study of the footprints. In order to facilitate this in 3D graphics, software packages that allowed user interactive tools to be included in the display had to be investigated. Methods of displaying the data also had to be investigated.

Java3D was selected to create the 3D models and user interactive tools that included measurement tools, gradient tools and profile tools. These tools were created for the Laetoli footprints but were applicable in other archaeological displays as well.

Acknowledgements

Firstly, I would like to thank the Lord Jesus Christ for His love, support and guidance during these last two years.

To my supervisor, Prof. Dr. Heinz Rüther, thank you for your supervision and help in this thesis as well as giving me the opportunity to do this project. To my co-supervisor, Ulrike Bruessler, thank you for your help and advice. Thanks also to Andrea, Sue, Nick, Sydney and Mike for your help.

To my Dad and Mom, Adeline and Brandt, thank you for your support and encouragement during my studies. To my family and friends, thank you for your support.

To the members of room 501, past and current (Simon Taylor, Simon Hull, Terry Richards, Justin Davey and Ulrike Bruessler), thanks for the timely distractions (stress relief) and help that you have provided.

Table of Contents

List of Figures

List of Tables

Glossary of Terms

API	An Application Program Interface (API) is a set of routines, protocols, and tools for building software applications.
Class	A category of objects in object orientated programming. It defines all the common properties of the objects to which it belongs.
Computer platform	The underlying hardware or software for a computer system.
Node	In tree-structures, a point where two or more lines meet.
Object	In object-oriented programming, a self-contained entity that consists of both data and procedures to manipulate the data.
Personal Computer	A computer based on an Intel microprocessor, or on an Intel-compatible microprocessor. High-end personal computers are equivalent to low-end workstations.
User Interface	The interface between a user and a computer program.
Workstation	A type of computer used for engineering applications (CAD/CAM), desktop publishing, software development, and other types of applications that require a moderate amount of computing power and relatively high quality graphics capabilities.

Chapter 1
Introduction

Computer graphics techniques are being used increasingly frequently to visualise complex data in archaeological investigations. Several projects involving the creation of detailed virtual reconstructions of archaeological sites have been undertaken in recent years. These virtual reconstructions have been done using three-dimensional (3D) computer graphics. This project describes the virtual reconstruction of the Laetoli footprints.

The Laetoli footprints are fossilised hominid footprints. They were discovered by Mary Leakey in 1978 in Laetoli, Tanzania and have been estimated to be three and a half million years old. The discovery was of great interest to archaeologists and the general public alike as it was the first evidence of bipedal walk. The first excavation in 1978 revealed two parallel tracks of hominid footprints where the footprints of the one track were significantly larger than those of the other track. After the site was studied and documented by a team of archaeologists, it was reburied to preserve the hardened volcanic ash from damage. Layers of sand, bio barriers and lava boulders were used in the reburial. (Bruessler, 2000)

A site inspection in 1985 showed that acacia trees had started growing over the buried footprints. The roots of the trees had damaged the footprints and some of the scientific value of the site had been lost. This resulted in two field campaigns, in 1995 and 1996 respectively, to conserve the site and prevent further deterioration of the footprints. The campaigns were managed by the Getty Conservation Institute in collaboration with the Government of Tanzania. Scientific interest in the footprints lead to the site being re-documented with higher levels of accuracy and detail. (Bruessler, 2000)

The Department of Geomatics of the University of Cape Town joined the two field campaigns to carry out a 3D documentation of the site. The survey was done by means of close range photogrammetry. The photogrammetric data processing included the processing of track images as well as individual footprint images. The resultant ortho-images and Digital Terrain Models (DTM) were used in this project as well as contour files.

The aim of this project was to develop a method of displaying the surveyed data of the footprints in such a way as to be useful to scientists and other interested people. This required that the data be used to construct accurate digital 3D graphics models of the footprints, which would support user

visualisation and interaction with the archaeological data. These capabilities had to be included into a software program that would be able to run on non-specialised computers with various computer platforms.

To fulfil the aims of the project, current and past archaeological reconstruction projects were studied and are briefly described in Chapter 2. The projects reviewed consisted of the creation of a virtual reality model of Stonehenge, the creation of a 3D model of the Florentine Pietà, the use of visualisation in the reconstruction of a pre-Inca temple, the virtual restoration of the Visir tomb and the visualisation of the Terramara of S. Rosa.

The methods of visualising data were also researched. Chapter 3 examines the definition of visualisation, the creation of visualisation packages, the use of 3D graphics as a visualisation tool, computer requirements for 3D graphics and existing 3D graphics authoring packages.

The selected 3D graphics authoring package is discussed in Chapter 4. A brief introduction to Java3D is given as well as descriptions on the methods used to create 3D graphics. Chapter 5 and Chapter 6 describe the construction of an application for the display of the Laetoli footprints as well as the construction of interactive tools. More specifically Chapter 5 discusses the construction of the 3D models and their appearances and Chapter 6 deals with the development of interactive tool such as measurement tools, gradient calculation tools and profile creation tools.

The final chapter, Chapter 7, contains the conclusions and recommendations of the project.

Constraints of this project included:

- Time - The development stage of the project was limited to 18 months.
- Experience - The development of useful archaeological tools required experience in the field of archaeology, which was not always available.
- Technology - Although computer technology is progressing at a rapid pace, certain functions in 3D graphics require large amounts of computational power, which is not available on current personal computers.

Chapter 2
Literature Review

Visualisation of archaeological sites or objects is not a new concept. Documenting data obtained from archaeological surveys in ways that allow the study and preservation of the data has been an ongoing process. With the advent of and advances made in high-end computer graphics, computers are being used more frequently in these studies. Archaeology is a wide-ranging field where recorded data from field surveys may be in many different formats. One of the more common representations of archaeological models is the creation of virtual reality walk-throughs of architectural ruins. In some cases, 3D models of artefacts or statues are created.

Questions have been asked whether architectural reconstructions of archaeological sites using computer graphics are helpful research tools or just pretty pictures that allow people to look at old places. In many cases, reconstructed models are simplified due to inaccurate measurement techniques, data storage limitations or insufficient knowledge of the model. For example, in Greek architecture, walls and columns are inclined in subtle ways to create optical effects. When a survey of the site is done, some of these features are not recorded, thus a 2D or 3D model created from the data will not be accurate. Subtleties such as these would therefore be lost to future generations. (Sims, 1997)

The rest of this chapter takes a brief look at some of the uses of computer graphics in representing archaeological finds.

2.1 A Virtual Reality Model of Stonehenge

A virtual reality (VR) model of Stonehenge is an example of an archaeological visualisation. It was created both for the purposes of education and entertainment. The researchers and developers of the project aimed to create an accurate model that not only portrayed the stones to within centimetre accuracy but also to represent the night sky above the Salisbury plain as viewed on any given day of the year. Two models were created. The first model was a photo-realistic model that included the surrounding landscape and was of high accuracy. This model required large amounts of computing power and thus another model was created that would be accessible by the general public. The second model was less detailed and was created in a Virtual Reality Modelling Language (VRML) environment, which could be viewed on the World Wide Web (WWW). Figure 2.1 shows a view of the Stonehenge virtual reality model. (Sims, 1997)

2.2 Creation of a 3D model of the Florentine Pietà

The Florentine Pietà is a marble statue carved by Michelangelo whilst he was in his seventies. Two years after he had stopped working on the statue, he attacked it with a sledgehammer. A servant interrupted the destruction of the statue and disposed of it. Another artist, a pupil of Michaelangelo, repaired some of the damage and completed some of the unfinished sections. (Abouaf, 1999)

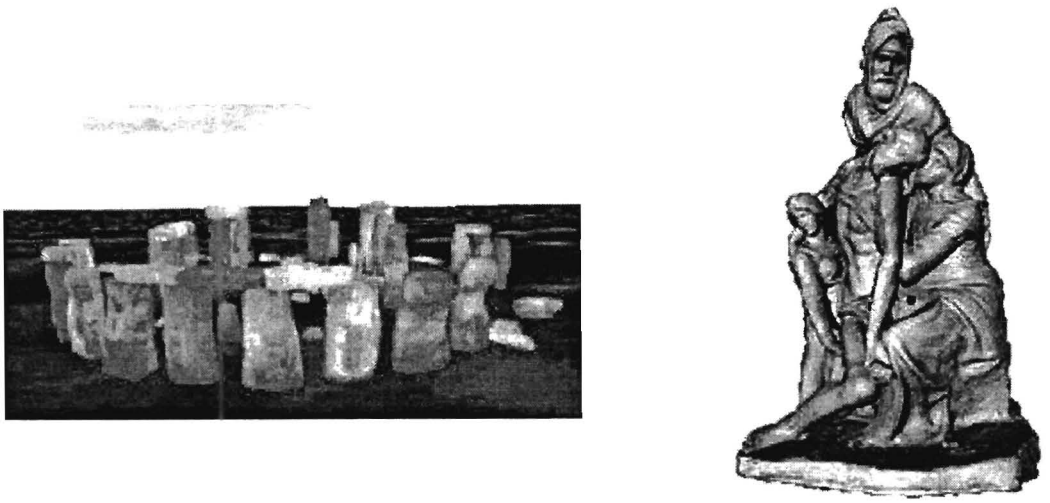


Figure 2.1 – A virtual reality model of Stonehenge (left) and an incomplete digital model of the Florentine Pietà (right)

The statue is exhibited in the Museum of the Opera of Santa Maria del Fiore. A team of researchers from IBM's Thomas J. Watson Research Centre are creating a high-resolution virtual model of the statue. This model is to be created to answer questions such as which portions of the statue represent the original work by Michelangelo and which portions represent the restoration and new work by the second artist. Techniques in 3D digital photography, database management, compression and detail visualisation were used to enable examination and analysis of the statue in ways not permitted on the original sculpture. Two surveys of the statue were done and IBM expects to be working on the data set beyond the year 2000. Figure 2.1 shows an image of the current digital model of the Pietà.

Jack Wasserman, a renowned art historian, focuses on five advantages associated with the use of the technology used to create the digital model of the Florentine Pietà (Abouaf, 1999):

- Convenience and interaction – Allowing users to interact with the digital model enables them to view the statue from perspectives and resolutions not otherwise available.
- Accuracy – The recording of the digital data is accurate to one millimetre, allowing accurate measurements to be made on the digital model.
- Problem Solving – Different perspectives of the statue may lead to new insights in the way it was designed. For example, Christ's torso and right arm are proportioned too long with respect

to the rest of the statue. This appears odd when the statue is viewed at eye level. Elevating the virtual model in the computer to the height it might have occupied if it had been installed above an altar would result in the proportions of the statue appearing more realistic.

- Documentation – The data can be documented allowing it to be distributed and viewed by users who are not able to see the real statue. The data can also be stored for future generations.
- Motivation of new research – As more data becomes available researchers become more interested as they are able to analyse the data and formulate new theories.

2.3 Using Visualisation in the Reconstruction of a Pre-Inca Temple

Another case of visualisation being used in archaeology is the excavation of a pre-Inca temple in Peru. The temple under excavation, called the Huaca Cao Viejo, is one of three stepped pyramids built out of sun-dried bricks at the El Brujo site. Within the temple, archaeologists have uncovered the remains of a ritual enclosure. The temple is hypothesised to have been used as a sacrificial alter where the figures on the ceiling and the wall played a large role in the belief systems of the inhabitants. The ceiling of this temple was of great archaeological interest since it is the only one of its kind known to have figures painted on it. Unfortunately, some time in the past the ceiling collapsed and broke apart. About 3000 pieces of ceiling were recovered, some big and some small. The interpretation of the paintings required that the ceiling be restored. (Kalvin *et al.* 1996)

In order to reconstruct the ceiling, the remaining pieces were digitised. This was done on site due to their delicate nature. The digitised pieces were then moved to a workstation for matching and manipulation with programs built with the *IBM Visualisation Data Explorer* visualisation toolkit. It was hoped that the pieces could be reconstructed digitally like a large puzzle. The matching and alignment of the pieces were based on:

- colours of the pieces;
- interpretation of the figures on the pieces;
- the textures of the pieces;
- and cane markings on the back of the pieces (the cane markings resulted from the construction of the ceiling).

The relationships between the pieces were stored in a database. Queries made to the database resulted in the selection of pieces, which were aligned on a computer screen. Orientation and thickness of the cane markings were used to guide and confirm possible alignments. At the time of the article, 218 pieces had been digitised and plans were being made to digitise the remainder of the 3000 pieces.

2.4 Virtual Restoration of the Visir Tomb

The Visir tomb is an Egyptian tomb of the seventh century BC. The tomb has been archaeologically explored since 1974 during which much of it has been documented. Thieves have plundered a great deal of the tomb and other parts have been moved to museums around the world. The first phase of the restoration of the tomb began in 1985. It consisted of the (Palamidese *et al*, 1993):

- Reconstruction of damaged or missing parts.
- Restoration of the decorations, which included the integration of lacking texts and scenes.
- Conservation and consolidation of existing mural paintings on the ceiling and walls.

As with the Huaca Cao Viejo temple described before, a database was constructed which contained images of the blocks that were part of the wall and ceiling. Visualisation of the digitised blocks required accuracy for the archaeologist and realism for virtual restoration. The results of the visualisation were promising and consisted of 3D displays of certain areas of the tomb though certain questions such as how much to restore or preserve and what must be done with intrusive features (such as a Roman wall obscuring Egyptian paintings) had to be raised. With the improvement of visualisation, increasingly realistic tours within the virtual monuments will be created.

2.5 Visualisation of the Terramara of S. Rosa

From 1500 BC in Emiliae (northern Italy), the Terramare culture developed. The Terramare are habitations built on pilings and protected by a defensive wall that screened the inhabitants from floods in flat countrysides where seasonal rains are violent. The S. Rosa Terramare is a middle and late Bronze Age site and has been excavated since 1984. It covers about seven hectares of ground and was damaged in the nineteenth century by quarry works. (Forte, 1995)

The visualisation of this area was done by creating a landscape model using a digital terrain model (DTM) and digital images from aerial photographs. In this case, the collection of DTM data was done using a total station. The collection of DTM points was concentrated on the earthwork and other features of interest. A total of 450 points were measured. A DTM model was then generated and the aerial photographs were mapped onto the resulting surface. The resulting digital model allowed a user to interactively move through the virtual landscape and view the virtual structures of the Terramara.

The above five examples all use computer graphics to represent archaeological data in a way that can be used to further human knowledge of the archaeological object. There are many types of

archaeological data and the methods used to model this data may differ from case to case. The next chapter discusses visualisation, which is the basis of representing archaeological models on computers.

Chapter 3
Visualisation

In this chapter, visualisation as a method of creating mental images of data and visualisation as a method of creating computer images of data to facilitate the creation of mental images will be discussed. 3D graphics as a visualisation tool will be introduced and the necessary computer requirements and programming packages for 3D graphics will be discussed.

3.1 What is Visualisation?

"visualize or –lise vb. To form a mental image of (something incapable of being viewed or not at that moment visible). – visualization or –lisation n." (The Oxford Paperback Dictionary)

A useful definition from Foley and Ribarsky (1994) describes visualisation as the binding (or mapping) of data to representations that can be perceived. The types of binding could be visual, tactile or a combination of these.

Visualisation can then be said to be the formation of an image in the human mind of some form of data. These mental images assist a user's understanding of complex material. Unfortunately, in some cases, a user may be unable to form a mental image of certain types of data or the image they form may be incorrect. Reasons for this may be:

- Differing perceptions of data from person to person, causing incorrect or unexpected mental images
- Too great a volume of data for the human mind to visualise.
- The data having such a form as to be incomprehensible or unknown to the human mind.

To avoid these problems an image of the data must be created. Visualisation in Scientific Computing (ViSC), also known as Scientific Visualisation, is the method of using the capabilities of computers to display data in such a way as to allow a human's visualisation ability to be enhanced. In the rest of this project the term 'visualisation' has been used in a way that incorporates both visualisation and scientific visualisation.

3.2 Scientific Visualisation

Scientific visualisation is used in almost all areas of science and engineering. Some examples of the uses of scientific visualisation include cartography, remote sensing, analysis of archaeological data,

physical chemistry and drug design, biochemistry, medical science, archaeological reconstruction, oceanography and meteorology.

McCormick *et al* (1987) define Scientific Visualisation as the use of computer graphics to create visual images that support the understanding of complex and often massive numerical representations of scientific concepts or results. Haber and McNabb (1990) define ViSC as the use of computer imaging technology as a tool for comprehending data obtained from simulations or physical measurements. It is comprised of computer graphics, image processing, computer vision, computer aided design, geometric modelling, approximation theory, perceptual psychology and user interface design. Brodlie *et al* (1992) state: "Scientific visualisation is concerned with exploring data and information in such a way as to gain understanding and insight into the data" and that the goal is to enable a deeper level of understanding of the data under investigation using the powerful visualisation abilities of a human. In order to achieve this, aspects in computer graphics, user interface methodology, image processing, system design and signal processing must be utilised.

The above definitions show that scientific visualisation makes use of computer graphics to aid the human mind in visualising large amounts of data. These graphical representations of data on a computer screen may be useful for a number of reasons such as (Medyckyj-Scott, 1994):

- A user may find graphical representations easier to remember than numerical representations.
- Graphical representations assist a user in the comprehension of complex 2D and 3D spatial relationships. They may also expose unexpected spatial relationships or phenomena otherwise unseen.
- Graphical representations may allow problems to become easier to understand and solve when data is displayed in different forms.
- Users become more motivated and comfortable when viewing graphical representations of data. This results in users being in a better mental state to execute their tasks.

Scientific visualisation offers more than mere static displays. Interaction and animation of data, collected by means of physical measurements or by computer simulation, have become possible due to the twin developments of computing power and capacity for data collection.

In order to facilitate the extraction of ideas and to allow interactivity, tools are created that enable the manipulation and presentation of the accumulated data. The type of visualisation and the audience to whom it is directed determines the nature of the tools. Questions such as the size and nature of the audience, the degree of abstraction (symbolic or detailed and realistic) and the degree of interactivity must be examined. For the layperson, in most cases, static images and animations

may be suitable and tools that allow interactivity are not required. The non-scientific audience want as much information as possible from a visualisation without having to deal with abstract concepts whereas scientists require interactivity to pose queries and formulate theories. (MacEachren et al 1994)

The three principal elements of the human-computer interaction that form the basis of scientific visualisation are the *users*, the *tasks* and the *tools* (Medyckyj-Scott, 1994).

3.2.1 The User

The user is a person that interacts with an application on a computer, captures visual output from the user interface and provides input into the system. The user interface connects the world of the user to the internal data paths of the system. Users are the most variable component of the three elements of human-computer interaction. They are individuals that react to computer interaction in different ways. The following points describe some of the problems a user introduces to the human-computer interaction (Medyckyj-Scott, 1994):

- The users abilities in understanding conventions in human-computer interactions and their preferences in the way they interact with a system differ.
- The users familiarity with the variations in commands that are needed.
- The users different thought patterns result in different pattern recognition and problem solving abilities.
- The users cultural backgrounds may differ resulting in different training. This may result in users seeing and interpreting concepts and meanings in different ways.
- The users expectations due to previous experience of similar interaction may cause misconceptions.

The points listed above may change over time leading to alterations in abilities, expectations and preferences of the user. This will result in the user comprehending the information incorrectly.

In order to make a successful visualisation it is imperative that the user is taken into account. Many visualisation operations on the data are performed in the mind of the user. The computer can ease the user's role in the creation of mental images of data by allowing various graphical representations and manipulation of the data. A computer may allow multiple views of the visualisation. This allows a user to compare visually between various data sets.

The computer can also enable a user to interact with a scientific visualisation. Interaction allowing users to manipulate and modify the parameters of the visualisation. This enables and encourages greater exploration of the data. A user will benefit from the ability to interactively modify

parameters in their problem and observe the results in real-time. The interaction between the user and the computer is the process whereby the user requests that certain tasks be performed by the computer. It is the sequence of events where each event comprises of an execution phase and an evaluation phase. The execution phase is normally supported by user interface tools, which enable users to execute commands. The evaluation phase contains the results of the execution phase displayed in textual form, graphical form or a combination of the two (Medyckyj-Scott, 1994). Interaction may include:

- Movement – animation or the defining of viewpoints may be used allowing the data to be viewed from various locations.
- and User-designer duality – allowing a user to become a graphic author by determining the design of the visualisation and making use of the result.

3.2.2 The Task

The task is an activity that must be fulfilled in order to achieve an objective. Tasks may be split into three categories, namely: input, decision and output. The input stage of the task consists of receiving and understanding of data. Decisions stages consist of thought, determination and planning of procedure. Output stages consist of communication and manipulation of the refined data. By decomposing the task, it becomes possible to define the type of interaction required. Questions such as the type of visualisation tools required for specific tasks and whether they vary from one occasion to another must be answered. The frequency of the tasks must also be determined. If a tool is used regularly, it must be readily available to the user. (Medyckyj-Scott, 1994)

3.2.3 The Tools

The tools are designed to display, operate and manipulate data. The data may be 2D, 3D or even 4D (the fourth dimension being time) and may originate from either real world measurements or computer generation. A tool must be able to transform data stored in the computer into a form that can be displayed on a computer screen. The way the data is displayed on the screen should be as realistic as possible, allowing it to be interpreted by the user.

In older workstations, the tools required that users type in their commands on a command line. Newer workstations offer the user a graphical user interface (GUI). A GUI allows a user to input commands using graphical components on the computer screen. These components can be made to represent real world objects that have the same function. For example, a switch displayed on a computer screen will be interpreted by the user as having the same function as that of a real world switch.

Tools should provide fast response times to user actions. This prevents interruptions to the task at hand. In some cases response times will be long and a progress indicator should be used to notify the user on the status of the task. The tools must make optimal use of screen space. Multiple windows displayed on a computer screen are able to increase viewing space but not necessarily the visual scope as a user can sometimes not see relationships or find windows containing important data. Too many windows as well as small portions of the visual scene being displayed due to limitations of screen resolutions may cause the user to become disorientated. (Medyckyj-Scott, 1994)

3.3 Creating a Scientific Visualisation

There are certain steps in creating a scientific visualisation. The visualisation pipeline (see Figure 3.1) is described by Brodlie (1994). The input to the pipeline is the raw data obtained from either computer simulations, measurements, or a combination of the two. The first step is to build a model based on the input data either by interpolation or by approximation. Once the model has been created it has to be represented geometrically. The second step maps the model to an abstract visualisation object, which results in a geometrical object. The visualisation technique used in this step would depend on the nature of the data and the type of information required from the model when visualised. The third step renders an image from the geometry and the fourth step displays the rendered image on a computer screen. These steps may also apply to the creation of other graphics representations, such as computer games.

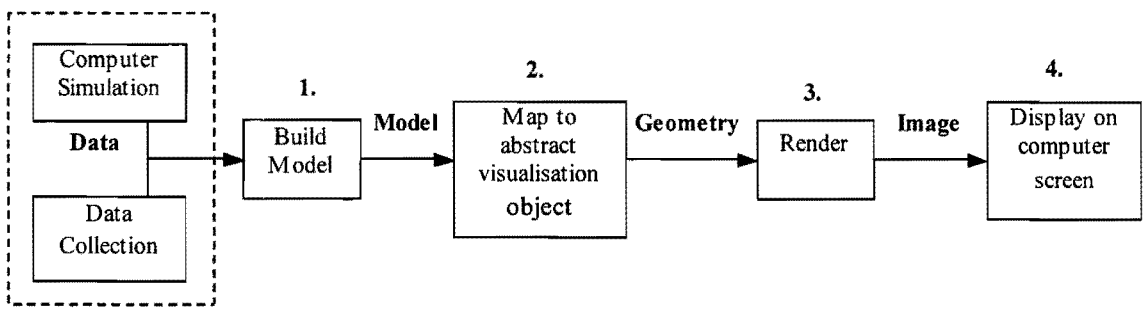


Figure 3.1 - The Visualisation Pipeline

3.4 Attributes of Visual Data

For visual data to be useful for the user, it must have certain attributes. These include desirable levels of intensity, size, duration, separation by distance, difference in form and contrast. In many visualisations, it is necessary to detect areas of change or other forms of variation in data. Some guidelines that enable enhanced detection of variations in visual data are listed below (Schenk, 1994):

- Difference in form between two or more objects is easier to detect than difference in size. This can be seen in Figure 3.2 A and B. In B the use of different geometrical shapes is much easier to detect than the difference in size between the two circles in A.
- Increased contrast between objects enhances detection. The result of increasing the contrast of the two circles in A can be seen in Figure 3.2 C.
- Regular simple geometric shapes are easier to identify than unfamiliar asymmetric shapes. In D two non-regular geometrical shapes have been displayed. It will be easier for the user to remember and identify, at a later stage, the regular geometrical shapes in B than the shapes in D.

In many cases features such as contrast, size, shading, colour and form are used to communicate various information to the user in a legible and unambiguous manner. This may be seen clearly in various types of maps where areas of interest are symbolically displayed in a manner recognisable to the user. The memory of the user becomes an important part in the visualisation process as symbols and features containing information about the data need to be recognised and remembered. In cases where data needs to be recognised, a legend becomes necessary.

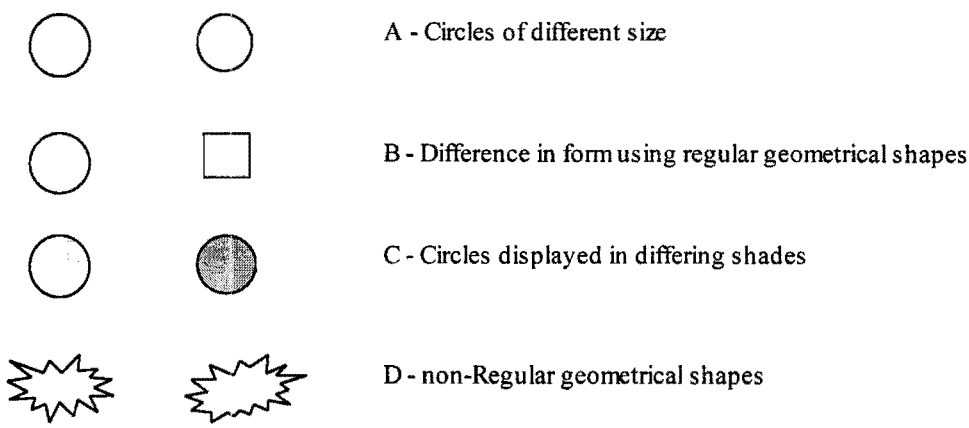


Figure 3.2 - Guidelines for Enhanced Detection

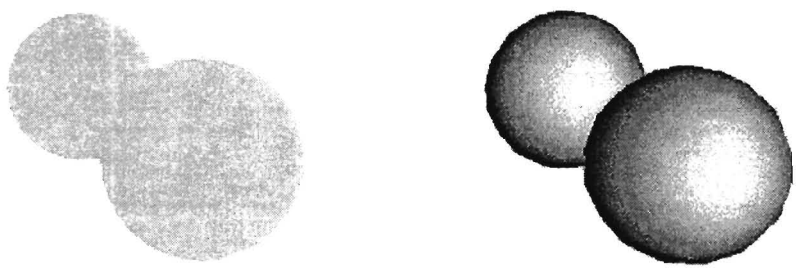
3.5 3D Graphics as a Visualisation Tool

The main thrust of computer graphics has been to turn the computer screen into a window to a virtual world. This has lead to the simulation of realism by means of 3D graphics, which increases the capacity of humans to generate mental images of 3D objects. 3D graphics can be said to be a method of scientific visualisation.

3D Graphics is the field of computer graphics concerned with generating and displaying three-dimensional objects in a two-dimensional space (i.e. the display screen) and includes operations such as hidden-surface removal, surface texturing, lighting, movement and virtual worlds. Pixels in

2D graphics have the properties of position, colour and brightness. The positions of the pixels are recorded as x and y co-ordinates which correspond to co-ordinates on the screen. Examples of 2D computer graphics are graphs, bar charts or photos in digital format.

In 3D graphics, a depth property is added to the 2D properties of the pixels creating an image. This property indicates a pixel's position on an imaginary z-axis pointing out of the screen, giving rise to shading and hidden surface removal. By defining 3D graphics in this manner, it may be said that a photo in digital format is a 3D graphics object as it has shading and hidden surface removal. A true 3D graphics object has 3D attributes. A photo consists of only 2D information. This results in 3D graphics being interactive as the 3D attributes of shading and hidden surface removal can be altered in real time. In Figure 3.3 two images are shown, one representing two circles in two dimensions and the other showing two spheres in three dimensions. In the left image it is not possible to say which circle lies in front of which if in fact they are overlapping circles. The right image, by adding a depth quality, demonstrates the use of shading and hidden surface removal. It becomes clear which sphere is in front and which is behind. 3D graphics also support multiple objects that may interact with one another. (PC Webopedia, 1999)



2D Representation 3D Representation

Figure 3.3 - 2D and 3D Graphics Representations

The display of 2D or 3D pixel information on a computer screen is known as rendering. 3D rendering is the process of adding realism to computer graphics by adding 3D qualities such as shadows and variations in colour and shade as well as displaying the pixel information on the computer screen. Considerable computing power and memory are required and in the past this was only available on powerful workstations. Present day personal computers, with 3D graphics accelerators that contain their own memory and specialised microprocessors, are able to handle many of the 3D graphics rendering operations currently available. (PC Webopedia, 1999)

3.6 Computer Requirements

In most scientific visualisations, there are large amounts of data that are processed. Most visualisation tools, in order to realise good results, require great computing power and good

graphics systems. Computer graphics Application Programming Interfaces (API) have evolved along with computer hardware and user requirements.

In the early eighties, 3D graphics applications were developed in a programming language called FORTRAN, which was run on state of the art raster terminals, plugged into mainframes and minicomputers. Developers were limited to primitives and wire frame surfaces. A primitive is an object from which higher-level, more complex objects and operations can be constructed. In graphics, primitives are elements such as lines, polygons and curves that can be combined to create more complex graphical images. The advent of workstations increased the capabilities of 3D graphics significantly. The C programming language, making calls to computer graphics libraries such as PHIGS and PEX, was used to create solid models. Object oriented programming languages such as C++ became more popular as workstations became more powerful. C++ and OpenGL became the standard for 3D graphics development. Their capabilities routinely included texture mapping, additional control over the rendering process and lighting models (see Figure 3.4). (Sun Microsystems, 1999b)

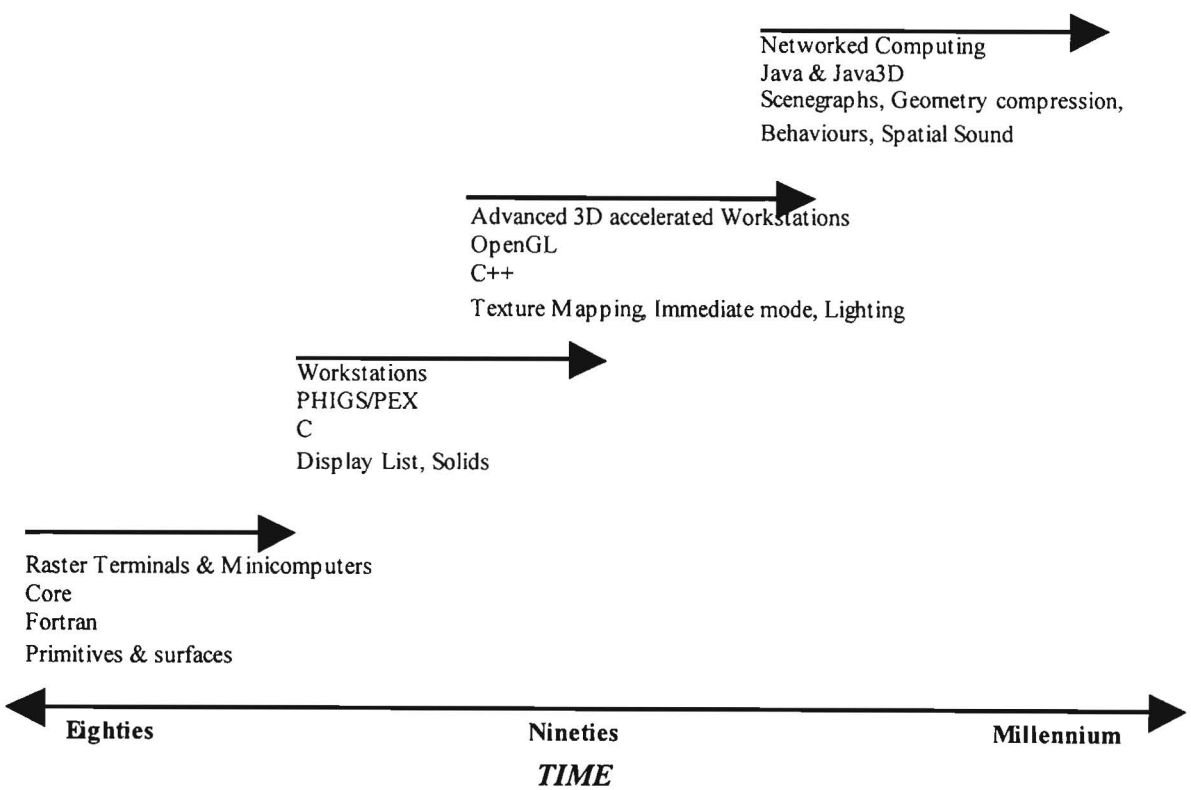


Figure 3.4 – The History of 3D Graphics

The current age is the network age where large amounts of data are shared across networks. Java is one of the major languages being used because it is platform independent. OpenGL is still utilised for 3D graphics and is used by Java3D as an interface to the computer hardware. The 3D models

created with Java3D can be distributed over the network but a workstation with good performance, large amounts of memory and disk space and powerful graphics facilities in terms of available colours and display speed, is still needed. Modern day workstations are becoming faster and have greater abilities to perform real-time processing of data. (Sun Microsystems, 1999b)

3.7 Choosing a Package

As 3D computer graphics have become more popular, new tools have been created to generate and display 3D graphics objects. Three of the more common tools that can be used to create interactive 3D graphics are the OpenGL API, the Java3D API and the Virtual Reality Modelling Language (VRML).

3.7.1 VRML (Virtual Reality Modelling Language)

VRML is not a programming language like C or Java, nor is it a "mark-up" language like HTML. It is a modelling language used to describe 3D graphics scenes. Most of the commonly used 3D graphics utilities such as hierarchical transformations, light sources, viewpoints, geometry, animation, fog, material properties and texture mapping are defined by it. VRML is a multi-platform file format for publishing 3D web pages and is not a programming library for application developers nor is it an API even though it includes scripting language. (Hartman and Wernecke, 1996)

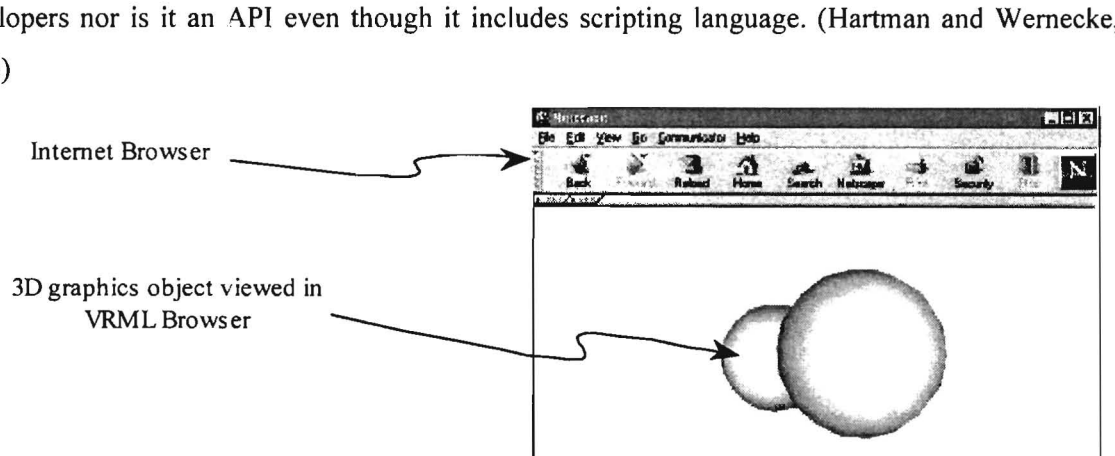


Figure 3.5 – Example of how a VRML file is viewed

Since VRML is a 3D graphics file format, a program is required to view the file. These programs are known as browsers and are created for various platforms. A common VRML browser on the Windows platform is *CosmoWorlds*, which runs as a plug-in for Internet browsers such as Netscape or Microsoft Internet Explorer. The following figure (see Figure 3.5) shows a VRML file describing two spheres being viewed in a VRML browser that is embedded in an Internet browser. The VRML browser may include various tools, the most common being movement and viewpoint

tools, which allow a user to explore the objects described by the VRML file from various perspectives.

Although VRML is not an API, it does have scripting capabilities that allow a user to perform simple interactions with the 3D objects created. In order to create more advanced user interactive tools an event-based model of interaction is used. In this model, events are used to send changes from an external language to the VRML browser and vice versa. The external language used is normally Java due to its portability over the Internet.

3.7.2 OpenGL

OpenGL is "the most widely adopted graphics standard." (OpenGL Overview, 1999) It is an environment for developing portable, interactive 2D and 3D graphics applications and was introduced in 1992. All UNIX-based platforms, as well as Windows based platforms, support 2D and 3D graphics applications developed in the OpenGL API. Software developers have access to geometric and image primitives, display lists, modelling transformations, lighting and texturing, anti-aliasing, blending, and many other features.

OpenGL is a graphics library and in order to use it, a programming language such as C++ has to be utilised to create a user interface whereby the user can interact with a 3D model created using the functions from OpenGL. By using C++, the program can no longer be run on all software platforms. Java can be used to write graphics applications using the graphics functions of OpenGL, which will be portable over various platforms.

3.7.3 Java3D

The Java3D API is a full-featured, high-performance 3D graphics architecture that integrates smoothly as an extension to the Java programming language. It is an API that yields a high degree of interactivity while preserving true platform independence and it provides the functions for creation of imagery, visualisations, animations, and interactive 3D graphics application programs (Sun Microsystems, 1999a). Java3D will be discussed in more detail in the next chapter.

Examples of the code required to create a sphere with a light source using the above three methods are displayed in the Appendix. It can be seen that the VRML code is significantly shorter than that of the Java3D and OpenGL code. The reason for this is that, as mentioned before, VRML runs in a browser. Code for a user interface is therefore not required.

Chapter 4

Java3D

This chapter will discuss the Java3D API and how to create 3D graphics using it. It will be assumed that the reader is familiar with the Java structure.

4.1 The Java3D API

The Java3D API is an interface for writing programs to display and interact with 3D graphics. It is a hierarchy of Java classes, which serve as the interface for 3D graphics rendering. A programmer works with high-level constructs to create 3D geometric objects. These geometric objects reside in a 'virtual universe', which is then rendered. A virtual universe is therefore a conceptual space in which visual objects 'exist' and may vary in size from the astronomic to the subatomic. The details of rendering are automatically handled by the API making it easier for a user to display 3D graphics on a computer screen.

A Java3D program creates instances of Java3D objects and places them in a scene graph data structure. The scene graph is an arrangement of 3D objects in a tree structure. It specifies the content of the virtual universe and how it should be rendered. The Java3D API defines over 100 classes, which are considered as the core classes. Other classes known as the utility classes are also available. These Java3D classes can be split into four major categories, namely (Bouvier, 1999):

- *Content loader* - read 3D scene files from other 3D graphics packages and creates Java3D representations.
- *Scene graph construction aids* - enable a user to add and remove objects from the scene graph.
- *Geometry classes* - are used in the creation of geometric objects.
- *Convenience utilities* - extend a core class to make them easier to use.

4.2 The Construction of a Scene Graph

A Java3D virtual universe is created from a scene graph. The scene graph is assembled from objects that define geometry, sound, lights, location, orientation and appearance of visual and audio objects.

The scene graph is a data structure composed of nodes and arcs where the nodes are the data elements and arcs are the relationships between the data elements. In the scene graph the nodes are representative of Java3D classes. The arcs represent two kinds of relationships between the Java3D nodes. The most common relationship is the parent-child relationship. This means that a group

node on the scene graph may have many children but only one parent and a leaf node has one parent and no children (see Figure 4.1). The second type of relationship is the reference relationship. (Bouvier, 1999)

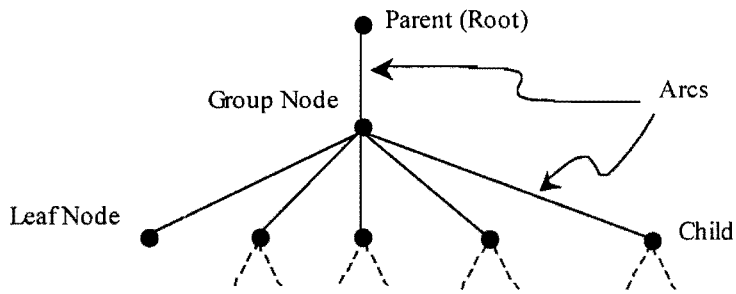


Figure 4.1 – A Tree Graph with Nodes and Leaves

The scene graph is a tree structure constructed of *Node* objects in parent-child relationships. A scene graph has a root from which other nodes are accessible by following the arcs. The arcs cannot form cycles and only one path exists from the root of the tree to each of the leaf nodes. The path from the root to the leaf node is called the scene graph path of which there is only one for each leaf node in the scene graph. The scene graph path will completely specify the attributes of the leaf node. The attributes, also known as the state information, include the location, orientation and size of the visual object.

The root of the scene graph is a *Locale* object, which provides a reference point in the virtual universe. It can be thought of as a landmark used to determine the location of visual objects in the virtual universe. More than one *Locale* object can be specified for a Java3D program. This results in the definition of more than one virtual universe. A drawback is that there is no means of communicating among virtual universes. It is recommended that one virtual universe be used in each Java3D program. (Bouvier, 1999)

Each *Locale* object may serve as the root of multiple sub-graphs of the scene graph. The roots of the sub-graphs are known as *BranchGroup* objects. The sub-graphs are also known as branch graphs. There are two different categories of branch graphs, namely (Bouvier, 1999):

- View branch graph – This branch specifies the viewing parameters, which include location and direction.
- Content branch graph – This branch specifies the contents of the virtual universe such as the geometry, appearance, behaviour, location, sound and lighting.

Java3D consists of classes that create the scene graph. The *Virtual Universe*, *Locale*, *Group* and *Leaf* classes create the hierarchy of the scene graph. All objects other than the *Virtual Universe* and *Locale* objects are scene graph objects, which are the superclass for almost all core and utility classes in Java3D. The *SceneGraph* object has two subclasses, namely: *Node* and *NodeComponent*. The following points describe these classes in more detail (Bouvier, 1999):

- The *Node* class – This class defines important common methods for its subclasses. Subclasses of the *Node* class compose the scene graph. The *Node* class is never used directly as it is an abstract class and only methods defined in it can be used. The following classes are subclasses of the *Node* class:
  1. *Group* class - This class is used to specify location and orientation of visual objects in the virtual universe. The *BranchGroup* and *TransformGroup* classes are subclasses of the *Group* class.
  2. *Leaf* class - This class is used in the specification of shape, sound and behaviour of visual objects in the virtual universe. The *Shape3D*, *Light*, *Behaviour* and *Sound* classes are subclasses of the *Leaf* class. These classes do not have any children but they may reference *NodeComponents*.
- *NodeComponent* class - This class is the superclass used in the specification of geometry, appearance, texture and material properties of a *Shape3D* (*Leaf*) node. It is not part of the scene graph but is referenced by it. A *NodeComponent* may be referenced by more than one *Shape3D* object.

Figure 4.2 shows an example of a scene graph. The scene graph is rooted at the *Locale* object, which is attached to the virtual universe. From the *Locale* there are two branches, the left-hand side branch is the content branch graph and the right-hand side branch is the view branch graph. The *Shape3D* node, as described before, is a leaf node and it has no children. The *Appearance* and *Geometry* node components are referenced to the *Shape3D* node by reference arcs. In the same way, the other objects in the view branch graph are referenced to the *View Platform* leaf node. The one difference in this branch graph is that there is an extra node, namely the *TransformGroup* node. This node is a sub-class of the *Group* node (Bouvier, 1999).

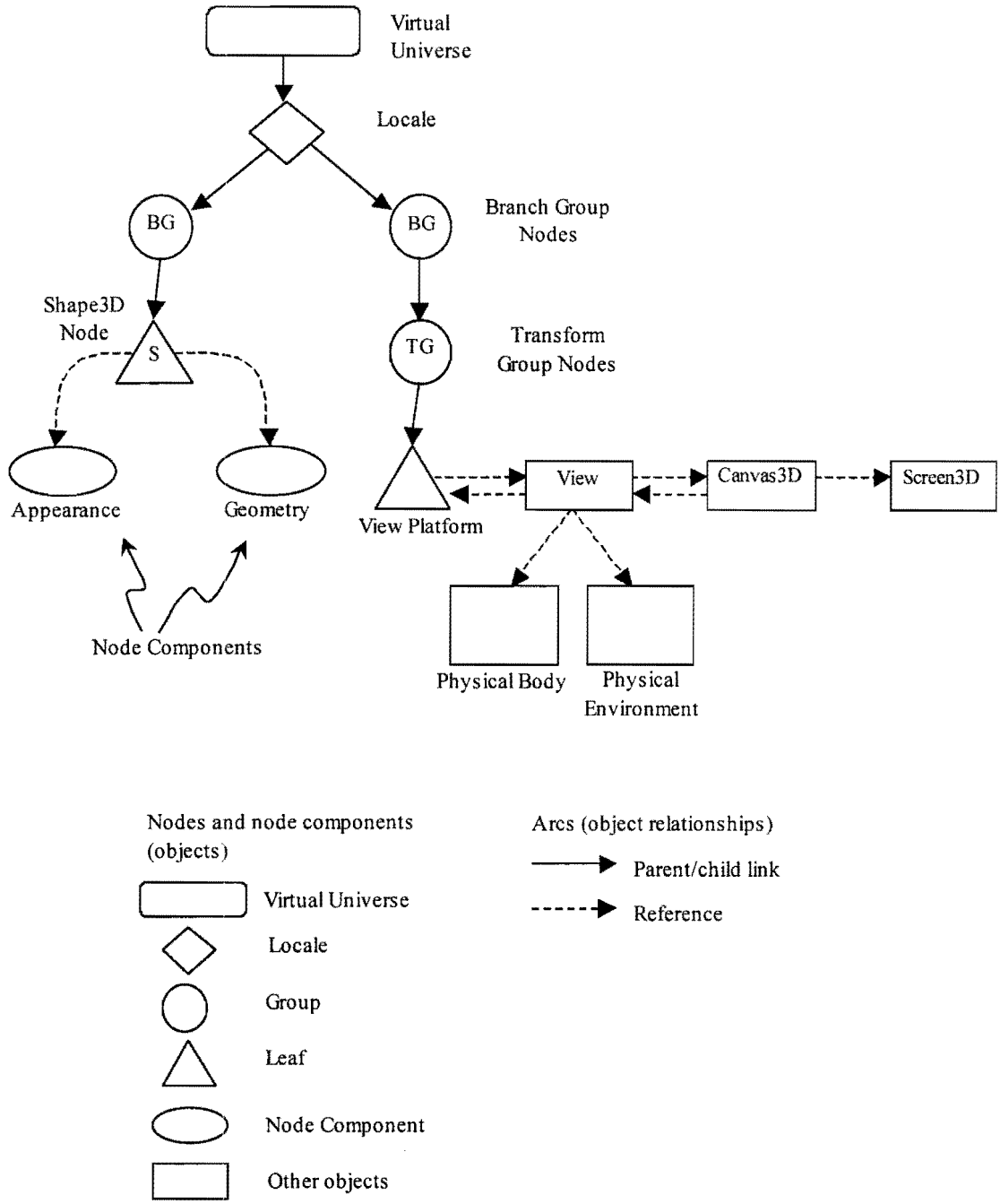


Figure 4.2 – Example of a Scene Graph

4.3 Creation of a Simple Java3D program

The basic procedure for the creation of a Java3D program can be seen in Figure 4.3. This example ignores the detail but illustrates the concept of Java3D programming where branch graphs are created for the scene graph. Since the view branch graph of many Java3D programs will be similar, a utility class was provided that combined the second, third and fourth steps into one class. This

class is called *SimpleUniverse*. By simplifying the creation of the view branch graph, a user can concentrate on the content of the program.

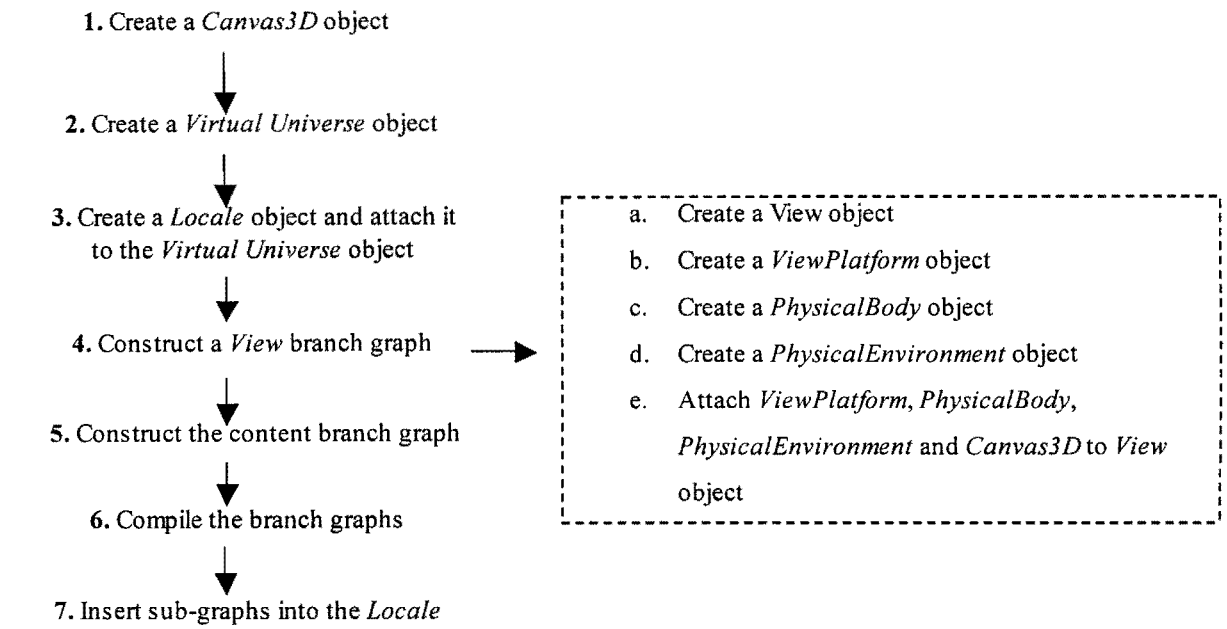


Figure 4.3 – Structure of a Simple Java3D Program

The *Canvas3D* object, mentioned in the example above, provides an image in a window on the computer screen. It is the image plate of the virtual universe. The *View* object is required to allow the user to view objects in a virtual universe. It contains parameters needed to render a 3D scene from a viewpoint. The *ViewPlatform* object controls the position and orientation of the viewpoint and the *PhysicalBody* object contains specifications on the location of the user head. These specifications include the position of the left and right eyes of the user. The *PhysicalBody* object is used to set-up input sensors for head tracking devices. In step six the branch graphs are compiled. This is to done to allow the information specified in the branch graphs to be converted to a more efficient form for rendering.

4.4 The Java3D Co-ordinate System

The term 'virtual universe' commonly refers to a three dimensional space in which 3D objects can be inserted. Each *Locale* object specified in a Java3D program establishes a virtual world Cartesian co-ordinate system from which it references visual objects. When one *Locale* object is present, only one co-ordinate system is used. The co-ordinate system used in a Java3D virtual universe is a right-handed system. The x-axis points to the right, the y-axis points upwards and the z-axis points out of the screen towards the viewer (see Figure 4.4). The distance units are in meters.

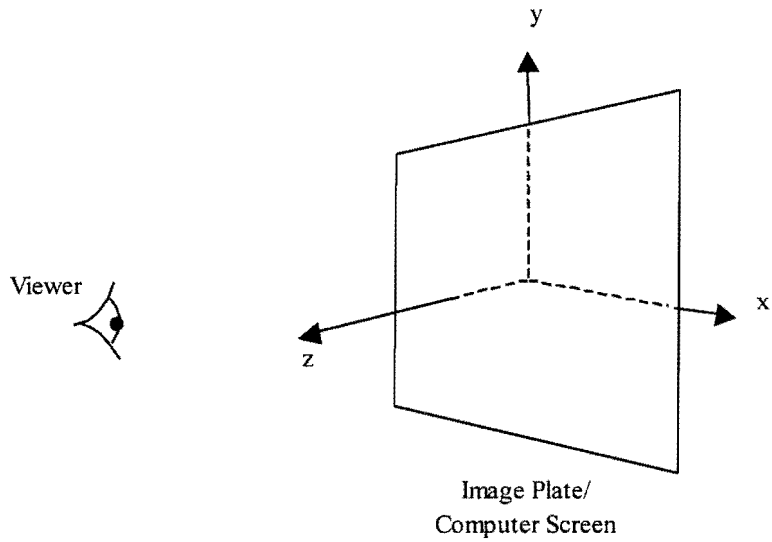


Figure 4.4 – The Java3D Co-ordinate System

4.5 Creating 3D Geometry

There are three ways to create geometry objects in Java3D.

1. The first method is to use the utility classes to create objects such as rectangles, cylinders, cones and spheres.
2. The second method allows a user to specify vertex co-ordinates for points, lines and polygon surfaces.
3. The third method is the use of content loaders, which load geometric objects from other 3D specifications such as VRML.

The *Shape3D* scene graph node is a leaf node and it defines visual objects in Java3D. It does not contain information about the shape or colour of an object. This information is stored in the *NodeComponent* objects that are referred to by the *Shape3D* object. The *Shape3D* node can refer to one *Geometry* node component and one *Appearance* node component.

4.6 Interaction

As mentioned before, the inclusion of interaction and animation in a Java3D program allows a virtual world to become more interesting and useful to a user. Interactive tools create changes in the virtual world in response to user actions. Animation is defined as changes in the virtual world caused without direct user action. Animations usually correspond to the passage of time and may be activated by a user making use of some form of switch. In Java3D, both interaction and animation are specified through the use of the *Behaviour* class.

The *Behaviour* class provides the mechanism to include code in a Java3D program that alters the scene graph. It is a link to code that provides changes to the graphics of the virtual universe. At some stimulus it changes the scene graph or objects in the scene graph. The stimulus can be the press of a key, a mouse movement or click, the collision of objects in the virtual world, the passage of time, some other event, or a combination of these. These changes include addition and removal of objects from the scene graph, the changing of attributes of the objects, the rearranging of objects in the scene graph, or a combination of these. (Bouvier, 1999)

The *Behaviour* class is part of the scene graph. The *Behaviour* node is inserted in the scene graph and it references the object that is to be changed. The locations of the *Behaviour* nodes are normally dependent on the effect on the *scheduling bounds* and code maintenance. A *scheduling bounds* is a region that surrounds an object to which the *Behaviour* class is being applied. The activation volume of a view must intersect the *scheduling bounds* of a behaviour for the behaviour to be active. Behaviours will only be executed in response to a stimulus when they are active. An example of this is when an object in the virtual world has been translated to a point outside the viewpoint's field of view. In some cases, when an object is no longer visible, it is not necessary to continue translating it and the behaviour is deactivated. Figure 4.5 shows an example of the placement of a behaviour node in a scene graph.

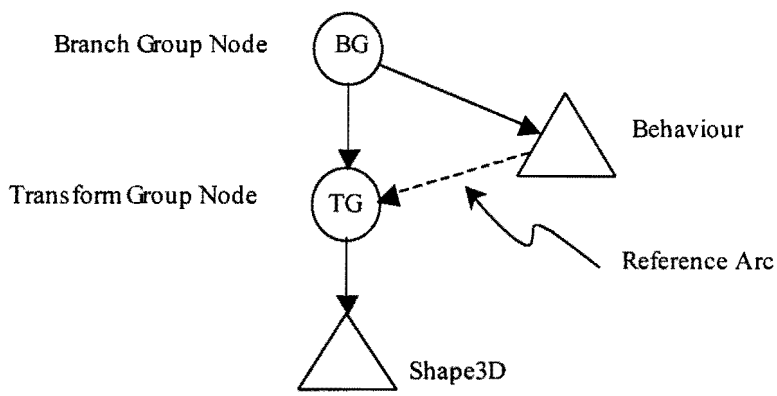


Figure 4.5 – Position of a Behaviour Node in a Scene Graph

4.7 A Simple Example Program

Figure 4.6 contains uncompiled Java and Java3D code that demonstrates the creation of an object in a virtual universe. In this program the *SimpleUniverse* class was used, thus the viewing parameters did not need to be set. Figure 4.7 shows the scene graph path for the program. As explained above, there is no view branch graph as it is specified in the *SimpleUniverse* class. The only branch graph that is present is the content branch graph, which is attached to the simple universe. The first node in the branch graph is the *TransformGroup* node (TG). This node specifies

the position and orientation of all objects below it in the scene graph path. The next node is the *Shape3D* node, which has reference arcs to the *Geometry* and *Appearance* node components.

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;

public class Start extends Applet {

 public BranchGroup createSceneGraph() {
  BranchGroup sceneRoot = new BranchGroup();
  TransformGroup sceneTG = new TransformGroup();
  sceneRoot.addChild(sceneTG);
  Appearance app = new Appearance();
  Cube cube = new Cube();
  Shape3D shape = cube.getChild();
  shape.setAppearance(app);
  sceneTG.addChild(shape);

  return sceneRoot;
 }

 public Start() {
  setLayout(new BorderLayout());
  Canvas3D canvas3D = new Canvas3D(null);
  add("Center", canvas3D);
  BranchGroup scene = createSceneGraph();
  SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
  simpleU.getViewingPlatform().setNominalViewingTransform();
  simpleU.addBranchGraph(scene);
 }

 public static void main(String[] args) {
  Frame frame = new MainFrame(new Start(), 256, 256);
 }
}
```

Imports the base classes o which the program runs

Method to create the Content Branch Graph

Creation of the Geometry

Adding of Geometry to Shape3D node

Adding of Appearance to Shape3D node

Creation of the Simple Universe

Figure 4.6 – The Code for a Simple Java3D Program

In the program a cube was created that had different colour faces. The first image of the cube created by the Java3D program in Figure 4.7 shows the cube from the top. The second image shows the cube, which has been rotated. The *TransformGroup* object handles the rotation of the cube.

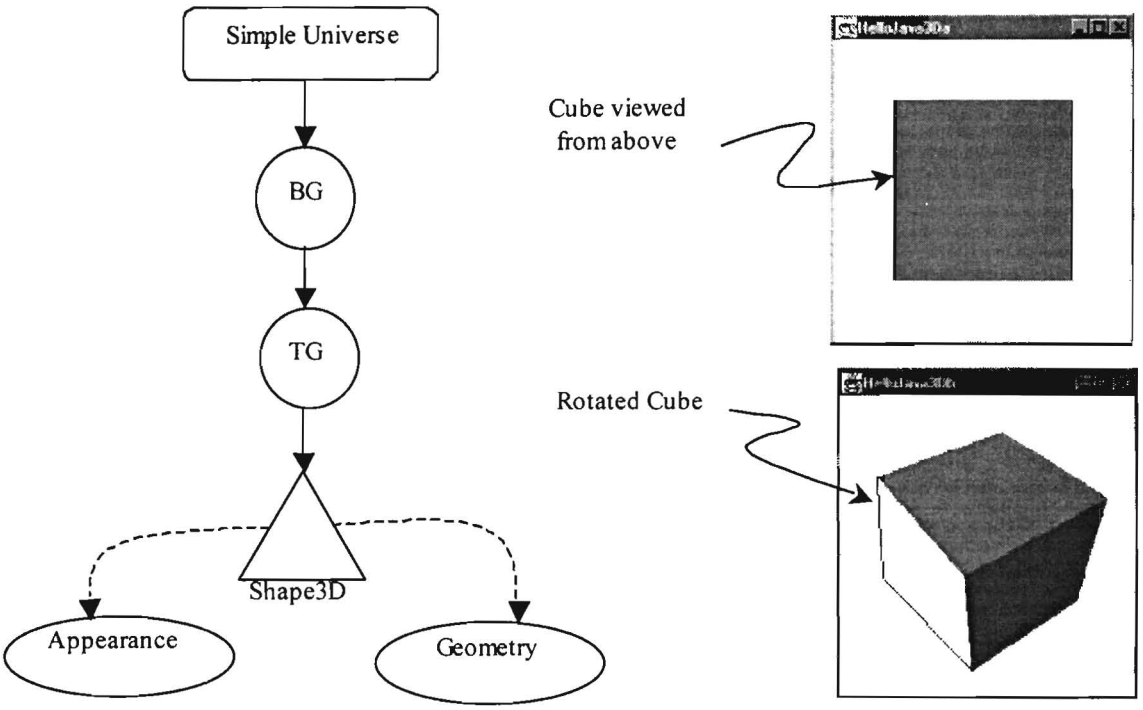


Figure 4.7 – The Scene Graph of the Simple Java Program as well as the Output

4.8 The Java3D Structure Used

At the beginning of the project involving the Laetoli footprints, a scene graph was created. As the project progressed, changes to the scene graph were made when visualisation tools were added. Figure 4.8 shows the scene graph used in the project. The basic structure is the same as in the example of Figure 4.2, where there are two branches off the *Locale* object, namely the view branch graph and the content branch graph. The structure of the view branch graph is the same as it is in Figure 4.2 and was therefore not drawn.

The first node in the branch graph has two branches. The right hand branch goes to a *Switch* object. A switch is an object that determines which child below it in the scene graph is rendered. In the case of the light switch, either the *Directional Light* or the *TransformGroup* (TG) node can be activated. The lights are discussed in section 5.4. The branch to the left of the *BranchGraph* (BG) node goes to a *TransformGroup* node, which is used to change the position and orientation of the objects below it. The three branches from this point will be described from left to right:

- The first branch contains the measurement spheres (see section 6.3). These spheres also have their own *TransformGroup* node. Although they do not belong to the *Shape3D* class, as they are provided as a Java3D utility class, they do have an *Appearance* node component.
- The second branch contains the main geometry of the program. There are three separate *Shape3D* nodes, each with their own *TransformGroup* node. The three *Shape3D* nodes are as

follows (left to right): flood plane (see section 6.1), contours (see section 5.7) and object (see section 5.1). Each of these nodes has *Geometry* and *Appearance* node components.

- The third branch contains the tools for the movement (see section 5.2), measuring (see section 6.3) and fly-through tools (see section 5.8). These tools allow a user to interact with the objects in the virtual universe and are therefore extensions of the *Behaviour* class. There is a switch function for these behaviours as only one of them can be active at any one time. The reference links have not been shown. The reference links of the movement and fly-through behaviours refer to the viewpoint parameters in the view branch graph. The measurement behaviour references the measurement spheres discussed previously.

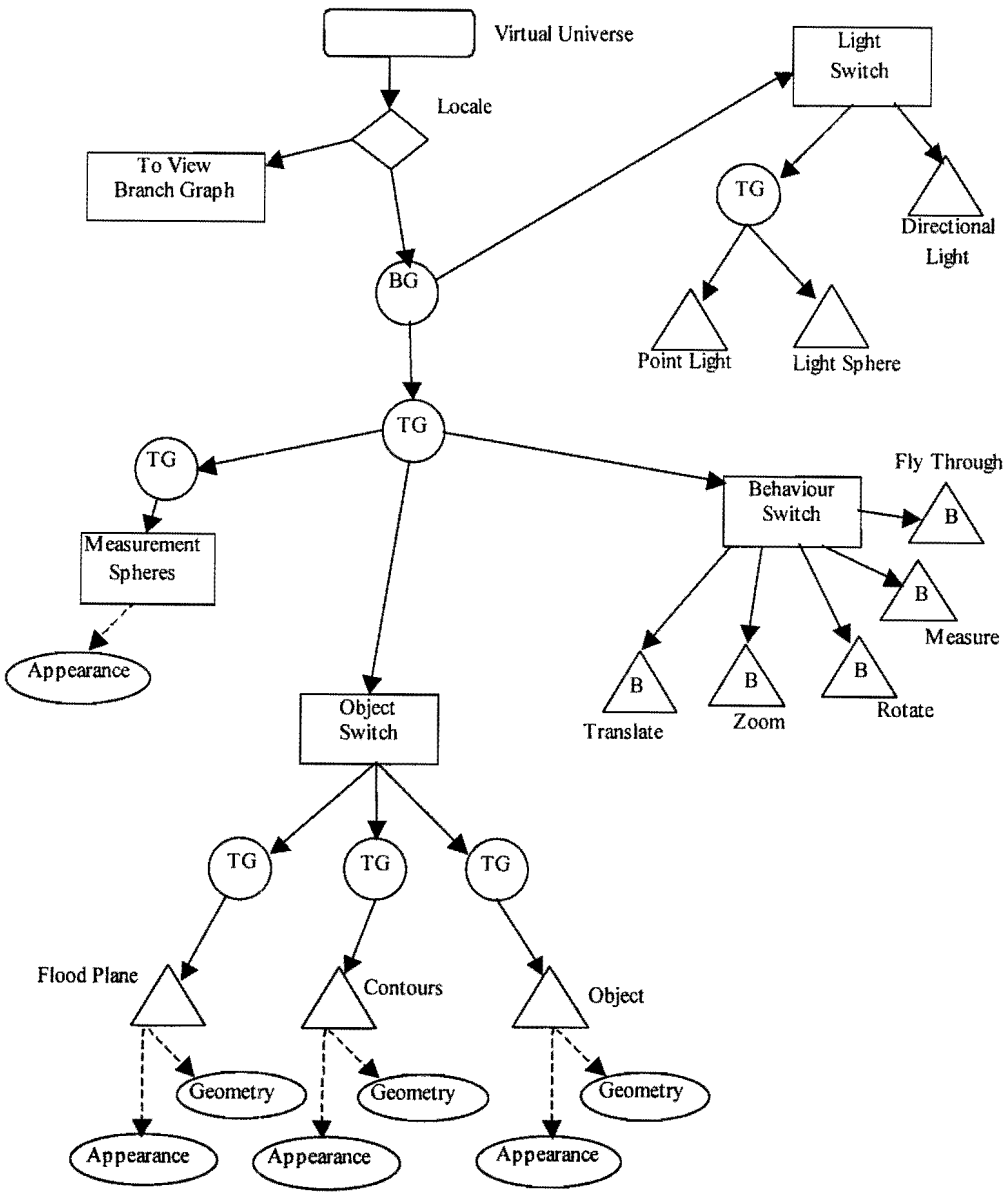


Figure 4.8 – Scene Graph for this Project

Chapter 5 and Chapter 6 discuss the creation of 3D surfaces and their properties as well as the construction of user interactive tools.

Chapter 5

3D Graphics with Java3D

This chapter discusses the creation of 3D graphics using Java3D.

5.1 The surface

This section describes the different types of surfaces that can be created, the data that was used to create the 3D surfaces and the algorithms used.

5.1.1 3D Surface Types

A 3D surface can be displayed in four different ways (Stynes, 1996):

- Rendering a surface of points. Not much can be determined from this except to see where individual points are and to get a vague understanding of the nature of the surface.
- Wire frame rendering. Points in a grid are connected to adjacent points by lines. This reveals a little more about the nature of the surface but it may still be unclear especially as the viewpoint is moved further away from the surface.
- Filled polygon rendering. A solid appearance is created by filling the polygons of the wire frame and shading them according to the direction in which they reflect light. The polygons are normally triangles.
- Gouraud shaded rendering. The most realistic representation of a surface, it blends the shading in each polygon to create a less angular surface. This method is usually the most informative method of displaying a surface.

Even though the most realistic way of rendering a surface is preferable, a simple wire frame may sometimes be favoured in terms of speed of rendering when interaction is required as there are fewer surface calculations required by the computer.

5.1.2 The 3D Data

The Digital Terrain Models (DTM), obtained from the photogrammetric data processing stage of the Laetoli project, specify the geometry of the footprints and are fundamental to the creation of the 3D surfaces. The DTM files consist of columns of x, y and z values (see Table 5.1). The co-ordinates are arranged in a grid format with the spacing between the points being 2.5 mm in both the x and y directions. In the DTM files used, the x values are kept constant for the first row while the y values are incremented by 2.5 mm. At the end of the first row, the x value is incremented by 2.5 mm and the y values begin from the original value and are incremented by 2.5 mm as before. The files consist of between ten and twenty thousand points.

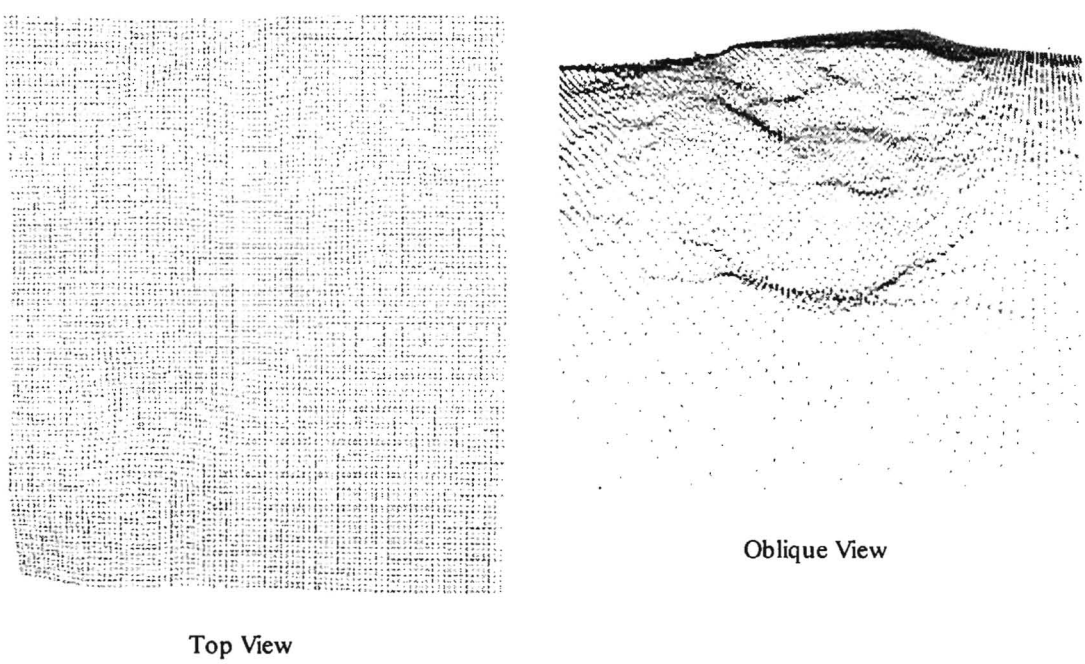
In Java3D the foremost step to rendering the co-ordinates in 3D is to determine methods of reading the data so that it could be rendered in the various ways described in section 5.1.1. Three arrays are created for the input of the x, y and z co-ordinates respectively. These arrays are used to store the co-ordinates for use in the creation of the 3D surface and other tools.

Table 5.1 – Example of a DTM file

102235	98465	1008.599
102235	98467.5	1009.628
102235	98470	1010.66
:	:	:
:	:	:
102237.5	98465	1009.633
102237.5	98467.5	1010.575
:	:	:

5.1.3 Rendering a Surface of Points

The first method of displaying the points described in section 5.1.1 is described in this section. Each co-ordinate is displayed as an individual point. As points do not have surface qualities they do not reflect light and are not visible. In order for the points to be visible, they must be given an ambient lighting quality (discussed in section 5.6.1).



Top View

Oblique View

Figure 5.1 – Surface Rendered as Points

From a top view little can be seen other that a rectangular scatter of points. The choice of the projection used to display the points on the computer screen will affect the perceived shape of the surface (see Section 5.3). In Figure 5.1, a perspective projection has been used to display the points in the virtual universe.

From an oblique view (see Figure 5.1) a vague representation of the surface can be seen. The effectiveness of the view of the points will be dependent on the nature of the surface in terms of height variations. The points render quickly in the virtual universe, as there are few operations or calculations that need to be performed by the computer.

5.1.4 Wire Frame Rendering

The next method of display is the wire frame. The wire frame is essentially displayed as a collection of lines that define a surface. A co-ordinate index informs the API between which points to render lines. The co-ordinate index is an array of integer values where each integer value represents the position of a co-ordinate in the array of co-ordinates. A line is rendered between the co-ordinates represented by the first and second integer values in the co-ordinate index. The third and fourth points in the co-ordinate index reference the next line and so on (see Figure 5.2). The co-ordinate index therefore contains an even number of entries. Due to the fact that a co-ordinate index must be created to render the surface in the form of a grid, more calculations are required and the process of creating this model is more difficult.

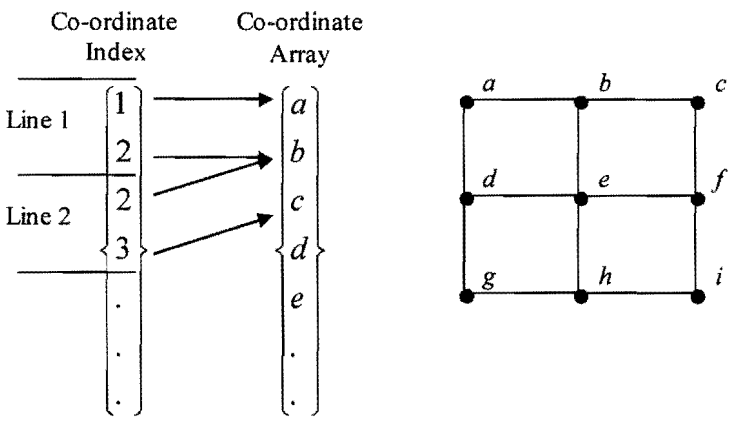


Figure 5.2 - A Co-ordinate Index for a Grid

In the case of small surfaces, the co-ordinate index can be manually created but as the surface becomes larger an algorithm is needed to index the co-ordinates. In the case of the Laetoli footprints there are thousands of co-ordinates in the DTMs that need to be indexed. Figure 5.3 shows the flowchart used to create the co-ordinate index.

The first step of the flowchart consists of creating three arrays containing the x, y and z values of the co-ordinates. In the second step the number of columns and rows are calculated. Since the co-

ordinates are in the form described in Table 5.1, where the x values remain constant while the y values are incremented by a constant value, it becomes easy to determine the number of columns. All the x co-ordinates with the same value are in the same row. A simple counter can be created to count all the x co-ordinates of the same value. Once the number of columns has been calculated the number of rows can be calculated by dividing the number of co-ordinates in the DTM by the number of columns.

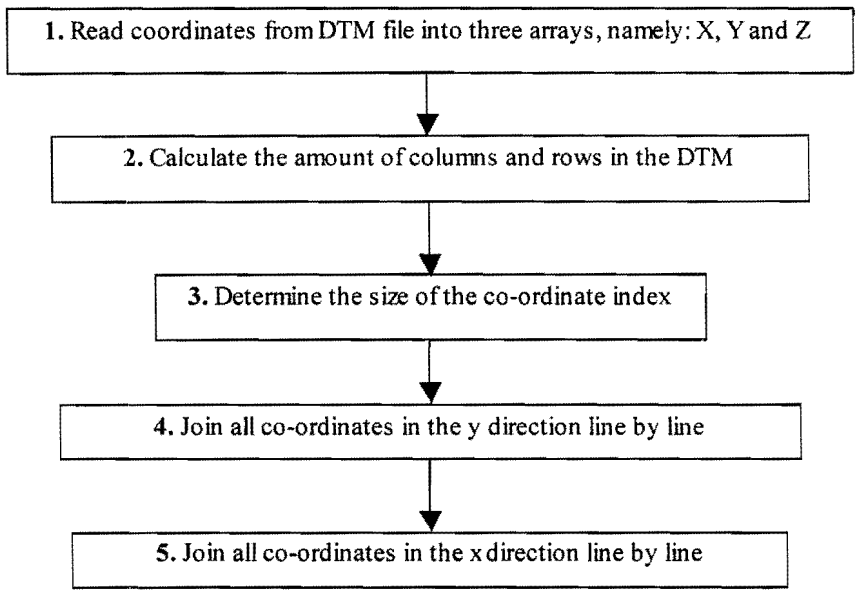


Figure 5.3 – Steps to Creating a Co-ordinate Index

The next step is to determine the size of the co-ordinate index. The following equation was used:

$$Size = 2 * [(R-1) * C + (C-1) * R] \qquad \textbf{Equation 5.1}$$

Where:

Size is the size of the co-ordinate index
C is the number of columns
R is the number of rows

Determining values of the co-ordinate index is done in two parts. Firstly, the indices joining the DTM co-ordinates in the y direction are calculated (fourth step) followed by the indices in the x direction (fifth step). Calculation of the indices in the y direction, which correspond to the vertical lines in Figure 5.5, is simple as the first co-ordinate in the DTM is connected to the second and so on. At the end of the row the x co-ordinate changes and the indices required to draw the line are not be added to the co-ordinate index. Instead, the point at the end of the row is skipped and the next row is started. The algorithm for this can be seen in step 4 of Figure 5.4.

Calculating the indices in the x direction, which correspond to the horizontal lines in Figure 5.5, is more complicated as the first point in the DTM has to be connected to the first point in the next

row. In order to determine the index position in the DTM of the first point in the next row the number of columns is added to the index of the first point. For example, the index for the first line of a DTM with 100 columns would be [0, 100] (Note that the index of the first point is 0 and not 1). The next lines index would be [100, 200]. An index value for the last point in the row and the first point in the next row is not required and is left out. The algorithm for this can be seen in step 5 of Figure 5.4.

Step 4:

i = 0

For n = 0 to n < 2R*(C-1) in steps of 2

    If $x_i = x_{i+1}$ then i = i+1

    I[n] = i

    I[n+1] = i

    i = i + 1

Step 5:

j = 0, m = 0, q = 0

For m = n to m <n + 2C*(R-1) in step of 2

    If j = R-1 then j = 0 and q = q + 1

    I[m] = j * C + q

    I[m+1] = j * C + C + q

    j = j + 1

Where:

I[] is the co-ordinate index array

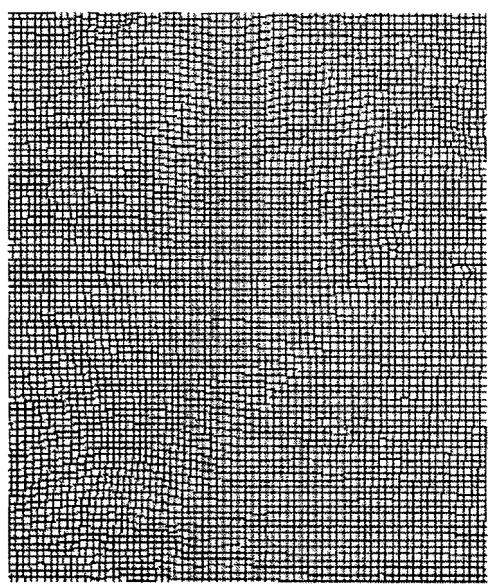
x is the x component of the co-ordinates

i, j, m, n and q are variable integer values

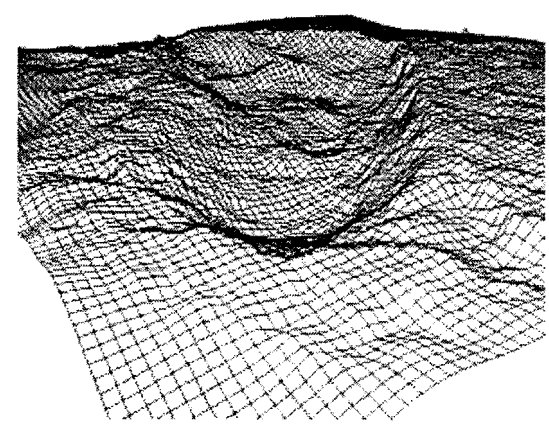
C is the number of columns

R is the number of rows

Figure 5.4 – Algorithms for vertical and horizontal lines



Top View



Oblique View

Figure 5.5 – Surface Rendered as a Grid

As in the case of describing the model by points, the lines of this model do not have reflective attributes and require an ambient lighting quality. The grid model is useful for large DTMs (ten thousand points or more) as it renders at acceptable speeds on lower end computers and the shape of the surface may be easily recognised. (Figure 5.5)

5.1.5 Filled Polygon and Gouraud Shaded Rendering

The third and fourth methods of rendering a surface (described in section 5.1.1) are similar as they are constructed in the same way and use the same co-ordinate index. The co-ordinate index used in these methods is slightly different from the co-ordinate index used in the previous method involving grids. In order to create a surface, a series of planes need to be created and joined together. By indexing three points together, triangular planes are created (see Figure 5.6). The direction in which the triangular planes are indexed is important. Graphics systems often only draw the one side of a surface to increase rendering speeds. The back of the surface will therefore be invisible. In Java3D, all the triangles must be indexed in an anti-clockwise direction to ensure that the whole surface will be visible from the same direction. If the back face of the surface were required to be visible instead of the front face, the triangular planes would have to be indexed in a clockwise direction.

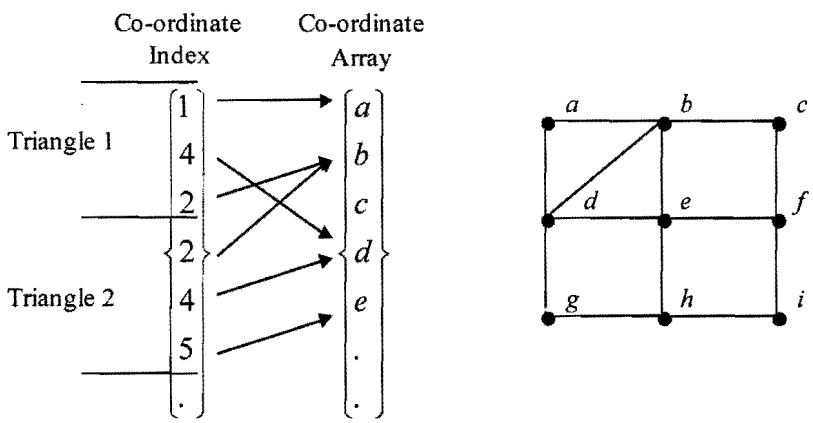


Figure 5.6 - A Co-ordinate Index for a Triangle

The first difference in the creation of a solid surface, as opposed to a grid surface, is the number of indices in the co-ordinate index. The following equation is used to calculate the size of the co-ordinate index:

$$Size = 6 * (R-1)(C-1)$$ **Equation 5.2**

Where:

Size is the size of the co-ordinate index
C is the number of columns
R is the number of rows

The algorithm for the creation of the triangles can be seen in Figure 5.7. In the indexing of the triangles, it is easier to index two triangles at a time for every loop in the algorithm. For the first

triangle the first two vertices are in the same row. The third vertex is in the next row and is referenced by adding the number of columns to the index value of the second vertex. The second triangle is created in a similar fashion except that the second and third vertices are in the next row. When the last point in the current row is reached it is ignored and the next row is started.

```
k = 0
For i = 0 to i < R
        For j = 0 to j < C
                T[k] = (i/R, j/C)
                k = k+1
```

Where:

T[] is the array of texture co-ordinates

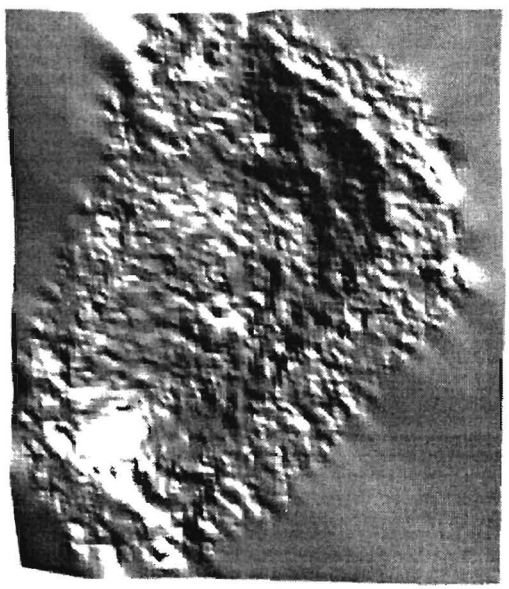
i, j, k are variable integer values

C is the number of columns

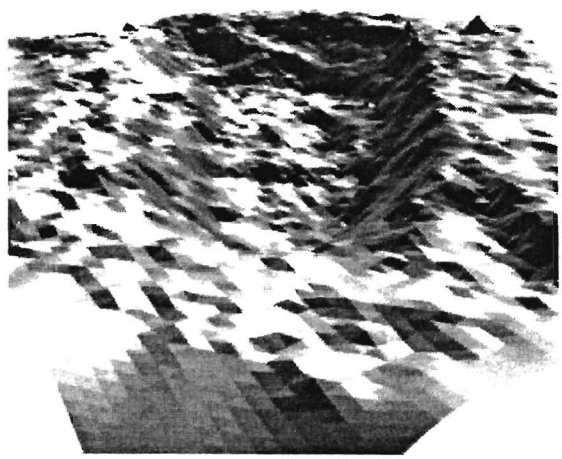
R is the number of rows

Figure 5.7 – Algorithm for Triangles

The filled polygon rendering consists of all the triangles rendered together to form a surface. This surface is very faceted due to creases between the triangles forming the 3D surface (see Figure 5.8). In order to get a more realistic appearance these creases need to be smoothed.



Top View



Oblique View

Figure 5.8 – Polygon Filled Rendered Surface

By specifying a crease angle in Java3D, a smooth surface can be achieved. Each triangular plane has a normal vector associated with it. This normal vector is used in lighting calculations to determine the direction of reflected light. The crease angle is the angle between the normal vectors of adjacent planes. If the angle between the normal vectors of any two adjacent planes is less than

the specified crease angle, the crease between the two polygons will be 'smooth shaded'. Figure 5.9 demonstrates how the crease angle is measured. In the figure, it is assumed that the crease angle has been set to 45 degrees. In the left figure, the angle between the normals is less than the stipulated crease angle value and the crease will be 'smooth shaded'. In the right hand figure, the angle between the normals is greater than 45 degrees. The edges between the two triangles will not be smooth shaded and will appear faceted.

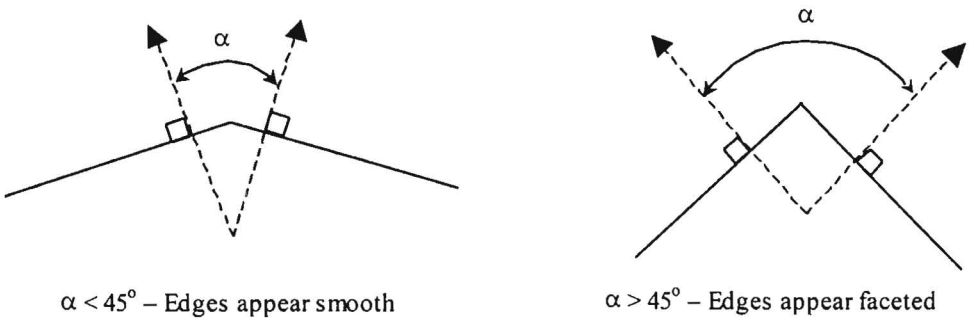


Figure 5.9 - Filled Polygon Rendering and Gouraud Shaded Rendering

In Java3D the Gouraud shading technique is used to 'smooth shade' the creases by eliminating intensity discontinuities. This shading process consists of four steps (Foley and van Dam, 1982). Firstly, the surface normals need to be calculated. The second step is to calculate vertex normals by averaging the surface normals of all triangles that share the vertex (see Figure 5.10). Thirdly, the vertex intensities are calculated by using the vertex normals and a shading model. A shading model is used to specify how the vertex intensities should be calculated. In the fourth step, each polygon is shaded by linear interpolation of vertex intensities along each edge and then between edges along each scan line (see Figure 5.11). The Java3D API takes care of the shading and the result can be seen in Figure 5.12. This form of rendering provides most realistic reconstruction of the footprint surface. It is also the slowest method, as the computer must perform many calculations.

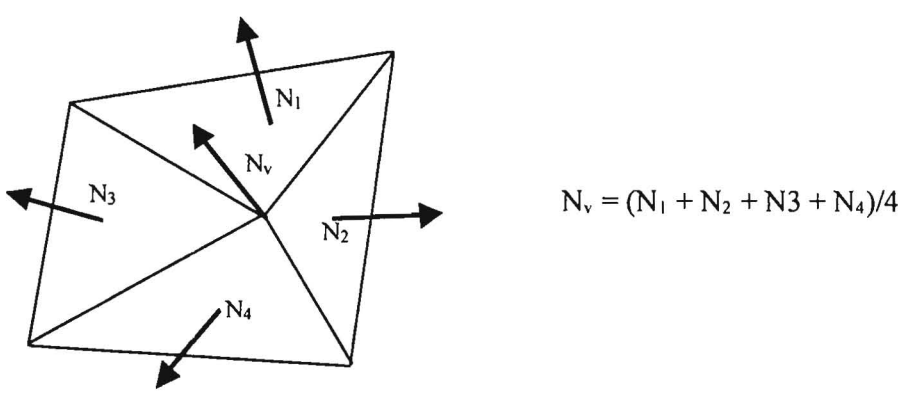
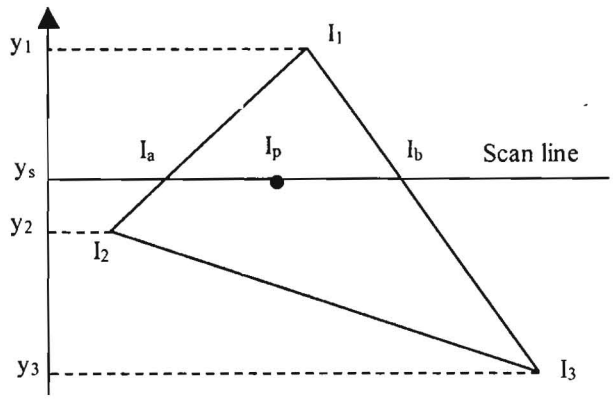


Figure 5.10 – Calculation of Vertex Normals



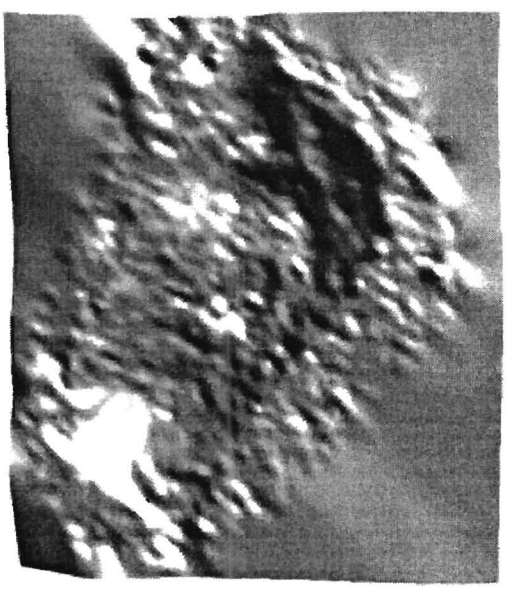
$I_a = I_1(y_s-y_2)/(y_1-y_2) + I_2(y_1-y_s)/(y_1-y_2)$

$I_b = I_1(y_s-y_3)/(y_1-y_3) + I_3(y_1-y_s)/(y_1-y_3)$

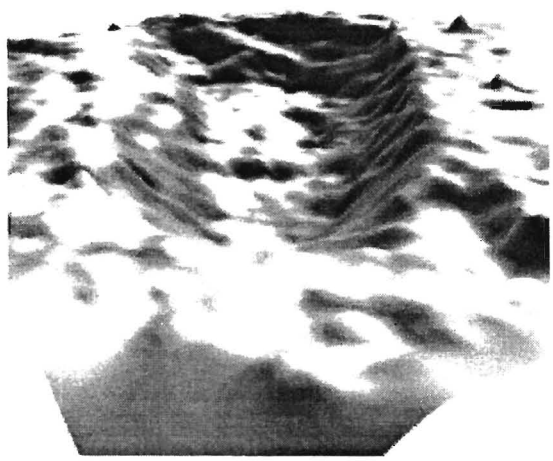
$I_p = I_a(x_b-x_p)/(x_b-x_a) + I_b(x_p-x_a)/(x_b-x_a)$

Where I is the intensity value.
I_a is interpolated from I_1 and I_2, I_b from I_1 and I_3 and I_p from I_a and I_b

Figure 5.11 - Intensity Interpolation



Top View



Oblique View

Figure 5.12 - Gouraud Rendered Surface

5.1.6 Example of a Co-ordinate Index for a Surface

A simple example of creating a co-ordinate index for a surface will be looked at here and will be revisited when necessary. For this example, a square surface created by two triangles will be considered (see Figure 5.13).

For simplicity, the heights of the four co-ordinates are set as zero. Due to the fact that the surface is created by means of triangles, the co-ordinate index size must be a factor of three. It can be seen in the example that the size of the co-ordinate index is six and therefore fulfils this requirement. The

first three values in the co-ordinate index refer to the first triangle and the next three refer to the second triangle. Before continuing it is important to note that in Java the first element in an array is referenced as zero and the second element as one and so on. Thus, the first triangle is created by joining the second, first and third co-ordinates found in the array of co-ordinates. The second triangle is created by joining the second, third and fourth co-ordinates in the array of co-ordinates. Another point to note is that both the triangles are referenced in an anti-clockwise manner, which will mean that both the triangular surfaces will be visible from the same direction. Once the co-ordinate index has been completed, a function in Java3D that requires both the array of co-ordinates and the co-ordinate index is used to render the surface.

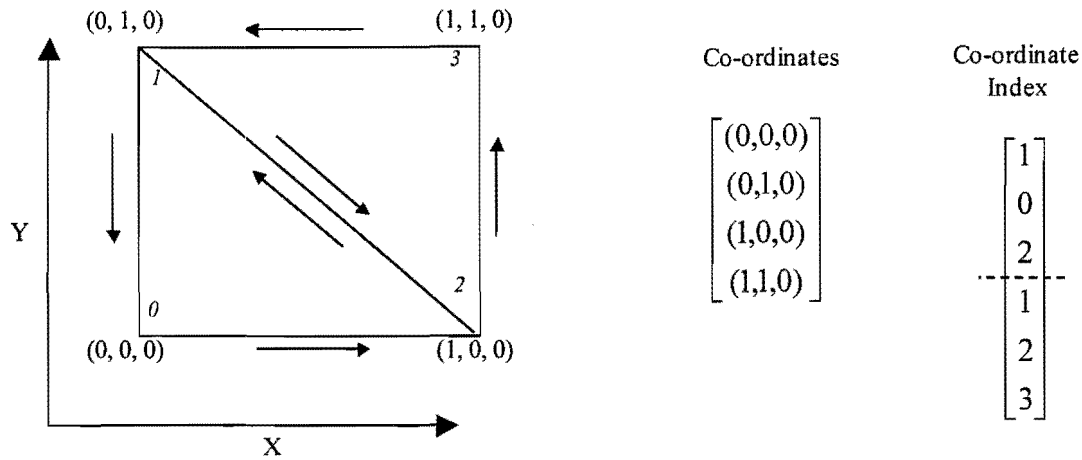


Figure 5.13 – Example of a Surface Created by Triangles

5.2 The Viewpoint and Movement Tools

The viewpoint is one of the most important features of 3D graphics. The amount of visual information that may be extracted from a visualisation depends on a users view of the object. Changing direction, position, height and angle of the viewpoint alters the view of an object. The various angles that the 3D object is viewed at give different impressions of the surface. Oblique vertical angles provide a user with strong impressions of the relief of the surface, although horizontal distances become difficult to gauge. Certain areas also become hidden or obscured. Overhead views provide the viewer with a better perception of horizontal distances, but heights become more difficult to judge. In order to visualise 3D objects displayed on a computer screen the user must be allowed to view the objects from various positions. (Stynes, 1996)

Movement tools were created to enable a user to alter their position in the virtual universe in order to see the 3D objects from various perspectives. Either the viewpoint or the 3D object can be

moved to alter the viewer's perspective of the scene. Moving the viewpoint is the preferred option in many cases since objects rendered in the same scene and not visible in a particular field of view will not need to be rendered. This will lead to fewer calculations performed and rendering will be quicker. Moving all the objects simultaneously will result in the co-ordinates of every object having to be recalculated even if they fall outside the field of view of the viewpoint. Referencing the co-ordinates of objects in the virtual universe is also easier if their positions remain constant.

In this project the initial viewpoint is positioned above the 3D surface facing downwards. Thus, the maximum x and y values of the DTM are positioned on the right and top of the screen respectively. The height of the viewpoint above the surface of the footprint has to be calculated to allow the whole footprint to be visible in the viewpoint's field of view. Since the surface areas of the individual footprints are of different sizes, a unique value has to be calculated for each footprint. The default field of view angle in Java3D is 45°. Due to the structure of the DTM, the maximum extent of the surface will be in either the x or the y direction. The maximum extent of the surface in the horizontal direction will be the larger of the two values determined by the following differences: $x_{max} - x_{min}$ and $y_{max} - y_{min}$. Equation 5.3 is used to calculate the height of the viewpoint above the surface.

$$height = \frac{S_{max} - S_{min}}{2 * \tan \frac{45^{\circ}}{2}} \qquad \textbf{Equation 5.3}$$

Where:

$S_{max} - S_{min}$ is the maximum extent of the surface in the horizontal plane.

Three tools were created for the movement of the object, namely:

- Rotate tool – Rotating the viewpoint causes it to move along the surface of an imaginary sphere. Initially the centre of the sphere coincides with the centre of the footprint. The viewpoint always points towards the centre of the sphere. The initial radius of the sphere is the height of the viewpoint above the centre of the footprint calculated in Equation 5.3.
- Zoom tool – The zoom tool will translate the viewpoint along the z-axis of the virtual world co-ordinate system, thus changing the radius of the imaginary sphere.
- Pan tool – The pan tool translates the viewpoint in the horizontal plane. Translation of the viewpoint will affect the rotation of the footprint by translating the imaginary sphere.

The creation of these tools was an intricate process and required knowledge of 3D transformations. Java3D provides utility tools for rotating, translating and zooming of objects. These tools did not perform the movement of the viewpoint in a manner that suited the specific case. For example, the rotation tool provided translated the viewpoint to the origin, rotated it by the desired amount and

then translated it back to its original position. This caused the surface to become hidden, as it did not lie within the viewpoint's field of view. Custom tools had to be written to avoid this problem.

In order to move the viewpoint the user must click and drag their mouse on the window containing the 3D object. The x and y components of the distance are calculated from the start and end points of the mouse drag. The distance and direction that the mouse is dragged will determine the movement of the viewpoint. The x and y values measured are modified slightly for use in the transformation equations (mentioned below) as the unmodified measurements result in movements that are either too large or too small.

In 3D graphics, 4x4 matrices represent 3D transformations. 4x4 matrices are used instead of 3x3 matrices to enable a consistent means of representing 3D transformations. It also becomes easier to build up combinations of transformations using 4x4 matrices. The 3D point (x, y, z) is represented in homogenous co-ordinates as (W.x, W.y, W.z, W) where W is not equal to zero. If W is not equal to one then W is divided into the first three homogenous co-ordinates to obtain the 3D Cartesian co-ordinate point. Translations in 3D are a simple extension of 2D translations (see Equation 5.4). The *Dx*, *Dy* and *Dz* values in the matrix represent translations in the x, y and z directions respectively. Modified x and y values obtained from a mouse drag when the pan tool is active will be substituted into the *Dx* and *Dy* values of Equation 5.4 respectively. In the case of the zoom tool, only the modified y value is required and it will be substituted into the *Dz* value.

$$[x \quad y \quad z \quad 1] \cdot T(Dx, Dy, Dz) = [x + Dx \quad y + Dy \quad z + Dz \quad 1] = [\hat{x} \quad \hat{y} \quad \hat{z} \quad 1]$$

Equation 5.4

Where:

$$T(Dx, Dy, Dz) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Dx & Dy & Dz & 1 \end{bmatrix}$$

Is the translation matrix

and

$(\hat{x} \quad \hat{y} \quad \hat{z})$ Are the co-ordinates of the new point

The rotations are slightly more complicated. For a rotation of a point about the x-axis the modified y value will be substituted into θ of $Rx(\theta)$ in Equation 5.5. A rotation of a point about the y-axis is done by substituting the modified x value into θ of $Rx(\theta)$ in Equation 5.5. Rotations about the z-axis are not required for the movement tool and are not calculated.

$$[x \quad y \quad z \quad 1] \cdot R(\theta) = [x' \quad y' \quad z' \quad 1]$$ **Equation 5.5**

Where:

$$R(\theta) = Rx(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ For rotations about the x-axis

$$R(\theta) = Ry(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ For rotations about the y-axis

and

$(x' \quad y' \quad z')$ are the co-ordinates of the rotated point

Since not all computers have the same screen resolutions, certain settings to control the sensitivity of the mouse are provided. If a user with an 800x600 screen resolution drags the mouse from the top of the screen to the bottom of the screen the measured y value would be 600. Performing the same mouse drag on a screen with a resolution of 1024x768 would result in a y measurement of 768 (see Figure 5.14). The same physical measurement at the two different screen resolutions results in different measured values. Higher screen resolutions result in movements of the mouse being more sensitive and the movement tools are difficult to control.

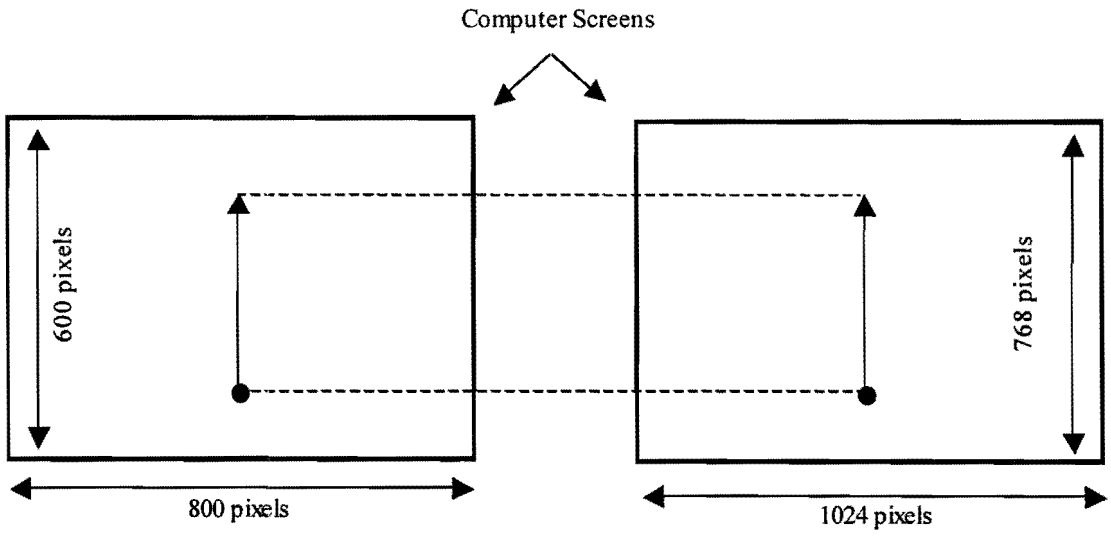


Figure 5.14 – Screen Resolutions

In the visualisation program for the Laetoli footprints, three settings are available to allow the user to alter the sensitivity of the mouse namely, 'tortoise', 'hare' and 'cheetah'. The tortoise setting is the default setting and it multiplies the measured x and y values by a factor of 0.01. The hare and cheetah settings multiply the measured x and y values by 0.1 and 1.0 respectively. Since the rotation tool accepts an angle in radians, multiplying the measured x and y values by the various movement speed factors cause excessive rotation of the viewpoint. Multiplying the measured x and y values by 0.1 and 1.0 is only useful for the pan and zoom tools.

Custom viewpoint positions are available to allow the user to view the footprints from various positions without having to be familiar with the movement tools. Five viewpoint positions are available namely 'top', 'north', 'south', 'east' and 'west'. The viewpoint for the top view is the original viewpoint and has been discussed previously. A command was added that allows a user to return to that position if they have become 'lost' or disorientated after using the movement tools. North, south, east and west correspond to directions along the y, -y, x, and -x axes of the virtual world respectively.

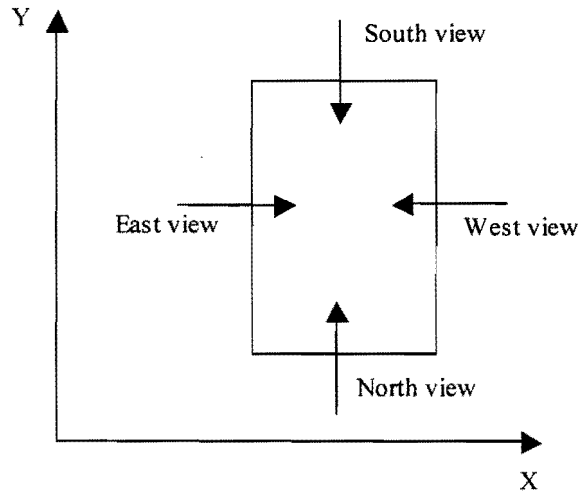
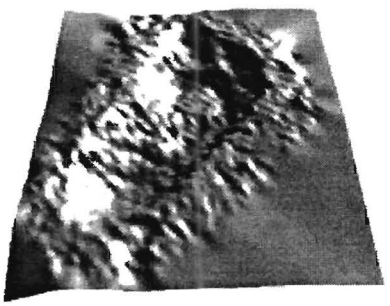


Figure 5.15 – Directions of Pre-defined Views

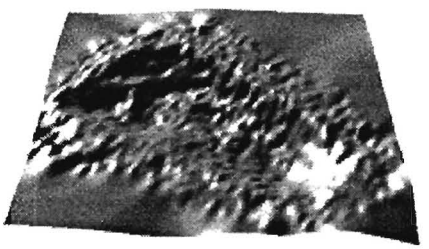
The north viewpoint position is placed south of the footprint and looks north. In order to do this the viewpoint has been rotated by 45° about the x-axis giving an oblique view of the surface. The rotation is calculated in the same way as rotations for the rotation tool. The south, east and west viewpoint positions require two rotations namely, a rotation about the z-axis and a rotation about either the x-axis or y-axis (see Figure 5.16).

As a user moves further away from an object in the real world, less visual detail can be perceived. The same concept applies in 3D graphics. In some cases, a technique that varies the amount of detail in a visual object based on its distance away from the viewpoint is required. The general term

for this technique is Level of Detail (LOD). Reducing the complexity of the visual object as it is moved further away will not affect the visual result depending on the distance away from the viewpoint Decreasing the amount of detail will affect the amount of rendering computation required. If it is done well, significant computational savings can be made without visual loss of content. (Hartman and Wernecke, 1996)



North View



East View

Figure 5.16 – Customised Views

If an LOD function were required for a 3D object, two or more objects would have to be created. The first object would be the normal object that contained all the DTM data. The second object would be created from fewer points of the same DTM. This would result in fewer triangles needing to be rendered. A simple method of reducing the complexity of the object would be to remove points from the DTM. Random removal of points from the DTM would result in new indexing methods being required to create the triangles, and information such as break lines and edges would be lost. A simple method to remove points in a systematic way would be to remove every second row and column. This would cause the DTM to be four times smaller and fewer triangles would be necessary to create the surface of the object. After the two objects have been created, they can be used by the LOD function in the following way. As the viewpoint is moved further away from the object of greater detail a point is reached where its visual appearance will be the same as that of the object of lower detail. At that point a switch is activated and the second object replaces the first object.

In the case of the Laetoli footprints, it was deemed unnecessary to add LOD functions. In the case where more than one footprint is displayed at a time, LOD functions would be useful. They would enable the user to view all the footprints in their proper positions without having to render all the DTM data.

So far, only the 3D positions of the viewpoint have been discussed. Other variables may also be altered to change the way 3D objects in the virtual universe are displayed on a computer screen. These variables include changing the projection (discussed in section 5.3) and changing the width of the viewing angle (know, in Java3D, as the field of view).

The field of view (FOV) of the viewpoint may be altered to fall between zero and 180 degrees. The default FOV in Java3D is 45°. As the angle is increased or decreased, it appears as if the object moves further away or closer to the viewpoint respectively. This is not true, as the geometrical positions of both the viewpoint and the object remain the same. In Figure 5.17 the different areas covered by the various FOV angles can be seen. If the angle is 90° the whole area covered will be displayed on the screen making the surface of the object appear smaller. Conversely, only the area covered by the viewpoint with an FOV of 15° will be displayed on the screen making the surface of the object appear closer.

Figure 5.18 shows a footprint that has been viewed using two different FOV settings from the 'north' viewpoint position. The first setting of 15° required that the viewpoint be moved further away from the footprint to allow it to be displayed on the screen. In the image of the footprint with the FOV setting of 90° the viewpoint had to be moved closer to the surface. By examining the differences between the two images, it can be seen that the convergence of parallel lines in the second image is much more rapid than in the first. This is due to the fact that the smaller the FOV angle, the more the view tends towards an orthographic view.

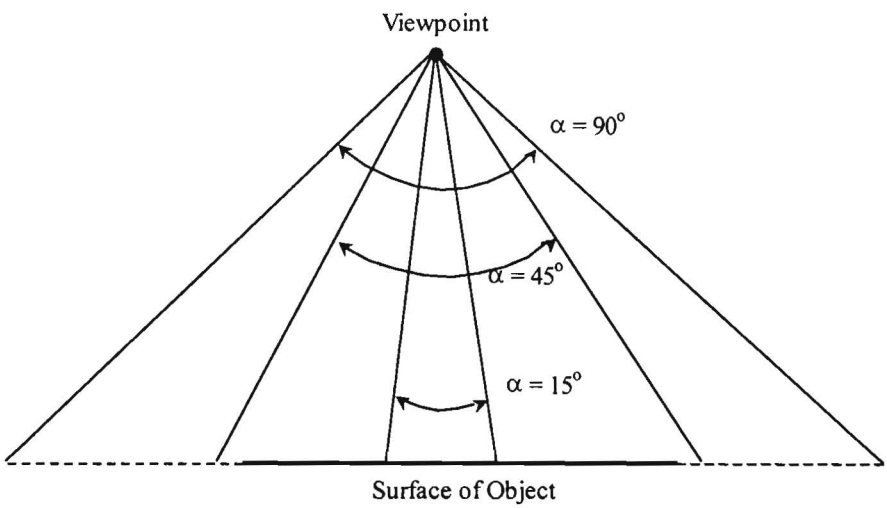


Figure 5.17 – Area of Surface seen with different FOV Angles

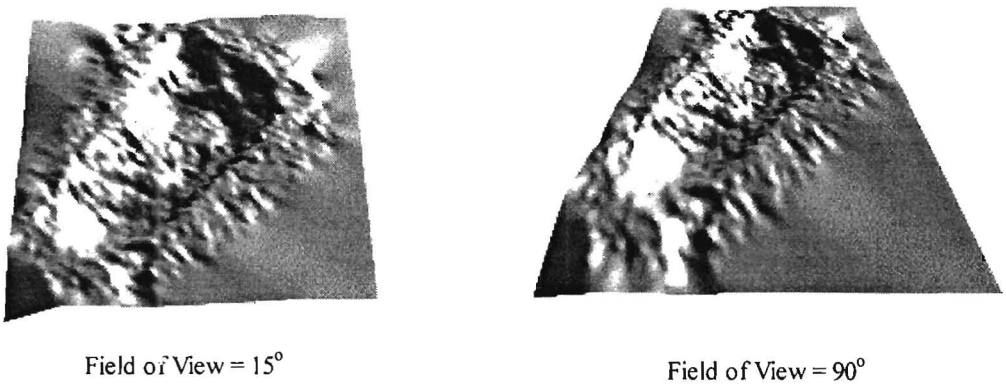


Figure 5.18 – Surface Viewed with different FOV Angles and Object Distances

5.3 Projections

Projections are needed to convert a three dimensional view of the world to a two dimensional image on a computer screen. A projection is a mathematical function for converting a system in n-dimensional space into a (n-1)-dimensional space (where n is an integer value). Two types of projections used commonly in computer graphics are *perspective* and *parallel* projections. The distinction between the two projections is the relation of the centre of projection to the projection plane. If the distance from the centre of projection to the projection plane is finite, then the projection is a perspective projection. A parallel projection results from an infinite distance between the centre of projection and the projection plane. Figure 5.19 demonstrates the difference between the two projections. (Foley and van Dam, 1982)

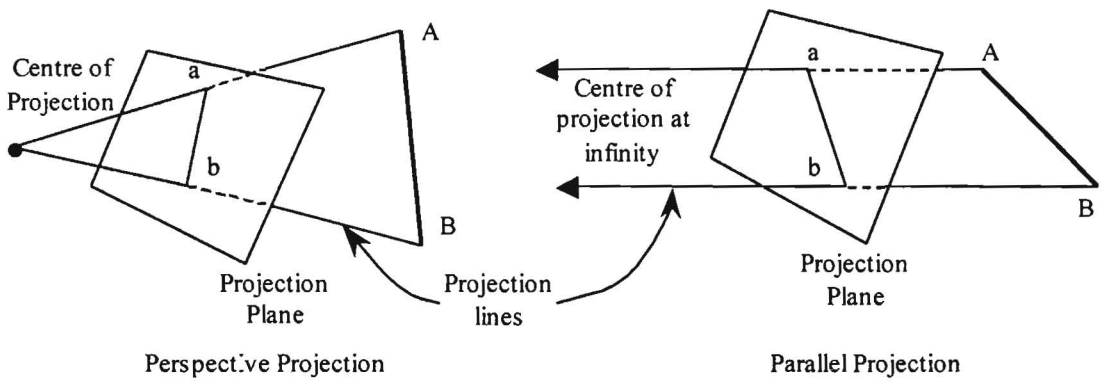


Figure 5.19 – Perspective and Parallel Projections

A parallel projection is a projection where all projection lines are parallel. These projections require fewer calculations than the perspective projections and therefore rendering speeds are increased. The disadvantage of this projection is that there is a lack of perspective and depth

realism. A parallel projection can be either orthographic or oblique (see Figure 5.20). An orthographic parallel projection's projection lines are all perpendicular to the projection plane. In the case of the footprints, the projection plane is in the horizontal plane and each point creating the surface is projected vertically upwards onto the screen. Thus, the size of the footprint on the screen is its actual size unless it has been previously scaled. In an oblique parallel projection, the projection lines are parallel to each other but the angle between them and the projection plane is not perpendicular.

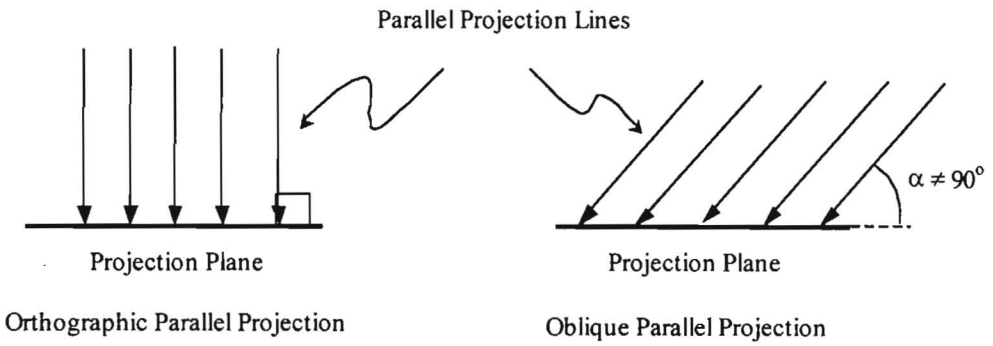


Figure 5.20 – Orthographic and Oblique Parallel Projections

The perspective projection is a projection where all projection lines originate from a point called the perspective centre. This system creates the same visual effect as those of photographic systems and the human visual system. It is used when a degree of realism is required though it is not particularly useful for displaying the exact shape and dimensions of the object. In other words, distances and angles are only preserved on faces that are parallel to the projection plane. Another point to note is that, in general, as with the human visual system, parallel lines are not projected as such.

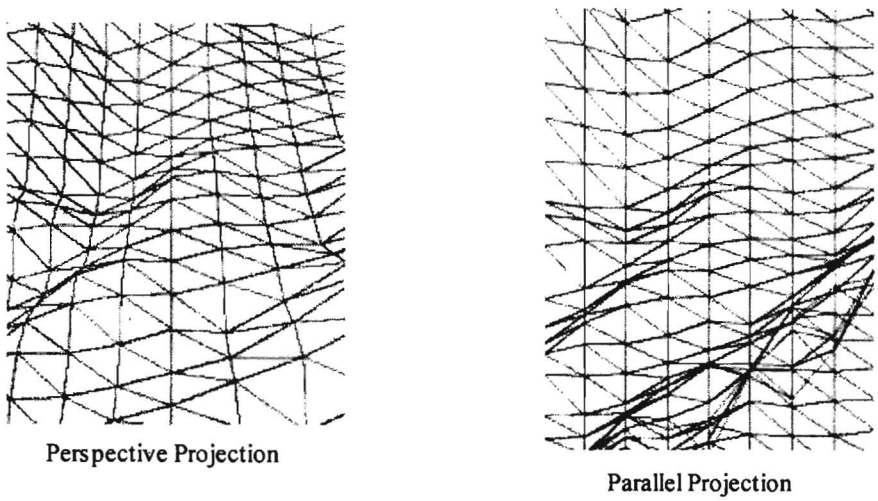


Figure 5.21 - Oblique Perspective and Parallel Projections

In Figure 5.21 oblique perspective and parallel projections of a section of a footprint are shown. It can be seen in the perspective projection that the parallel lines appear to be converging on a single point, whereas the corresponding lines in the parallel projection remain parallel. When changing from the perspective projection to the parallel projection a scale factor has to be applied to fit the whole of the footprint on the screen.

5.4 Light

The position of a light source can be crucial in forming the user's perception of a surface. The direction from which the light strikes a surface determines the different features that are highlighted. Inappropriate lighting of a surface may lead to misleading displays such as the inversion of relief. Java3D provides different types of light sources that can be used, namely: ambient lights, directional lights, point lights and spotlights. In the creation of the footprints, only the directional and point light sources are used.

The first light source discussed is a directional light source (see Figure 5.22). This light source resembles the light from the sun where the beams of light are considered parallel. The direction of the light can be set by means of a vector direction from the origin. In order to illuminate the surface of the footprint in a natural way a vector direction of (-1.0, -1.0, -1.0) is used for the direction of the light.

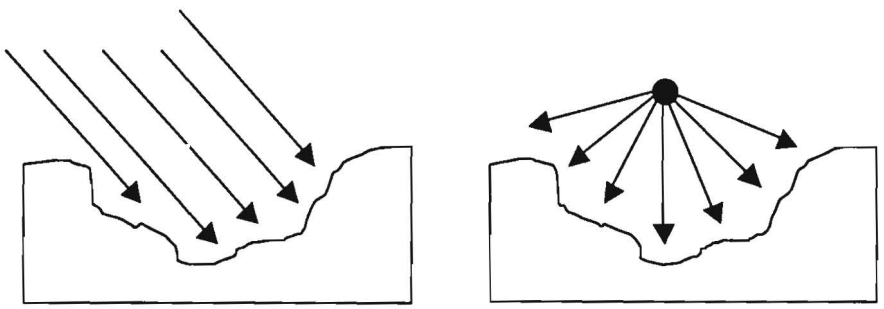


Figure 5.22 – Directional and Point Light Sources

As mentioned previously, the direction of the light is important as a surface may be interpreted incorrectly by the shadows formed. The direction of the directional light source is fixed and because each surface is different, it is necessary to have a moveable light source to highlight different areas of the surface. A point light is used as the moveable light source. Beams of light originating from this light source are omni-directional and attenuate with distance (see Figure 5.22).

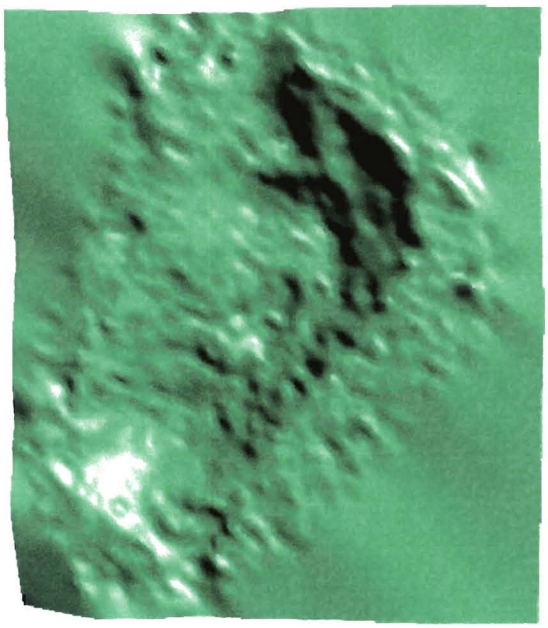


Plate 1 – A single colour applied to the 3D surface

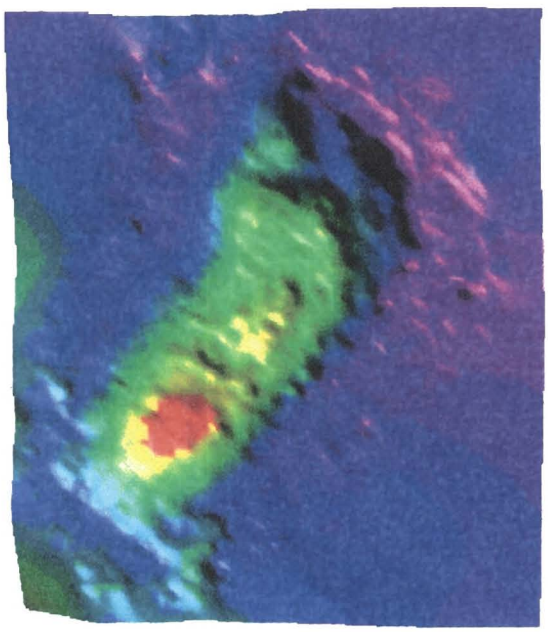


Plate 2 – Graduated colour applied to the 3D surface

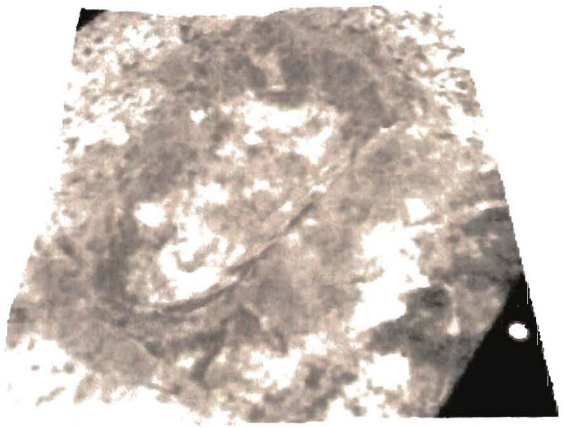


Plate 3 – Ortho-image overlaid onto the 3D surface

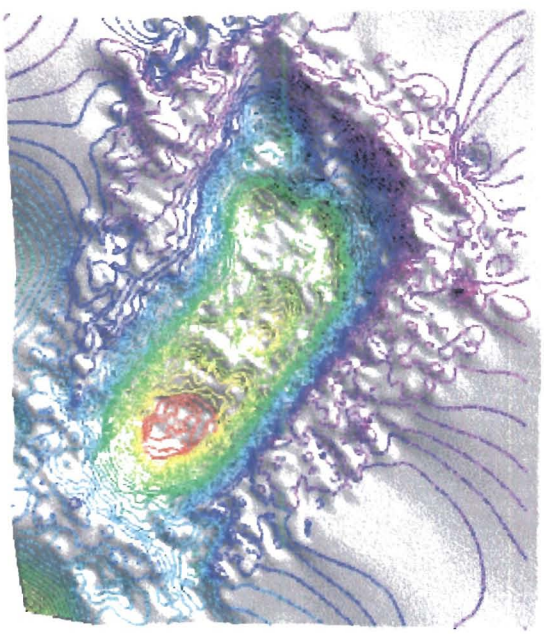


Plate 4 – 3D contours overlaid onto the 3D surface

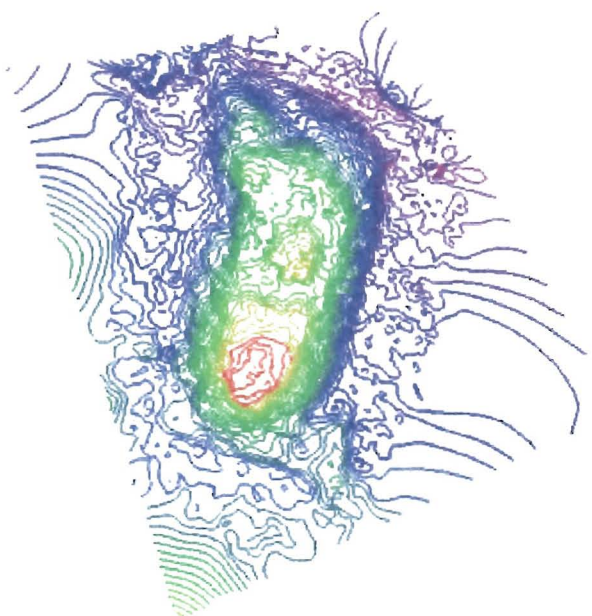


Plate 5 – 3D contours displayed without the 3D surface

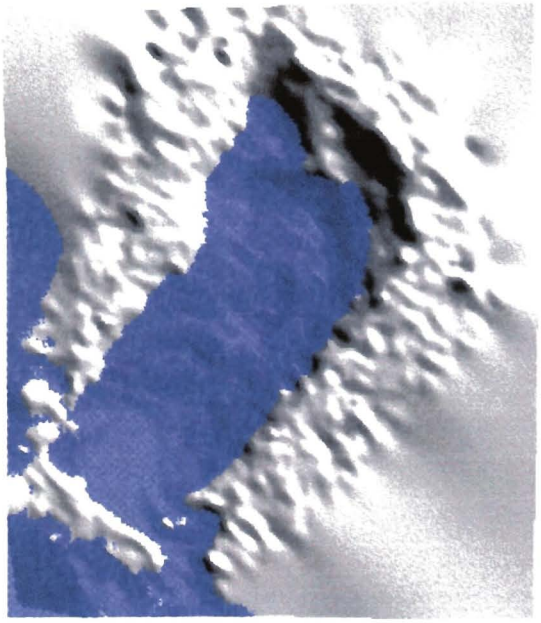


Plate 6 – Flood Plane raised through the 3D surface

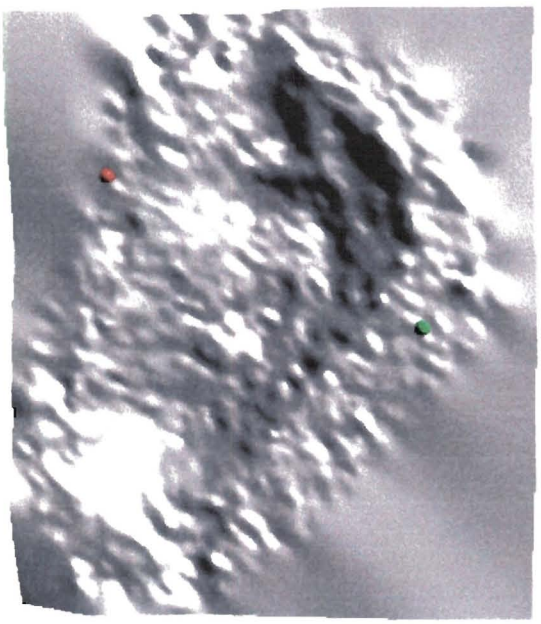


Plate 7 – Measurement spheres on 3D surface at points where measurements were made

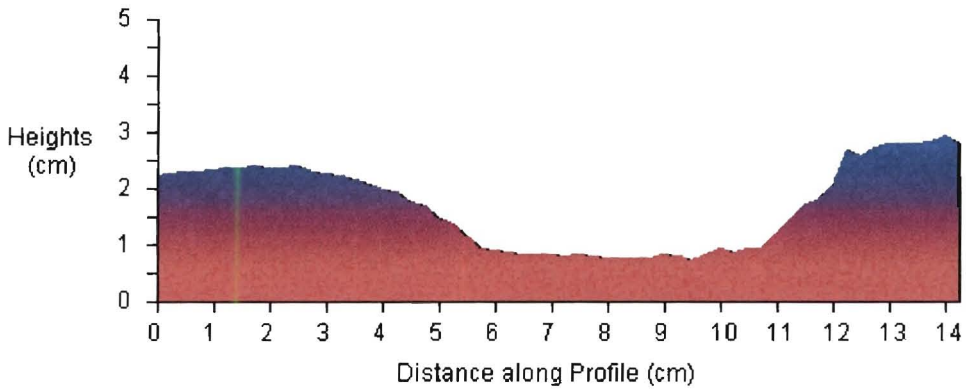


Plate 8 – Profile through footprint between measured points seen in *Plate 7*

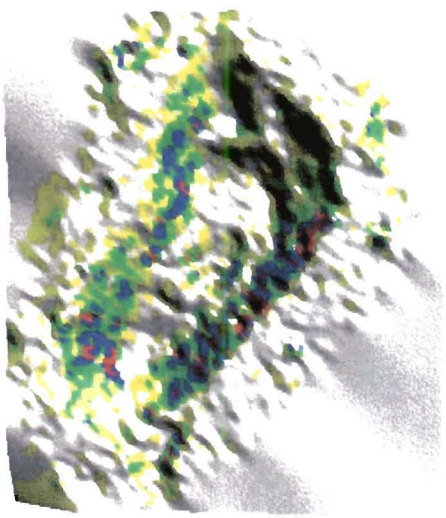


Plate 9 – Gradients calculated in the x direction and overlaid onto 3D surface

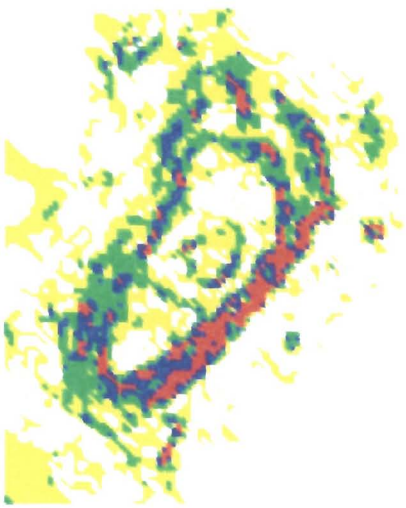
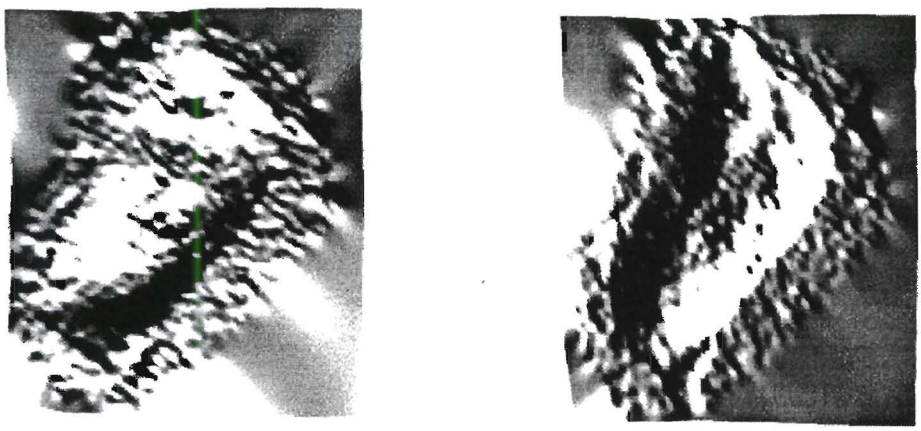


Plate 10 – 2D representation of gradient values seen in *Plate 6*

A switch is available that allows a user to active the point light source. When the point light source is activated, the directional light source is removed and vice-versa. The position of the point light source can be altered enabling a user to highlight different features on the surface. Moving the light source closer to features of interest emphasises the shape of the surface by greater contrast between areas of light and shade. As the light is moved further away, its intensity attenuated thus creating less contrast between areas of light and shade. Figure 5.23 shows a point light source shone onto the surface of a footprint from two different directions. The picture showing the light from the west can be easily misinterpreted if it is not known from which direction the light originated. The footprint may appear to be a bulge on the surface rather than a hole i.e. the inversion of relief.

Another factor that must be considered is that not all graphics cards render 3D graphics in exactly the same way. Some graphics cards cause the light to be either too bright or too dim. When too much or too little light is reflected off the surface, it becomes difficult to perceive the contrast between areas of light and shade. A tool is therefore available to change the intensity of the point light source.



Light from the South Light from the West

Figure 5.23 – Point Light Source from Different Directions

5.5 Colour

"Colour is an immensely complex subject that deals with both physics and physiology" (Foley and van Dam, 1982). The colour of an object depends not only on the object itself but also on the light source illuminating the object and the user's visual system.

In Java3D, there are three ways to specify the colours for a visual object, namely: specifying the colours of each vertex, using a function called '*Colouring Attributes*' and lastly by defining the material of the object. When more than one colour specification has been made, two simple rules

determine which colour specification takes preference. Firstly, material colours are only used when rendering objects lit by a light source. Secondly, the '*Colouring Attributes*' method is only used when rendering unlit objects. Specifying colours of the vertices takes precedence over the other two methods and will be discussed in this section. The material colour specification will be discussed in section 5.6.1.

There are various ways that colours may be defined. In Java3D, colours may be defined by either mixtures of hue, saturation and lightness or by mixtures of red, green and blue (RGB). The use of colour in 3D graphics is different to the use of colour in 2D graphics. On a 3D surface colours are affected by areas of light and shade. A user's perception of the colour of a surface may be affected if a grey scale colour scheme has been used in the representation of the surface. The grey scale colours of the surface will conflict with the grey scale colours of light and shade. In order to alleviate this problem, combinations of changes in hue and lightness need to be used. For example, a simple colour graduation from red to yellow to white would involve changes in hue (from red to yellow) and lightness (from yellow to white).

Variations in hues are useful as they often act as a depth cue. This can be seen with the use of colour schemes in maps displaying terrestrial data. When a user looks at a map, they see areas displayed using variations in hues and lightness. They will perceive areas that are green to be low altitudes, areas of brown to be higher altitudes and areas of white to be the tops of mountains. These perceptions come from the user's knowledge of landscape in the real world, where low-lying areas are normally greener than higher areas and mountain tops are covered in snow. These colours can be used similarly in 3D graphics where the appropriate hue can reinforce the relative relief of a surface. This becomes important when the surface is viewed from above, where perception of relief is entirely dependent on the illumination of the surface. (Stynes, 1996)

In order to specify the colour of the surface, every vertex of every triangle creating the surface of the object is given a colour value. As in the creation of the surface where a co-ordinate index is required, so too the application of colours requires a colour co-ordinate index. In Figure 5.24 an example is given on the creation of a colour index. In the example, the square formed by two triangles is coloured. The bottom left and top right corners have been coloured red and blue respectively and the other two corners have been coloured green. The first step required to set the colour of the object is to create an array of colours that the user wishes to use. In this case, an array of three colours is required, namely red, green and blue. The colours are defined in terms of the RGB colour scheme. The second step is to determine the colour of the first vertex. The first vertex is defined by the first element in the co-ordinate index and in this case it is vertex number two. From the figure, it can be seen that the second vertex is green thus the first element in the colour

index should point to the colour green in the array of colours. Remembering that the first element in an array in Java3D is referenced as zero, the first value of the colour index will be one, which refers to the second element of the array of colours. The rest of the colour index is created in the same way. In order to set the colours of all the vertices, the colour index must be the same size as the co-ordinate index. Another point to note is that all the values in the colour index are integer values.

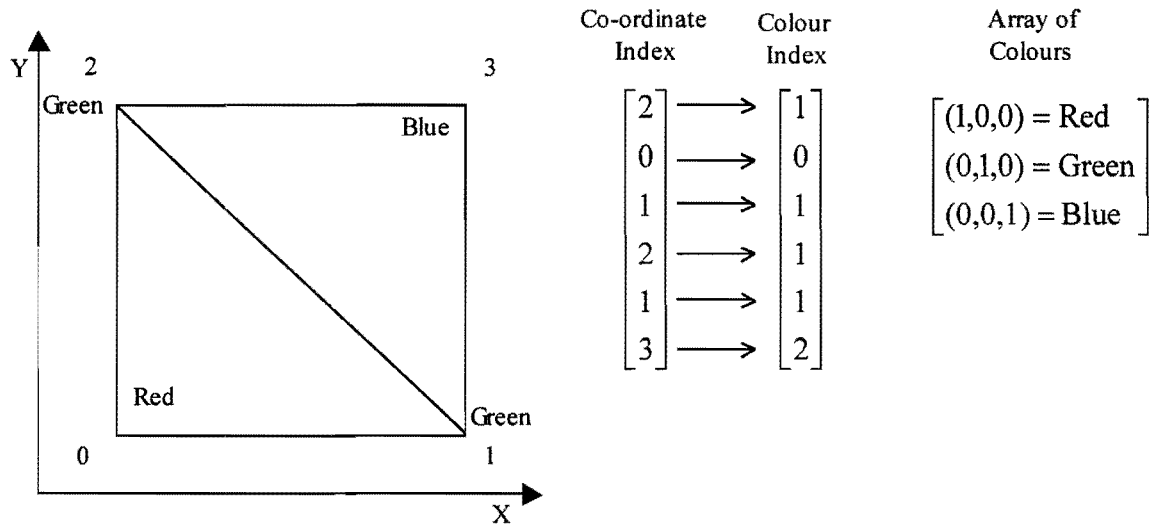


Figure 5.24 – Example on the Creation of a Colour Index

Once all the vertices have been set to their correct colours, Java3D uses the Gouraud shading algorithm to colour all the pixels of the triangles between the vertices. For example, the colour will be graduated smoothly along the edge of a triangle where one vertex is green and the other vertex is red. The colour interpolation will work in a similar fashion to the intensity interpolation described in section 5.1.5, using colour values instead of intensity values.

In some cases, sharp colour edges are required. Using the example in Figure 5.24 a red triangle and a green triangle can be created. This would be done by using the colour index [0, 0, 0, 1, 1, 1]. From this colour index it can be seen that two vertices in the same position can have different colours. One vertex belongs to one triangle and the other vertex belongs to another triangle. A Gouraud shading between vertices of the triangle would result in the whole triangle being the same colour. The bottom triangle in the example would be red and the top triangle would be green. A sharp edge between the two triangles will therefore be defined.

In the program created in this project the user is given the ability to select a colour. The footprint is then displayed using the selected colour. Setting the colour of the surface of the object to one

colour is simple as the array of colours consists of one value and the colour index will consist of one integer value, namely zero. ***Plate 1*** shows a footprint where the colour has been set as green.

Creating a graduated colour display of the footprints is slightly more complicated. It requires the colouring of all the vertices of the triangles according to their heights (see ***Plate 1***). The range between the highest and lowest vertices of the footprints is divided into fifteen equal sections. Each of the fifteen sections is assigned a colour, thus the array of colours consists of fifteen values. Figure 5.25 shows how vertices are assigned colours based on their heights. For example, point one falls in the yellow range and is thus assigned the colour yellow. Point two is assigned the colour green and so on. As mentioned before, the colours of the edges of the triangles between vertices are interpolated between the vertex colour values. Fifteen colours are used to make the change of colours smooth. If fewer colours are used, Gouraud colour interpolations will be between colours such as green and red, which will detract from the visual appearance of the object.

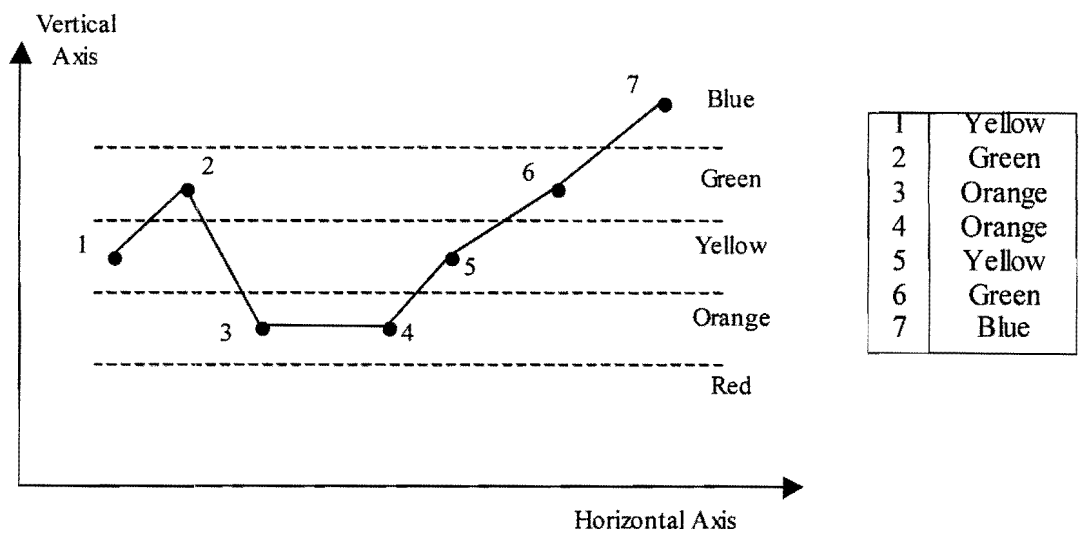


Figure 5.25 – Applying a Colour to a Vertex based on Height

The colours chosen for the graduated colour display of the footprints are based on the light spectrum where the lowest vertices are coloured red and the highest vertices are coloured violet. The lowest point on the surface defining the footprint is most often the heel and colouring it red highlights it (see ***Plate 2***). Other colour schemes would be applicable in other circumstances, for example, the 3D mapping of the terrestrial data. In this case, the standard colours used in atlases would probably be more useful and familiar to users than the colour scheme used for the footprints.

A drawback with the setting of the vertex colours is that once they are set they cannot be removed. Another drawback is that the material colour and 'Colouring Attributes' methods cannot display

the surface in multiple colours. The implication of these two drawbacks is that the colours of each vertex must be specified if multiple colours are required. This results in non-optimal use of memory storage space when the surface is set to a single colour, as the colour index is a large array consisting of a single value. In later versions of Java3D this problem will be solved, as there will be a function that will allow the colours of the vertices to be removed thus allowing the use of the other two colouring methods.

5.6 The Appearance

The appearance node in Java3D specifies the material, textures, transparency and structure of an object.

5.6.1 Material

The material object defines the appearance of an object under illumination. In other words, the material defines the reflective qualities of the surface of an object and these colours can only be seen when the surface is illuminated. It defines the ambient, emissive, specular and diffuse colours as well as the 'shininess' of an object. These qualities are described below (Bouvier, 1999):

- Ambient colour – The ambient colour of an object is seen under ambient lighting, which is a constant low-level light in a scene. For example, a light shone on to the surface of a sphere would only be visible on the one side. The dark side of the sphere will be lit by ambient light. The default ambient colour is dark grey, representing an object in shadow.
- Emissive colour – An object with an emissive colour will appear to glow with that colour. It is the colour that the object emits and it does not cast light onto other objects in the scene.
- Specular colour – Specular reflections occur naturally with smooth objects. In general, the smoother a surface is, the more intense the specular reflection. Specular colour defines the colour of the specular reflection on a 3D surface. The default colour of the specular reflection is white, though it can be set to make the light source appear to be a different colour.
- Diffuse colour – Diffuse reflection is the normal reflection of light from the surface of an object. The colour of an object can be set using this method rather than the method described in section 5.5.
- Shininess – The 'shininess' determines the intensity of the reflected light and is only used in calculating the specular reflections of an object.

The material can be set to give the object different appearances such as plastic, metal and stone. Tools are available that allow the user to change the material of the object. The result of changing the material of the surface is not as effective as expected. This is due to the colour of the surface of the footprint (discussed in section 5.5) overriding the colour of the material and it is not possible to remove a colour once it has been set.

5.6.2 Textures

Up to this point, the detail of visual objects has been provided by geometry. Some objects require large amounts of geometry to define their shapes. This requires large amounts of memory and rendering computations. Textures add the appearance of surface detail to visual objects without adding more geometry.

An example of this would be to render a carpet in a 3D model. To create an accurate geometrical model of the carpet each fibre would have to be modelled. The amount of computational power required for this would be unacceptable. A possible alternative would be to create a flat polygon consisting of many vertices. The vertices would then be assigned colours to give variations in colour over the surface. If the vertices are close enough together, a model of the carpet can be created. Although it would require less memory and calculations than the previous method, the requirements are still too great. Much simpler geometrical surfaces can be used in texture mapping. An image of a carpet mapped onto a flat rectangular surface will give the appearance of surface detail and will require minimal geometrical rendering.

In Java3D, functions are provided that map images onto geometrical surfaces. This is known as texture mapping. Mapping of textures on to a geometrical surface requires that the location of the texture (image) on the geometry be specified. Before this is done, the image must be prepared by setting the transparency, colours and size to the required values. The image must then be saved in an image format that Java3D can use such as the JPEG or GIF image file formats. The images used in the case of the footprints are ortho-images that were created in the photogrammetric process. This resulted in each DTM having a corresponding ortho-image of the same size. The contrast and brightness of the ortho-images were edited to be more visually appealing when mapped onto a 3D surface. Once the images have been edited, they are referenced in the appearance node, which is a node component of a leaf node (discussed in section 4.2). The position of the texture on the 3D surface is set in the geometry of the 3D object by means of specifying texture co-ordinates. The method of realising this will be explained below.

When mapping a texture to a surface the position of the texture on the surface must be specified. This is done using texture co-ordinates. A texture co-ordinate is a co-ordinate that specifies a point on the texture (a pixel on an image) to be mapped to a vertex on a 3D object. Texture co-ordinates are specified in the *s* (horizontal) and *t* (vertical) dimensions of the texture image space. Both the *s* and *t* co-ordinate values range between zero and one (see Figure 5.26). The texture co-ordinate, for example, of the centre of a texture (image) would be (0.5, 0.5).

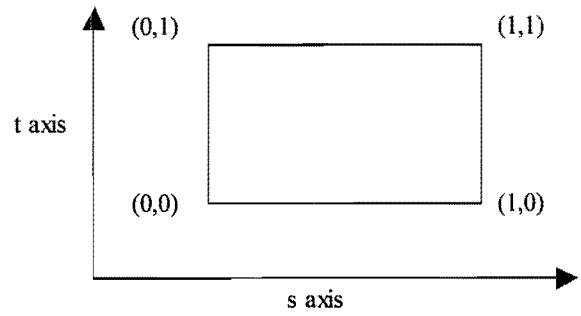


Figure 5.26 – Texture Map

Texture mapping is the process of mapping a texture onto a 3D surface by mapping texels to pixels. The pixels are the picture elements used to display the geometry of the 3D object. Texels (*Tex*ture *El*ement) are the base units of the texture in the same way as pixels are the base elements of an image. Texture mapping begins with the specification of texture co-ordinates for the vertices of the geometry. This means that for every vertex the texel value (colour value) corresponding to the texture co-ordinate associate with the vertex will be mapped to that vertex. The texture co-ordinates of pixels lying between vertices are determined by tri-linear interpolation of the vertices' texture co-ordinates. Once the texture co-ordinates of the pixels between the vertices have been calculated they are assigned the colour value of the corresponding texels. (Bouvier, 1999)

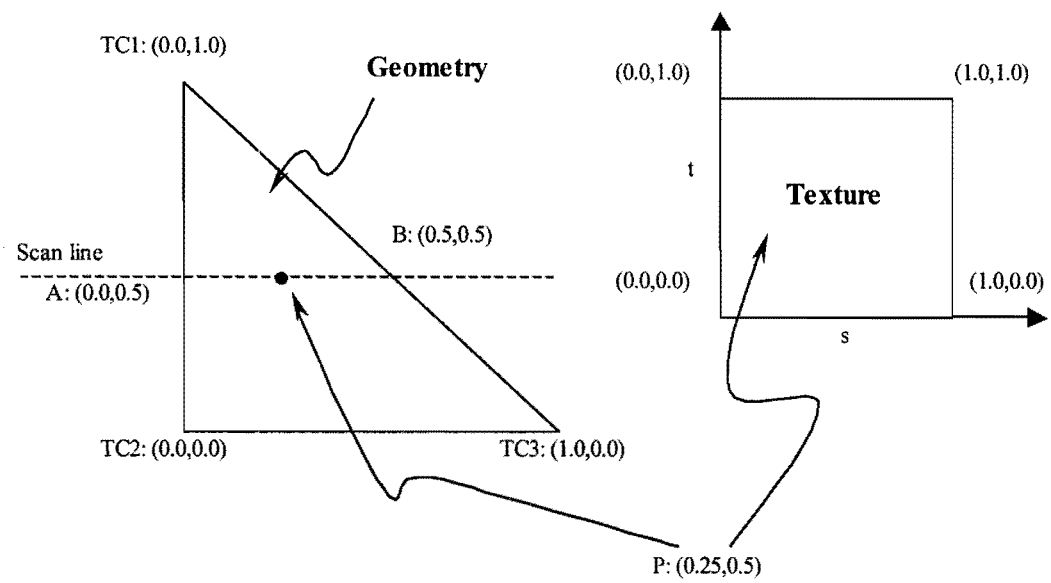


Figure 5.27 – Tri-linear Interpolation for an Example Pixel

Figure 5.27 demonstrates the process of tri-linear interpolation for an example pixel. When the object is rendered, the rendering is done in scan line order. The pixel P in the triangle (part of the geometry) is roughly in the centre of the current scan line. Texture co-ordinates have been assigned to each of the three vertices creating the triangle and are labelled as TC1, TC2 and TC3. The

texture co-ordinates are the starting points for the tri-linear interpolation. The first of two linear interpolations determine the texture co-ordinates along the edges of the triangle at the scan line. These two points are labelled A (interpolated between TC1 and TC2) and B (interpolated between TC1 and TC3) in the figure. The third interpolation is done along the scan line between these two points. The resulting texture co-ordinates for point P in this example would be (0.25, 0.5). Once the texture co-ordinate has been calculated for point P, the texel corresponding to the texture co-ordinate on the texture is selected. (Bouvier, 1999)

When the texture co-ordinates are calculated for each pixel, rarely does one texel map directly to one pixel. Pixels may be either smaller or larger than texels. If the pixels are larger than the texels, a minification filter is used to map the multiple texels to a single pixel. Conversely, if the pixels are smaller than the texels, a magnification filter is used that maps a texel on to multiple pixels. (Bouvier, 1999)

In the case of magnification, each texel will appear as several pixels. The result of this is that individual texels may be seen in the rendering. This is called 'texelisation'. In Java3D, the magnification filter allows a user to use different options in determining pixel values from texel values. The first method is to use nearest neighbour sampling and the second method is to use linear interpolation sampling. Nearest neighbour sampling will find the closest texel to the pixel under investigation and apply the texel's colour to that pixel. Linear interpolation sampling will interpolate among the neighbouring texels to the pixel and will map the interpolated colour to the pixel. The nearest neighbour sampling will be quicker than the linear interpolation sampling as fewer calculations are required. The advantage of using the linear interpolation method is that the appearance of texelisation may be minimised.

In the minification filter, where the pixels are larger than the texels, the texels must be minified to fit to the pixels. The problem is that a pixel value can have only one colour and yet several texels supply colour values. As in the magnification filter, nearest neighbour and linear interpolation sampling methods can be used.

Extending the simple example discussed in section 5.1.6 it can be seen how a texture is mapped onto a surface (see Figure 5.28). The first thing to note is that there will be as many texture co-ordinates as there are vertices creating the surface. In the same way as the co-ordinates creating the surface are indexed, so too are the texture co-ordinates indexed. The number of texture co-ordinates will be the same as the number of co-ordinates used to create the surface. The texture co-ordinate index is required to map each texture co-ordinate to a vertex. In this case, and the case of the footprints, the co-ordinate index is the same as the texture co-ordinate index. In the figure, the first

value in the index is one and it represents second elements in both the array of co-ordinates and the array of texture co-ordinates. This will mean that the second texture co-ordinate will be mapped onto the vertex of the triangle represented by the second element in the array of co-ordinates. The rest of the texture will be indexed in the same way. Java3D deals with the mapping of texels to pixels and only the texture co-ordinates need to be specified.

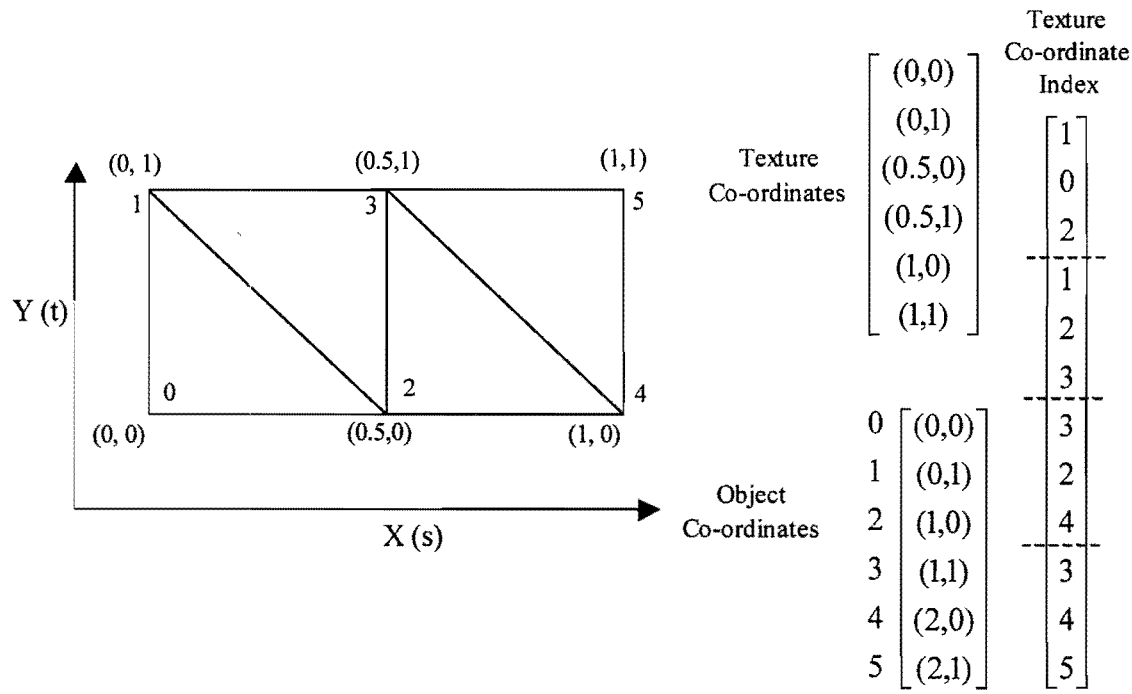


Figure 5.28 – Texture Mapping Example

As mentioned previously, the photogrammetric process resulted in ortho-images being created from the DTMs. DTMs of each footprint therefore have a corresponding ortho-image. An ortho-image of a footprint is mapped onto the 3D surface in the same way as described in the example above. As described previously a texture co-ordinate is required for each point in the DTM to successfully map the ortho-image onto the surface. The texture co-ordinate array would therefore have more than ten thousand co-ordinates for each footprint. Figure 5.29 shows the algorithm used to create the array of texture co-ordinates for the footprints.

```
k = 0
For i = 0 to i < R
        For j = 0 to j < C
                T[k] = (i/R, j/C)
                k = k+1
```

Where:

T[] is the array of texture co-ordinates

i, j, k are variable integer values

C is the number of columns

R is the number of rows

Figure 5.29 – Algorithm to Create Texture Co-ordinates

Due to the large amount of data required with the mapping of textures, rendering of the 3D surface becomes slower when a texture is overlaid. Another point that must be made is that the texel colour will replace any prior pixel colours. If a 3D surface is a specific colour and a texture is overlaid, the texture will take precedence over the colour. ***Plate 3*** shows a footprint with an ortho-image overlaid onto the surface.

5.6.3 Transparency

The transparency attribute changes the transparency of an object. A good example for the use of the transparency attribute is creating glass objects, such as windows, in a virtual building. In the case of the footprints, the flood plane (discussed in section 6.1) is given a transparency attribute, which allows a user to view the surface of the footprint under flood plane.

5.6.4 Structure

Java3D provides a means whereby the structure of the surface may be altered by displaying it as either a scatter of points, a grid or a solid. This differs to the various methods of displaying the surface described in section 5.1.1. When altering the structure, the surface characteristics remain the same, for example, the normal vectors defining the direction of reflected light remain unchanged. In the case of the points, only the pixels, which represent the vertex points on the surface, will be drawn. Thus, the normal vector values are still known and lighting calculations can be performed. The same concept is used for the grid. This enables the user to more easily define the shape of the surface when viewing the points or the grid. A disadvantage is that rendering speeds of the points or the grid are no better than that of the surface. The grid structure also varies from the grid structure described in section 5.1.1 in that it is triangular. (See Figure 5.30)

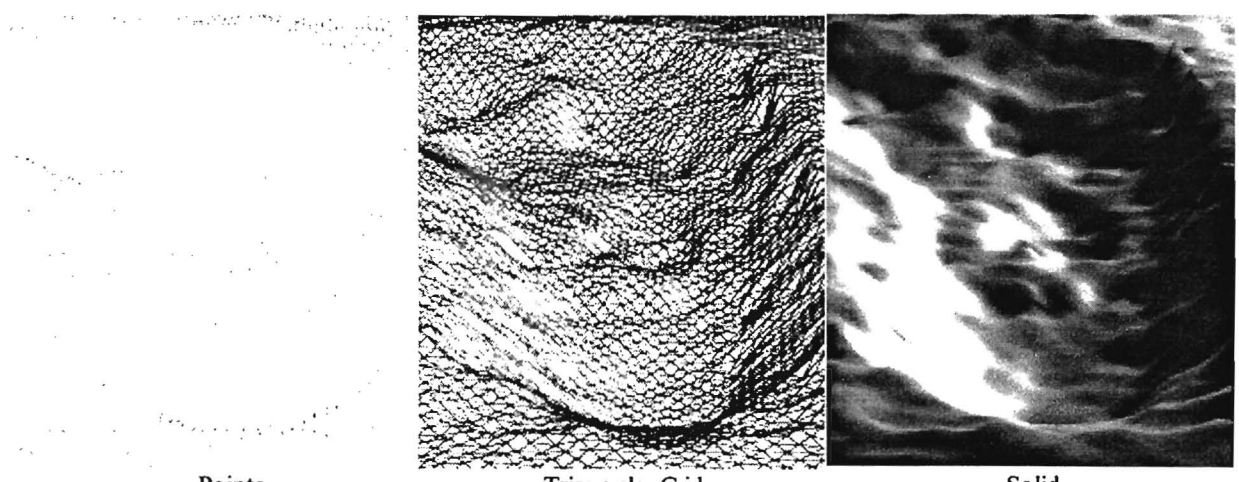


Figure 5.30 – A Footprint Displayed in Three Different Structures

The advantage of being able to alter these attributes is that the surface does not have to be recalculated in order to view the surface represented as points or as a grid. This may be useful when viewing a surface that has a spike in the DTM. By altering the structural appearance of the surface, the user is able to determine whether the spike is due to one or more outliers in the DTM.

5.7 Contours

Before the visualisation project began, contours for the footprints were created using the DTMs. The contours were created using ArcInfo, a GIS package. They were stored in the Drawing Interchange File (DXF) image format. The DXF image format is one of the more intricate image formats as it is a vector based as opposed to a raster based image format. Each DXF contour file is about 6Mb in size. Distributing files of this size requires large amounts of storage space and since most of the data in the file is not required, a program that extracted the relevant information was written in Java.

POINT	X	Y
A	0.0	0.0
B	1.0	1.0
C	1.0	2.0
D	2.0	2.0

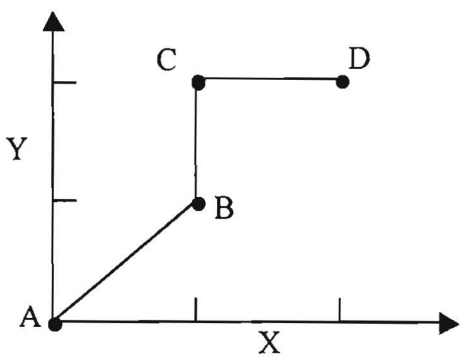


Figure 5.31 – Description of a Polyline

In the DXF file, the contours were stored as polylines. A polyline is a line created by joining successive co-ordinates in a list (see Figure 5.31). The output file from the Java program is a text file that consists of three columns representing the x, y and z values respectively. The start and end of the polylines in the DXF file are recorded by adding a specific marker into the list of co-ordinates in the new file. This marker is the value –9999.99 and it is placed into the x, y and z positions of the columns in the text file. The resulting contour data files are approximately one tenth the size of the original files.

Two methods of displaying the contours are made available in the visualisation program. The first method is to display the contours in a separate window that can be viewed at the same time as the 3D surface is viewed (see Figure 5.32). This is a static display, the only functionality being the ability to zoom in or out. The advantage of this method is that the user can compare the contour map with the 3D surface and locate areas of interest that may have been overlooked.

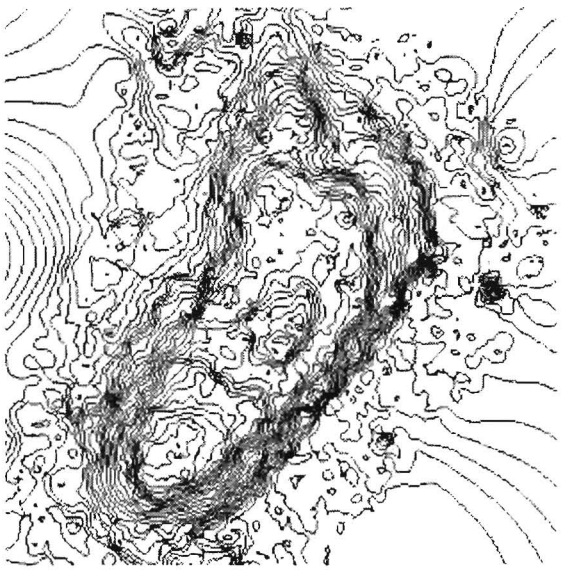


Figure 5.32 – 2D Contours

The second method is to display the contours overlaid onto the 3D surface (see ***Plate 4***). The 3D contours are created in much the same way as the surface displayed as a grid in that the contours are a collection of lines as is the grid shaped surface (see Section 5.1.4). Consecutive points in the new contour data files are indexed as a polyline, for example, point 1 is indexed to point 2 and point 2 to point 3 and so on. At each marker, the value –9999.99, a new polyline is started by indexing the first value after the marker to the second value after the marker. A switch is supplied that allows the 3D contours to be added or removed from the virtual universe. Likewise, the 3D surface can be displayed or hidden. This allows the user to view the 3D contours independently of the surface (see ***Plate 5***). The contours are coloured according to the graduated colour scheme used in the display of the surface. As lines do not have normal vectors, lighting of the contours to make them visible proved to be a problem. An initial solution to the problem was to set the material of the lines so that they would have an emissive light. Unfortunately, as discussed in section 5.6.1, this is not possible as the colours of the lines override the material settings. In order to solve this problem, normal vectors that project directly upwards (in the direction of increasing z) are assigned to the beginning and end of each line. The problem with this method is that if the 3D contours are rotated by large amounts they become difficult to see as the light is reflected in only the one direction.

5.8 Fly-Through

A fly-through is an animation constructed by means of translating the viewpoint through a set of co-ordinates. Java3D supplies classes for the animation of objects in the virtual universe. The Java3D class used for the fly-through is called a *RotPosPathInterpolator*. It defines a behaviour that modifies the rotational and translational components of its target transform group by linearly

interpolating among a series of predefined position and orientation pairs. The target transform group in this case is that of the viewpoint. An *Alpha* object is used to generate the interpolated values. The *Alpha* object is a class in Java3D used to create time varying functions. The *Alpha* object will map an action to time. When creating the fly-through it is used to define position and orientation of the viewpoint at specific time values.

The fly-through for the footprints is created in such a way as to position 'knot points' a certain distance beyond the extremities of the footprint. A knot point is one of two points between which position and orientation values for the viewpoint are interpolated. Eight knot points are used in the formation seen in Figure 5.33. The viewpoint's vertical position is slightly above the surface resulting in an oblique view. One of the advantages of the method in which the fly-through is created is that it does not need to be altered for each footprint.

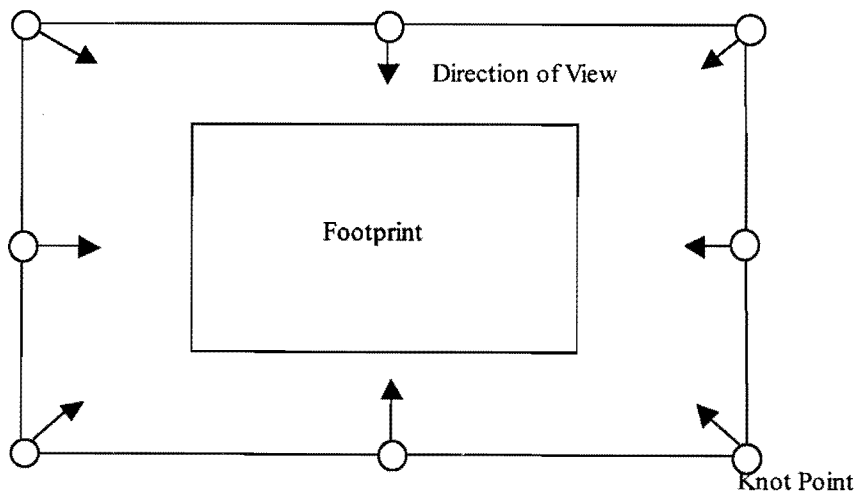


Figure 5.33 – Structure of the Fly-Through

Chapter 6
Interactive 3D Graphics

The previous chapter described the creation of 3D surfaces, viewpoints, movement and appearance of 3D objects. This chapter will discuss the user interactive tools that were created for the footprints. These tools include the flood plane, measurement tools, profile tools and gradient tools.

6.1 The Flood Plane

The flood plane (or cut plane) is the first interactive tool that was implemented in the project. A simple rectangular shaped plane is created, the size being the same as that of the DTM. The appearance of the flood plane is set to resemble water as the concept of rising and ebbing water supports a user's perception of a level plane. In order to set the appearance of the flood plane to resemble water, the material and transparency had to be set to appropriate values (see section 5.6).

Once the plane is created it is positioned in the 3D scene at the appropriate location. The initial position of the flood plane is set as the lowest point of the DTM. Another factor that was considered was that the user might not want to see the flood plane. A switch is therefore available that allows a user to either show the flood plane or hide it.

The next step is to allow user interaction in raising and lowering the flood plane. A tool is required that allows the user to manipulate the height of the flood plane. The tool used for this is a slider bar, which is a Java GUI component, which is similar to a scrollbar. As a user reduces or increases the value of the slider bar the flood plane lowers or rises respectively. The minimum value of the slider bar corresponds to the lowest point of the DTM and the maximum value the highest point in the DTM. In order to move the flood plane independently of the 3D surface it has to have its own transform group (see section 4.8). Movement of the plane is done by changing the z-value in the translation matrix seen in Equation 5.4. (Section 5.2).

There are various benefits of the flood plane. One of the benefits is that a user may define a level plane on the surface of the footprint. This level plane defines the outline of contours. Another benefit for the user is that the highest and lowest points on the 3D surface may be identified. From observation, raising the flood plane to a certain level defines the shape of the footprint (see ***Plate 6***).

6.2 Height exaggeration

Creating height exaggeration relative to horizontal distances will affect the emphasis and perception of relief. The exaggeration can be set to emphasise the qualities of the surface that are of current importance. Often less height exaggeration gives better overall impressions whereas greater height exaggeration allows a user to examine smaller areas more closely. In general, the smaller the scale of the terrain, the larger the corresponding height exaggeration should be. (Stynes, 1996)

A tool is made available to exaggerate the heights of the footprint. As with the flood plane a slider bar is used allowing a user to choose a value for the vertical exaggeration. Changing the vertical scale of the object is achieved by setting the *Sz* value in Equation 6.1 to the amount by which the heights are to be exaggerated. Both the *Sx* and *Sy* must equal 1.0 as only the heights are to be exaggerated.

$$[x \quad y \quad z \quad 1] \cdot S(Sx, Sy, Sz) = [Sx \cdot x \quad Sy \cdot y \quad Sz \cdot z \quad 1] = [\hat{x} \quad \hat{y} \quad \hat{z} \quad 1] \qquad \textbf{Equation 6.1}$$

$(Sx = Sy = 1)$

where

$$S(Sx, Sy, Sz) = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ is the scaling matrix

and

$(\hat{x} \quad \hat{y} \quad \hat{z})$ are the scaled co-ordinates

When exaggerating the heights of the surface, the positions of other objects such as the flood plane have to be considered. Scaling of the footprint results in the position of the flood plane not corresponding to the position of the footprint. All the objects that require scaling are put in the same transform group (see Figure 4.8). The scale factor is applied to the transform group and therefore to all the objects within it.

The vertical exaggeration of the footprints is able be set to three times the normal in steps of 0.25. This exaggeration emphasises the shape of the footprint. Using data of smaller scales, an exaggeration of three times may not be sufficient to emphasise changes in height on the surface. In such cases, a greater height exaggeration would be required. An example of this would be to view a 3D model of the African continent. The ratio of heights to horizontal distance is very small. Heights may be in the range of zero to five kilometres whereas horizontal distances are measured in thousands of kilometres. In order to see height variations the scale of the heights must be exaggerated by large amounts, especially if the variations in heights are small.

Since the bottom of the surface cannot be seen (only one face of the triangles creating the surface is visible – discussed in Section 5.1.5) a function is required that inverts the surface. In order to invert the 3D surface all the DTM heights are multiplied by negative one. The surface is then redrawn. The result of this is that the object appears as a cast of the original (see Figure 6.1). The scale of the inverted object can then be altered in the same way as the normal surface. It must be noted that the inverted footprint is a mirror image of the original.

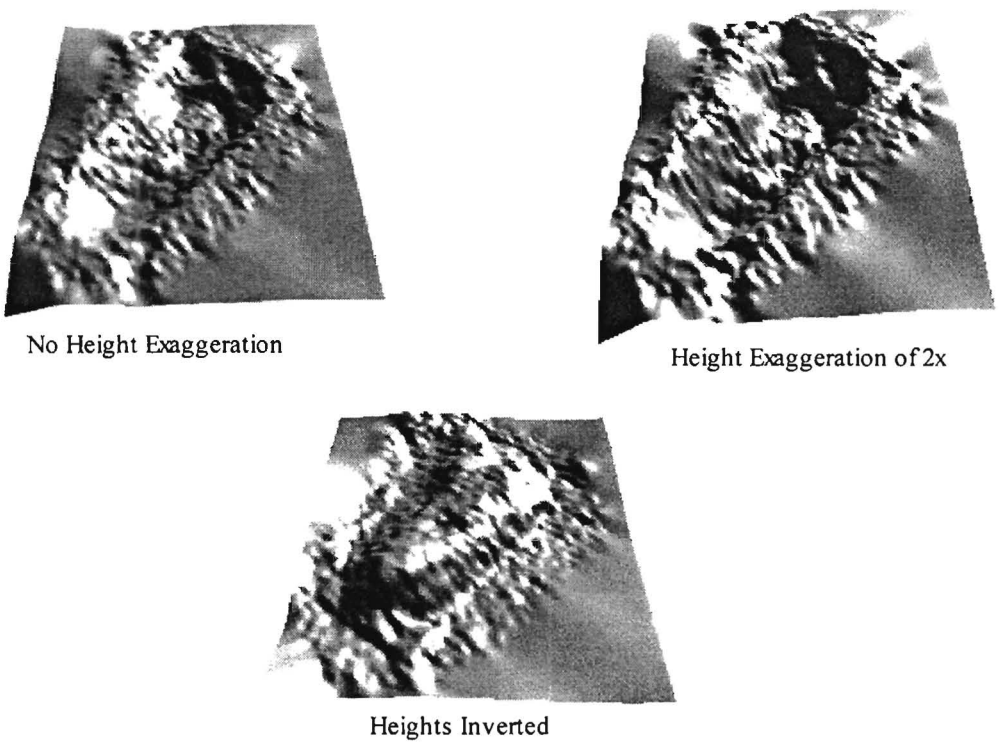


Figure 6.1 – Exaggeration and Inversion of Heights

6.3 The Measurement Tools

The co-ordinates on the surface of the footprint are determined by enabling the measurement tools and selecting points with the mouse. A click of the mouse on the 3D surface returns the x, y and z co-ordinates of the point selected. These measurements are listed in a table in the order in which they are made. For the first measurement, only the x, y and z values are inserted into the table. If a second measurement is made the x, y and z co-ordinates are listed in the second row of the table. 2D distance, 3D distance and change in height values are calculated between the first and second measurements and inserted into their corresponding columns in the second row of the table. Successive measurements follow the same procedure (see Table 6.1).

For each measurement made, a method of displaying the position of the measurement on the surface of the footprint had to be implemented. Creating a small sphere at the origin of the co-

ordinate system and translating it to the required position on the 3D surface marks the position of a measurement. Another problem is the identification of individual measurements from a series of measurements made. To this end, a function is made available that alters the colour of the sphere when a row in the measurement table is selected. The initial colour of the spheres is set to green. When a measurement is selected in the table, the corresponding sphere colour is set to red. A function was also added that allows a user to delete a measurement. If the second measurement in Table 6.1 were deleted, the distance calculations would be updated by performing the distance calculations between the first and third measurements.

Table 6.1 - An Example of Recorded Measurements

Point No.	X	Y	Z	2D Distance	3D Distance	ΔH
1	100.00	100.00	35.00			
2	200.00	150.00	43.00	111.80	112.09	8.00
3	150.00	200.00	27.00	70.71	72.50	-16.00

Java3D provides various means of selecting objects in a scene by clicking on them with the mouse. One of these methods is to select the closest vertex on the object to where the user clicked with the mouse. The position of the vertex would be known and the co-ordinates used in measurements. This method would not provide accurate measurements. It was also not available in the version of Java3D used. Another means of performing measurements on the 3D surface had to be developed.

$$X = x_o + x_d * dist$$
$$Y = y_o + y_d * dist$$
$$Z = z_o + z_d * dist$$

Equation 6.2

Where:

X, Y and Z are the co-ordinate components of the measured point;

x_o, y_o and z_o are the co-ordinate components of the origin of the vector (ie. the viewpoint);

x_d, y_d and z_d are the components of the direction vector and

dist is the distance to the intersection. (length of the vector)

The following method is used when making measurements on the surface of the footprints. When a mouse click occurs on the 3D surface, 2D screen co-ordinates are returned. The screen co-ordinates are transformed by Java3D into 3D image plate co-ordinates. A unit vector is created between the viewpoint co-ordinates and the selected image plate co-ordinates. The unit vector is used to initialise a 'pick ray'. The pick ray is a vector that is projected into the virtual world from the viewpoint through the position of the mouse pointer on the image plate (see Figure 6.2). When the

pick ray intersects the surface of an object in the virtual world, it returns the distance between the viewpoint and the point of intersection. Once the location of the viewpoint and the direction and length of the pick ray are known, Equation 6.2 is used to calculate the co-ordinates of the mouse click on the 3D surface:

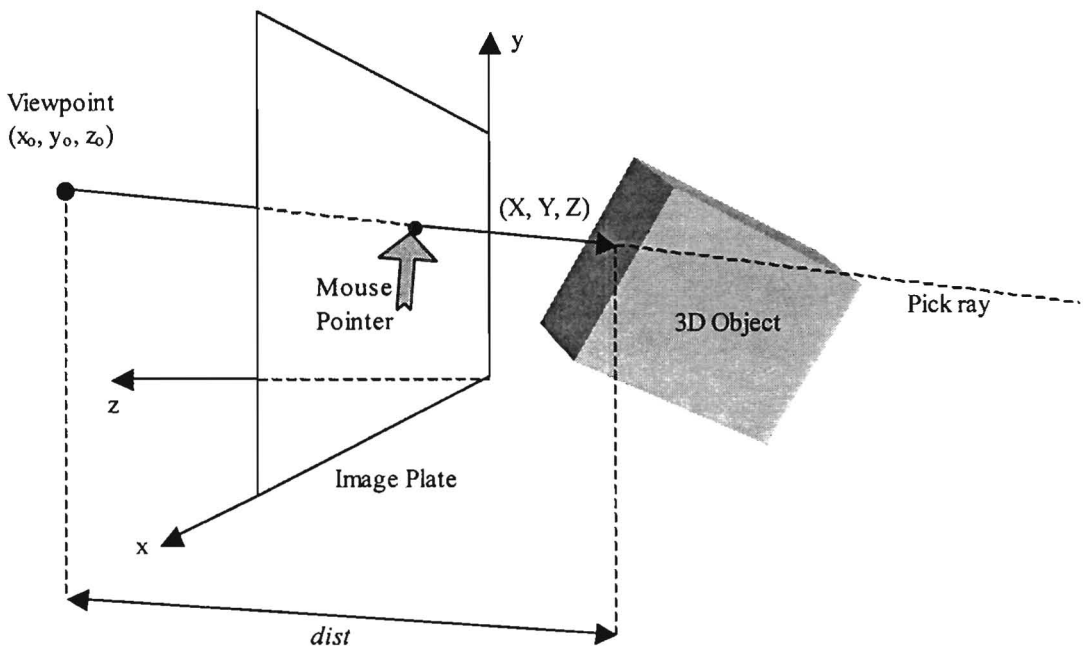


Figure 6.2 – A Pick Ray

Plate 7 shows a footprint on which two measurements have been made.

6.4 Creation of a Profile

The creation of vertical profiles of a 3D surface in the visualisation program is dependent on the measurement stage. Two measurements are required for the profile, namely a starting point and an ending point. The first step is to determine the number of points that are required for each profile. This is done by calculating the distance between the selected start and end points of the profile (see Figure 6.3).

The distance is then divided by the spacing between the points. In the case of the footprints the spacing between the points on the profile is set at 2.5mm, which is the same as the spacing between DTM points. A profile along the length of a 25cm footprint will therefore consist of 100 points.

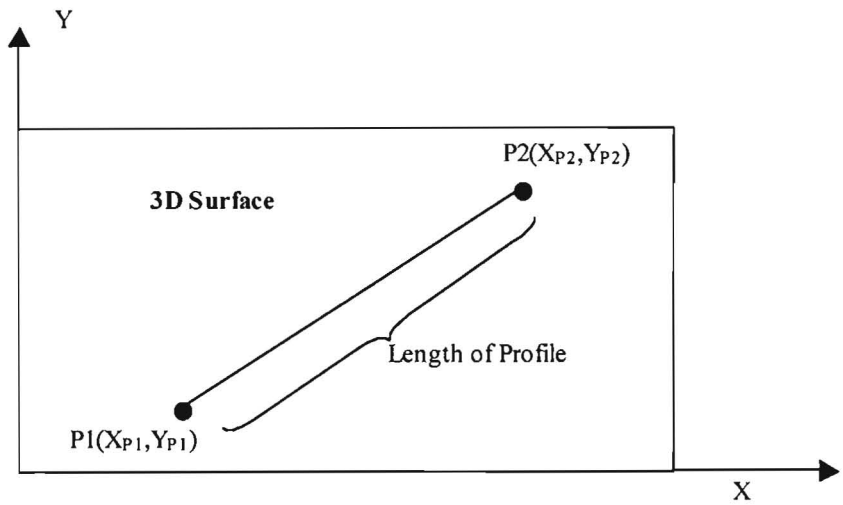


Figure 6.3 – Calculating the Length of a Profile

The next step requires that the x and y co-ordinates of the profile points be known. The following equation is used to calculate these co-ordinates:

$$x_p = X_s + n \cdot (0.25 \cdot Cos\ \theta) \qquad \textbf{Equation 6.3}$$
$$y_p = Y_s + n \cdot (0.25 \cdot Sin\ \theta)$$

Where:

x_p, y_p are the x and y co-ordinates of a point on the profile;

X_s, Y_s are the x and y co-ordinates of the starting point of the profile;

n is an integer value representing the number of the point in the profile and

θ represents the direction of the profile.

Calculating the number of points in the profile and their respective horizontal positions is a simple task. The next task concerns the mapping of height values to profile points. Three different methods of obtaining these heights were attempted. The decision on which method to utilise was determined by observing the speed and accuracy of the creation of a profile.

Method 1:

This method uses an algorithm similar to that of the measurement tool. An origin and direction for a pick ray are required for this method. The origin is chosen to be a point on the z-axis above the surface of the footprint. An accurate direction for the pick ray is required to intersect the surface of the footprint at the correct position in order to calculate the height of the profile point. The direction is calculated by determining the direction of the vector from the origin to the profile point. The profile point has an x and y value and a temporary z value. The temporary z value is the same for each point on the profile and is determined by calculating the average z value of the two

measured points. From Figure 6.4 it can be seen that the projection of the pick ray through the profile point with the temporary z value results in a new z value being determined. This new z value may be inaccurate, as the length of the pick ray from the origin to the point of intersection could be incorrect. (See section 6.3 on calculating co-ordinates from a pick ray and origin).

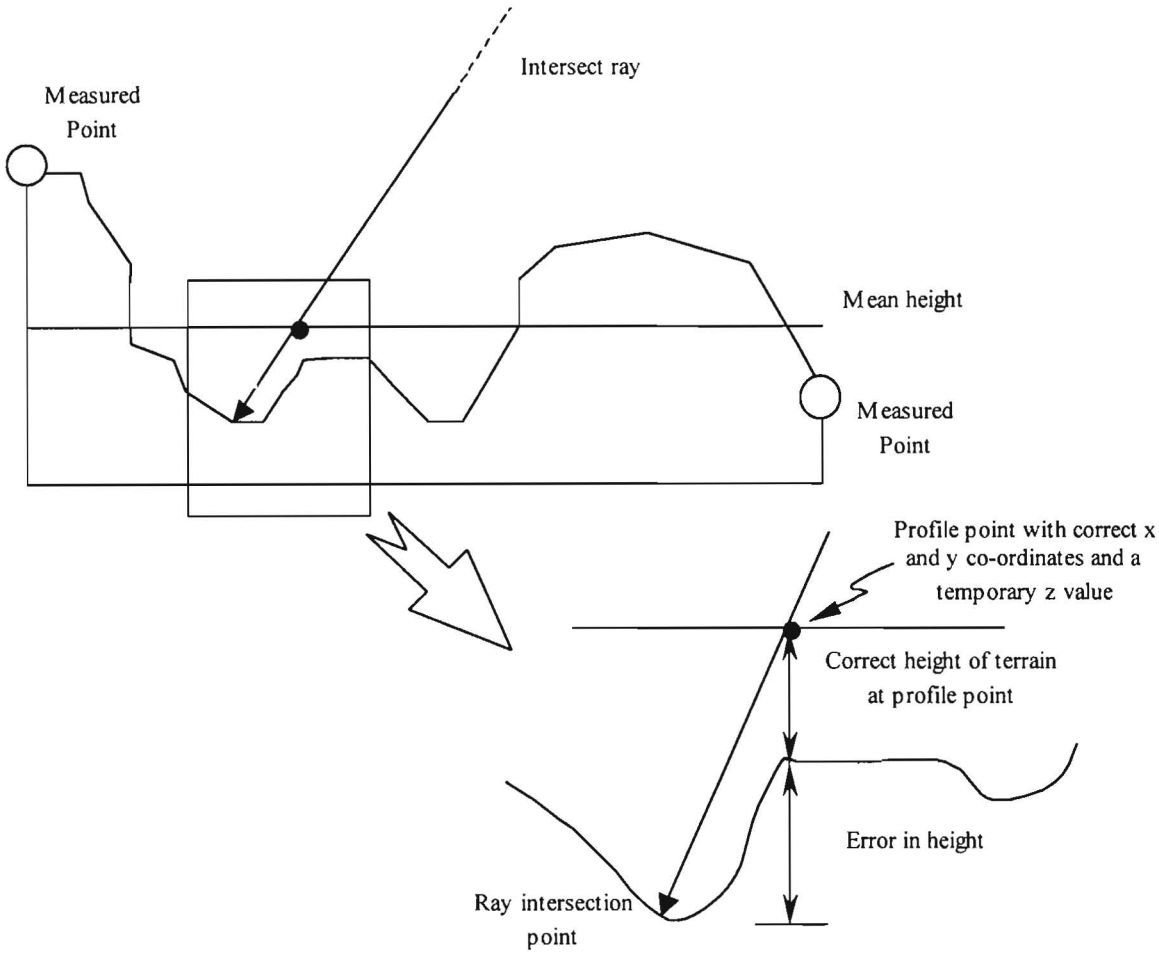


Figure 6.4 – Errors Due to Oblique Rays

In order to achieve more accurate results for the length of the pick ray, certain modifications could be applied:

- Iteration of the method above. This would be done by updating the temporary z value with the new z value. As the iterations progressed the difference in z values would become less until a specified difference had been reached.
- The height of the origin above the surface could be increased, minimising the angle of the pick ray from the vertical. The further away the origin of the pick ray is from the surface, the closer the point of intersection on the surface will be to the required point of intersection. The intersected point may be close enough to the actual point that no iterations will need to be done.

- The origin of the pick ray could be moved. Knowing the x and y value of the profile point the origin of the pick ray could be placed directly above the profile point. The direction of the pick ray would therefore be directly downwards and the intersection on the surface of the footprint would be at the correct point.

Although this method provides accurate results, it is slow due to the many calculations that have to be performed.

Method 2:

Once the x and y co-ordinates of the profile points are known, the co-ordinate list of the DTM is searched for the closest two points. The heights of these two points are known and a weighted mean is performed to determine the height of the profile point (see Figure 6.5). The following equation is used to calculate the height of the profile point:

$$Z_P = Z_{P1} - \frac{d_1(Z_{P1} - Z_{P2})}{D_{P1\_P2}} \qquad \textbf{Equation 6.4}$$

Where:

Z_P is the height of the profile point;

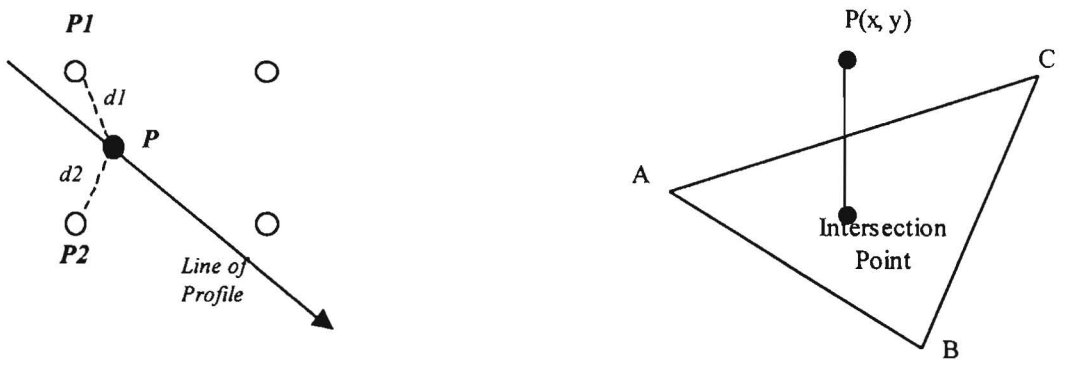
$P1$ and $P2$ are the closest two points;

Z_{P1} and Z_{P2} are the height values of $P1$ and $P2$ respectively;

d_1 is the distance from $P1$ to the profile point; and

D_{P1_P2} is the distance between $P1$ and $P2$.

Although this method is not the most accurate, few calculations are required and the profile is created more quickly than in method 1.



Using 2 points for the profile

Using 3 points for the profile

Figure 6.5 – Calculating a Profile Using Two Points and Three Points

Method 3:

The third method is a variation of the second method. In this method, the closest three points are selected. These three points are used to form a plane (see Figure 6.5) using the following equations:

$$Ex + Fy + Gz = D$$ **Equation 6.5**

where

$$D = Ex_0 + Fy_0 + Gz_0$$

$$E = (B_y - A_y)(C_z - A_z) - (C_y - A_y)(B_z - A_z)$$

$$F = (B_z - A_z)(C_x - A_x) - (C_z - A_z)(B_x - A_x)$$

$$G = (B_x - A_x)(C_y - A_y) - (C_x - A_x)(B_y - A_y)$$

and

$$A = A(x, y, z)$$

$$B = B(x, y, z)$$

$$C = C(x, y, z)$$

As the x and y co-ordinates of the profile point values are known, they can be substituted into the equation for the plane to determine the height. More calculations are required in this method than in the previous method, causing it to be slower.

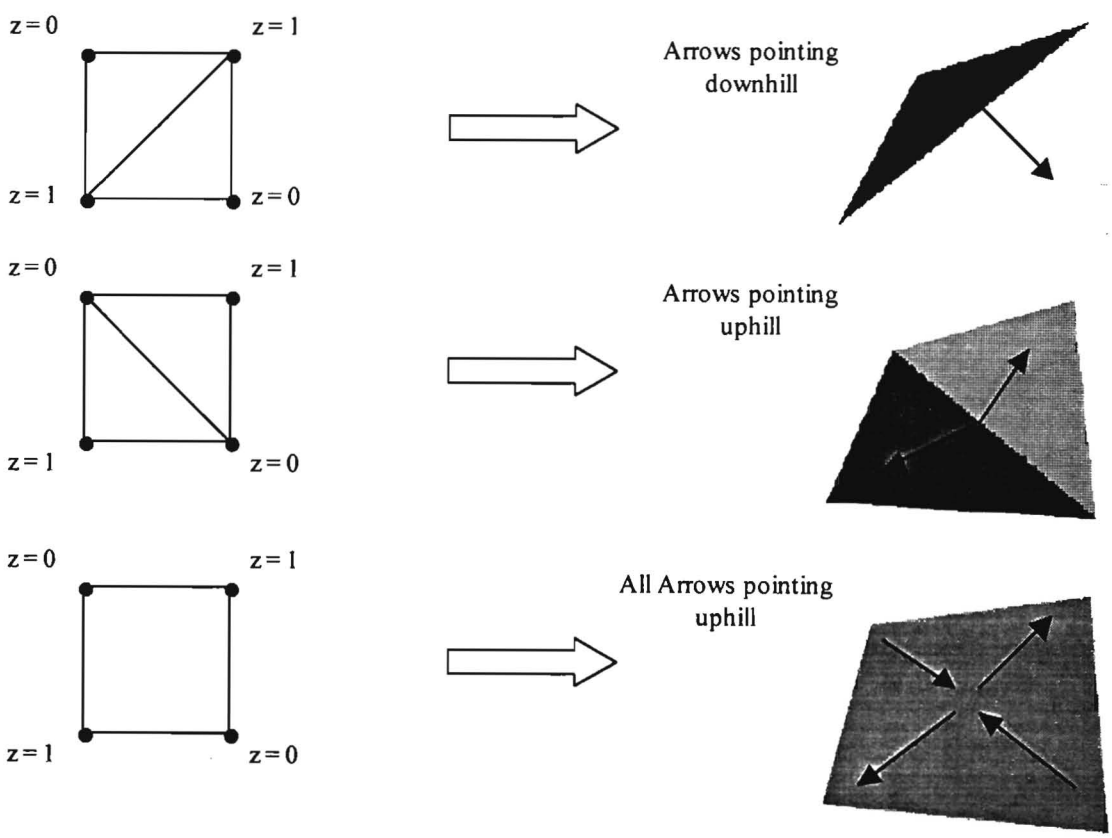


Figure 6.6 – Different Means of Indexing Triangles

The accuracy of this method depends on the triangle formed by the three closest points. In the case of the DTMs for the footprints, the triangles will always be the same size. The problem occurs when the triangle selected is not the same as the triangle that was rendered. In Figure 6.6 two surfaces were created using the same co-ordinates and viewed from the same position. The only difference is in the way the triangles are referenced to create the surface. It can be seen that the way the surface is created may determine the shape of the surface. In the case of the footprints where the triangles are much smaller, the difference will not be noticeable. Measurements made on the surface by means of the pick ray or by the selection of the three closest points to create a plane may be inaccurate. In order to overcome this problem, a square formed by four points in the DTM in which the profile point lies, must be determined.

The square formed by the closest four points of the DTM will surround the profile point. Approximating the surface of the square may be done using a hyperbolic paraboloid. The equation for the hyperbolic paraboloid can be seen in Equation 6.6. Figure 6.7 shows how the hyperbolic paraboloid is defined by four points in a square. (Kraus, 1997)

$$Z = a_0 + a_1 X + a_2 Y + a_3 XY \qquad \textbf{Equation 6.6}$$

Where:

Z is the height of the profile point that is to be calculated;

X and Y are the x and y values of the profile point relative to the first point of the square;

The coefficients a_i are determined by solving the following matrix where:

Δd is the length of the sides of the square and

Z_i is the height of the i^{th} point of the square:

$$\begin{bmatrix} a_o \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1/\Delta d & 1/\Delta d & 0 & 0 \\ -1/\Delta d & 0 & 1/\Delta d & 0 \\ 1/\Delta d^2 & -1/\Delta d^2 & -1/\Delta d^2 & 1/\Delta d^2 \end{bmatrix} \begin{bmatrix} Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \end{bmatrix}$$

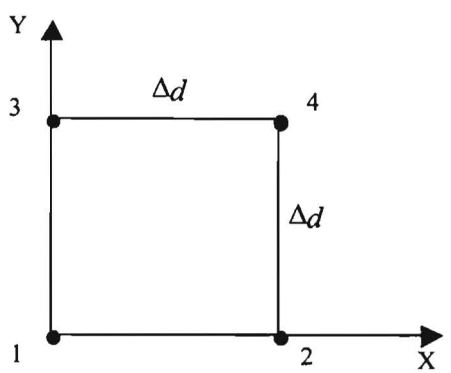


Figure 6.7 – Structure of Square for Hyperbolic Paraboloid Equation

All three methods have their advantages and disadvantages (see Table 6.2). It was decided that great accuracies would not be required, as there is no noticeable difference in the display of the calculated profiles when displayed on the computer screen. The choice of which method to use in the calculation of the gradient is based on the speed at which it is calculated. The second method is selected, as it is the fastest method. ***Plate 8*** shows a profile through a footprint between the measured points displayed in ***Plate 7***.

Table 6.2 – Advantages and Disadvantages of Methods used to Calculate the Profile

Method Number	Advantage	Disadvantage
Method 1	The amount of equations required are not dependent on the number of points in the DTM. They are dependent on the number of points in the profile	Pickrays require many calculations to determine the intersection point on a 3D surface. This results in Method 1 being slow.
Method 2	Fewer calculations are performed in this method than in the other two methods, thus it will be faster.	The heights calculated in this method are not as accurate as in the other methods.
Method 3	This method provides the most accurate results	This method is slow due to the amount of calculations required.

6.5 Gradients

Various types of data may be overlaid onto the surface to facilitate a user's understanding of the nature of the surface. In the 3D display of a surface, it is sometimes difficult for a user to perceive slopes. In order for the user to calculate a gradient on the surface, a horizontal and a vertical distance would be required between the points of interest and the gradient would have to be calculated manually. The horizontal and vertical distances would be obtained by using the measurement tool. This method would be time consuming thus a tool was created that calculated all the gradients on the surface. These gradients were then overlaid onto the surface.

For every point in the DTM, a gradient is calculated. The vertex corresponding to the DTM point for which the gradient is calculated was assigned a colour based on its gradient value. As the DTM is a grid, gradients can be calculated in four directions (see Figure 6.8) namely: x-direction, y-direction and the two diagonal directions.

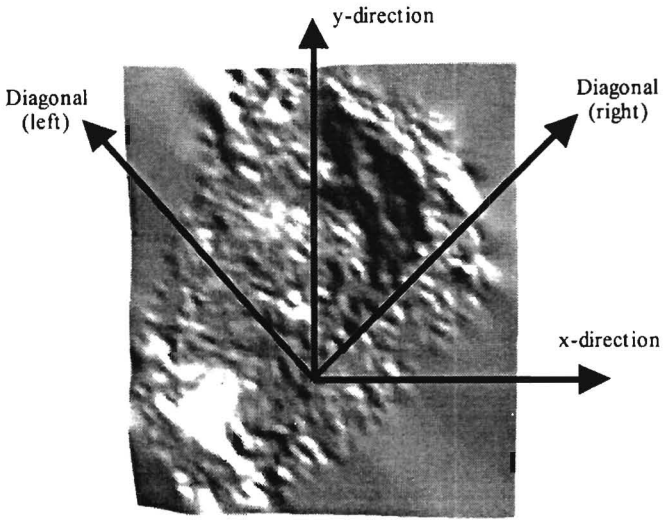


Figure 6.8 – Directions for the Gradient Calculations

In the case of calculating the gradients in the x-direction the two neighbouring points, one on the left and one on the right of the point for which the gradient is being calculated, are used. A best-fit line that minimises the sum of the vertical distances is calculated for the three points (see Figure 6.9) using Equation 6.7. Since the horizontal spacing between the points used in the gradient calculation is the same only the two outside points ((x_0, y_0) and (x_2,y_2) in Figure 6.9) are required for the calculation of the gradient for the best fit line.

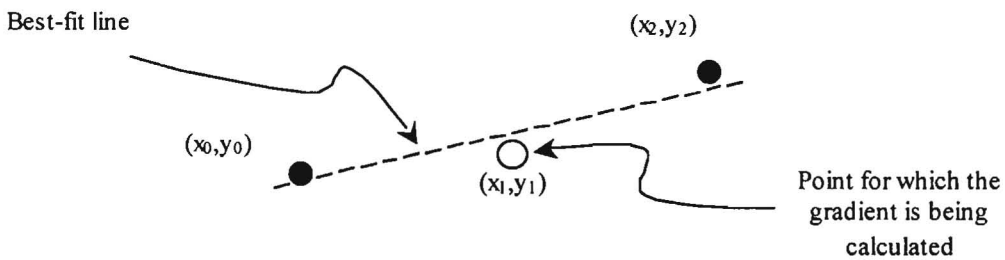


Figure 6.9 – Best-Fit Line Calculated from Three Points

In the case of calculating the gradients in the y-direction, the x_k value in the above equation would be the y co-ordinate. When the diagonal gradients are calculated, x_k in the above equation has to be calculated by determining the spacing between adjacent points in either the x or y direction (if the DTM is a square grid) and multiplying that value by $\sqrt{2}$ (see Figure 6.10).

$$Gradient = \frac{\left(\sum x_k\right)\left(\sum z_k\right) - n\sum x_k z_k}{\left(\sum x_k\right)^2 - n\sum x_k^2}$$

Equation 6.7

(All the sums run from $k = 1$ to $k = n$)

Where:

n is the number of points;

x_k is the x co-ordinate;

and z_k is the height co-ordinate.

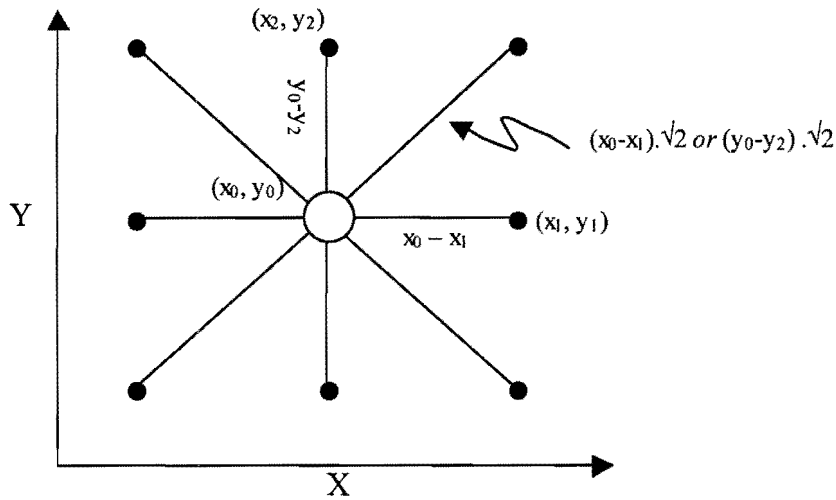


Figure 9

Figure 6.10 – Calculation of Gradients for Oblique Grid Points

Once the gradient of each vertex has been calculated, a colour is applied according to its value. ***Plate 9*** demonstrates a footprint with the gradient calculated in the x-direction. ***Plate 10*** shows a 2D representation of the gradients seen in ***Plate 9***. The 2D representation gives an impression of the shape of the footprint. Five colours are used to group the gradient values, namely red, blue, green, yellow and white (see Table 6.3).

Table 6.3 – Colours used to Represent Gradient Values

Colour	Gradient (g)
Red	$g > 1.00$
Blue	$1.00 < g < 0.75$
Green	$0.75 < g < 0.50$
Yellow	$0.50 < g < 0.25$
White	$g < 0.25$

The gradients can be calculated in any of the four directions depending on the user's selection. An option is provided that takes the results from all the gradient calculations and only displays the greatest gradient values. In order to give the user more functionality, another function is supplied, which enables the user to alter the range of the colour groups used to group the gradient values. The user is required to choose a value between 0° and 90°. This enables the user to display, in red, vertices with a gradient greater than the selected value. The user interface for the gradient tool lists the altered gradient group values shown in Table 6.3.

Chapter 7
Summary and Analysis

Before commencing with the analysis of the project a brief summary of what has been done will be established.

7.1 Summary

In the creation of 3D graphics using Java3D, a virtual universe is required in which 3D objects can be placed. In this project, the 3D objects placed in the virtual universe are 3D surface models of the Laetoli footprints. These models are created using 3D data in the form of DTMs. The DTMs of the footprints were created in the photogrammetric process of the Laetoli project.

The surface of a footprint defined by a DTM can be displayed in a 3D environment of a Java3D virtual universe in four ways. The surface of the footprint, known as the geometry, can be displayed as either a scatter of points, a wire frame surface, a filled polygon surface or a Gouraud shaded surface. The choice of display will effect rendering speeds and realism of the 3D models. As the realism of an object increases, rendering speeds drop and vice versa. Since realistic models of the footprints are required, the Gouraud shaded surface type, considered the most realistic representation, is used to display the footprints even though rendering speeds are slower than in the other displays.

Once the 3D model (geometry) is inserted into the virtual universe, a viewpoint has to be specified. Without a viewpoint, a user will be unable to see the 3D surface in the virtual universe. Since the project has been created to allow user interaction with the 3D models, movement tools are made available. The movement tools allow a user to define the orientation and position of the viewpoint. Changing of orientation and position of the viewpoint is done by providing the user with rotation, translation and zooming tools. These tools allow a user to view the 3D models of the footprints from various perspectives. Since these movement tools can be difficult to use when rendering speeds are low custom viewpoint positions are available. Other viewing parameter can also be altered in the virtual environment. In the visualisation program, a user is given the choice of two different projections namely a perspective projection and a parallel projection. In the perspective projection, the viewing angle (field of view) can be altered.

Once the object has been added to the virtual universe and it is in the field of view of the viewpoint it is still not visible. To make the 3D object visible it must be illuminated by a light. Two different

lights may be used by the user namely, a directional light and a point light. A user is able to select which light source they would rather use. The directional light is a fixed light source whereas the point light source can be moved to different positions. By moving the light source, different features on the 3D surface can be highlighted.

The light source enables the 3D surface in the virtual universe to be visible to the user. The colour of the 3D surface at this stage is a greyish colour. In order to enhance the visual appearance of the 3D object, a colour value is applied to each vertex. A user is given the choice on whether to display the surface in a single colour or to apply a graduated colour scheme. The graduated colour scheme is based on variations in height where the lowest points in the 3D surface are red and the highest points are violet. Another function that is available is the fly-through. The fly-through is an animation that moves the viewpoint through a set of pre-defined positions. It provides a sequence of different perspectives of a 3D surface.

After the colours have been applied to the surface, the appearance node of the 3D object is set. In the appearance node the reflective qualities, the transparency and the structure are set as well as the textures for texture mapping. The reflective qualities (material) of an object are dependent on the lighting of the 3D surface. The parameters of the material can be set by the user. Due to the vertex colours overriding the material colours, the material settings do not work as well as they should. For example, the diffuse colour of a 3D object will be overridden by the colour of the vertices. The structure of the 3D object can be altered by the user using the structure settings. The structure settings allow the 3D surface to be displayed as a solid, a triangular grid or as points. The texture setting in the appearance node is required for the mapping of the ortho-image to the 3D surface. A user is given the capability, when required, to overlay the ortho-image onto the 3D surface and remove it. With the mapping of the ortho-image onto the 3D surface, the rendering becomes slower.

Since the contours are available, they are used to enhance the visual display of the 3D surfaces. This is done by overlaying 3D contours onto the corresponding 3D surfaces of the footprints. As more data is added to a 3D scene, more calculations are required and rendering becomes slower. A user is therefore given the option of adding or removing the 3D contours from the virtual universe. Another option that is given enables a user to view the 3D contours independently of the 3D surface by removing the 3D surface from the virtual universe.

After the 3D models have been created and their appearance set, interactive tools are implemented.

- The first interactive tool is the *flood plane*. The flood plane is a level plane that can be raised or lowered through a footprint by a user.

- *Exaggeration of heights* is the next interactive tool. This tool allows a user to exaggerate or invert the heights of a footprint.
- The *measurement tools* allow a user to click with the mouse on a 3D object and obtain 3D co-ordinates. These co-ordinates are used to return 2D and 3D distances as well as the difference in heights between two measured points
- The measurements are used in the *creation of profiles*. A profile is created through two measured points selected by a user.
- The final interactive tool is the *calculation of gradients*. This tool allows a user to select the direction in which the gradients are to be calculated as well as highlighting gradient values greater than a specified value in red.

7.2 Analysis

The aims of this project were to create a method of accurately displaying the surveyed data of the Laetoli footprints in such a way as to be useful to scientists and other interested people (see Chapter 1). In order to fulfil this aim, 3D graphics was used as a visualisation tool. The Java3D API, an extension of the Java Development Kit, was used to produce an application that created 3D graphics models and implemented interactive tools that would be available on different computer platforms.

7.2.1 3D Graphics Packages

In Chapter 3, different software packages used to create 3D graphics applications were discussed. An analysis of these packages resulted in Java3D being selected too create the 3D visualisation. Reasons for this are that VRML files require browsers to display their content and the creation of interactive tools is more complicated. If the visualisation were to be displayed on the Internet, VRML would have been chosen. Taylor (2000) created a similar project of the Laetoli footprints for the Internet using VRML. The OpenGL method was rejected for this project as the creation of interactive tools would require the use of an external programming language such as C++. This would result in the 3D-visualisation application being limited to a single computer platform. Java3D, a new extension to the Java programming language, is platform independent. The 3D abilities of the Java3D API are of a high standard and in addition to creating a visualisation package for the footprints, the API was able to be tested for future use.

7.2.2 Measurement Accuracies

One of the aims of the project is to be able to create measurements on the 3D surface accurate to the one millimetre. In order to test the accuracy of the measurement tools (discussed in section 6.3), measurements made in the visualisation program had to be compared to measurements from another source. ArcView, a GIS package, was used to display the contours of the footprints. Measurements were taken between various points on the contours. In the 3D model, the contours

were overlaid onto the surface and corresponding measurements were made. The reason the contours were used was that it enabled the selection of the same points on the 3D surface to be measured. Distances between the measured points were calculated. The distances from the ArcView measurements and the distances from the visualisation package measurements were compared. The average resulting differences between the measurements were in the order of 0.5 mm. These discrepancies were deemed acceptable as they fell within the prescribed limits of 1 mm.

7.2.3 Hardware Considerations

One of the concerns of the 3D visualisation of the footprints at the beginning of the project was that computer graphics capabilities were going to be a problem as rendering speeds of complex visual objects such as the footprints were very slow. Development of the visualisation program was commenced on a Pentium 133 computer with a 2Mb graphics card. This resulted in very slow rendering speeds limited to one or two frames per second. With the rapidly advancing field of computer graphics, 3D rendering is becoming quicker. On a Pentium II 333 with a Riva TNT 16Mb AGP graphics card, rendering speeds of up to seven frames per second are achieved.

7.2.4 Computer Platforms

Testing of the visualisation on various computer platforms was not as successful as anticipated. In order to run a Java application on different platforms, a run-time environment for that platform is required. Since Java3D is relatively new, the run-time environments are only available on certain platforms such as the Win32 platform and the Sun platform. Testing of the visualisation program was only done on Windows 95, 98 and NT systems. On all these systems, the application ran successfully. The use of a Unix based platform could not be attempted as the Java run-time environment and the Java3D extension for the Linux platform (a Unix based platform) was not available at the time of testing.

Chapter 8
Conclusions and Recommendations

8.1 Conclusions

Archaeological visualisations using 3D graphics have become an important factor in the display and study of archaeological data. Scientific visualisation is a wide-ranging discipline in which there are many sub topics. Interactive 3D graphics is a small part of scientific visualisation. This project concentrated on the 3D visualisation of the Laetoli footprints. Other data, such as textual data and 2D image data consisting of maps and drawings, could be added to the project to create a more conclusive scientific visualisation.

In terms of 3D graphics, there are limitations in the creation of accurate and realistic 3D models of the Laetoli footprints. One of these limitations is the requirement of large amounts of data to create accurate 3D surfaces. Computer hardware capabilities limit the possible detail that can be achieved. This results in a balance having to be achieved between computer hardware capabilities and the detail of the 3D models. In the case of this project, DTMs of between ten and twenty thousand points were used. DTMs of twenty thousand points or more resulted in very slow rendering speeds and DTMs with fewer than ten thousand points resulted in a loss of surface detail. With the advance of computer graphics hardware abilities, more detailed 3D models will be possible.

In other archaeological 3D visualisations that have been looked at, there are few interactive tools available for the user to study and gain greater insight into the nature of the 3D models. In this respect, the 3D visualisation of the Laetoli footprints was successful. With the integration of Java and Java3D, there are great possibilities in the creation of further tools that would be useful to the study of the Laetoli footprints. Although this project concentrated on creating 3D models of the Laetoli footprints, the same methods used to display and interact with the 3D models could be used on other spatial data.

8.2 Recommendations

The effectiveness of the visualisation was determined by giving a presentation to archaeologists from the University of Cape Town Department of Archaeology. Although the archaeologists who viewed the visualisation did not specialise in the specific field of the footprints they were able to make comments based on their expertise in other fields. Some of their comments were

implemented and those that could not are mentioned below along with some other principles and functions.

8.2.1 The GUI

The Graphical User Interface (GUI) is a graphical interface between the user and the program on the computer. It is an interface that takes advantage of the computer's graphics abilities to make the program easier to use. A GUI is important as it frees the user from complex command languages with which they may not be familiar. In this project, the emphasis was placed on the 3D display of the footprints and the creation of interactive tools. The GUI created for the visualisation program requires more attention as it is not a user-friendly environment in which the 3D models of the footprints can be studied.

8.2.2 Editing of Points

The visualisation uses DTMs obtained from a photogrammetric process to create a 3D surface. Once the data has been loaded and viewed in a 3D environment, errors in the data may become apparent. Errors such as spikes, where one of the z values of a point in the DTM is excessively large or small, are sometimes present (see Figure 8.1). A tool that allows a user to select and alter the height value of the point under question would be useful.

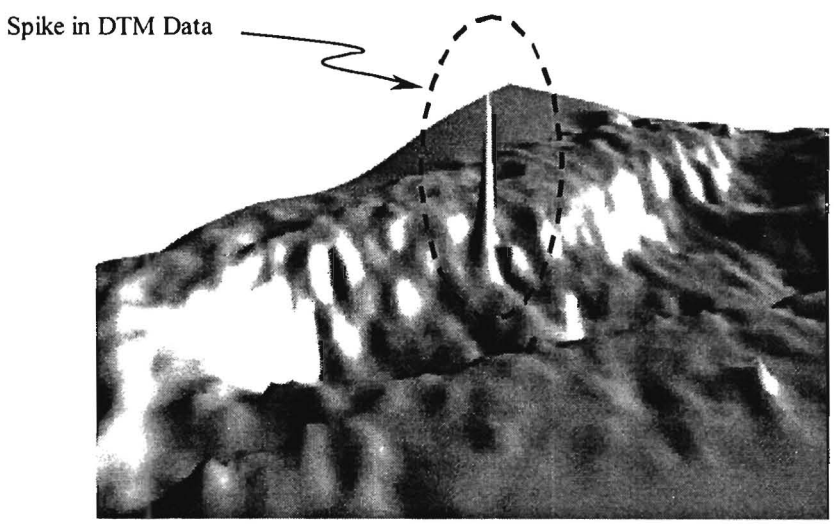


Figure 8.1 – Spike in the DTM

8.2.3 Creation of Reference Files

In section 3.7.1 it was mentioned that 3D objects in VRML are stored in text files. These files describe the geometry and appearance as well as the translational properties of the 3D objects. A similar format in Java3D may become useful in projects such as the visualisation of the footprints. Data pertaining to the creation of the 3D models such as co-ordinates, co-ordinate indices, colour indices, texture co-ordinates and texture co-ordinate indices could be stored in a simple text file.

The data from the text files would result in fewer calculations being executed thus increasing the performance of the computer.

8.2.4 Greater Input Functionality

At present, the visualisation program only has the ability to read and create surfaces from the regular DTM data used in the project. Other functionality that is useful and could be included for use in other projects is the ability to perform triangulations of irregular scatters of points. The triangulation is necessary for the indexing of co-ordinates to create a surface. By using a Voronoi triangulation, flat areas would consist of a small number of triangles and rugged areas would consist of more triangles. A surface can be more accurately defined using this method as features such as break lines can be depicted. Depending on the nature of the surface, Voronoi triangulation may result in fewer triangles being created thus improving rendering speeds.

8.2.5 Additional Data

In the conservation of the footprints, much data was collected for later studies and research. Some of the data consisted of condition reports. The condition reports were tracings of surface data onto paper, which was then scanned or digitised into digital format. The surface data consisted of roots, damage caused by roots and insects, and cracks. With the data in digital format in the form of an image, it can be overlaid onto the surface of the footprints. This would provide extra information thus adding to the effectiveness of the visualisation.

8.2.6 Comparison of Footprints

A useful tool, which was not implemented but would be useful, was the ability to compare selected footprints. In order to do this, two footprints would have to be rendered side by side. This would mean that, for the smaller footprints, at least twenty thousand points would have to be rendered. With these added points and other calculations required in 3D rendering, current personal computers would provide very slow rendering speeds. With the progress of computer graphics capabilities, more of these functions will become possible. Another tool that was not implemented for the same reasons was a 3D walk-through. This would allow the user to navigate through a virtual universe in which all the footprints were rendered.

References/Bibliography

Abouaf, J. (1999), The Florentine Pietà: Can Visualisation Solve the 450-Year-Old Mystery?, IEEE Computer Graphics and Applications, Potel, M. (ed.), Jan/Feb 1999, pp 6-11.

Boochs, F.; Garnica, C.; Wolter, F. (1998), Determination and Interactive Vislualization of 3D-objects, *International Archives of Photogrammetry and Remote Sensing*, Vol. XXXII, Part 5, pp. 316-322.

Bouvier, D.J. (1999), *Getting Started with the Java 3D™ API: A Tutorial for Beginners,* Sun Microsystems, Online. Available: http://java.sun.com/products/java-media/3D/collateral/#tutorial, 28 Jan 2000.

Brodlie, K.W, Carpenter L.A, Earnshaw R.A, Gallop J.R, Hubbold R.J, Mumford A.M, Osland C.D, Quarendon P, (1992), *Scientific Visualization: Techniques and Applications*, Springer, New York.

Brodlie, K. (1994), A Typology for Scientific Visualization., *Visualization in Geographical Information Systems*, Hearnshaw, H..M., Unwin, D..J. (ed.), Wiley & Sons, New York.

Bruessler, U. (2000),*Development of an Integrated Information System for Archaeological Heritage Documentation*, Unpublished Ph.D. thesis

Carey, R. & Bell, G (1997), *The Annotated VRML 2.0 Reference Manual*, A-W Developers Press, New York.

Colombo, L. (1998), Photogrammetric Imaging for virtual Reality: an example of settlement documentation, *International Co-operation and Technology Transfer*, Vol. XXXII (6w4), pp. 111-116.

Davies, C. & Medyckyj-Scott, D. (1994) Introduction: The Importance of Human Factors, *Visualization in Geographical Information Systems*, Hearnshaw, H. M. & Unwin, D. J. (ed.), Wiley & Sons, England, pp 189-192.

Day, B. (1999), *3D graphics programming in Java, Part 1: Java 3D*, Online. Available: http://www.javaworld.com/javaworld/jw-12-1998/jw-12-media.html, 13 Feb 1999.

Foley, J. & van Dam, A. (1982), *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, USA.

Foley, J. and Ribarsky, W. (1994), Next-Generation Data Visualization Tools, *Scientific Visualization,* Academic Press, pp. 103-127

Forte, M. (1995) Scientific Visualisation and archaeological landscape: the case Study of Terramara, Italy, *Archaeology and Geographical Information Systems,* Lock, G. & Stancic, Z., Taylor and Francis Ltd Publishers, pp. 231-237.

Haber, R.B. & McNabb, (1990), Visualization Idioms: A Conceptual Model for Scientific Visualization Systems, *Visualization in Scientific Computing,* Gregory M. Nielson and Bruce Shriver, IEEE Computer Society Press, Washington, pp.74-93.

Hartman, J. & Wernecke, J. (1996), *The VRML 2.0 handbook – Building moving worlds on the web,* A-W Developers Press, USA.

Hayden, B. (1993), *Archaeology – The Science of once and future things,* W.H. Freeman and Company.

Java 3D On-line Interest Group, Archives 1997-99 Online. *Internet Resource,* Available: http://java.sun.com/products/java-media/mail-archive/3D/index.html, 23 Mar 1999.

Kalvin A, Remy A, Ardito O, Morla K, Nolasco E, Prado J, Franco R, Murga A (1996) Using Visualisation in the Archaeological Excavation of a pre-Inca Temple in Peru, *IEEE Visualization Conference*, Yagel, R. & Nielson, G. (ed.) pp 359-361.

Kraus, K. (1997), *Photogrammetry – Advanced Methods and Applications*, Vol. 2, 4th edition, Institute for Photogrammetry and Remote Sensing, Vienna University of Technology, Dümmler/Bonn.

MacEachren *et. al.*(1994), Introduction to Advances in Visualizing Spatial Data, *Visualization in Geographical Information Systems*, Hearnshaw, H..M., Unwin, D..J. (ed.), Wiley & Sons, New York.

McCormick, B., DeFanti, T.A., Brown, M.D. (1987), Visualization in Scientific Computing, *ACM SIGGRAPH Computer Graphics,* Vol. 21 (6)

Medyckyj-Scott, D. (1994) Visualization and Human-Computer Interaction in GIS, *Visualization in Geographical Information Systems*, Hearnshaw, H. M. & Unwin, D. J. (ed.), Wiley & Sons, England, pp 200-211

OpenGL Overview, OpenGL, Online. Available: http://www.opengl.org/About/About.html, 18 Aug 1999.

Palamidese, P., Betro, M., Muccioli, G. (1993), The Virtual Restoration of the Visir Tomb, *Proceedings Visualization '93*, Nielson, G. & Bergeron, B. (ed.), pp 420-423.

PC Webopedia, Online Dictionary, Online. Available: http://webopedia.internet.com/, 20 Aug 1999.

Reilly, P. (1992), Three-Dimensional modelling and primary archaeological data, *Archaeology and the Information Age: a Global Perspective*, Routledge, London, pp 147-173.

Robertson, P. & De Ferrari, L. (1994), Systematic approaches to visualization: Is a Reference Model Needed?, *Scientific Visualization*, Academic Press, pp.287-305

Senay, H. & Ignatius, E. (1994), A Knowledge-Based System for Visualization Design, *IEEE Computer Graphics and Applications,* Nov 1994, Vol:14, No:6, pp.36

Sims, D. (1997), Archaeological models: Pretty Pictures or Research Tools?, *IEEE Computer Graphics and Applications*, Potel, M. (ed.) Jan/Feb 1997, pp 13-15.

Schenk, T. (1994), Concepts and Algorithms in Digital Photogrammetry, *ISPRS Journal of Photogrammetry and Remote Sensing*, Vol. 4, No. 6, pp2-8.

Stynes, K. (1996), *Introduction to surface visualisation*, Online. Project Argus, Available: http://www.geog.le.ac.uk/argus/Tutorials/SurfViz/TerrainPages/Tools/index.html, 20 Aug 1999.

Sun Microsystems (1998), *The Java 3D API: White Paper,* Online. Available: http://java.sun.com/marketing/collateral/3d_api.html 11 Oct 1999.

Sun Microsystems, *Java 3D API Collateral*, Online. Available: http://java.sun.com/products/java-media/3D/collateral/j3d_api/j3d_api_3.html, 11 Oct 1999a.

Sun Microsystems, *The Evolution of 3D Graphics*, Online. Available: http://java.sun.com/products/java-media/3D/collateral/presentation/sld002.html, 11 Oct 1999b.

Taylor, S. (2000), *(Unknown Title)*, Masters Thesis.

Wehrend, S. & Lewis, C. (1990), A Problem-Oriented Classification of Visualization Techniques, *Proceedings IEEE Visualization '90*, October, IEEE Computer Society Press, pp.139-143.

Wood, M. & Brodlie, K. (1994), ViSC and GIS: Some Fundamental Considerations, *Visualization in Geographical Information Systems*, Hearnshaw, H. M. & Unwin, D. J. (ed.), Wiley & Sons, England, pp 3-8.

Appendix

OpenGL Program for a Sphere with a Light Source

```
#include <GL/glut.h>
#include <stdlib.h>
/*Initialize material property, light source, lighting model, and depth buffer.
*/
void init(void) {
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glShadeModel (GL_SMOOTH);

  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_DEPTH_TEST);
}

/*function for the display*/
void display(void) {
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glutSolidSphere (1.0, 20, 16);
  glFlush ();
}

/*function to reshape the object*/
void reshape (int w, int h) {
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
    glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
    1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
  else
    glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
    1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

/*assign keyboard functions*/
void keyboard(unsigned char key, int x, int y) {
  switch (key) {
    case 27:
    exit(0);
    break;
  }
}

/*Create the window for the 3D object*/
int main(int argc, char** argv){
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
  init ();
  glutDisplayFunc(display);
```

```
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return 0;
}
```

VRML Code for a Sphere with a Light Source

```
#VRML V2.0 utf8
Group {
  children [
    WorldInfo {
      title "Sphere Demo"
    }
  # Creation of a light source
  PointLight{
    location 3 3 3
    attenuation 1 0 0
    on TRUE
    radius 5
  }
  # Creation of a sphere
  DEF SPHERE Shape {
    appearance Appearance {
      material Material {
        diffuseColor 0.8 0.8 0.8
        emissiveColor 0 0 0
        shininess 0.2
        specularColor 0 0 0
        ambientIntensity 0.2
      }
    }
    geometry Sphere {
      radius 1
    }
  ]
}
```

Java Code for a Sphere with a Light Source

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.Sphere;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;

public class HelloUniverse extends Applet {
  public BranchGroup createSceneGraph() {
    // Create the root of the branch graph
    BranchGroup objRoot = new BranchGroup();
    // Create the transform group node and initialise it to the
    // identity.
    TransformGroup objTrans = new TransformGroup();
    objRoot.addChild(objTrans);
    Appearance appearance = new Appearance();
    Color3f AmbientColour = new Color3f(0.8,0.8,0.8);
    Color3f EmmisiveColour = new Color3f(0.1,0.1,0.1);
    Color3f DiffuseColour = new Color3f(0.5,0.5,0.5);
    Color3f SpecularColour = new Color3f(1.0,1.0,1.0);
    Material material = new Material(AbientColor, EmmisiveColour,
      DiffuseColour, SpecularColour, 50f);
    appearance.setMaterial(material);
```

```
    // Create a sphere and add it to the scene graph.
    objTrans.addChild(new Sphere(0.4,appearance));

    Point3f position = new Point3f(1.0f,1.0f,1.0f);
    Point3f attenuation = new Point3f(1.0f,0.0f,0.0f);
    //Create a light and add it to the scene graph
    PointLight light = new PointLight(true, SpecularColour, position,
      attenuation);
    objTrans.addChild(light);
    // Have Java 3D perform optimizations on this scene graph.
    objRoot.compile();
    return objRoot;
  }
  public HelloUniverse() {
    setLayout(new BorderLayout());
    GraphicsConfiguration config =
      SimpleUniverse.getPreferredConfiguration();
    Canvas3D c = new Canvas3D(config);
    add("Center", c);
    // Create a simple scene and attach it to the virtual universe
    BranchGroup scene = createSceneGraph();
    SimpleUniverse u = new SimpleUniverse(c);
    // This will move the ViewPlatform back a bit so the
    // objects in the scene can be viewed.
    u.getViewingPlatform().setNominalViewingTransform();
    u.addBranchGraph(scene);
  }

  // The following allows it to be run as an application
  // as well as an applet

  public static void main(String[] args) {
    new MainFrame(new HelloUniverse(), 256, 256);
  }
}
```