DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING

Development of a 3-D Headset Tracking Interface for True 3-D Visualisation

Shane Fox

August 2004

MASTER OF ENGINEERING

IN

ELECTRONIC SYSTEMS

Supervised by **Dr. Derek Molloy**

Acknowledgements

I would like to thank my supervisor Dr. Derek Molloy for his help, guidance and commitment to this project. Thanks are also due to Cathy for her advice, continuous support and excellent proof reading skills. Finally to my family and friends without who's support and encouragement the last few years in college would have been a great deal more difficult.

Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed: ..... Date: .....

Abstract

This thesis describes the interfacing of a Head Mounted Display (HMD) and Motion Tracking Device into both a test scene and virtual environment applications. Firstly, a stereo test scene is created to demonstrate the capability of the HMD to display stereoscopic images. The second part of the project involves developing the virtual environment application that has been partially developed by the Vision Systems Group. The application is being developed to allow OpenGL objects to be viewed and manipulated in a virtual environment. This project sets out to extend the application to incorporate stereo viewing and improved manoeuvring of the objects by interfacing the application with the HMD and motion tracker.

Firstly, the thesis explains the technique that allows humans to see in 3-D. Following this, a method for implementing stereoscopic vision in a computer environment is described in detail, together with the actual implementation of stereoscopic imaging into both the test scene and the virtual environment applications. The thesis concludes by discussing the effectiveness of stereoscopic imaging in a computer environment and the problems encountered during the implementation stage. Future extensions and improvements are also suggested.

Table Of Contents

Table of Figures

Chapter 1 – Introduction

This introductory chapter provides a general introduction to the dissertation. It begins by stating the motivation behind the project and describes the current state of 3-D interaction technologies. An insight into the common problems and difficulties associated with 3-D interaction are presented. The aims of the project are then outlined together with the approach taken. Finally, the structure of the dissertation is presented.

1.1 Background Information

1.1.1 Growing Demand for 3-D and Virtual reality applications

In recent years much work has been done in the area of virtual reality and three-dimensional (3-D) computer applications. There have also been great advancements made in 3-D graphics hardware and rendering systems. These advancements have allowed 3-D applications to be developed, which greatly enhance the user's experience of an application. 3-D and Virtual reality applications can essentially be used for any task from educational, through gaming and into extremely complex applications, which have serious usability problems such as flight simulation software. Two examples of 3-D software applications are shown below in Figure 1.1.

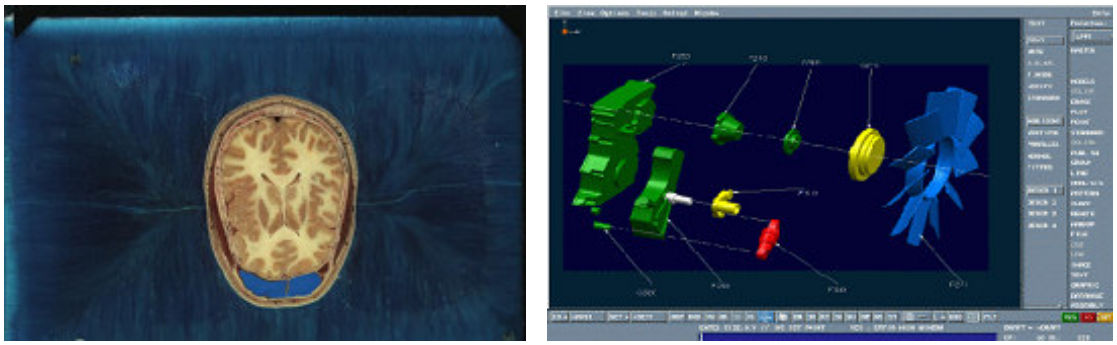

Figure 1.1: Examples of 3-D applications [1]

However, despite all the hype about these new technologies, neither technology has exploded into our lives and everyday use. The use of 3-D applications has increased reasonably in popularity mainly due to the ever-increasing gaming market and to a lesser extent in computer drawing applications such as "AutoCAD". A problem with these 3-D

applications is that the standard method of viewing a computer application is through the use of a monitor. This, however, is not adequate for modern 3-D computer applications as the user is restricted to viewing a 3-D scene on a conventional 2-D flat screen monitor, which nullifies the entire true 3-D experience. Modern computer software applications are also exceedingly advanced and can be cumbersome and complicated for the user to operate due to the large number of controls to operate. Thus, the regular method of interacting with a computer application through the use of a keyboard and mouse is no longer adequate for advanced computer applications, as the large number of controls to manipulate is much too difficult.

For true 3-D, a virtual reality environment is required to exemplify the experience and increase the user's interaction with this environment. This can be achieved through the use of a Head Mounted Display (HMD) and Motion Tracking Device, which allow the user to "step" into the virtual reality environment and view the environment in true 3-D, using stereo viewing. Tremendous progress in stereoscopic output devices started the move of Virtual Reality from research laboratories to industry, arcades and the home. Real applications where virtual reality systems could be employed are when:

- the real environment is hazardous or costly,
- the environment encompasses large virtual spaces,
- a large number of parameters are to be manipulated by the operator,
- tasks are of a hands-busy nature,
- the perspective is important.

1.1.2 3-D Interaction

In the previous section (Section 1.1.1), the increasing popularity of 3-D and virtual reality applications are discussed and also the hardware issues. The hardware issue is, however, not the main stunt in the development of virtual reality. There is still a large area of this new technology that is underdeveloped and requires much improvement if the technology is to progress. This area is how to interact with the virtual environments that have been created. The interaction within the virtual environment should allow users to perform tasks in three dimensions rather than being limited by the 2-D conventional graphical interfaces. Applications of immersive and desktop virtual environments, augmented reality and ubiquitous computing all require efficient and usable 3-D interaction. However, spatial interaction is not well understood and presents significant new challenges that are not addressed satisfactorily by traditional 2-D human-computer interaction [2]. The universal

interaction tasks of any virtual environment are outlined in “The Art and Science of 3D Interaction”, presented at the IEEE Virtual Reality'2000 [3]:

- navigation,
  - Travel
  - Way-Finding
- selection,
  - Choosing one or more objects from a set
- manipulation,
  - Specification of object position and orientation
  - Specification of scale, shape, etc.
- system control.
  - All other interactions usually accomplished via commands.

An example of a scientific visualisation 3-D software application, where an interface has been developed with most of the above interaction tasks (except Navigation), is the “3D-DOCTOR” (Figure 1.2). The 3D-DOCTOR is an advanced 3-D image processing, modelling and measurement software for MRI, CT, PET, microscopy, and industrial 3-D imaging applications [4].

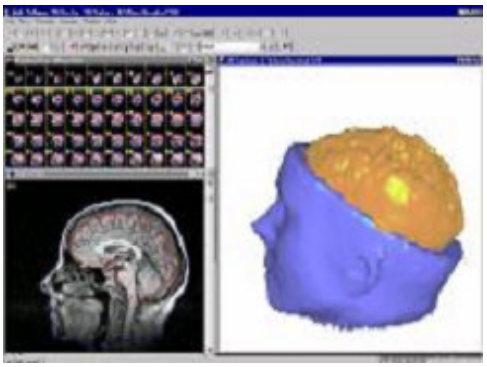

Figure 1.2: 3-D Doctor Application

1.1.3 Problems with 3-D Interfaces

However, despite all the advantages 3-D interfaces have to offer there are still major issues that have to be addressed. In a report from "The Challenges of 3D Interaction" workshop, it states that 3-D graphics applications are “significantly more difficult to design, implement and use than their 2-D counterparts” [Herndon, et. al. 1994]. In an experiment conducted by

Luczak's in 1991, the usability of 2-D and 3-D CAD systems was investigated. The results showed that users' performance with 2-D CAD systems was better than with a 3-D system and the average time spent on test tasks was about 55% longer for users of 3-D CAD systems than 2-D [5].

There are a number of reasons for this poor reflection of 3-D interfaces; below are listed some of the common problems that must be overcome to implement a user-friendly, multi-functional 3-D interface:

- additional dimension to manipulate and control,
- difficult to pick and place objects accurately,
- hard to map onto 2-D displays and devices,
- occlusion – where an object that blocks another is assumed to be in the foreground,
- calibration is difficult,
- lag makes people feel ill.

1.2 Aims of the Project

The aim of this project is to develop a 3-D Headset Tracking Interface for True 3-D Visualisation. This can be divided into two parts:

(i) To create a test scene, which demonstrates the stereo abilities of the HMD and utilises the Motion Tracking Device to further enhance the user's experience of being present in a virtual environment,

(ii) To expand the software environment that has been partially developed in the Vision Systems Group by developing an interface to view the scenes in true 3-D.

The software environment that the Vision Systems Group has developed is an application called OpenGLViewer. It allows an object to be viewed in 3-D and some basic manoeuvring of the object is provided such as rotation, zoom, etc. The main aim of this project is to develop an interface that allows the user to interact with the 3-D environment by providing features that permit the user to manipulate the image in a true 3-D environment. These features are provided through the use of both the HMD and a tracking device.

The interface is to be developed using the Microsoft Foundation Classes (MFCs). These classes provide a "Windows" style interface to an application and add a vast amount of

functionality such as status bars, drop-down menus, etc. OpenGL is the graphics software language that is used to extend the partially developed software environment by complementing its operation with the 3-D stereo headset and tracking device so the scenes can be viewed in true 3-D. OpenGL is also to be used to create a test scene for the 3-D stereo headset and tracker to demonstrate it's stereo capabilities in a true 3-D environment.

1.3 The Document Structure

This thesis is laid out as follows:

Chapter 1: This chapter presents a general introduction to the dissertation and states the motivation behind the project. The aims of the project are outlined and the general approach taken is described.

Chapter 2: The technical background for the project is described by outlining the equipment and software that is used in the project.

Chapter 3: The overall design and structure of the applications developed in the dissertation are provided. The design decisions for using the particular languages are discussed and the fundamentals of stereoscopic imaging are presented.

Chapter 4: This chapter describes the implementation of the test scene created and the developments made to the OpenGLViewer application to incorporate stereoscopic imaging.

Chapter 5: The results obtained from testing the two applications are presented. The successes and drawbacks of the final implementations and possible reasons for the problems encountered are discussed.

Chapter 6: Concludes the thesis by presenting a critical assessment of the overall successes and shortcomings of the project. Areas of future work are also provided.

1.4 Conclusion

The first chapter presents a general introduction to the project. It begins by stating the motivation behind the project and provides an insight into the problems associated with 3-D interaction. There follows an outline of the project together with the approach taken. Finally, the structure of the dissertation is specified.

The subsequent chapter outlines the technical background associated with the project by providing a description of the equipment used in the project and also an insight into the software languages used.

Chapter 2 - Technical Background

2.1 Introduction

This chapter provides the technical background for the project by describing the equipment, software development tools and software languages used in the project.

To begin with a summary of the equipment utilised in the dissertation is supplied, which outlines their purpose, functions and inclusion in the project. After that two development tools, Visual C++ and Visual Studio .NET are briefly summarised. Finally, the software languages used in the implementation of the project are outlined. These include OpenGL and the Microsoft Foundation Classes (MFC). The basics of both languages are provided to give the reader an insight into these technologies.

2.2 Equipment Used

Originally, in the interim report (submitted April 2004), three main pieces of equipment were outlined that would be employed in the implementation of the project. These included:

- the Head Mounted Display,
- the Intertrax^2 Motion Tracker,
- the LCD Shutter Glasses.

The LCD Shutter Glasses were to be included in the implementation of the project, if time permitted, but unfortunately this was not possible. This, however, is only a minor misdemeanour as the HMD is the main piece of equipment for viewing the stereo scenes and it is also a much more sophisticated device that provides better quality results.

2.2.1 The Head Mounted Display (HMD)

The model of HMD used in the project is the I-glasses SVGA display (Figure 2.1) supplied by I-O Display Systems. This plugs directly into the graphics card and allows the wearer to view an OpenGL stereo application in true 3-D (stereo). This type of device works by tricking the brain into thinking that it is viewing a 3-D image. Each eye views a slightly different image that the brain merges into a single image achieving stereovision just as the normal human visionary system provides this facility. The difference between the images is the position of their viewing point. Both images viewpoints are on the same horizontal axis but are slightly offset. This positional difference replicates the separation between a

human's eyes and allows each view to provide visual information the other does not. Stereoscopic imaging is discussed later in section 3.3.

The I-glasses SVGA display provides the following features [6]:

- resolution: 1.44 million pixels (800 x 600 RGB colour sequential system),
- input: SVGA (800 x 600) at 56-100Hz,
- display refresh rate: Double the input refresh rate up to 60Hz. Above 60Hz the input refresh rate equals the input refresh rate,
- field of view: 26.5 degrees diagonal,
- eye relief: 25mm,
- colour depth: 18 bit,
- convergence: 7.5 feet,
- focus distance: 13 feet at the centre,
- display: Stereo Supported,
- audio: Full Stereo Supported.

Figure 2.1: I-glasses SVGA HMD

2.2.2 The Motion Tracking Device

The final piece of hardware that is used in the project is the motion tracker called the Intertrax^2, which is developed by a company called "InterSense" see Figure 2.2. The Intertrax^2 is one of the smallest high performance head trackers available. It conveys extremely fast, accurate orientation tracking via the USB port of the PC and has an on-board microprocessor so all calculations are handled onboard and there is no burden placed on the host computer. InterTrax^2 is designed with the following features [7]:

- three degrees of freedom angular tracking (Pitch, Yaw and Roll),
- zero jitter and high stability,
- angular range: Pitch ±80°

Yaw ±180°

Roll ±90°,

- USB interface,
- on board microprocessor performs sophisticated algorithms to compute the orientation of the tracker in space,
- internal update rate: 256Hz,
- internal latency: 4 milliseconds,
- weight: 39 grams,
- power: 5 volts via USB

Figure 2.2: InterTrax2 Motion Tracking Device

This device will be used in conjunction with the HMD and can be sited on top of it. This allows the user's head motion to be tracked and these changes are used to update the view of the 3-D stereo scene. This enhances the clients's stereoscopic experience as natural head movements are used to look around the scene instead of keyboard or mouse movements and it provides the wearer with a sense of being present in the 3-D environment. The interface of the Motion Tracking Device is discussed in detail later in section 4.3.

2.3 The Development Tools

Two development tools are used in the project; these are Microsoft Visual C++ 6 and Microsoft Visual Studio .NET 2003. The reason two development tools are used is because the first part of the project, which involves developing a stereo test scene for the HMD and motion tracker, is developed in Visual C++. However, it became necessary to switch to the .NET development tool as the software environment that has been partially developed by the Vision Systems Group has been designed using the .NET tool.

Both the Visual C++ package and .NET include a C++ compiler, a debugger, a resource editor and the Microsoft Foundation Classes (MFC) libraries. The MFC libraries are used to add functionality to the applications and are discussed in more detail in section 2.5. Visual Studio .NET also offers the Microsoft .NET Framework runtime environment, which provides powerful tools for designing, building, testing and deploying applications.

2.4 OpenGL – The Industry Standard Graphics API

OpenGL is a software interface to graphics hardware [8]. It is a powerful modelling and low-level rendering software library, available with extensive hardware support and allows the rendering of multidimensional objects into a framebuffer. It deliberately only provides low-level rendering and modelling capabilities, which is advantageous to the programmer as it allows great flexibility and control. OpenGL provides the programmer with a set of functions that can be used to produce interactive 3-D applications. In Figure 2.3 below, a screenshot from an OpenGL based computer game highlights the power and the degree of realism that can be achieved using OpenGL.

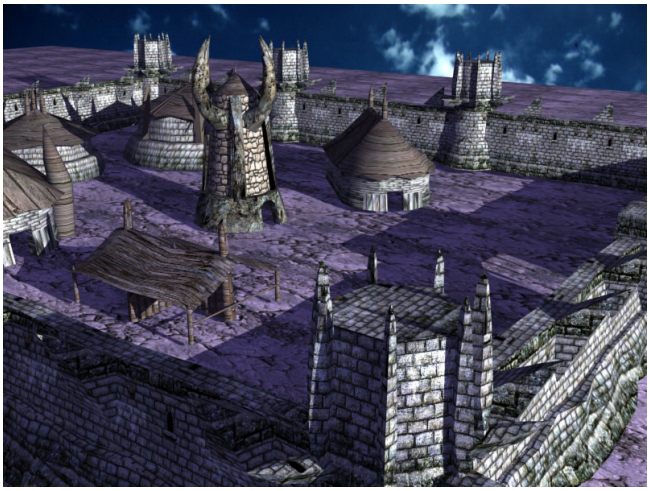

Figure 2.3: Screenshot of an OpenGL based computer game

2.4.1 Brief History of OpenGL

The OpenGL API began as an initiative by Silicon Graphics, Inc. (SGI) to create a single vendor and platform independent API, for the development of 2D and 3D graphics applications. Prior to the introduction of OpenGL, there was no standard graphics library and hardware vendors had their own specific graphics libraries. This made it very expensive and difficult for software developers to support versions of their applications on multiple hardware platforms, hence the need for and development of OpenGL. OpenGL

solves this problem as it is an open source API and is thus vendor and platform independent.

In 1992 the development of OpenGL was handed over to an independent consortium called the OpenGL Architecture Review Board (ARB). The ARB consists of major graphics vendors and other industry leaders. Its current members, as of October 2003, are *3Dlabs, Apple, ATI, Dell Computer, Evans & Sutherland, Hewlett-Packard, IBM, Intel, Matrox, NVidia, SGI* and *Sun*. The primary role of the ARB is to establish and maintain the OpenGL specification by managing the features of the OpenGL specification [9].

2.4.2 OpenGL Basics

OpenGL is designed to be an efficient, hardware and platform independent graphics interface. OpenGL achieves this by only providing a small set of geometric primitives - points, lines and polygons, however this is not a major hindrance as these geometric primitives can be used to build high-level rendering and modelling libraries. The OpenGL Utility Library (GLU) is one such library that is included as part of the OpenGL implementation. GLU provides features that range from simple wrappers around OpenGL functions to complex components supporting advanced rendering techniques [10].

OpenGL also provides no commands for performing windowing tasks or obtaining user input. However, these operations can be implemented using another external library called GLUT. This library is discussed in section 2.4.4.

2.4.3 OpenGL as a State Machine

OpenGL internally acts as a state machine. This means that it can be put into various states that remain in effect until they are changed. For example, setting the current colour to red ensures that all objects are drawn in red until the colour is changed again. States are set using the API to change the state variable that OpenGL preserves for a particular state. Objects are created with attributes taken from the current state along with the current transformation. The state also specifies which drawing and rendering operations take place. Other common states control the current viewing and projection transformations, drawing

modes, light characteristics and material properties of the objects been drawn. When rendering it is vital to ensure that all states are set correctly as unexpected results can occur.

2.4.4 OpenGL Utility Toolkit (GLUT) Library

GLUT is a utility library that addresses the difficulties caused from the fact that OpenGL does not provide routines for interfacing with a windowing system or input devices. GLUT provides methods that allow this functionality to be implemented into an application and still remain platform independent. GLUT is designed for constructing small to medium sized OpenGL programs, as it is not a full-featured toolkit. The latest version of GLUT and the version used in this project is GLUT 3.7 [11].

2.4.5 Support and Documentation

Due to the fact that OpenGL has been around for a number of years now and because it is so widely used throughout the industry, there is much support available. There are a large number of tutorials available on the Internet, which facilitate beginners in learning the basics of OpenGL.

The official Website for OpenGL is OpenGL Org [12], which provides a large amount of useful information about OpenGL such as documentation, coding resources and sample applications. It also includes many links to other sites with tutorials and references to OpenGL books that are available.

Another invaluable Website that provides excellent OpenGL tutorials is the *NeHe Productions* Website [13]. This site has a large selection of OpenGL tutorials that progress from the basics of OpenGL to very advanced OpenGL topics.
Two books are used in this project; the *OpenGL Programming Guide* [8] and *The OpenGL Super Bible* [14]. Both provide detailed insights into the language and are great learning aids.

2.5 The Microsoft Foundation Classes (MFC)

The MFC library is a set of data types, functions, classes, and constants used to create applications for the Microsoft Widows family of operating systems. When writing a Windows program you create objects of these predefined MFC classes. These objects allow an application to communicate with Windows, process Windows messages and send messages to other MFC objects. MFC uses C++ to simplify Windows programming by making use of C++ encapsulation and inheritance utilities.

In the early days of MFC, applications were built with two primary components: an application object representing the application itself, and a window object representing the application's window. The primary duty of the application object was to create the window object, which in turn, processed messages. Thus MFC was little more than a thin wrapper around the Windows API, i.e. it attached an object-oriented interface onto Windows enabling dialog boxes, etc., which were already present in Windows in some form.

MFC 2.0 made a major breakthrough by introducing the Document/View architecture, which is still used in the current version of MFC and is discussed later in section 2.5.3.

2.5.1 MFC Basics

To program an MFC application new classes are derived from the architectural classes in the MFC library. These new classes can inherit and replace existing behaviour as appropriate and are known as an application framework. As a result of using application-framework classes to drive a program and wrapper classes to access Win32 objects, programs written using MFC are a bit like a sandwich. The application supplies the filling while MFC supplies the bread (see Figure 2.4) [15].

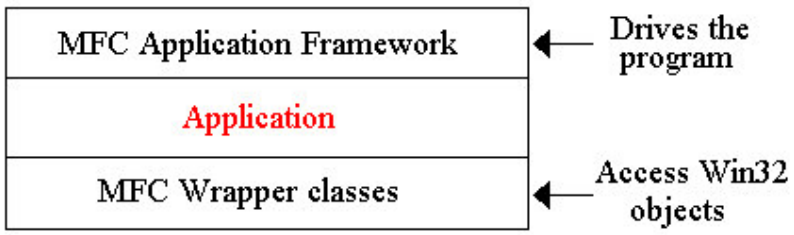

Figure 2.4: MFC Application Structure

The main classes in MFC are:

- **CObject** - Most classes are derived from this class,
- **CWinApp** – Called whenever an application is created but only gets called once. It handles the event loop activity and initialisation,
- **CWnd** – Collects all the common features found in windows, dialog boxes and controls,
- **CFrameWnd** – Derived from CWnd and is used to implement a normal framed window application,
- **CView** – Allows a user access to a document through a window.

2.5.2 The AppWizard

The AppWizard is the simplest way to create a new MFC application. The AppWizard is present in both Visual C++ and Visual Studio .NET. Creating a windows program involves setting up the same basic, error-free code for almost every application. The AppWizard removes all the drudgery of creating a new project by removing all the tedious steps and by simply specifying the major properties of the new application; it generates the basic code for the application. This code can then be modified to create a customised application.

The AppWizard can generate several types of applications, all of which use the application framework in different ways. Single Document Interface (SDI) and Multiple Document Interface (MDI) applications make full use of a part of the framework called the Document/View architecture, which is discussed in the next section It is, however, not necessary to use the Document/View architecture in an MFC application but there are numerous advantages for doing so.

2.5.3 The Document/View Architecture

The Document/View Architecture is the cornerstone used for creating MFC-based applications. It allows a separation to be made between the various parts that constitute a computer program; this includes the code that manages an application's data and the code that presents the data to the user. MFC does this by using numerous different classes that work together as an assembly. The three main parts of the Document/View architecture are a frame, one or more documents and the view.

In the Document/View model, the document contains the code, which manages the application's user data. The document part of the architecture is represented by a document class, which is created by deriving an object from the **CDocument** class in MFC. A document has two primary functions; to provide a programmatic interface for accessing the document data and to save and restore the data.

The view is the part of the architecture the user sees and works with. It is the code that draws the data on an output device for the user to manipulate. A view must be associated with a document. The view can also be more than just a display mechanism for a document; it can also be a two-way interface between the document and the user, thus, the view provides both the output of information from the document and accepts input from the user for the document. For this to occur the document must provide a way of encapsulating its data for the view to access [16]. A view can be created by either directly using one of the derived classes from **CView** or deriving a custom class from **CView** or one of its child classes.

The final component of the Document/View architecture is the frame. Every application must have at least one frame. The view implemented by an application is a borderless child window, which contains no menu of its own. A frame is thus required to place the view window within and allow it to have a physical workspace. This allows the user to move, size and resize the object. There are two types of frame windows, which include, a single document interface (SDI) and a multiple document interface (MDI) frame window. Figure 2.5 illustrates the main components of the Document/View architecture.

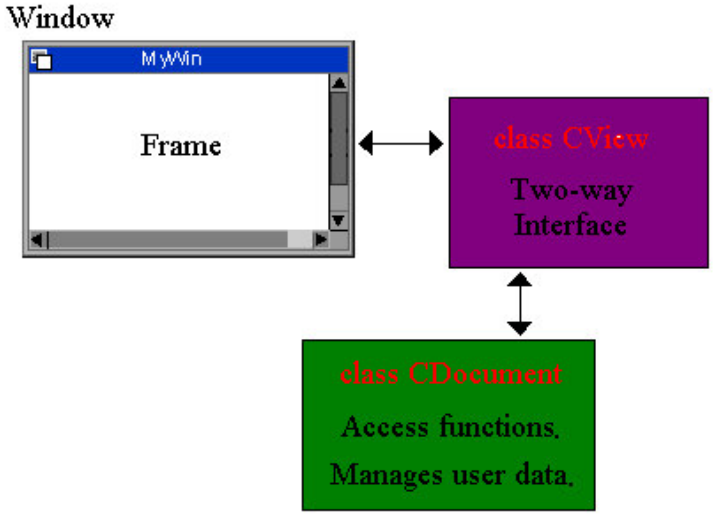

Figure 2.5: The Document/View Architecture

2.5.4 Messages and Events

The Microsoft Windows operating system leaves it to each object within an application to make requests for resources, etc. it may require as this takes the burden off the operating system and places it on the application. The objects make requests by sending messages to the operating system. There are many different types of messages and in MFC all messages begin with **WM_**, which stands for Window Message.

Only windows can receive messages and a complicated framework is used to implement the command-routing mechanism that routes command messages. Command messages (**WM_COMMAND**) are generated when menus, toolbar buttons, and other user interface events occur. The frame window handles most command messages but they are not confined to being dealt with in just the frame class. An ID associated with the menu item, etc. is passed with the message as a parameter. The window procedure for handling the message will have a case for **WM_COMMAND** and a handler for the parameter.

In order to send a message, a control must create an event. To distinguish between the message and the event, a message's name as stated earlier begins with **WM_**, whereas the name of an event usually starts with **On** (e.g. **OnClose()**), which indicates an action. To summarise, the message is what needs to be sent and the event is the action of sending the message.

2.6 Conclusion

This chapter begins by introducing the hardware devices that are used in the project, these include the HMD and the Motion Tracking Device. A description and the specifications of each device are given. Following that the two development tools, Visual C++ and Visual Studio .NET used in the dissertation are briefly outlined.

The fundamentals of the two main software languages, OpenGL and MFC, which are used in the implementation of the project, are then examined. This provides a background of basic subject matter in these languages, which are discussed subsequently in more detail in chapters three and four.

In the following chapter (Chapter 3), the overall design and structure of the applications developed in the dissertation is provided. This includes the design decisions for using the particular languages. The fundamentals of stereoscopic imaging and a method for implementing stereo in a software environment are described.

Chapter 3 - Design Architecture

3.1 Introduction

This chapter begins by discussing the major design decisions made within the project before any of the implementation could be carried out and provides reasons for the decisions taken.

Firstly, the motives for using the various software languages are explained and the advantages they provide over other similar languages are presented. Following that, the method of implementing stereo is revealed in detail. This involves describing the fundamentals of stereoscopic imaging and the technique used for applying stereo in this project.

3.2 Software Design Decisions

There are many design decisions made throughout this project but one of the first and possibly most critical involves deciding on the software languages to design the applications with. This obviously would have an effect on the project as a whole, as many aspects of the applications to be implemented would be affected. Aspects such as the functionality, design, speed of operation, appearance of the applications and many more features are influenced by these decisions. The languages finally chosen are C++ using MFC and for the graphics language, which is used for the all the 3-D rendering and modelling, OpenGL is chosen. The reasons for choosing these languages are discussed in the next two sections (Sections 3.2.1 and 3.2.2).

3.2.1 Reasons for choosing C++ and the Microsoft Foundation Classes

The primary reason for using C++ is simply to utilise its performance capability. Compared to Java, C++ is much faster. C++ and Java have some similarities, such as Local Method Declarations and Method Overloading but there are also many differences. The major examples of their differences are:

- no pointers in Java,
- global variables are not allowed in Java,

- the method of referencing objects is different, in C++ the default is by value whereas in Java the default object manipulation method is by reference,
- no multiple inheritance in Java,
- C++ has a preprocessor,
- Java does not facilitate operator overloading.

For the requirements of the project, C++ is chosen over Java due to its many features, power and speed, it would be the most suitable language in which to implement the project. C++ was chosen over C (despite C being faster again) because of its object-oriented approach to programming, which in a project of this scale simplifies the management of the program's structure and allows a more suitable structured model of the application to be developed. Another key reason for choosing C++ is because it incorporates the MFC libraries that are used to develop the interface for the applications in a Microsoft Windows style, a feature that Java is lacking.

The major factor in choosing to use MFC is also due its speed when compared to other similar languages. Furthermore, MFC is extremely powerful and saves a large amount of coding time. The key advantages of MFC are [17]:

- substantially reduces the effort to write an application for Windows,
- execution speed comparable to that of the C-language API,
- minimum code size overhead,
- easier use of the Windows API with C++ than with C,
- easier-to-use yet powerful abstractions of complicated features such as ActiveX, database support, printing, toolbars, and status bars.

However, by using MFC the application's portability will be limited to the Microsoft Windows operating system as MFC is not platform independent. This design decision can be justified by the fact that Microsoft Windows is the predominant operating system and this problem could be overcome by allowing the user to disable the MFC functionality of the application, hence allowing it to operate on most platforms.

3.2.2 Motives for using OpenGL

There are many graphics languages available on the market today, as the industry has grown immensely in the last few years. The primary contenders are, however, Microsoft's Direct3D, which is part of the DirectX suite of libraries, and an open standard language by Silicon Graphics Inc., called OpenGL. After much scrutiny and deliberation of the two languages, OpenGL was chosen as the graphics language, which best suits this project.

The primary reasons for choosing OpenGL over Direct3D are listed below:

- operating Speed of OpenGL is very fast compared to other similar software, as functions are built into the graphics card,
- it allows low-level rendering of images, which other similar products do not,
- OpenGL is the Industry Standard**.** It can be used on multiple platforms, and is vendor neutral. No other graphics API operates on a broader range of hardware platforms and software environments. Whereas Direct3D is only compatible with Microsoft Windows operating systems,
- OpenGL is hardware independent, as it provides no commands for performing windowing tasks or obtaining user input,
- OpenGL is very stable. Any proposed changes have to be approved by the OpenGL Architecture Review Board, which ensures that any changes made are in the interest of all the hardware vendors.
- the OpenGL architecture makes it easy to port OpenGL programs from one system to another. It is also an open source API, thus making it portable and vendor independent.

3.3 Stereoscopic Imaging

Most people take stereoscopic vision for granted, as seeing in true 3-D is completely natural for humans. It is only when one looks at a computer or television screen or other such devices that it becomes apparent there is something missing. These devices are of course missing the third dimension or depth dimension and thus everything appears very flat even when displaying 3-D images. To overcome this problem and add greater reality to these devices, a number of ways of adding stereoscopic imaging have been researched in recent years and are continually improving.

Before implementing stereoscopic vision on a computer screen or other such device, one must understand just how the human vision system allows 3-D vision. The key to stereoscopic vision is having two eyes or two views of the same scene. First both eyes must focus on the same target, this is known as binocular vision. However, because a small horizontal distance, called the interocular distance, offsets the eyes, each eye sees a unique view of the scene from its own perspective. The two slightly different images (called a stereo pair) are sent to the brain, which integrates the images into a single stereoscopic image that gives the perception of depth.

3.3.1 Viewing Stereo Effects

There are many ways of viewing stereo pairs; one of the first methods used in the early days of computer graphics was to use colour filter glasses (red/blue glasses) to view a superimposed image. This superimposed image consisted of a left and right view of the image that had been turned into the corresponding colour of the left and right lens of the glasses. When viewed through the red/blue glasses it gave a crude effect of a stereoscopic image, but the major drawback with this technique is that only grey scale images can be viewed. Figures 3.1 and 3.2 below show an example of this technique taken from the mars lander [18].

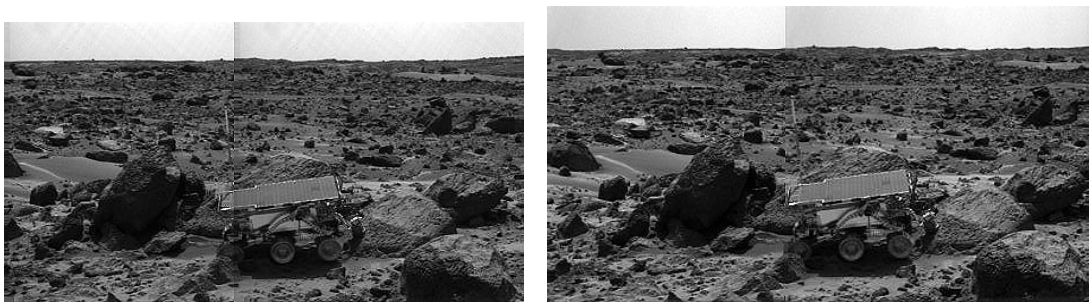

Figure 3.1: (a) Left image (b) Right image

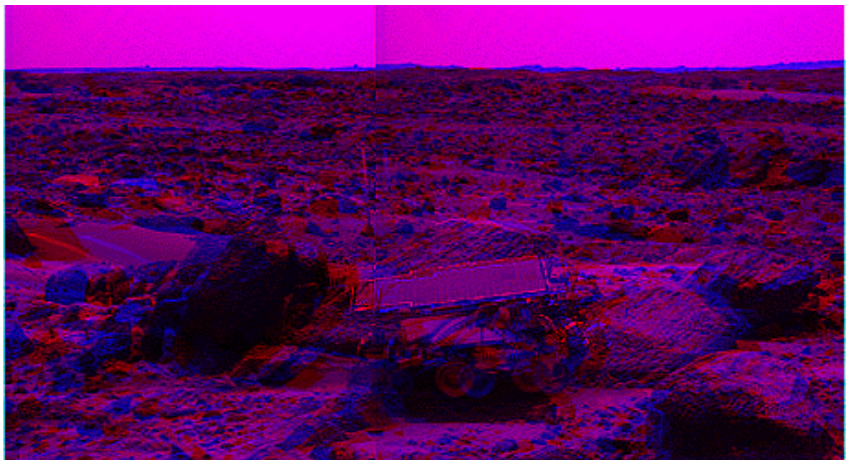

Figure 3.2: Red/Blue integrarated image

Another, more sophisticated approach, is to display the left and right images alternative on a high refresh rate monitor. A pair of LCD Shutter Glasses is then used to view the stereoscopic image on the monitor. These glasses allow the correct image for each eye to be viewed by alternatively blocking a lens of the glasses in synch with the image being displayed on the monitor. If the images are refreshed fast enough, the result is a flicker free stereoscopic image, however, very fast refresh rates are required to completely eliminate the flickering problem. This approach is described in the original interim report for the project and would be implemented if time allowed but unfortunately this was not possible.

Probably the best approach for viewing stereoscopic images is through the use of a HMD as shown previously in Figure 2.1. These devices are normally helmet-like and have two separate displays so each eye can view the image intended for it, thus creating the stereoscopic effect. The advantages of this approach are that it offers a better quality and more realistic 3-D effect than any other method. Furthermore, if used in conjunction with a Motion Tracking Device looking around the scene can be controlled by the user's head motion, which further increases the virtual reality affect. Due to the above reasons this approach is implemented in this project.

3.3.2 Retinal Disparity

Retinal Disparity is the difference in the images projected onto the retina of the eye due to the interocular distance. Retinal disparity is caused by the fact that each of our eyes sees the world from a different viewpoint. For example, in if both eyes are fixated on a point, F1, then an image of F1 is focused at corresponding points in the centre of the fovea of each eye. Consider another point, F2, at a different location. F2 would be imaged at points in each eye that may not be the same distance from the fovea. This difference in the distance is the disparity [19]. Figure 3.3 demonstrates retinal disparity.

For adults, the average eye separation is two and a half inches or 64 millimetres apart. The disparity is processed by the brain into a single stereoscopic image. This ability of the brain to integrate two different, although similar, images is called fusion and the resultant sense of depth is called stereopsis[1] [20].

[1] Stereopsis is a Greek work, which means, "solid seeing".

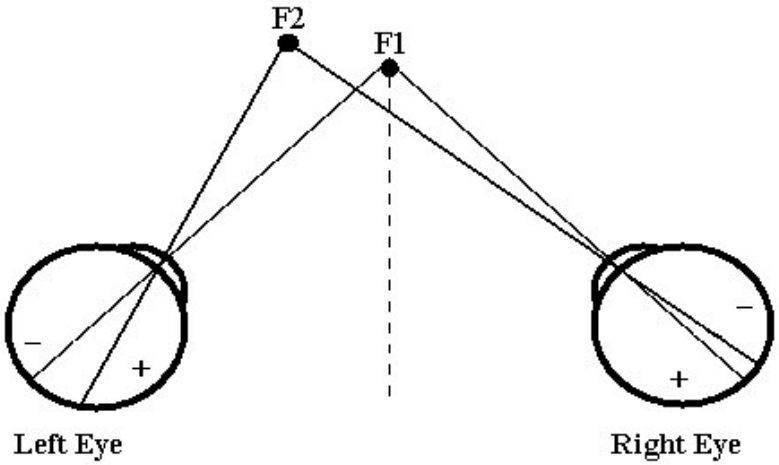

Figure 3.3: Disparity [19]

If an object is closer than the fixation point, the disparity will be a negative value. This is known as crossed disparity because the two eyes must cross to fixate the closer object. The opposite of this is uncrossed disparity. This occurs when the object is farther than the fixation point, thus the disparity is a positive value and the two eyes must uncross to fixate the further object. These two cases of disparity are illustrated in Figure 3.4 below. The final case of disparity is called zero disparity, which occurs when the object is located at the fixation point.

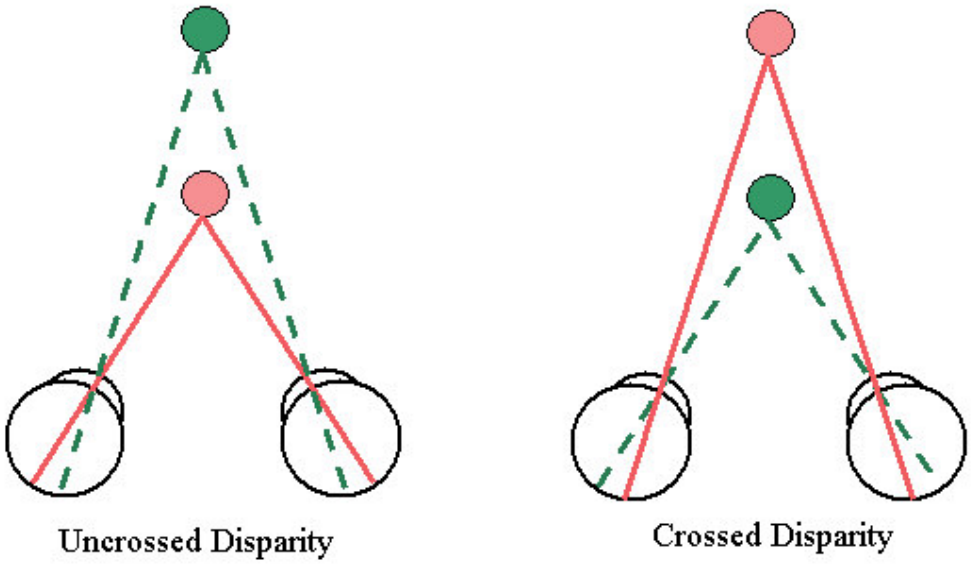

Figure 3.4: Types of disparity [21]

3.3.3 Parallax

Following retinal disparity, the next key element in producing the stereoscopic depth effect is called parallax. Parallax is the horizontal distance between corresponding left and right image points measured at the display screen. The display screen in question is a stereoscopic display, which differs from a planar display, as it is able to display parallax values of the image points. The stereoscopic image as stated previously, is composed of two images generated from two related perspective viewpoints, and it is these viewpoints that are responsible for the parallax content of the view.

Parallax and disparity are closely related entities. It is parallax, which produces retinal disparity, and disparity in turn produces stereopsis. Parallax is measured at the display screen, whereas disparity is measured at the retina. Parallax may also be given in terms of angular measure, which relates it to disparity by taking into account the viewer's distance from the display screen [20].

Parallax is thus the key to creating stereoscopic images. In this project the HMD is the device used to implement parallax, which in turn induces retinal disparity in the eyes and hence produces the stereo effect.

There are three types of parallax; the first case is zero parallax, which is illustrated in Figure 3.5 below. In this case the image points of the two images (stereo pair) exactly coincide or lie on top of each other. Thus, when the observer's eyes look at the display screen they are converged at the plane of the screen, hence zero parallax.

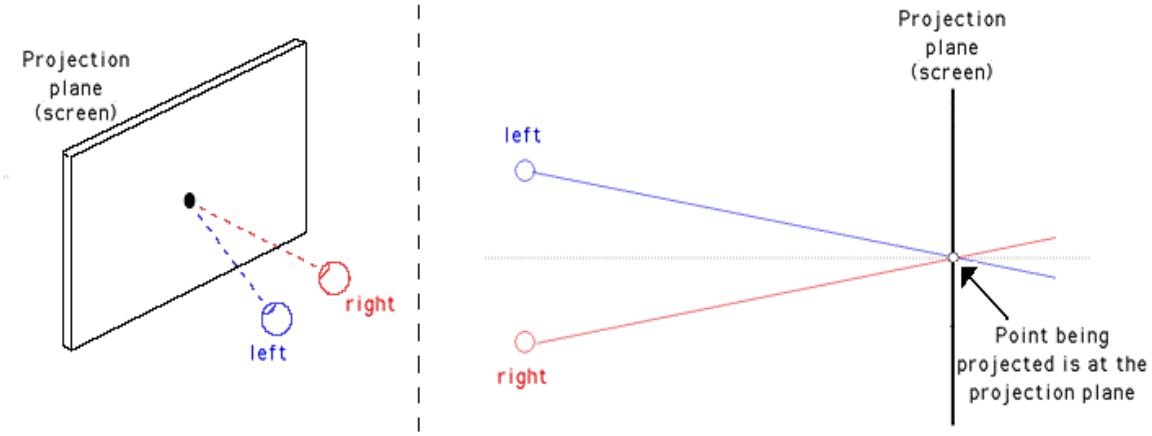

Figure 3.5: Zero Parallax

The second type is positive parallax or uncrossed parallax, which occurs when the axes of the left and right eyes do not cross each other when their view is fixed on an object. Any positive parallax between zero and the interocular distance produces images that appear to be behind or into the display screen. However, stereo pairs with parallax values close or equal to the interocular distance produce discomfort for the viewer when viewed on a small screen. Figure 3.6 below, demonstrates positive parallax.

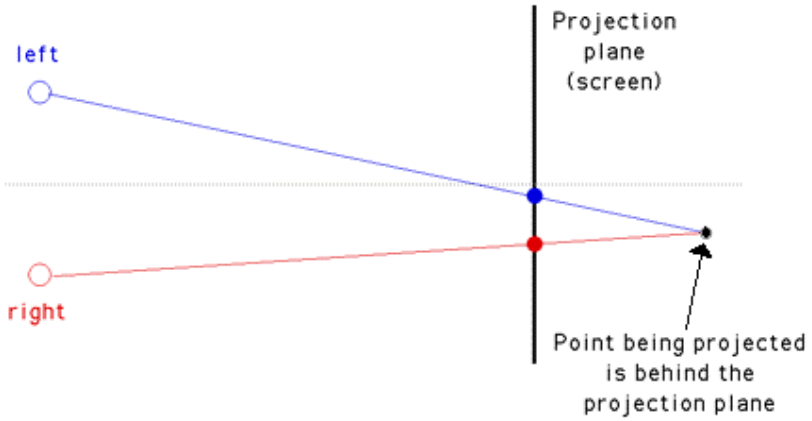

Figure 3.6: Positive Parallax

The final type of parallax occurs when the eyes axes have to cross to focus on the object, this type of parallax is called negative or crossed parallax. Objects with negative parallax appear closer than the plane of the display screen; they appear to pop out of the screen. This is shown below in Figure 3.7.

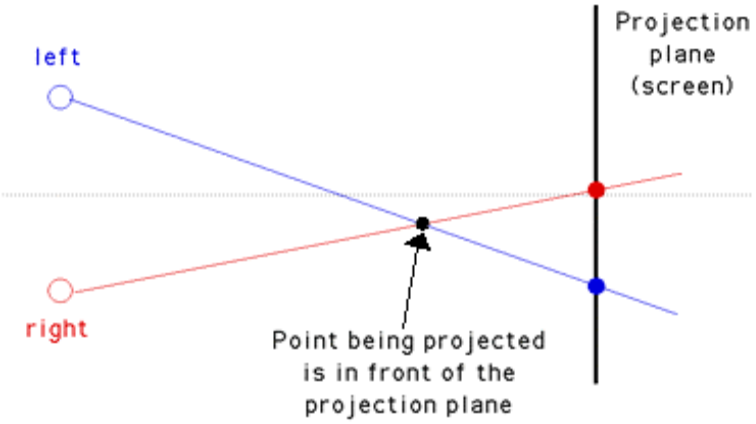

Figure 3.7: Negative Parallax

3.3.4 Interleaved Stereo Images

The technique of interleaving stereo pairs has been in use since almost the beginning of stereoscopic imaging research. This is due to the fact that the standard television formats available (PAL and NTSC), allowed it to be transmitted using existing hardware. The odd and even fields of these formats accommodated the stereo pairs, as the left and right images could be interlaced into the odd and even fields.

There are two principal techniques to extract the stereo pairs from the interleaved image signal and present them to the correct eye for stereoscopic viewing. The original method is to use a pair of LCD Shutter Glasses that alternatively block one eye, whilst presenting the other eye with its correct image. This method has been discussed previously in section 3.3.1. The problems with this method are flickering, due to halving the refresh rate and also poor quality images, as the resolution has also been halved. The three Figures below 3.8, 3.9 and 3.10, are an example of an interleaved image.

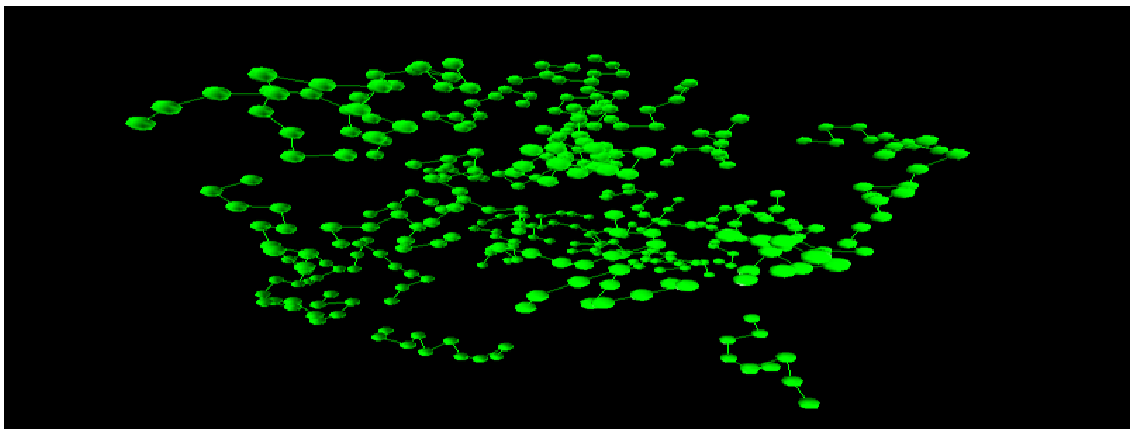

Figure 3.8: Left image

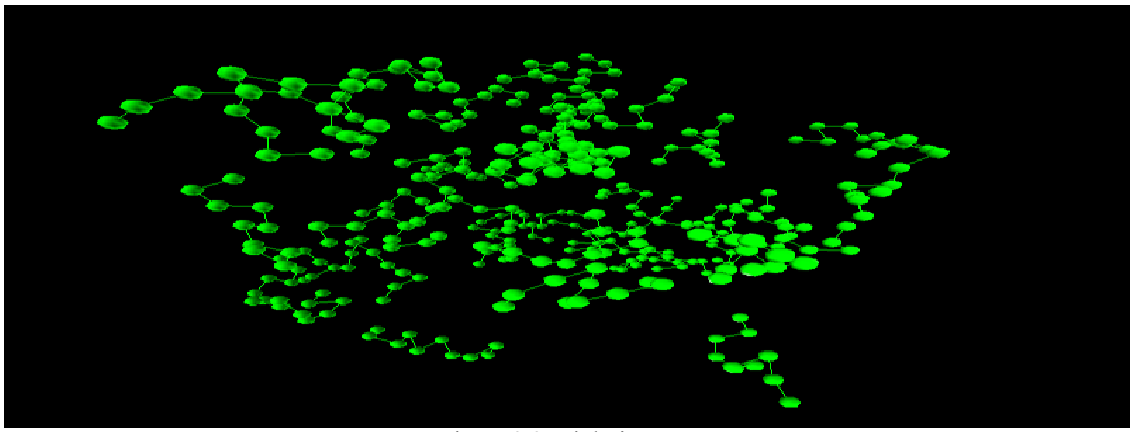

Figure 3.9: Right image

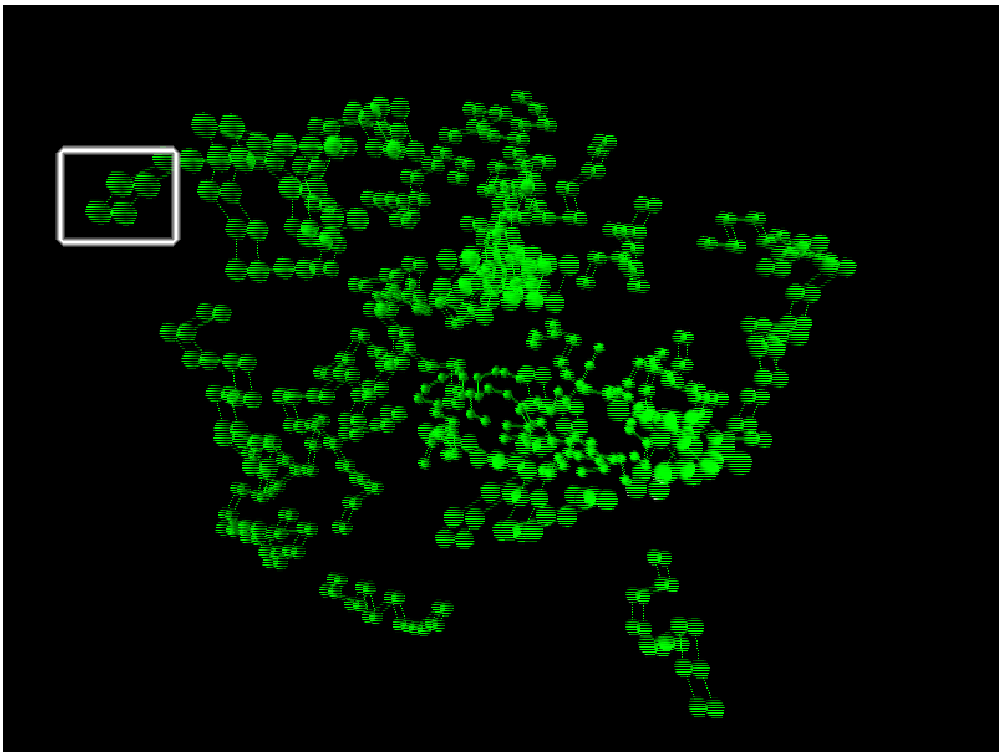

Figure 3.10: Interleaved image

The second technique for extracting the stereo pairs from the interleaved image signal is to use a HMD with two Liquid Crystal (LC) displays. The operation of the HMD is also outlined in section 3.3.1; however, the advantage of using this device with the interleaved image signal is not discussed. This advantage is due to the persistence of LC displays, which results in an almost flicker-free image but the reduction in the resolution is still present.

The interleaving technique is not, however, used in this dissertation. Instead a similar technique, which produces better quality images as the resolution is not reduced, is implemented. This technique, which is portrayed in the next chapter, section 4.2, does not compress the stereo pairs into a single interleaved image, although when displayed on a standard display screen (as opposed to the HMD), the alternating stereo pairs generate a similar image to that shown in Figure 3.11 above, but without the loss of vertical resolution. The actual image displayed is extremely difficult to capture as it is composed of two separate images with only one being displayed on every frame update. Figure 3.11 is also included to demonstrate the effect of alternating two similar images on a display screen and although it is very difficult to illustrate on a 2-D plane, a sense of depth can to some extent be perceived.

3.4 Conclusion

In this chapter some of the major design decisions made within the dissertation before any of the implementation could be carried out are discussed. These mainly involve the software languages used to implement the two applications. Following this, stereoscopic imaging is described in detail.

In the succeeding chapter (Chapter 4), an in-depth disclosure of the implementation of first the stereo test scene and then the OpenGLViewer application is presented.

Chapter 4 – Implementation

4.1 Introduction

This chapter describes the implementation of the test scene created to demonstrate the capability of the HMD to display stereoscopic images, and also the developments made to the OpenGLViewer application to incorporate a stereo mode.

First the method of exercising stereoscopic imaging is depicted and the incorporation of this technique into the two applications is portrayed. Following this, the interface of the Motion Tracking Device is described in detail. The preceding section explains the technique used to implement full-screen and this leads into an outline of the development of the stereo test scene. This involves using many OpenGL features, such as texture mapping to create a 3-D mountain scene that can be viewed in stereo using the HMD.

Finally, this chapter covers the expansion of the OpenGLViewer application by enabling it to incorporate stereo viewing and the use of the motion tracker to extend its movement capabilities.

4.2 Implementing Stereo Rendering

To implement stereoscopic imaging, various hardware devices are required. In this project, the major device, which has already been discussed in previous sections of the dissertation, is the HMD. The next essential piece of hardware is the graphics card. A modern, powerful, graphics card is necessary, which includes quad buffers and can function in stereo mode. The graphics card, which this dissertation employs, is the GeForce4 Ti, produced by NVidia. This card provides much flexibility regarding its stereo properties and allows adjustments to be made to many stereo settings. For example, enabling stereo can be set to toggle mode or it can be permanently switched on.

One of the first stages of the project involved setting up the graphics card to enable stereo mode. This involves installing a driver for the graphics card that allows stereo. Most graphics card manufactures provide these stereo drivers, as downloads from their Websites; in this project's case the Website is: http://www.NVidia.com. After installing the driver,

testing was carried out to ensure that stereo was operating correctly. NVidia includes a test for stereo in the driver software. This test was carried out and the graphics card and driver were found to be fully stereo compliant.

4.2.1 Quad Buffering

Quad buffering as the name suggests involves the use of four buffers and is the ability of the graphics card to write to a separate buffer on request. In OpenGL this request is made through the use of the `glDrawBuffer` command. Quad buffering is necessary to implement stereovision.

In normal non-stereo mode, OpenGL uses standard double-buffering. Double-buffering involves a front and back buffer. Typically, the drawing is done to the back buffer and when complete, it is swapped to the front buffer or display buffer for displaying using the function `SwapBuffers()`. If double-buffering were not used, the program would display the drawing of the frame as it happens, hence animations would appear extremely jagged. This occurs, due to the fact that the computer is not quick enough to clear the screen, draw and display the next frame, without a noticeable difference to the human eye. It simply cannot produce a high enough refresh rate to acquire reasonable quality animations.

For stereoscopic images, both the left and right views (the stereo pair) must be displayed to each eye simultaneously. Thus, for smooth animations on a stereoscopic display, an additional two buffers or double-buffered memory is required, hence the term quad buffering. The two double-buffers are divided into left and right buffers. The front left and back left buffers, handle the frames for the left eye. Similarly, for the right eye, the front and back right buffers are used. In practice, the `glDrawBuffer()` command is used with a parameter specifying the correct buffer to draw to. The parameters this function accepts are `GL_FRONT`, `GL_BACK`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT` and `GL_BACK_RIGHT`.

4.2.2 Setting up a Stereo Window

To create a stereo window using the OpenGL API, it is necessary to utilise OpenGL's stereo buffering support. Firstly, it is good practice to check that the graphics hardware can support quad buffering before initialising a stereo window. This can be achieved by querying the graphics driver using the following code:

```
unsigned char ucTest;
bool g_bStereo;

glGetBooleanv (GL_STEREO, &ucTest);

if (ucTest){        // yes stereo supported
      g_bStereo = TRUE;
      MessageBox(NULL,"Stereo Available","Stereo
      Status",MB_ICONINFORMATION);
}else{        // stereo support not available
      g_bStereo = FALSE;
      MessageBox(NULL,"Stereo UnAvailable","Stereo
      Status",MB_ICONINFORMATION);
}
```

If stereo buffering is supported by the graphics hardware, this code returns true and it is then safe to initialise an OpenGL stereo window. There are two means to initialise a stereo window. The first technique involves using a Microsoft Window's API call that supports OpenGL. This call is to the `SetPixelFormat()` function and is used in conjunction with a format descriptor flag `PFD_STEREO`, which enables stereo mode. The code for implementing this is shown below:

```
PIXELFORMATDESCRIPTOR pfd;
pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR); //Size of the structure
pfd.nVersion = 1;     // Always set to one.

// Pass in the appropriate OpenGL flags.
pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
PFD_DOUBLEBUFFER | PFD_STEREO;

int iPixelFormat;
// Gets a pixel format that best matches that described in pfd
if( (iPixelFormat = ChoosePixelFormat(m_hDC, &pfd)) == FALSE )
{ // Displays error message on failure
      MessageBox(NULL, "ChoosePixelFormat failed", MB_OK);
      return FALSE;
}
// This sets the pixel format that we extracted from above
if (SetPixelFormat(m_hDC, iPixelFormat, &pfd) == FALSE)
{ // Displays error message on failure
      MessageBox(NULL, "SetPixelFormat failed",  MB_OK);
      return FALSE;
}
```

The second technique for enabling stereo buffering support, involves the use of the GLUT library, as described in section 2.4.4. A single call to the function `glutInitDisplayMode()`, with the necessary parameter to enable stereo is all that is required. The parameter to request stereo is `GLUT_STEREO`. The snippet of code below, illustrates the use of the `glutInitDisplayMode()` function, it also first checks if stereo support is available by calling the `checkStereo()` method, which uses the same code as described earlier to check the availability of stereo.

```
if(Stereo.checkStereo())// Checks if stereo is supported.
{
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH |
      GLUT_STEREO);
}else{ // Stereo not supported
      MessageBox(NULL,"No Stereo",MB_ICONINFORMATION);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
}
```

4.2.3 Stereo Rendering

There are two main techniques to render stereo pairs to the correct individual eye displays of the HMD. Both techniques involve setting up the projection of a virtual camera. The first technique of generating stereo pairs is called the "Toe-In" method. This method is an incorrect method, as it introduces vertical parallax, however, it is often used as some rendering software do not support the features required to implement the correct "Off Axis" technique. Vertical parallax is the difference in the vertical height of corresponding points within a stereo pair and can cause much discomfort to the viewer, as their eyes try to adjust to accommodate the misaligned stereo pair images.

Toe-In Projection

The toe-in method is similar to applying a simple rotation to a pair of images. The two cameras used in this type of projection have fixed and symmetric apertures, which are both pointed at a single focal point. Images created using this method will still appear stereoscopic but the vertical parallax it introduces will cause increased discomfort levels. The introduced vertical parallax increases out from the centre of the projection plane and is more important as the camera aperture increases. Figure 4.1, demonstrates the "toe-in" projection method.

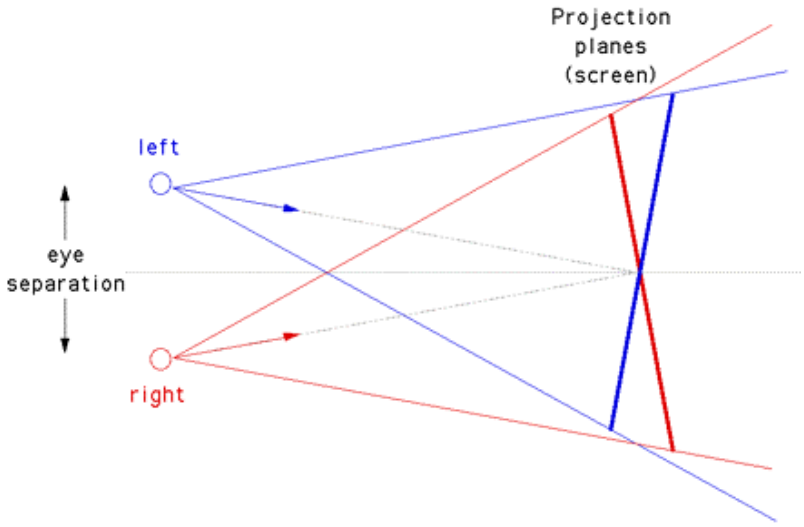

Figure 4.1: Toe-in projection

Off-axis Projection

The off-axis technique for creating stereo pairs generates no vertical parallax and is hence, a much better approach than the "toe-in" method, which can cause discomfort for the viewer. The feature that some graphics rendering software do not support and is vital to the implementation of this technique is a non-symmetric camera frustum. OpenGL does support this feature using the `glFrustum()` function, which allows the perspective projection to be described, whilst the view vectors for each camera remain parallel. The off-axis projection technique for generating stereo pairs is illustrated in Figure 4.2 below.

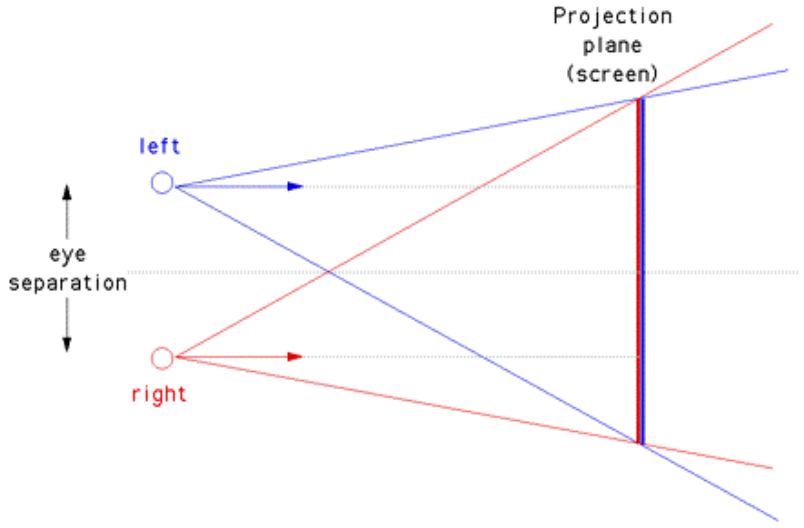

Figure 4.2: Off-axis projection

The Off-axis projection technique is the technique implemented in the project to produce the stereo pairs necessary for stereoscopic imaging. A snippet of the code used in the stereo mountain scene is shown below, however, this code is also very similar to that used in the OpenGLViewer application with a few minor modifications. To condense the snippet of code only the code for drawing the right eye image is shown, as the code for the left eye is almost the same. There are three important points to be noted about this portion code with respect to creating stereo pairs; the first is the derivation of the eye positions. The `Cross()` function called returns the cross product of the two vectors passed in, i.e. a vector that is perpendicular to both input vectors. From this the two eye positions are derived. The next point to be made regards the setting of the viewing volume using the `glFrustum()` command. Finally, the view is drawn to the back-right buffer using the `glDrawBuffer(GL_BACK_RIGHT)` function and the camera position is passed to the `SetViewByTracker(normal)` method, which is only used in the test scene application. This method sets the camera position, view and up direction by calling a function named `gluLookAt()` and integrating the current camera position with the orientation data from the tracker.

```
if(g_bStereo)
{     /* Derive the two eye positions */
      g_Camera.Cross(g_Camera.vView,g_Camera.vUpVector, temp);

      normal = g_Camera.Normalize(temp);
      normal.x *= g_Camera.eyesep / 2.0;
      normal.y *= g_Camera.eyesep / 2.0;
      normal.z *= g_Camera.eyesep / 2.0;

      // Sets the current matrix mode to projection
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity(); //Clears the currently modifiable matrix

      left  = - ratio * wd2 - 0.5 * g_Camera.eyesep * ndfl;
      right =   ratio * wd2 - 0.5 * g_Camera.eyesep * ndfl;
      top    =   wd2;
      bottom = - wd2;


      // Describes a perspective matrix that produces a perspective
      // projection.
      glFrustum(left,right,bottom,top,near1,far1);

      glMatrixMode(GL_MODELVIEW);
      glDrawBuffer(GL_BACK_RIGHT);

      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glLoadIdentity(); // Reset The matrix
```

```
// Passes in the right camera position to be incorporated
with the current camera position and head tracker orientation
data.
g_Camera.SetViewByTracker(normal);
```

4.3 Interfacing the Motion Tracking Device

The Motion Tracking Device used in the dissertation is the InterTrax^2, which is described earlier in section 2.2.2. This device is equipped with eight sensors that allow extremely stable angular velocity measurements to be made. These measurements are continually analysed by a built-in microprocessor that computes the trackers orientation in space.

The mode of operation the motion tracker is used in, is called "Native Mode". This is an advanced mode, which allows three degrees of freedom tracking and has a very high level of precision. This mode is the mode that is integrated into the stereo test scene and the OpenGLViewer application. There is another mode called "Game Compatibility Mode", which is less advanced, as it only offers two degrees of freedom. This mode is, however, not available on the USB model of the InterTrax^2 and hence cannot be used in this project [22].

4.3.1 Setting up the Motion Tracker

The steps required to set-up the motion-tracking device are:

- attach the InterTrax^2 to the top of the HMD using the self-adhesive pads,
- plug the tracker cable into the USB port,
- to initialise the tracker it should be placed in an upright position and held motionless for at least five seconds,
- after successful initialisation of the tracker, requests for tracker data from other applications can be made.

4.3.2 Reset Button

The reset button is located on the top right of the InterTrax^2 tracking device and is the only button on the device. This button can be used for setting the orientation of the device within a 3-D application. When pressed an application can use the current position of the tracking device as the straight and level positions of the tracker thus orientating the 3-D application

view around the position of the tracker. The reset button can be used at any time to change the orientation again.

4.3.3 Software Interface of the Motion Tracking Device

For a software application to interface the motion tracker, InterSense has provided a dynamic link library file (DLL) called *isense.dll*. This file can be used to initialise, access and retrieve data from any InterSense tracker. The DLL provides an easy to use API, which maintains compatibility with existing devices and also makes the applications forward compatible with all future InterSense products. It also simplifies communication and can detect, configure, or acquire data from up to four different trackers. The DLL also provides the highest possible performance for all InterSense trackers [22].

The three main files of the DLL API are:

(i) *isense.h* - This is a header file that contains function prototypes and definitions. This file should not be modified.

(ii) *isense.cpp* – This file contains DLL import procedures. It is included instead of an import library to ensure compatibility with all compilers.

(iii) *isense.dll* – This is the InterSense DLL. It should be placed in the Microsoft Windows system directory or in the working directory of the application.

The API provides functions to perform all necessary operations that involve interfacing the motion tracker. Details of these functions can be found in the InterSense manual [22] and in the *isense.h* file. The main functions used in this project are:

- ```
  ISD_TRACKER_HANDLE ISD_OpenTracker(   HWND hParent,
                                        DWORD commPort,
                                        Bool infoScreen,
                                        Bool verbose)
  ```

  - Function to detect and open any trackers attached to the PC. Returns FALSE if it fails.

- ```
  Bool ISD_CloseTracker( ISD_TRACKER_HANDLE handle)
  ```

- This function de-initialises the tracker. It closes communication ports and frees any resources allocated to the tracker. Returns FALSE on failure.

- `Bool ISD_GetData( ISD_TRACKER_HANDLE handle, ISD_TRACKER_DATA_TYPE *data)`

  - Function that retrieves the data from all configured stations. Data is placed in the ISD_TRACKER_DATA_TYPE structure. Returns FALSE if it fails.

- `Bool ISD_ResetHeading( ISD_TRACKER_HANDLE handle, WORD stationID)`

  - Resets the current heading to zero.

The code that is used in the dissertation to access the motion tracker and acquire data from it is shown below. This same code is used to interface the tracker in both the stereo test scene and the OpenGLViewer application. The first function of the InterSense API, that must be called is `ISD_OpenTracker()`. This method detects any trackers attached to the computer and returns a handle for accessing them. It is normally called in the constructor of a class, as it only needs to be called once and this also helps to prevent any lag from occurring during the program's operation. If no motion-tracking device is found, a negative number is returned. The next important function is the `ISD_GetData(handle, &data)` method. This function retrieves the orientation data from the tracker. It is vital that this method is called at a reasonable rate to ensure the user gets a smooth movement when they look around the scene using the HMD; otherwise the display could become erratic, due to scene updates occurring too slowly. This method is thus called on every frame update in the implementation of the two applications.

```
// Detect the tracker. Returns a negative number on failure.
handle = ISD_OpenTracker( NULL, 0, FALSE, FALSE);

// Failed to detect tracker.
if(handle < 1)
{
      ::MessageBox(NULL,"Failed to detect tracker",
      "Error",MB_ICONINFORMATION);
}else{
      // Get the orientation data. Must be called at a
      //resonable rate
      ISD_GetData(handle, &data);

      // Rotation on the Y-Axis in degrees (Yaw).
      m_yrot = -data.Station[0].Orientation[0];
```

```
            // Rotation on the X-Axis in degrees (Pitch).
            m_xrot = data.Station[0].Orientation[1];
}
```

4.4 Modifying the display for optimum results

In order to fully appreciate stereoscopic imaging in a virtual environment, the user's view should not be obstructed or intruded by anything other than the virtual environment itself, as this would completely diminish the virtual experience. Furthermore, the HMD operates optimally at a certain resolution (800 x 600) and refresh rate (100Hz) settings. Therefore, to achieve the paramount perception of being present in the virtual environment, both of the aforementioned requirements must be fulfilled.

4.4.1 Full-Screen and the Display Settings

To satisfy the first requirement of nothing but the virtual environment being displayed in the user's view, the program's window is made full-screen. To achieve a full-screen view, the basic idea is to adjust the size and position of the application's window so it is larger than the screen by the exact amount to make the active view just fill the display. There is no explicit command available in MFC to exercise full-screen mode, thus a method to do so, is incorporated into both the test scene and OpenGLViewer applications.

In the test scene application, the user is queried to enable full-screen mode on start up of the program, see Figure 4.3 below. This gives the user the option of selecting windowed mode if for some reason this mode is required. If the user prompts "yes", then the application's window is immediately changed to full-screen by a method called `ChangeToFullScreen()`.

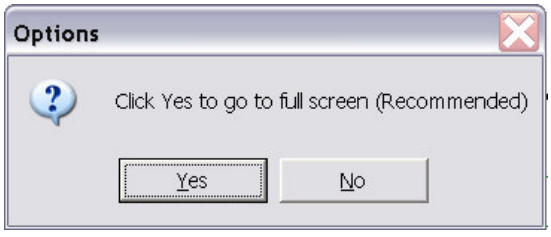

Figure 4.3: Enabling full-screen in the test scene application

This method modifies the device mode by setting the resolution to be 800 x 600 and changes the window's size to be equal to the screen size. Furthermore, the refresh rate is set to 100Hz; the required setting for optimal performance by the HMD. A snippet of the

method's code is shown below. The important lines are setting the `dmSettings` and calling the Windows operating system method `ChangeDisplaySettings()`, which applies the `dmSettings` and removes the Windows Start menu bar.

```
DEVMODE dmSettings;         // Device Mode variable
// Makes Sure Memory's Cleared
memset(&dmSettings,0,sizeof(dmSettings));

dmSettings.dmPelsWidth = 800;    // Selected Screen Width
dmSettings.dmPelsHeight = 600;   // Selected Screen Height
dmSettings.dmFields = DM_BITSPERPEL |DM_PELSWIDTH | DM_PELSHEIGHT;
dmSettings.dmDisplayFrequency = 100;  // Set refreshrate to 100Hz

// This function actually changes the screen to full screen
// CDS_FULLSCREEN and gets rid of the start bar. Always want to
// get a result from this function to check if it failed.

int result = ChangeDisplaySettings(&dmSettings,CDS_FULLSCREEN);
```

The OpenGLViewer application applies full-screen in a different manner to the approach taken in the test scene. This is due to the fact that the program is an MDI application and it is not desirable to have the entire program operating in full-screen mode, as the test scene program operates. The technique for implementing full-screen in the OpenGLViewer application is presented later in section 4.6.1.

4.5 Mountain Test Scene

To test the ability of the HMD to operate in stereo mode, it was decided to create a stereo mountainous scene. The Motion Tracking Device is also to be incorporated into this test application to enhance the user's stereo experience, as it increases the sense of being present in the virtual world by allowing head motions to control looking around the scene. The basis for the stereo mountain scene is a tutorial available for download from a Website called "GameTutorials" [24]. This tutorial provided the fundamental starting point for the development of the stereo test scene. There are many stages in creating such a scene, these are described in the following sections.

4.5.1 Height Maps

The most important characteristic of a mountainous scene is of course the mountains themselves. There are numerous techniques to create a mountainous terrain, but one of the most popular and the method implemented in this dissertation, is through the utilisation of a height map.

A height map basically represents low and high points on the landscape. It can be created using any form of data as the input. For example, an audio stream could be used to create a visual height map representation. There are also many height map editor applications available on the Internet that facilitate the creation of height maps. Typically, zero or the colour black are used to represent the lowest point in the height map and the highest point is either represented by the value 255 or the colour white. An example of a height map is shown below in Figure 4.4. A similar height map is used in the creation of the mountainous scene, but it is scaled up in size greatly and the viewer is positioned on the landscape floor.

Figure 4.4: Example of a height map

Two files contain the code for the height map that is used in the test scene; these are *Terrain.h* and *Terrain.cpp*. Firstly, three important variables are defined, these include:

```
// This is the size of our .raw height map
#define MAP_SIZE      1024
// This is the width and height of each triangle grid
#define STEP_SIZE          16
// This is the ratio that the Y is scaled according to the X and Z
#define HEIGHT_RATIO  1.5f
```

The `MAP_SIZE` is the dimension of the height map; a size of 1024*1024 is used. The `STEP_SIZE` is the size of each quad that is used to draw the landscape. This variable can be used to control the degree of precision that the terrain is drawn in. A lower step size

increases the smoothness of the landscape but it also decreases the performance of the application. Finally, the `HEIGHT_RATIO` is used to scale the landscape on the y-axis. Increasing this variable produces higher and more defined mountains, whereas decreasing it has the effect of creating flatter mountains.

The next stage is to load the height map data in from a file into an array, which is used to render the height map data. A *.RAW* file called *Terrain.raw* contains the values for the height map, which range from 0 to 255. A method called *`LoadRawFile()`*, which is contained in the *Terrain.cpp* file is used to perform this operation. It first opens the file for reading in read/binary mode and if this is successful, it continues by loading the *.RAW* file into an array called `g_HeightMap` using a pointer (`pHeightMap`) to this storage location. The important line of code in this method is:

```
fread( pHeightMap, 1, nSize, pFile );
```

The first remaining parameter after `pHeightMap` specifies that one byte is to be read at a time. *nSize* is the maximum number of items to read (i.e. the image size in bytes; the width * height of the image) and finally *pFile* is a pointer to the file structure.

Following successful loading of the *.RAW* file, the final stage is to actually draw the terrain. A snippet of the `RenderHeightMap()` method used to draw the landscape is shown below. *X* and *Y* are variables used to loop through the height map data and *x*, *y* and *z* are more variables required to render the strips making up the landscape. Rendering the terrain with `GL_QUADS` is changed to use `GL_TRIANGLE_STRIP` instead, as this removes the need to pass in the same vertex more than once, due to the fact that each two vertices are connected to the next two. To do this in a single strip, it is necessary to reverse the order of every alternate column. An analogy can be drawn here to that of mowing a lawn. Go to the end, turn around and come back the way you came. If this is not implemented carefully, it can result in polygons stretching across the whole terrain. The variable used to track the direction of drawing the strips is called, `bSwitchSides`. The final operation is to draw the terrain from the height map. This is executed by simply walking through the array of height data and using this data to plot points. The columns of the height map array are plotted first, followed by the rows. A screenshot of the height map that the snippet of code creates without the line of code to bind the texture to the terrain is shown in Figure 4.5. The power of texture mapping is discussed in the next section 4.5.2.

```
void RenderHeightMap(BYTE pHeightMap[])
{
      int X = 0, Y = 0;     // Variables to walk the array with.
      int x, y, z;          // Variables for readability

      // To keep track of the drawing direction
      bool bSwitchSides = false;

      // Make sure our height data is valid
      if(!pHeightMap) return;

      // Bind the terrain texture to our terrain
      glBindTexture(GL_TEXTURE_2D, g_Texture[0]);

      // We want to render triangle strips
      glBegin( GL_TRIANGLE_STRIP );

      // Go through all of the rows of the height map
      for ( X = 0; X <= MAP_SIZE; X += STEP_SIZE )
      {
      // Check if we need to render the opposite way for this
      // column.
            if(bSwitchSides)
            {
                  // Render a column of the terrain, for this
                  current // X. Start at MAP_SIZE and render down to
                  0.
                  for ( Y = MAP_SIZE; Y >= 0; Y -= STEP_SIZE )
                  {
                        // Get the (X, Y, Z) value for the bottom
                        left vertex.
                        x = X;
                        y = Height(pHeightMap, X, Y );
                        z = Y;

                        // Set the current texture coordinates and
                        // render the vertex.
                        SetTextureCoord((float)x, (float)z );
                        glVertex3i(x, y, z);

                        // Get the (X, Y, Z) value for the bottom
                        // right vertex
                        x = X + STEP_SIZE;
                        y = Height(pHeightMap, X + STEP_SIZE, Y );
                        z = Y;

                        // Set the current texture coordinate and
                        // render the vertex
                        SetTextureCoord( (float)x, (float)z );
                        glVertex3i(x, y, z);
                  }
            }
```

Figure 4.5: Screenshot of the terrain created using a height map

4.5.2 Texture Mapping

Texture mapping is one of the major advancements made in computer graphics during the last ten years. Texture mapping is the process of fitting images onto polygons, thus greatly enhancing their appearance by making them look more realistic. For example, to create a realistic brick wall without texture mapping, each individual brick must be drawn separately and even then the wall would appear too smooth and regular to look realistic. However, using texture mapping, a real image (e.g. photograph, etc.) of a brick wall can be mapped onto a single polygon. This texture mapped brick wall would look much more realistic, due to its irregularities and would also require a lot less processing power to create. In OpenGL, texture mapping ensures that, as the texture-mapped polygon is transformed and rendered the correct procedures are used to modify the textured polygon appropriately. For instance, if the view is changed to perspective, then further away bricks look smaller than closer bricks.

There are four steps required to use texture mapping:

(i) specify the texture,

(ii) indicate the method to apply the texture to each pixel,

(iii) enable texture mapping,

(iv) draw the scene, supplying both texture and geometric coordinates [8].

A method called `CreateTexture()` is used in the test scene to fulfil the first two steps required to use texture mapping. By calling this method the texture is specified as a parameter passed into the method. This is exercised in the program by a line such as:

```
CreateTexture(g_Texture[0],"Terrain.bmp");
```

This `CreateTexture()` method, first opens the texture image file and if this executes successfully the bitmap image is loaded. Following this, the texture is binded to a texture target using the code, `glBindTexture(GL_TEXTURE_2D, texture)` and a mipmap is then created. Mipmapping[2] is a technique used in OpenGL of specifying prefiltered texture maps of decreasing resolutions, called mipmaps. These mipmaps can then be used instead of the full resolution texture map for textured objects that are viewed at a distance. This alleviates the problem, of introducing visually disturbing artefacts, which would occur by simply scaling down the texture map [8]. Mipmapping is performed in the program using the following lines of code:

```
// Build Mipmaps (builds different versions of the picture for
// distances - looks better)
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, pImage->sizeX,
      pImage->sizeY, GL_RGB, GL_UNSIGNED_BYTE, pImage->data);

// Assign the mipmap levels and texture info by specifying the
// minification filtering methods.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
      GL_LINEAR_MIPMAP_NEAREST);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

Finally, the `CreateTexture()` method frees up the texture data and image data. The third step required to use texture mapping can be applied very simply by using the OpenGL call to enable texture mapping:

```
glEnable(GL_TEXTURE_2D);
```

The final requirement for using texture mapping is to draw the scene, supplying both textural and geometric coordinates. The last section 4.5.1, touched on this subject, as the method for drawing the height map, `RenderHeightMap()`, contained code for specifying both the textural and geometric coordinates. The texture coordinates are set by calling the method, `SetTextureCoord()`. This method sets the textural coordinate of the terrain based on the *X* and *Z* coordinates using the OpenGL command `glTexCoord2f()`.

Figure 4.6 illustrates texture mapping being applied to the terrain generated in the previous section.

[2] Mip stands for the Latin *multim im parvo* and means “many things in a small place”.

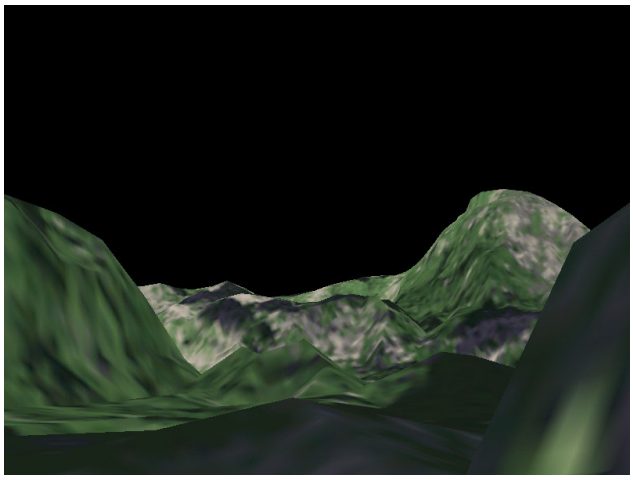

Figure 4.6: Screenshot of texture mapping being applied.

4.5.3 Skybox

A skybox is a technique of suggesting a larger environment space. It is basically a box with images of the surrounding environment texture mapped onto the inside of the box. The box is position so as to surround the viewing position. The skybox position relative to the viewing position must remain constant.

A common problem with implementing skyboxes is that thin lines can appear at the seams of the skybox, which diminish the affect of the illusion. The cause of these lines is due to OpenGL linearly interpolating a texel at the edge of a texture with a texel at the opposite edge. This can be solved using texture clamping, which instructs OpenGL not to interpolate between the opposite edges of the texture. The following OpenGL function calls enable texture clamping:

```
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S,GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T,GL_CLAMP_TO_EDGE);
```

In the stereo test scene a method called, `CreateSkyBox()`, which is in the *Main.cpp* file is used to create the skybox. Six textures are used in the creation of the skybox, allow of which use the same `CreateTexture()` method as described in section 4.5.2, to handle the creation of the textures. A snippet of the `CreateSkyBox()` method is shown below (Only the code for creating the back side of the cube is shown):

```
void CreateSkyBox(float x, float y, float z, float width, float
height, float length)
```

```
{
      // Turn on texture mapping if it's not already on
      glEnable(GL_TEXTURE_2D);

// Bind the BACK texture of the sky map to the backside of the
// cube
      glBindTexture(GL_TEXTURE_2D, g_Texture[BACK_ID]);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
            GL_CLAMP_TO_EDGE);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
            GL_CLAMP_TO_EDGE);

      // This centers the sky box around (x, y, z)
      x = x - width  / 2;
      y = y - height / 2;
      z = z - length / 2;

      // Start drawing the side as a QUAD
      glBegin(GL_QUADS);

            // Assign the texture coordinates and vertices for the
            // backside.
            glTexCoord2f(1.0f, 0.0f); glVertex3f(x + width, y, z);
            glTexCoord2f(1.0f, 1.0f); glVertex3f(x + width,
                  y+height, z);
            glTexCoord2f(0.0f, 1.0f); glVertex3f(x, y + height, z);
            glTexCoord2f(0.0f, 0.0f); glVertex3f(x, y, z);

      glEnd();
```

A screenshot of the skybox created in the test scene is shown in Figure 4.7 below. The image on the left Figure 4.7(a), illustrates the skybox with no textures mapped onto the inside of the box. The image on the right, Figure 4.7(b), displays the complete skybox and demonstrates the use of texture clamping to remove edge boundaries from being displayed.

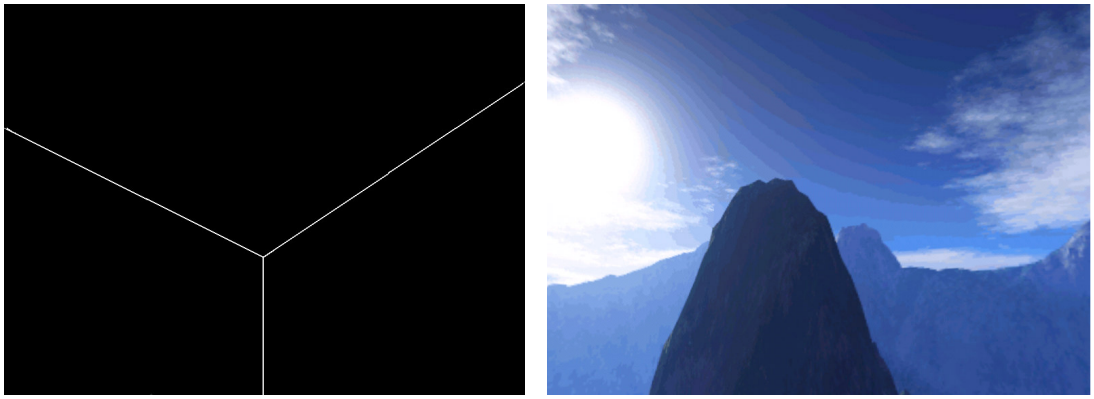

Figure 4.7: (a) Skybox with no texture mapping (b) Complete skybox

4.5.4 Movement within the scene

In order to provide a true sense of being present in a virtual environment, the user must be allowed to look and move around the environment freely. The test scene application provides this functionality through the use of the Motion Tracking Device and the keyboard arrow keys. The user simply looks in any direction and the scene is updated accordingly. If the user wants to move forward, backward, right or left in the direction they are facing, the corresponding arrow keys are used to facilitate this requirement.

Allowing the user to look around the scene, is performed in the program by a method called `SetViewByTracker()`, which is located in a file called *Camera.cpp*. This method first retrieves the orientation data from the tracker as described earlier in section 4.3. After, successfully acquiring this data, the camera position and view are set up. This is exercised by incorporating the new head position data into the previous camera set up. Following this, the OpenGL function `gluLookAt()` is used to define the viewing matrix. This function takes nine parameters as input. The first three parameters define the x, y and z coordinates of the position of the camera, the subsequent three parameters specify the x, y and z coordinates of where the camera is pointing towards and finally, the last three parameters define the up direction.

This `SetViewByTracker()` method, is called from within the `RenderScene()` method, which is subsequently called at a period equal to the frame rate. This ensures that when the user looks around the scene the motion is smooth. If the test scene application is in stereo mode then the `SetViewByTracker()` method is called at twice the rate, as two views of the scene are required to be updated.

To facilitate movement around the scene using the arrow keys, a method called `CheckForMovement()`, is called on every frame update. This method checks if an arrow key has been pressed and if so reacts with the appropriate action. This action basically involves adding or subtracting a function of the current yaw value to the correct positional vector of the camera. Thus, the direction the user is looking in is amalgamated with the direction of the arrow key pressed to provide a realistic sensation of movement. The movement is completed when the `RenderScene()` method calls the `SetViewByTracker()` method, which updates the user's view by changing the camera

position and view. A sample of the `CheckForMovement()` method is shown; the forwards or up arrow is all that is presented, as the code for the other directions is similar:

```
// Check if the Up arrow key is pressed
if(GetKeyState(VK_UP) & 0x80) {
      // Add the acceleration to the X position

      m_vPosition.x -=(float)sin(m_yrot*m_piover180) * 0.65f;

      // Add the acceleration to the Z position
            m_vPosition.z -= (float)cos(m_yrot*m_piover180) * .65f;
}
```

A problem with this approach for moving around the scene is that it does not take into account the fact that the terrain is uneven. If the above method were used, as is, then the user would walk through the terrain instead of over it. To solve this problem, extra code is required to ensure that the user is always kept at the level of the landscape. This is achieved by checking the position of the camera with respect to the position of the height map. If the camera is below the terrain level, an integer of ten is added to the camera's position to keep it above the ground level. The camera's view must also be adjusted by the difference between the camera's new position and its previous position. Similarly, if the user is walking down from a height the camera's view and position are adjusted accordingly by subtracting ten.

A sample of the code that is used in the program is shown below, this code is called on every frame update to ensure that the changing of the camera's view and position does not introduce a bumpy effect into the user's view:

```
// Returns the current camera position
CVector3 vPos         = g_Camera.Position();
CVector3 vNewPos    = vPos; // Used to set the new camera position

// Check if the camera is below the height of the terrain at x and
// z, if so add 10 to make it so the camera isn't on the floor.
if(vPos.y < Height(g_HeightMap, (int)vPos.x, (int)vPos.z ) + 10)
{
      // Set the new position of the camera so it's above the
      // terrain + 10
      vNewPos.y = (float)Height(g_HeightMap, (int)vPos.x,
            (int)vPos.z) + 10;

      // Get the difference of the y that the camera was pushed up.
      float temp = vNewPos.y - vPos.y;

      //  Get the current view and increase it by the difference
      the position was moved
      CVector3 vView = g_Camera.View();
```

```
        vView.y += temp;

        // Set the new camera position.
        g_Camera.PositionCamera(vNewPos.x,  vNewPos.y,  vNewPos.z,
                                vView.x,    vView.y,    vView.z,    0, 1, 0);
}
// Check to ensure the camera position is not to far above the
// terrain level. Occurs when walking down a slope.
else if(vPos.y > Height(g_HeightMap,(int)vPos.x,(int)vPos.z ) -
10)
{
        etc.
```

The test scene application developed thus fulfils all the necessary requirements, as outlined previously to demonstrate the stereo capability of the HMD. It presents a full-screen stereo view of a mountainous scene, which integrates the tracking device to allow the user to look around the scene by using their head motion to control the view. Furthermore, the application provides the functionality to move around the scene using the arrow keys and it is the combination of these abilities that gives the user a sense of being present in a true 3-D environment. Figure 4.8 below illustrates a screenshot of the complete test scene application.

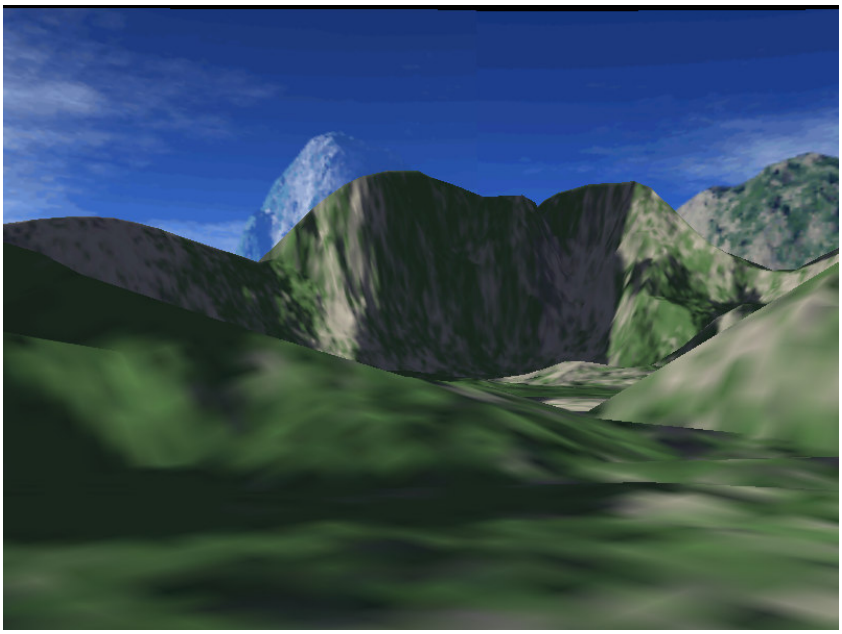

Figure 4.8: Screenshot of the complete test scene application

4.6 OpenGLViewer Application

The OpenGLViewer application, as previously stated is a partially complete project that allows OpenGL scenes to be viewed in a virtual environment. It has been developed in the Microsoft .NET environment using C++, OpenGL and MFC. This dissertation extends the OpenGLViewer application by applying three concepts that increase its functionality and allow stereoscopic imaging to be incorporated. These concepts include the ability to:

(i) view the active display in full-screen,
(ii) view the OpenGL scenes in stereo using the HMD,
(iii) integrate the Motion Tracking Device to allow the user to manipulate the OpenGL scenes using head motions.

The first step taken to incorporate these concepts into the OpenGLViewer application was to acquire an in-depth knowledge of the inner workings of the application. This was obtained by carefully scrutinising through the application's code so as to build up a working knowledge of its features, operation and structure. Once this step was completed it was possible to begin adding the three concepts described above to the application.

4.6.1 Adding Full-Screen

The motives for adding full-screen to the OpenGLViewer application are, firstly, to improve the view of the OpenGL scene being displayed, and to facilitate the stereo mode feature by enabling it to provide a realistic virtual experience (with a view free from any intrusive objects) when using the HMD. Full-screen mode is executed in a different manner than in the test scene application. A new sub-window is created when full-screen mode is initiated. This is made possible by the fact that the OpenGLViewer program is a MDI application, which means the user can open more than one document in the application without having to close the other documents. This method of creating a new sub-window to supply a full-screen view is advantageous, as it provides the user with increased functionality by allowing both the normal view and full-screen view of the same OpenGL scene to be displayed simultaneously in a multiple monitor environment.

Firstly, to enable full-screen the user right-clicks in the active window's region. This displays a menu, allowing the user to choose full-screen or stereo mode. A diagram of this is shown in Figure 4.9. After the user selects full-screen mode, a new window is created by initialising an instance of a class called *CSubWindow*. On instantiation of this class an event

named `ON_WM_CREATE()` is generated, which sends an MFC style Windows message to create the new sub-window. The sub-window created is a child of the default view window (Figure 4.9), which is constructed from a class called *COpenGLViewerView* and hence, the new sub-window inherits the same properties. After the sub-window is created, the OpenGL objects are added to the sub-window's view by calling a method within the *CSubWindow* class called `NewView()`.

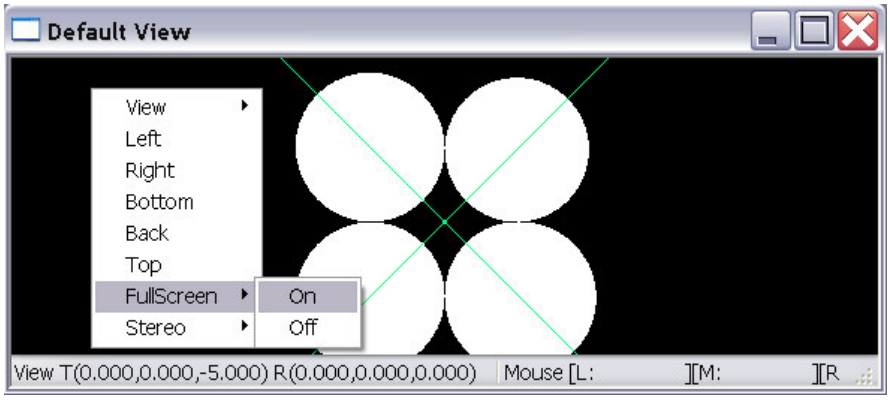

Figure 4.9: Enabling full-screen

Following this, a method named, `GoFullScreen()` is used to provide the full-screen functionality. This method is contained in a class called *CFullScreenHandler*, which encapsulates all the full-screen functionality. The `GoFullScreen()` method operates in a slightly different manner than the `ChangeToFullScreen()` method, described in section 4.4.1, which is used to execute full-screen mode in the test scene application. The reason for using a different technique to facilitate full-screen is due to the fact that the `ChangeToFullScreen()` method also sets the screen resolution and refresh rate for optimal performance when using the HMD in stereo. This is not required in the full-screen mode of the OpenGLViewer application, as stereo mode is enabled separately, plus the user can employ the application without the HMD. The facility to change the screen settings is explained in the next section 4.6.2, where a description of the implementation of stereo mode in the OpenGLViewer application is provided.

The `GoFullScreen()` method utilises three functions called `GetSystemMetrics()`, `GetWindowRect()` and `SetWindowPos()`. `GetSystemMetrics()` retrieves the dimensions of the screen size. `GetWindowRect()` returns the size and location of a frame or view window, whereas `SetWindowPos()` sets the position and size of a frame or view

window. Through a combination of these methods the position and size of both the frame and view windows is set so as to situate and size the view frame to exactly fill the entire screen. A screenshot of the full-screen mode is shown below in Figure 4.10.

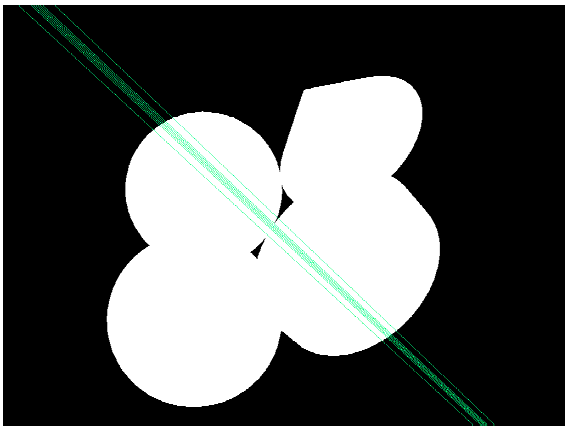

Figure 4.10: OpenGLViewer application operating in full-screen mode

4.6.2 Stereo mode

The functionality of stereoscopic imaging is added to the OpenGLViewer application in a similar manner to the method used to add this feature to the test scene application. One major difference is that a new window is created to view the stereo images. This has the same added advantage, as described for full-screen mode in the preceding section.

The approach for implementing stereo mode is also very similar to that described for full-screen mode, as its procedure follows the same course of events. Firstly, the user enables stereo mode by selecting it from a menu, as shown earlier in Figure 4.9. This calls a method in a class called *COpenGLStereoViewer*, which instantiates an instance of the *CSubWindow* class, thus, creating a new sub-window in a similar fashion, as that stated in the former section. However, as the sub-window created has been instantiated from a different class (*COpenGLStereoViewer*) than that used to create the full-screen window it has been created with different properties. The property that enables stereo is the `PFD_STEREO` flag, which is passed into the pixel format descriptor, as explained before in section 4.2.3. Following this, the OpenGL objects are added to the stereo window's view in a similar fashion to the full-screen mode by calling a method within the *CSubWindow* class called `NewStereoView()`. Finally, the screen is changed to full-screen mode by calling a method named `GoStereoFullscreen()` in the *CFullScreenHandler* class. This method also changes the screen settings to facilitate the HMD by providing the optimal settings.

The stereo-pairs required for stereoscopic imaging are rendered in the `OnPaint()` method of the *COpenGLStereoViewer* class. This rendering action is identical to that used for the test scene application, as it employs the off-axis projection technique; this has formerly been described at the start of this chapter in section 4.2.3.

4.6.3 Incorporating the Motion Tracking Device

The motion tracker is used in the OpenGLViewer application, to permit the user to manipulate their view of an OpenGL object in the virtual environment by using their head movements to control the view. The user can look around an object by simply moving their head in the desired direction.

Originally, the mouse controlled the operations for manoeuvring objects in the OpenGLViewer application. The user was able to zoom in and out on the objects, drag (pan) the objects and also orbit the objects about their axes. To zoom the user held down the 'z' key and pressed the right mouse button. Moving the mouse in a forward motion zoomed in and similarly a backwards motion zoomed out. The pan function was controlled by also holding down the 'z' key but the left mouse button was used to drag the objects in the desired direction. Finally, the user held down the 'o' key to enable the orbit function, which allows the objects to be rotated about their axes in the horizontal, vertical or both directions. The direction is selected using either the left, right or middle mouse buttons, which selects both directions (free mode), the horizontal direction or the vertical direction respectively.

To incorporate the tracking device into the application, it was decided that just the orbiting motions were to be controlled by the tracker. This simplified that integration process but it also provided a very reasonable level of control over the view using the tracker. The user for instance, would be able to look down to get a top view of the object, look up to view underneath the object, and left or right would produce the views of both sides. Changing the view to produce these viewing positions requires obtaining the yaw and pitch orientation values from the motion tracker.

The first phase of incorporation of the Motion Tracking Device into the application is to set-up the interface to the tracker. This is completed in the same manner as described earlier in section 4.4.3 by using the DLL designed by InterSense for this purpose and including the

isense.cpp, *isense.h* and *types.h* files into the OpenGLViewer application. The yaw and pitch values are then obtained from the device and used to control the orbiting of the OpenGL objects.

4.7 Conclusion

In this chapter the implementation of the two applications developed in the project are described. These are the creation of a test scene to demonstrate the HMD operating in stereo and the extension of the OpenGLViewer application to allow it to provide a stereo view of its virtual environment. Both the applications include the utilization of the tracking device to provide realistic viewing of the environments and further the user's experience.

In the next chapter, the outcome of developing both applications is discussed and the problems encountered during the implementation stage are outlined.

Chapter 5 - Results and Discussion

This chapter outlines the results obtained from the development of the two applications. It reviews the results obtained from testing the operation of the applications. It also describes the successes and drawbacks of the final implementations and discusses possible reasons for the problems encountered.

5.1 Testing of the stereo test scene application

The only true test for this application is simply to allow a client to operate it. In such a test the application performs excellently. It provides the client with a sense of being projected into a realistic looking virtual environment, where the mountains appear to have depth. Furthermore, the incorporation of the tracking device into the application is proven to be effective, as it allows the user to look around the scene with a realistic and smooth motion.

Another two simple tests that were applied to the application are:

- to toggle stereo mode on/off during operation. When stereo mode is switched off the depth is immediately removed from the scene, thus providing a distinct noticeable difference to the user. The toggle stereo feature is enabled/disabled using the ‘s’ key on the keyboard.

- to close one eye during operation in stereo mode. By focusing on any reference point within the virtual environment and alternatively opening and closing each eye while still focused on the same reference point. The user notices a difference between the views from each eye. This demonstrates that there is parallax between the two separate eyes views and thus, proves that the virtual environment is operating in true stereo.

5.2 Testing of the OpenGLViewer application

The second major part of the project entailed expanding the software environment that has been partially developed in the Vision Systems Group by developing an interface to view the OpenGL objects in true 3-D. This development was segmented into three areas that could be implemented separately. These areas include the functionality to:

(i) view the OpenGL scenes in full-screen,
(ii) view the OpenGL scenes in stereo using the HMD,
(iii) allow the user to manoeuvre their view of the OpenGL scenes using their head motion.

The three key milestones of the extension to the OpenGLViewer application are implemented in the .NET development environment, as described in section 4.6 with varying degrees of success. The first stage of the extension, to add the facility to change the view to full-screen was accomplished successfully. In the application, the client can now right-click the mouse on the required view to display a menu and simply select full-screen to enable it. The application creates a new full-screen display of the view, which still permits all the previous functionality to operate as normal, such as zoom, etc.

The next stage of the development, to allow the OpenGL scenes to be viewed in stereo, unfortunately did not produce as successful a result. The problem encountered was that the stereo mode of the NVidia graphics card could not be enabled, thus obviously impeding the creation of a stereoscopic view for the application. The code for this operation compiles and executes successfully but when trying to enable stereo it returns “Stereo Unavailable”, indicating that some problem occurred when communicating with the graphics card. This problem is quite unusual, as the same interfacing code that is used in the test scene application to enable stereo on the graphics card is also used in this application. If stereo mode could be enabled on the graphics card then the application should function in stereo, as the code to implement stereoscopic imaging is in place. Possible reasons for this problem are outlined in the next section (section 5.3).

The final stage of the extension made to the OpenGLViewer application was to integrate the motion-tracking device to enable the client to manoeuvre their view of the scene using head motions. This was implemented with a mixed degree of success. A similar problem to

that mentioned for enabling stereo mode occurred when trying to interface the tracking device. The code compiled and executed normally but the interface to request the orientation data from the tracker fails when requested. Again the code used is the same as that used in the test scene application, thus in theory there should be no problem applying this code to the OpenGLViewer application.

However, a technique to overcome this problem can be applied to demonstrate that the tracker has been integrated correctly into the OpenGLViewer application. By using another application called the InterSense Server, which is a demonstration tool provided by InterSense. To apply this technique the client must first execute the OpenGLViewer application followed by the InterSense Server in this order. This in effect calls the required DLL file, which allows the OpenGLViewer application to interface with the tracker. The motion tracker then operates correctly with the OpenGLViewer application. Thus, the effectiveness of the incorporation of the tracker into the application can be tested. Results from the application prove that the tracker operates excellently when tested in this manner and furthermore, when used in conjunction with the HMD provides a very useful additional feature to the OpenGLViewer application.

5.3 Possible Reasons for the Problems Encountered

There are a number of possible reasons for the problems encountered with the OpenGLViewer application that are outlined in the previous section. The possible reasons for the problems include:

- intermixing of two very different languages in the .NET development environment,

  - OpenGL operates as a state machine and incorporating this language into an Object Oriented language such as C++ can lead to difficulties. Such a problem could be the reason for the stereo mode not operating correctly, as it is when the application is creating a new OpenGL stereo window that it fails to enable stereo.

- interfacing the graphics card from within the .NET environment,

  - The NVidia graphics may not have full support for the latest versions of .NET or OpenGL, thus again leading to the stereo problem encountered.

- accessing the Intersense DLL from within the .NET environment,

- This has to do with the problem encountered with the motion-tracking device. An e-mail was sent to the producers of the tracker (InterSense) asking for any support available with this issue. Unfortunately, no response was received, thus one could draw the conclusion that there may be some issues with the motion tracker and the latest version of the .NET development environment.

All of these possible reasons were investigated but it was very difficult to find any solutions to the problems. The Internet was the chief source of material used for the investigation into the problems. It was found that other people have had similar problems but no real solutions were presented. A good discussion board Website that was found while investing these problems is called .NET 247 [25].

5.4 Conclusion

This chapter discusses the outcome of the two applications developed and reports the results from testing the operation of the applications. Following that, possible reasons for the problems encountered are presented.

In the ultimate chapter of the dissertation, the final conclusions and possible future work are described.

Chapter 6 - Conclusions and Further Research

The primary aim of the project was to develop a 3-D Headset Tracking Interface for True 3-D Visualisation. This was achieved through the development of two applications. The first application is a test scene to demonstrate the stereo capabilities of the HMD, which also utilises the motion-tracking device to further enhance a user's experience of being present in a virtual environment. The test scene application also served as a learning stage to acquire the necessary level of competence in OpenGL and the development environment so this knowledge could be applied to a further advanced application (i.e. the OpenGLViewer application).

The second major part of the project was to develop the OpenGLViewer application by increasing its functionality by including full-screen and stereo mode features and also integrating the motion-tracking device to allow manipulation of the OpenGL objects using the user's own head motions. These extensions were achieved with a mixed degree of success, as described in the previous chapter.

Overall, the main objective of the project to develop a 3-D headset-tracking interface for true 3-D visualisation is achieved through the stereo test scene application. This application performs well under testing and provides the user with a virtual environment, which demonstrates true 3-D. The incorporation of this technology and the technique for implementing stereoscopic imaging was then applied to the OpenGLViewer application with a mixed level of success. However, despite the few technical problems encountered, the project achieved its principal aims and is a success overall. If the few technical problems could be resolved, the OpenGLViewer application should also be fully functional.

6.1 Future Work

There are many possible areas to expand this project, as it entails such a broad topic of research. The areas of future work involve both the stereo test scene application and the OpenGLViewer application, and are described in the following sections.

The first area of future work is to solve the problems encountered with the development of the OpenGLViewer application. The possible reasons for these problems have been outlined in section 5.3 and these should provide a good starting position to enable the problems to be solved.

Another area of future work is to improve the stereo affect accomplished in the test scene application and also incorporated into the OpenGLViewer application (however, not fully operational). A technique to automatically control the disparity could be developed to achieve an improved stereo affect. This would simulate the phenomenon that occurs in reality when objects that are closer to the viewer appear to have more depth than objects further away. Thus, by including a facility to automatically control the disparity setting for objects at different distances, the natural affect that occurs in reality could be simulated within the virtual environment. A technique for implementing this phenomenon is described in a paper entitled "Automatic Disparity Control in Stereo Panoramas (OmniStereo)" [25] and this paper should provide a good starting position for this work.

A further area of future work would be to develop the motion of the tracking device in the OpenGLViewer application. At present, the motion tracker is limited to controlling the orientation of the OpenGL objects. This could be extended to allow the user's head motions to have more control by combining a panning facility with the motion tracker.

References

[1] Chenney, S., *3D User Interfaces,* Berkeley, 1997 http://bmrc.berkeley.edu/courseware/cs160/fall97/lectures/10-15-97/sld001.htm, (April 2004)

[2] ACM, SIGGRAPH and SIGCHI, *Symposium on Virtual Reality Software and Technology, 1999* http://www.cs.ucl.ac.uk/staff/m.slater/VRST99/ (April 2004)

[3] *The Art and Science of 3D Interaction* presented at the IEEE Virtual Reality'2000 New Brunswick, USA, March 18, 2000. http://www.mic.atr.co.jp/~poup/3dui/TUT3DUI/ (April 2004)

[4] 3D-Doctor, *Create more accurate 3D models using vector-based technologies* http://www.ablesw.com/3d-doctor/index.html (April 2004)

[5] Poupyrev, Ivan, "Research in 3D user interfaces", In Looking Forward, The IEEE Computer Society's Student Newsletter, 1995, Vol. 3, N 2, pp. 3-5 http://www.hitl.washington.edu/people/poup/research/papers/lookfwd.html

[6] Virtual Realities, *I-glasses PC* http://www.vrealities.com/igsvga.html (April 2004)

[7] Virtual Realities, *InterTrax2* http://www.vrealities.com/intertrax2.html (April 2004)

[8] Neider, J., Davis, T. and Woo, M., "OpenGL Programming Guide", *The Official Guide to Learning OpenGL*, Release 1, 1993

[9] *About the OpenGL Architectural Review Board*, http://www.opengl.org/about/arb/overview.html (August 2004)

[10] Hawkins, K., Astle, D. and LaMothe, A., *What is OpenGL?* http://www.developer.com/tech/article.php/947051 (August 2004)

[11] Silicon Graphics Inc., *GLUT – OpenGL Utility Toolkit* http://www.sgi.com/software/opengl/glut.html (August 2004)

[12] OpenGL Official Website, http://www.opengl.org/ (August 2004)

[13] NeHe Productions, http://nehe.gamedev.net/ (August 2004)

[14] Wright, R., and Lipchak, B., "OpenGL SuperBible", 1996

[15] Feurer, Alan R., "MFC Programming", pp 6-8, 1997

[16] Blaszczak, M., "MFC 4", *Programming with Visual C++,* pp 115-119, 1996

[17] Microsoft Foundation Class Library, *General Class Design Philosophy,* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/_mfc_About_the_Microsoft_Foundation_Classes.asp (August 2004)

[18] Bourke, P., *Stereo Pair Photography*, July 1999 http://astronomy.swin.edu.au/~pbourke/stereographics/stereophoto/ (August 2004)

[19] William and Craig, *Seeing 3D from 2D Images*, http://www.cise.ufl.edu/~lok/teaching/dcvef03/Seeing%203D%20from%202D%20Images.ppt (August 2004)

[20] Lipton, L., "StereoGraphics Developer's Handbook", *A guide to creating stereoscopic images for StereoGraphics*, 1997, http://www.stereographics.com/support/downloads_support/handbook.pdf (August 2004)

[21] Depth Perception, *Multiple Cues & Development Aspects* http://www.public.asu.edu/~coberle/n_depth.ppt (August 2004)

[22] InterSense, *User Manual for InterTrax^2 Versions 1.0 and above*

[23] Bourke, P., *3D Stereo Rendering Using OpenGL (and GLUT)*, 1999, http://astronomy.swin.edu.au/~pbourke/opengl/stereogl/ (August 2004)

[24] GameTutorials, *Height Map tutorial*, http://www.gametutorials.com/Tutorials/opengl/OpenGL_Pg4.htm (August 2004)

[25] Matthew Reynolds' .NET 247, *Internet Discussion Board*, http://www.dotnet247.com/247reference/default.aspx (18/8/04)

[26] Pritch, Y., Ben-Erza, M. and Peleg, S., "Automatic Disparity Control in Stereo Panoramas (OmniStereo)", *The Hebrew University of Jerusalem, Israel*.