Evaluation of Virtual Reality Technology for Control Engineering

Prepared by : Bernard Kuc

Prepared for : The Department of Electrical Engineering at the University of Cape Town

29th April 1999
Revised 14th September 1999

Thesis prepared in partial fulfilment of the requirements for the Degree of MSc in Electrical Engineering

Acknowledgements

I would like to thank Mintek for providing the funds required to purchase the equipment, my supervisor Professor Martin Braae for his encouragement when things were not going too well.

Terms of Reference

This dissertation was required to evaluate the currently available low cost Virtual Reality hardware for its suitability in use in Control Engineering. The instructions given were:

- Review the currently available hardware.
- Purchase the most suitable hardware for setting up a virtual environment while keeping within the budget made available from Mintek.
- Using software incorporate the different items of hardware to function as a unified environment.
- Design and program a 3D virtual control room.
- Evaluate the performance and shortcomings of the hardware and the 3D environment
- Evaluate the suitability of the technology for use in a Control Engineering environment

The dissertation commenced in February 1998, and was due for the end of April 1999.

Synopsis

Virtual Reality technology has over recent years become available for commercial use. Where initially it had only been available to research centres and the military, it is now accessible to the industrial and commercial sectors. What this dissertation covers is the suitability of the low cost end of the Virtual Reality hardware for use in Control Engineering.

The use of Virtual Reality within Control Engineering could impart significant advantages over traditional control rooms currently in use in factories. The primary one, as rated by most commercial ventures, would be the cost saving of replacing all the physical hardware in a control room with virtual counter-parts in software. This is assuming that the Virtual Reality hardware will itself be of sufficiently low cost. The second is its ability to be used for operator training in instances where factors of safety and economics cannot allow for mistakes to be made on the real plant. A third advantage of a virtual control room, is its portability. As long as the factory can be accessed through some computer network, then the control room can be moved to any desired location. For example a copy of the control room for each factory can be maintained at head office where a chief engineer can occasionally check up on plant performance.

After a careful review of the available hardware, the following items were purchased:

- A Diamond FireGL 1000 Pro video accelerator card,
- A 5DT data glove,
- A Polhemus InsideTrak six degree of freedom tracking system with two receivers, and
- A General Reality CyberEye 200W head mounted display.

The virtual environment was built in Visual C++ due to the current lack of flexible and extendible higher level development tools, with OpenGL being chosen as the graphics library to perform the 3D scene rendering. There are currently only two 3D graphics libraries that are in common use, namely OpenGL and DirectX. DirectX, being promoted by Microsoft, is slowly gaining market share, however OpenGL is still the only graphics library being accelerated in hardware by the performance graphics workstation market. Choosing OpenGL makes the software easily portable to considerably more powerful graphics workstations.

The data glove uses the computers serial port to communicate with the computer. A class was implemented to encapsulate the commands exported by the data glove for easier use. The tracking system was a bit more intricate because it plugs directly into the PC. Windows NT does not allow user programs to directly access the computers hardware, and thus a device driver was written through which communication to the tracking system could take place. This was also encapsulated in a class with simplified functionality.

The HMD uses line sequential stereo, with the image for the left eye being drawn in the even lines, and the image for the right eye being drawn in the odd lines. To start off a stencil buffer is created. The stencil buffer is a single bit buffer, the size of the display area, that is kept on the graphics card. This buffer is then filled with alternate lines of 1s and 0s. When the image for the left eye is generated, the graphical system is told to only draw the pixels where the stencil buffer is equal to 1. Similarly when the image for the right eye is drawn, only pixels where the stencil buffer is equal to zero are drawn.

The virtual environment essentially consists of a square control room. The front facing wall contains a window through which the factory is visible. The factory consists of a heater, a liquid pump, a storage tank, a release valve, a thermometer and a flow rate meter. The left-hand wall contains three control panels. Each control panel has eight buttons through which the user controls how it interacts with the virtual factory. The three control panels, from left to right, control the temperature, flow rate, and liquid level subsystems of the factory respectively.

On the right-hand wall are three sets of three graphs. Each set belongs to its own control panel and displays the last 100 values for the setpoint, the control action, and the factory output. Actions performed on the PID panels correlate directly with effects of the actions being visible within the virtual factory. For example, increasing the temperature setpoint increased the flame size in the factory and the turns the colour of the liquid redder.

Performance testing of the hardware and the virtual environment began once the virtual control room was fully functional. Starting with the data glove, it was found that although the accuracy is sufficient for a virtual control room, the range over which orientation angles can be measured, is not all attitude and thus unsuitable for use in a virtual environment. The tracking system also proved sufficiently accurate over the specified range of 75cm. This range is however not large enough, as a range of 3m is required for a decently sized virtual control room.

The HMD was tested and found to have an effective resolution of 213 by 210 pixels per eye. This resolution made text unreadable without the user having to take a very close up look. The resolution inadequacy is however only a price factor rather than a technology limitation. The virtual environment was also performance tested. The scene complexity that can be achieved with a reasonable frame rate is highly dependent on the objects that are rendered and on what is visible at any specified stage. Checking for contact between the data glove and the buttons, and rendering 3D True Type fonts takes exceptionally long.

This dissertation concludes that the technology exists for use in control engineering. It is currently still not cost effective, but with time and larger volume cost savings, the technology will become ripe for mass usage within industry. Current low cost hardware it is not adequate, or reliable enough to be used to replace traditional control equipment, but can be used as a monitoring supplement. The technology is however advancing at a fair pace and attempts should begin for implementation on a real plant. Implementations should start off for just monitoring so that a failure in the virtual environment is not damaging to any critical application. However within about five years, the technology that can be used to build virtual control rooms that can totally replace traditional control equipment should become available.

Table of Contents

Chapter 1
Introduction

1.1 Dissertation Overview

Virtual Reality, very much a flavour of yesteryear, is slowly reaching its maturity, where it will become ready for mass usage in almost every environment. Now that most of the hype about the topic has dissipated, many people are considering the practical applications that can be achieved with today's hardware.

This study intends to find out if the technology that is currently available is of any use for Control Engineering. The low cost end of currently available Virtual Reality technology needs to be evaluated for its suitability, and applicability within the Control industry.

There are three predominant uses where Virtual Reality technology can be applied. The first use is for 'Simulation and Training'. A virtual control room has the ability to be used for operator training in instances where factors of safety and economics cannot allow for mistakes to be made on the physical plant. The other alternative of building a physical replica of a control room for a factory, especially one that operates just like the original, is often prohibitively expensive. With Virtual Reality, besides the initial computer costs, there is only the development time spent in building up a replica in software.

The second use is to replace a physical control room. The main advantage of replacing a physical control room would be the considerable cost reductions, assuming the Virtual Reality (VR) hardware was relatively inexpensive. Another advantage of having a virtual control room is that it does not need to be near to the physical factory. Thus a chief engineer at head-office can plug in his equipment and see the control room for any of the factories that the company owns. He / she is able to jointly see what the operator is seeing and help step him / her through any problem. Thus it is possible to avoid the inconvenience of having to physically go to the installation, or the difficulty of explaining a step by step procedure over the telephone. Finally, replacing a physical control room with a virtual counterpart makes redundancy inexpensive. If one computer fails, another computer with an identical copy of the software can easily and quickly be brought on-line.

The third use of VR technology is for plant diagnosis. The user can be enclosed in a virtual replica of the factory, and with the touch of a finger on a pipe is able to get the flow rate, temperature and density of the liquid in that pipe. With instant online information it will be considerably easier for him / her to find where a problem might be occurring.

Chapter 2 as a starting point gives a budget breakdown for this dissertation. Chapter 3 goes through each item of hardware and in detail describes how each item works. It also discusses the software that needed to written to enable the hardware to successfully function together.

Chapter 4 discusses the software that was written to build up a 3D virtual environment where the user can interact with a virtual control room. The control room contains a virtual factory so that the user can see the results of his actions in the control room occurring on a model factory. Chapter 5 then covers a detailed technical evaluation of the hardware individually and as a combined VR system. The report ends off with the conclusions in Chapter 6 and the recommendations in Chapter 7.

1.2 Literature Review

The literature review has been subdivided into a number of sections, namely: techniques, technology and applications. In the techniques section I describe papers of how researchers have gone about solving different problems encountered in Virtual Reality. In the technology section I review papers that describe new software and hardware technology being developed for the field of Virtual Reality. The applications section, which is by far the largest, containing reviews of 16 journal papers, describes research efforts into using the available Virtual Reality technology for different practical applications. This section illustrates some ingenious and novel uses that researchers have attempted to realise using current VR hardware.

1.2.1 Techniques Review

In [Rokita 1996] a method for adding another dimension of realism to images generated for Virtual Reality is proposed. The author begins by discussing the visual depth cues that influence the user's perception of space and reality, namely:

- occlusion,
- perspective,
- atmospheric effects,
- light and shade,
- binocular disparity,
- motion parallax,
- interposition,
- convergence (eye rotation towards the center of interest), and
- accommodation of the eye and the resulting depth-of-field effects.

The first four of these can be realised by conventional graphical workstations and form the basics of drawing a 3D picture. More powerful computers can render motion parallax, interposition, convergence and binocular disparity. Binocular disparity occurs where each eye sees a slightly different image. Motion parallax is when objects further away from the view point move less then objects closer to the viewpoint when the view point moves.

The author presents in this paper a method of adding depth-of-field effects to the image seen by the user. Depth-of-field is the effect of blurring that occurs to objects that are either in front of or behind the point at which the user is focussing. Traditionally, all computer images are totally in focus, from objects a few centimetres away from the viewpoint, all the way to objects a few kilometres away. This is not how the human eye sees objects as not everything can be in focus at the same time.

The paper discusses the reasons for the image blurring using basic lens theory of how objects beyond the focal point will not be focussed to a point on the retina but will be spread out over a larger area. Thus what would be a single rendered pixel of an image becomes a blurred ring of pixels if the object represented by that one pixel is out of focus. This leads to the conclusion that objects that are beyond or in front of the focal point need to be blurred before being displayed to the user.

An important requirement is that one needs to know where the user is focussing. This can in turn be found if we know which object the user is looking at. The author uses a system that is capable of measuring the direction that an eye is looking at to a resolution of 0.01°. By knowing the point on the display device that the user is looking at, the exact pixel of interest to the user can be calculated. Once this pixel is known, the value in the Z-buffer at that location will give the depth of that pixel. The Z-buffer is already used to calculate occlusion and interposition and thus this information is readily available to the computer system.

Thus the distance at which the user is focussing is calculated, and all objects beyond and in front of this distance need to be blurred. The size of the region, to which each pixel should be blurred to, is determined from the distance of this pixel to the depth at which the user has focussed. The author then proceeds onto a discussion of methods for generating defocus effects.

The first method calculated the intensity and depth of each pixel of the focussed image, and then each pixel of the defocused image is made up of the sum of the intensities from all the blur intensities of all the nearby pixels. Thus each pixel is drawn as a multi pixel splotch whose size is determined by how defocused the object needs to appear. This method is however computationally intensive. The second method, only suitable for ray traced images, uses a stochastic sampling technique. In this technique each ray that strikes an object gets separated into a number of rays to whose direction vectors a random direction is added whose size depends on how much defocused the object needs to be.

The author then proposed his own method of multiplying each pixel by a nine-pixel convolution mask. The number of times that each pixel is multiplied is determined by how much unfocussed the pixel should be. This filtering is easy to implement in hardware as it is simple, and is also a post processing technique unlike the second method proposed. The author ends by presenting a number of images showing the out of focus effect generated by this algorithm.
[Barrus 1996] describes a technique for creating very large environments that can be used and designed by many users. The authors use a concept of locales, which makes expansion of large environments possible with little or no degradation in speed for the end user.

Normally, creating virtual environments with a large spatial context, with many objects, and many users interacting with the environment simultaneously, causes considerable penalties in performance of the end user rendering system and of the underlying network architecture. The concept of locales is based on the idea that even in a very large virtual world, most of what a single user can observe at a given moment is local in nature.

One would expect a virtual world to be combined of a large number of small, localised activities. Locales divide a virtual world into chunks that can be processed separately. The user sees several locales at once, but these are generally the one that the user is in and those neighbouring this locale. As the user moves from one locale to the other, the environment and the set of neighbouring locales change seamlessly for the user. Locales are based around three key features, namely:

- Separate communication channels. Each locale is associated with a separate set of multicast addresses. This way, it only receives information on changes that occur in locales neighbouring it.
- Local co-ordinate systems. Each locale has it's own co-ordinate system.
- Arbitrary geometry and relationship. The shape, size, and orientation of locales can be chosen arbitrarily.

These three key features confer on the system a number of benefits:

- Efficient communication. Locale-specific communication channels lets a process attend to activities in some locales while completely ignoring what happens in others.
- Efficient culling. By designating only a small part of the virtual world as relevant, a locale neighbourhood serves as a highly efficient culling mechanism.
- Precise positions. Since each locale has its own co-ordinate system, a user always has high positional precision, even if the virtual world is very large.
- Easy combination of separately designed pieces. Locales are combined together to form a larger virtual world. Each locale needs to only know of the interfacing with neighbouring locales, and the author of the locale need not be concerned with other developments in other areas of the virtual world.
- The locally defined relationship between locales enables interesting effects, like having the inside of a building larger than the outside. Each local has a translation and rotation matrix for each of its neighbours that define how objects in that locale will be seen in the current one.

A problem faced when building a large multi-user system is the number of system updates that need to be maintained by each user. By using locales, the locale only listens to the multicast network addresses that could affect changes to it. Thus filtering out of unneeded information is done by the network card, which is designed for this task, and need not be done in software. The total processing that is required by a user is only proportional to the number of users nearby, and not by the total number of users in the system.

By having only local co-ordinate systems, one can avoid the problem of when a large virtual environment has to be divided within a fixed precision numerical value. The larger the system, the smaller the resolution of the objects themselves. Within a local, the positional precision of objects needs to only be subdivided over a small area, and thus no resolution problems occur.

The authors then proceed to describe a large-scale virtual world that they designed called Diamond Park. It contains a square mile of landscape, about a dozen buildings, two lakes, and two ornamental pools, all of which are made from a total of 62 locales.

The authors describe how they create a building with two vestibules through which the user needs to enter the building. These are designed such that the inside of the building cannot be seen from the surrounding park, and vice-versa. This way, neither the complicated outside environment nor the detailed interior of the building will ever have to be drawn simultaneously. This way the outside environment is a neighbour of a simple vestibule, which is a neighbour of another vestibule, which in turn is a neighbour of the building interior. With the computationally expensive building interior, and external environment not being neighbours, considerable computational time is saved.

The authors also describe transportation obelisks that have neighbours in different sections of the virtual environment. These effectively form virtual teleportation portals, where you walk into in one area of the virtual environment and walk out in another area. This is possible by defining neighbouring locales that are not physically nearby.

The authors thus show a method for improving the rendering speed and reducing computational load of collaborative, large-scale, multi-user virtual environments. Their research makes it possible to have virtual environments than can be supported by any single computational machine.

In [Forsberg 1997], the authors discuss a new method for interaction with objects in a virtual environment. They propose a method of gesture recognition because in some cases it is more direct than traditional widget-based interaction. The discussion focuses around a test application called Jot used for virtual reality modelling. In effect the authors wanted to create an interface as transparent as the interaction afforded by a pencil and a sheet of paper.

This application was developed because most people still prefer to design with paper and pencil, and only resort to computer-based solutions when they have to create or computationally analyse precise 3D models. Drawing with paper and pencil allows one to quickly and directly construct approximate shapes. One can sketch out complete illustrations without even having to change modes, as is common in computer based solutions.

Jot has no menus or buttons, but it presents the entire display screen as a blank canvas. In Jot, both the operation and its parameters are specified with strokes. For example, one can select, instantiate, scale and position a primitive object by drawing the primitive's silhouette. The interface was developed on the ActiveDesk that is a 1.2 meter by 0.9 meter desk which allows one to draw on it, similarly as one would on a drafting table. The large surface enable a wide range of drawing styles, from detailed fingertip motions to sweeping, full arm gestures.

Gestures are composed of multiple line strokes. When the system recognises a sequence of these line strokes, it performs the corresponding operation. The operations typically use the line strokes to further parameterise the operation. The information conveyed in the line strokes includes line lengths, drawing speed, and location relative to features of the image. For example, gestures that instantiate primitives provide enough information to select which primitive to create, determine its dimensions, and position it in 3D space.

Gestures are also used to instantiate widgets. For example to change an objects colour, one draws the letter C, to instantiate a colour wheel whose size and position are determined from the gesture. The user can then drag and drop colour from this colour wheel onto the objects. Objects are also grouped with the surface that they are placed on. If a surface moves, then any objects connected to it will also move.

Each different tool used by jot contains a physical stand-in that is attached to a 6 degree-of-freedom tracker so that the system knows which tool is being used. Thus tools such as pencils, erasers and knives can be used on the 3D object being designed. Putting all this together into an application creates a seamless and easy to use design tool.

Numerous other techniques have been developed to aid research into Virtual Reality and its application. I will now present brief summaries of some more of them. [Wiley 1997] presents a method for generating articulated figure motion for use of drawing virtual humans in virtual environments. Animation of figures in Virtual Reality is normally done by specifying in detail each frame in the animation sequence. Like cartoon animation this is a laborious process. In this paper, the authors present a method of interpolating intermediate frames, requiring considerably fewer predefined frames. By taking a source frame, requesting a destination frame, their approach calculates the path that each joint and limb needs to take to reach this final position while still maintaining consistency of the joints and limbs in the figure.

[Emering 1998] present a method for interaction inside a virtual environment using body actions. By keeping track of the limbs and joints of the user's body, a corresponding articulated figure could be drawn inside the virtual environment. This information is useful for collaborative work and human to human interaction within a virtual environment. It can however also be used to communicate with other virtual, computer-controlled humans. The authors present a method of interpreting body poses using a multilevel analysis. By starting with the simplest to calculate and moving down to the most processing intensive, a user's body language can be interpreted. They start by determining the centre of mass velocity, then the velocities of the hands and feet, then the centre of mass position, the positions of the hands and feet, and finally leaving the joint values to the end if the user's body pose has still not been interpreted.

A number of general tools for designing virtual environments exist. These are however so general that they are of little help in many projects, or not general enough for them to be used in specific projects. Any added generality in a design tool often makes it more difficult to use. [Bowman 1998] demonstrate a design tool that was developed for a small niche of virtual reality application, namely designing animal habitats. By placing numerous restrictions on the objects, textures, layout and positioning the authors were able to develop a tool that is simple enough for anyone to use. It enables the quick prototyping of virtual environments and evaluating their usefulness without committing much effort into their design.

[Cavazza 1998] present a paper on the motion control of virtual humans. They discuss the various types of virtual humans and their varying needs. Pure avatars are virtual humans that represent real humans for collaborative work in VR. The animation of these needs to correlate to the actual body and face of the user. Guided actors are virtual humans inside a virtual environment whose behaviour is controlled by an outside human. This type of virtual human would typically be used as a character in a computer game, whose actions and behaviour are controlled by the player.

Autonomous actors on the other hand are computer controlled. These need to demonstrate behaviour, typically perceiving other objects and virtual humans in the environment. These are typically used for animation sequences requiring virtual crowds of people. Finally the most intricate of virtual humans are classed by the authors as interactive-perceptive actors. These are aware of, and can communicate with, other actors and real people in the virtual environment. Communication depends on the actors' emotional state and facial emotions and speech may be co-ordinated between them. This final set of virtual humans will fill the new generation of computer games and virtual environments adding realism to the world of virtual reality.

The concept of virtual humans is taken further in [Kalra 1998], where the authors demonstrate techniques for constructing realistic virtual humans real-time. They show the contrasting requirements of real-time animation versus off-line rendering of virtual humans and describe methods for surface modelling, deformations, skeletal animation, locomotion, grasping, facial animation, shadows, clothes, skin and hair to ensure real-time rendering. The authors demonstrate numerous techniques for drawing of facial expressions, hands and articulated figures and how different methods effect the performance and frame-rate.

1.2.2 Technologies Review

[Pape 1996] presents a Virtual Reality Simulator for use in the development and testing of Virtual Reality applications. The need for such a simulator is identified by the author from the observation that many research labs have one or more workstations per user, and yet very few have sufficient Virtual Reality equipment due to their prohibitive costs.

This lack of Virtual Reality equipment could possible slow down the developments of applications utilising VR and ultimately the acceptance of Virtual Reality. Until VR hardware becomes lower in cost, the author feels that a simulator that will enable application developers and researchers to do their work on standard workstations until the final stages of testing is needed. With the Virtual Reality Simulator presented in this paper, optimal use of the Virtual Reality hardware can be made by eliminating the need for all the programming and debugging to be done on scarce and expensive resources.

The simulator operates by separating a virtual environment application into three basic parts:

- input (tracking and controlling devices),
- computation (simulation and management of the virtual world), and
- output (immersive display, spatialised audio and haptic feedback)

The input and output methods are what differentiate a VR application from desktop interactive graphics, and are also the components that are unavailable on ordinary workstations. The computation can be performed as is, but the inputs and outputs must be simulated using standard workstation hardware. Any limitations of the VR devices need to also be simulated as closely as possible.

Input to the VR system consists of 3D tracking data and control-device data. The tracking could be magnetic, acoustic, or mechanical, but ultimately all these systems report on the position and/or orientation of the user's head and other tracked objects. The simulator maintains this positional and orientation data for the user's applications and allows this data to be changed by the user using mouse and keyboard input. Although the mouse and keyboard can not be used as substitutes for the actual VR input devices, they allow the modification of the position and orientation data, howbeit in a difficult fashion, to test how the application will respond to the real devices.

The simulator also simulates the limitations of position and orientation in the VR system by limiting these values to a user configurable area. This simulates the limitations placed by the size of the room, the length of the Head Mounted Display cable, and the limited range of the tracking hardware itself.

Since a fully immersive view is not possible on a normal monitor, the simulator displays the view that will be available to immersive output device. The simulator can be configured to only show a limited resolution or field of view to closely match the target hardware of the application. In certain immersive VR environments, for example the CAVE and ImersaDesk, the display area is limited by projection screens. The simulator can thus be enabled to block out areas of the view that would not normally be available under such environments.

The simulator also provides a third-person viewpoint. This viewpoint allows one to view how the user can interact with his virtual world and also to judge size and distances relative to the user. In this view, icons for the user's head and other tracked objects are placed automatically by the simulator. The audio is also converted to a non-spatialised format and piped to the standard audio output.

Synchronisation between the components of a VR system needs to take place. If the development workstation is of a similar processor architecture to the target system then this is not a problem. However, normally the VR environment runs on a multiprocessor system not available to the developer. The Simulator simulates this by running each input, output and application system that would have been run on its own processor, as independent processes. In cases where the target system would use multiple computers connected using high-speed, low-latency network hardware, shared memory is used to simulate communication between the processes.

This architecture is useful because whether the user's application is running on the VR hardware or on the simulator, the application just accesses a certain memory location to acquire the inputs, for example the positions and orientations. This means that porting of the developed application will not require any source code modification to run on the target system as the application code does not make any device specific function calls. It also helps if certain VR hardware components are available on the development workstation. For example if a stereoscopic display is available to the developer on the workstation then the Simulator will just pass the data to the stereo display instead of simulating that component. In conclusion the author believes that this Simulator will speed the development of VR applications and thus ultimately the acceptance of this new technology for mass use.

[Salisbury 1997] presents the Phantom haptic interface. The Phantom has a stylus grip or fingertip thimble with which users can get force feedback and thus feel objects by probing them with their fingers.

The motivation for the Phantom interface is based on touch being one of the most fundamental ways in which users perceive and affect changes in the world around them. Touch interaction differs from all other sensory perception in that it is truly bilateral. Energy is exchanged between the user and the environment as the user pushes on an object and the object returns a resistive force. In this exchange, information and intent are conveyed in a physically direct and cognitively primitive way.

The Phantom interface permits users to feel the forces of interaction they would encounter while touching objects with the end of a stylus or the tip of their finger. Unique to the human's sensory modalities, the haptic system relies on action to simulate perception. While exploring an object, we actively scan our fingers across its surface and squeeze or palpate it to sense its physical properties. Thus, in almost all of the hand's activities, either to extract information from the environment, or to alter it, we use both the sensory and motor part of our haptic system.

Correspondingly, a haptic interface needs to sense our motor actions and display appropriate haptic images. In the real world, whenever one touches an object, it imposes forces back on the skin. These net forces, plus the posture and motion of hands and arms, are transmitted to the brain as kinaesthetic information. This is how we sense coarse properties of objects. The skin also detects finer detail, for example, spatial and temporal variations of force distribution on the skin, slipping of surfaces, fine textures, small shapes, and softness.

For a virtual environment (VE) to be realistic, it needs to match the abilities and limitations of the human sensory systems. It is known that the human tactile system can resolve vibrations of up to 1kHz, with sub-micron amplitudes being detectable at around 250 Hz. The kinaesthetic resolution in sensing the position of our fingers is about 1mm, with the ability to discriminate differences of about 10 percent for velocity and 20 percent for acceleration. The motor system's bandwidth for controlled motion is less than 10Hz, and the maximum controllable force exerted through the fingers is 50 to 100 Newton.

In matching human capabilities, the Phantom haptic interface trades off complexity to ensure high fidelity. Based on the observation that people perform satisfactorily in exploring and manipulating the world through a rigid stick, the Phantom operates on the principle that forces generated through a point contact, contain sufficient spatial and temporal information for users to understand the object that they are interacting with.

The Phantom interface measures motion along six degrees of freedom and can exert controllable forces on the user along three of these freedoms. Because the device does not constrain motions within its workspace, and because its inertia and friction are low, free motion feels free and comfortable. Two methods have been developed to encode 3D shapes. The first ensures solidity while touching polyhedra. It defines a local tangent plane constraint surface that is impenetrable and coplanar with the currently touched facet of the object. The second method models the stylus as a line segment to take into account its orientation in reflecting the forces. Faceted shapes can be made to feel smooth by perturbing the effective surface normal using a technique similar to Phong shading in graphics. Friction and simple textures can be added by perturbing the normal and tangential contact forces.

One of the first broad application areas of haptic technology is training people to perform real-world tasks. At MIT research is being performed in using the Phantom haptic interface together with a virtual workbench to train electronic technicians. Trainees can see a circuit board, feel the components of the circuit board with a probe, use a virtual multimeter at various contact points, and even haptically operate switches. At the Center for Human Simulation surgical procedures on the eye and knee are simulated based on highly detailed and realistic anatomical models. Boston Dynamics have developed a surgical simulation of an anastomosis procedure. The user can look at and feel a virtual blood vessel, use forceps to grasp a needle, and even suture the vessel.

Virtual Reality is a fast growing technology field with constant improvements in hardware and software. Technologies in Virtual Reality change fast and often to accommodate new research trends and interests. I now present a summary of a number of newer technologies that are pushing the forefront of Virtual Reality equipment.

[Cress 1997] demonstrate how they are able to add another sensory dimension to virtual reality applications by creating a vestibular display. The authors show how the realism of Virtual Reality experiences is enhanced when the user feels the tilt and roll as it occurs in the virtual environment. Although the graphical displays in flight simulators are very convincing, often complicated manoeuvres leave the user disorientated because there is a conflict between what the eyes are showing him / her and what the body is telling them. The authors show how applying a voltage just behind the ears of the user can stimulate their 8th cranial nerve. By altering the strength and polarity of the current flowing through the user's head they are able to generate the sensation of tilt and roll in the user. This greatly enhances the Virtual Reality experience.

[Ikei 1997] describes a vibration tactile display. The display consists of five by ten piano wires arranges in a ten by twenty millimeter rectangle. By applying a frequency of 250 Hz and changing the duty cycle of the input frequency, the extent by which the wires extend is changed. Thus by individually controlling the extension of each wire a surface texture can be created for the user's finger to feel. This vibration tactile display is connected to a computer mouse, so by moving the unit, the texture image displayed to the users finger changes based on where the cursor is pointing. Thus the user can move the unit over some virtual surface and feel a virtual texture through their fingertip.

A current project underway in Japan to ultimately enable anyone to telexist through a network is described in [Tachi 1998]. The author describes research currently underway for the remote control of robots through a virtual environment. Based on an idea at General Electric from the 1960, where a user would wear an exoskeleton that would magnify his / her strength by a factor of 20, current research is enabling users to use a remote robot for object manipulation and receive feedback from the robot. As the user grasps an object via the robot, the user experiences a counter force on his / her fingers, enabling the estimation of grasping forces.

[Billinghurst 1998] discuss three different techniques for displaying information on a wearable computer using the see-through head mounted display (HMD). The first is by having a head-stabilised view. In the head-stabilised view the data displayed in the HMD does not move with movement of the user. The second technique is by enabling the display to change with motions of the head. This technique, called body-stabilised display, is a bit more difficult to implement as it requires some form of head tracking, but it allows a user to pan through a large amount of information by only moving their head. This creates an effective multi-million pixel display on a low resolution HMD.

The third and final technique is the so-called world-stabilised view. Here the information displayed to the user is dependent on where the user's location is and what they are looking at. One example of this technique is described where a user gets annotations to the buildings around a university campus as he / she walks through it. The authors then proceed to describe how a user's ability to access and retrieve data increases with each of the above display system configurations.

A new method for creating tactile displays is shown in [Asamura 1998]. The authors show how the spatial accuracy and resolution of a tactile display is increased when the pressure actuators are mounted in solid housing rather than the traditional independent pin arrays. The authors also show another technique capable of stimulating both the first level and the forth level of pressure receptors in the skin. By stimulating both they are able to display much higher resolution of tactile images with a considerably larger sensitivity range. Their technique consists of an array of variable air pressure pumps that stimulate the first layer of receptors in the skin. The housings of these pressure actuators are also capable of vibrating. These vibrations generate larger amplitudes in tactile pressure and thus stimulate the bottom layer of skin tactile receptors.

[Goldiez 1999] discuss the advancements in desktop PCs and how these are making their way into the high performance graphics arena of Virtual Reality. The authors state how between May 1996 and May 1998, the Unix workstations improved in performance by only 70% in their graphics capabilities while PC had increased their graphics rendering capabilities by 2500% over the same period. Funded by the US Army, the research shows how seriously the effects of the PC's technological advancements, driven by the gaming industry, are being taken by the performance graphics community. The authors evaluate the possibility of replacing military simulation from dedicated Unix performance workstations to PCs using OpenGL and Windows NT.

The final application of Virtual Reality that I will describe comes from [Slater 1999] where the authors describe a research project into using Virtual Reality to help people overcome their fear of public speaking. Their research is still in the formative stages, and the authors describe the effects on patients that a crowd of virtual people can have. The patient is faced with a group of virtual avatars that surround him / her in a semicircle. Currently under user control, the group of avatars is capable of displaying positive or negative reinforcement to the patient. The avatars can display, smiles, clapping, nodding, standing ovations for positive reinforcement, and can yawn, frown, twitch, walk out, and look around when they need to display negative feelings. The effects of these emotions are evaluated by the authors on numerous subjects to test their perceived self-rating.

1.2.3 Applications Review

In [Leigh 1996] a paper is presented that deals with the design of a Virtual Reality environment in which multiple users can operate together towards a common goal. A large-scale virtual control room, as is the ultimate aim of this research, would also require a collaborative environment in which multiple operators could work together.

This paper outlines the functionality and uses of a collaborative virtual environment designed by the authors called Calvin (Collaborative Architecture via Immersive Navigation). The authors investigate the techniques required to support general collaboration in persistent virtual environments. The techniques include the representation of virtual co-presence, video and audio teleconferencing, VR interfaces, and database technology for sustaining persistence in virtual worlds.

Calvin supports two different perspectives resulting from different camera parameters to view and manipulate a design. Calvin defines an egocentric view as the mortal's view, and an exocentric view as the deity's view. This metaphor defines the relationship between participants in the two views and determines their roles in the collaboration. For example, a mortal might be a student in the environment, whereas a deity might be a teacher who possesses capabilities not provided to the student.

Co-presence between collaborators located at different laboratories is addressed through the use of Avatars. Avatars are the graphical persona donned by each participant. With each Avatar being visually different, participants are able to identify each other within the Virtual Environment. In Calvin, avatars are composed of a separate head, body and hand. With information from the tracking devices mounted on the users, gestures such as nodding or waving is possible within the virtual environment. Due to there being two types of users, mortals will appear as dwarfs to the deities, and a deity will appear as a giant to the mortals.

Persistence is supported within Calvin allowing collaborators to work synchronously and asynchronously from each other. Calvin manages a central repository of information maintaining the states of ongoing design projects. The repository contains a collection of objects occurring in the scene, the avatars, and a series of files containing information on the state of each object within the environment. Each user in turn maintains a user definition file for each scene that they are involved in, that contains the user's location, avatar, and whether they are a mortal or deity.

Object manipulation is performed with a wand. Users can move objects by pressing a wand button and then moving their arm in the direction that they want the object to move. Users can also move within the virtual environment. A downward directed collision algorithm is used so that users are able to climb stairs and other objects. For example if a user climbs a set of stairs, then Calvin automatically lowers the entire landscape to simulate the effect of raising the user.

Using this scheme allows one user to stand on an object, while another user lifts up that object. If one participant is a deity, then it is also possible for the deity to put their hand down on the ground and for a mortal to walk onto the hand. The deity can then move their hand with the mortal on it.

Menu selection can be performed in two ways in Calvin, namely the Virtual Visor and speech recognition. The Virtual Visor is used as a gaze directed input device. To make a selection, the user gazes at the appropriate menu option and presses a button on the wand. This provides for a considerable advantage over traditional methods of selection by reducing the number of operations requiring user arm movements. Many virtual systems induce user fatigue as a result of prolonged and repeated arm raising to make menu selections. In addition to the visor, Calvin augments menu selection with a speaker-independent speech recognition system. Audio feedback confirms visor menu selection and speech commands.

Calvin has been tested on three different collaborative design efforts. The first is in the design of the computer room at NCSA to house the Infinity Wall (a four-screen active stereoscopic projection environment and a number of workstations.) The second is in the design of the test-bed room for the Supercomputing 95 conference, consisting of a CAVE (Cave Automatic Virtual Environment,) two ImmersaDesks, seating for 200 people, and two Infinity Walls. The final collaborative test of Calvin was in the design of octagonal command and control centres (called Octmods) in a prototype of the Integration Test-bed room at the Advanced Research Projects Agency headquarters.

The author ends off with a few more ongoing research efforts in which Calvin is being used, one being learning in narrative-based virtual environments. This is an application if VR to create a family of educational environments for young users. It is based on constructionism, where real and synthetic users, motivated by an underlying narrative, build persisting virtual worlds through collaboration.

[Cremer 1996] present a driving simulator developed for use in research for both Virtual Reality and Medical Applications. The paper presents the obstacles encountered in developing a realistic VR driving simulator and how they were overcome.

For a driving simulator to be effective and realistic it must have:

- high-resolution visual, auditory and haptic feedback,
- numerically accurate, real-time simulation of complex physical systems,
- modelling and control of believable agents and scenarios,
- modelling of large-scale virtual environment databases, and
- a software architecture providing real-time database access and runtime co-ordination of multiple threads.

Although, for all of the above requirements considerable progress has been made, the authors believe that it is in the development and integration of all these elements that a successful simulation can be made.

The Iowa Driving Simulator (IDS) is a high-fidelity, operator-in-the-loop, ground–vehicle simulator. The simulator consists of a vehicle cab mounted inside a dome that rests on a 6-degree of freedom motion base. The base provides acceleration cues with a maximum instantaneous acceleration of 1G. The driver feels haptic feedback through the steering wheel, accelerator and the brake pedal. Within the dome, high-resolution graphics are projected onto a panoramic screen in front with 190° horizontal field-of-view, and one behind the cab with a 65° field of view. Directional audio cueing is provided by a multi channel audio system.

The IDS provides a safe, virtual environment in which to study driving behaviour. Since it began operating in 1992, the IDS has been used mainly in areas of transportation, virtual prototyping and medical research. It has also been used as a test-bed for simulator technology research. In transportation it has been used to test the use of in-vehicle equipment such as intelligent cruise control, advanced collision warning, roadway departure warning devices, heads-up displays and traveller information systems.

High fidelity dynamical models enable the use of the IDS to design vehicles by using the simulator instead of a physical prototype. The IDS has also been used to test the effects of various drugs on a person's driving ability.

Virtual driving environments require for the roadway traffic to be simulated on a microscopic level. Cars do not just jump from lane to lane, but take time along some path from the one lane to the other. The behaviour of any simulated vehicles must be consistent with what would be expected from a human driver. Global properties such as density and flow should also be controllable. Specific situation must also be created which require the co-ordinated behaviour of a number of vehicles. These situations must also be replicated to compare the performance across multiple tests.

Subjects will drive unpredictably, thus it is not possible to pre-assign roles that will be played by certain cars. For a car to run a red traffic light, it must be near the robot. It cannot just materialise there when the user's virtual vehicle approaches the intersection. In IDS the participating vehicles are selected at run-time. As the subject approaches the intersection, a director is used to conscript an approaching scenario vehicle to run the robot. This type of scenario planning requires co-ordination of the ambient traffic to create consistent circumstances for the event.

Driving simulation requires scene databases of large geographic regions with properly constructed roads, smooth terrain, natural foliage, and appropriate cultural features. To enable the simulation of the physical interaction between the ground and vehicle, driving surfaces must be modelled with very high resolution. To achieve this, in IDS three separate, but interrelated databases have been used.

The visual database contains texture-mapped polygons used in the image generation of the scene. The logical database is used for scenario control including a high level representation of the road network, the vehicles, buildings, signs, trees and other objects. The terrain database is used for calculating the vehicle dynamics. It contains a high-resolution grid of terrain elevations. It also contains information on the road surface type, roughness, friction and soil properties.

Thus the authors conclude that a deterministic, real-time software system that integrates components for user interaction, simulation, and scenario and scene modelling has been created. This simulator is not only useful in improving driving simulation, but also benefits the research areas of believable agents which can be transferred to target applications involving simulated walking humans and training in the operation of complex machinery.

In [Rosenblum 1996], two independent applications of Virtual Reality are discussed by the authors. The first involves the use of VR to help train fire-fighting crew aboard navy ships, and the second is to aid the development, prototyping and visualisation of potential new ship craft.

Ship fires are very dangerous if not dealt with immediately, especially on heavily armed navy ships. Often a few extra seconds means the difference of a number of lives. Thus fire-fighting crew aboard navy ships undergo intensive training. This training is however very expensive due to the scarcity of safe fires on ships to practice on. This makes this sort of training very suitable for VR. By modelling a variety of ships, fire fighters are able to familiarise themselves with ships or parts of ships that they have little or no actual experience with.

Because VR is intended to aid the training, the fire fighters must be allowed to concentrate on fire-fighting and not on navigating the environment. This places considerable strain of the VR system. Since Navy fire fighters are trained in fire-fighting procedures, the virtual environment is less a training tool than a mission preparation aid. In the virtual environment, fire fighters can practice standard procedures, practice tactics, and try out strategies without endangering personnel or property.

To demonstrate VR for shipboard fire fighting, several portions of a decommissioned ship were modelled. When 3D sound was added to the simulation, it was found that it aided inexperienced users move rapidly even to a fire they could not see. The environment was modelled with collision detection between the user and static objects in the VR environment. Travelling through the virtual environment was done through the use of a glove avatar, and navigation followed a "fly where you point" metaphor.

Statistic tests were performed which showed considerable improvement in performance of fire fighters that used VR for mission rehearsal. In a navigation test, these fire fighters performed about 30 seconds faster over a two-minute run to reach a predetermined location. All the members of the traditionally trained group, given directions to the fire, made at least one wrong turn, whereas only one of the VR training group did so.

The second part of this paper discusses how Virtual Reality was used in the design phase of a new ship. In an effort to design a new, better, and less crew intensive battleship, the US Navy has started on designs of the new ship as well as deciding on the contractors who will be designing and building the ship. As part of the review of submitted tenders, the authors were part of importing one of these designs into a virtual environment. They were supplied with AutoCAD files and had to convert these into a 3D virtual model.

The interior of the ship contained rooms only, as no detailed machinery, piping or other features were part of the design at this stage, and took 4000 polygons to model. The exterior hull required a further 15200 polygons. During this conversion it was found that a large section of the hull had the normal vectors pointing in the incorrect direction, that a few doors were missing, and that a step was missing on one of the stairs. It was also found that two abutting decks failed to match and a curvature feature not previously seen was also noticed.

Thus besides aiding in finding problems in the design, porting the design to a VR environment made it easier to choose among the bidding parties, as to which would be the best design.

A very novel application of Virtual Reality is presented in [Hodges 1996]. This paper presents an active and successful application that is financially viable and operational without the need to wait for better, cheaper, for faster hardware.

The authors start by discussing why a fear of flying is a serious problem. Between 10 to 25 percent of people suffer from a fear of flying. An additional 20 percent of the people that do fly require alcohol or sedatives to be able to do so. Besides the social embarrassment that accompanies a fear of flying there are also significant financial ramifications. In 1982, about 1.6 billion dollars of possible revenue was lost by airlines due to the portion of people that refuse to fly.

Treatment approaches include anxiety management methods, provision of accurate information regarding aeroplanes and flying, and exposure techniques. Actual exposure to flying after initial training in anxiety management has been consistently reported as effective. Exposure is usually provided in stages, with clients first practising going to the airport, seeing and hearing aeroplanes taking off and landing. Patients might later actually sit in a stationary aeroplane. Ideally an actual experience of flying conducted with the therapist would be the culmination of the therapy.

The difficulty and expense of using actual aeroplanes and flights for exposure have daunted many therapists and researchers. Fear of flying therapy is expensive. The therapist's time in organising exposure to aeroplanes and the cost of airfare make treatment prohibitively expensive. Also, not every airline will allow a person to sit in a stationary aeroplane to help them overcome their fear of flying.

Virtual Reality exposure, in which the patient is exposed to a virtual environment containing their feared stimulus, has been shown in a controlled study to be an effective treatment for acrophobia (the fear of heights.) The authors extended Virtual exposure to the treatment of the fear of flying by designing a virtual aeroplane for the patient, who wears a head mounted display with stereo headphones. The process of using virtual reality for therapy has many advantages, amongst them are:

- Therapist time is reduced because treatment can take place in the therapist's consulting rooms.
- Inside the VR environment, precise control over every element is possible. The patient can have a smooth ride, or a rough ride, as well as the ability to terminate the flight at any moment if anxiety levels rise too high.
- The patient can sit in a stationary plane for as long as they want to, to calm down in a virtual aeroplane.
- Since tests are conducted in the therapists' consultation rooms, patient confidentiality is preserved.
- The costs of airfare are substantial, whereas the hardware required for this simulation if of low cost.
- Many patients are also more willing to undergo therapy in a simulator, where they would never consent to boarding an aeroplane.

The major stages in the development of the virtual aeroplane were:

- creating a visual model of the passenger cabin of a commercial aircraft;
- creating an environment for the aircraft to fly through;
- designing the control sequence software for the different stages of an aeroplane flight; and
- designing the sound effects.

To be convincing the aeroplane had to be as realistic as possible, this meant that the model had to match the proper size and scale of a real passenger cabin of a commercial aircraft. Detail such as the fabric texture, colour of the seats, location of the seatbelt signs, aisle width, and window locations were important to provide a realistic feel to the patient.

A virtual world through which the aeroplane will fly also needed to be created. This scenery would be view by the patient through the window. This environment contained a runway, airport buildings, and other scenery. As the plane ascends, a texture map of an aerial photograph of a city is used for the ground, and clouds are added to give the impression of height. The complete environment designed by the authors consisted of 4770 polygons of which 1919 were textured.

The path of the plane's flight is defined as a cubic spline, with the location of the plane along this spline being dependent on the plane's speed and time from takeoff. The authors were capable of achieving 11 frames per second when the patient was looking around the cabin, and 19 frames a second when the patient was looking out of the window, on a Crimson Reality Engine (RE1).

Because there is a natural order of events that occurs in a flight, for example the plane taxis on the runway before taking off, to preserve realism, not everything could be controlled by the therapist. The therapist could extend the duration that the plane waits at the airport, can keep the plane taxing around the airport indefinitely, can choose from a smooth or rough takeoff, and numerous weather conditions during the flight. However once the plane is in flight, the plane had to land before it is able to take off again. This was done to preserve realism. At any stage however, the therapist is able to abort the simulation.

Gradually as the patient became more comfortable with flying, adverse conditions and aborted landings were simulated so that the patient would be capable to cope with adverse conditions on an actual flight. Bad weather conditions were accompanied by a dark sky and thunder sound effects, as well as a shaking view. Sound cues are crucial to elicit the proper fear response, thus from a quite plane before taxing, to the engine sounds, to the pre-flight and pre-landing briefings by the crew, even to the screeching of the wheels at touch down, all were incorporated into the design.

The paper then discusses a case study of a patient that underwent Virtual Reality therapy for her fear of flying. The patient wore a Virtual Research VR4 head mounted display with head tracking being done by the Polhemus Isotrak II. The authors present a battery of seven different tests that the patient had to do to evaluate her anxiety over flying. These tests were done before therapy commenced, after the patient was taught anxiety management techniques, after the VR treatment and then a month after all the treatment had passed.

The authors show results of how after the anxiety management techniques were taught, the patient's anxiety dropped by almost 30%, and then after the VR treatment they dropped a further 50% below the levels after the anxiety management training. Even though anxiety levels had increased a month after treatment had finished, anxiety levels were still lower than they were after only the anxiety management training.

There are many things that cannot be done well in Virtual Reality. High quality Virtual Reality equipment is expensive, and virtual environments are still cartoonish and not very realistic. There is always a lag between head movement and the tracker's response being visually shown. The areas over which trackers operate are also very small, and haptic cues are also very limited. However what the authors showed was that one needs to find applications were these problems are not critical.

VR technology is capable of giving a person the illusion that they inhabit a computer-generated world. For stimulating fear cues, the level of realism does not need to be very high as has been shown by this research. This makes VR ideal for phobia therapy.

In [Gaither 1997], the authors write about an effort between the Naval Research Laboratory and Mississippi State University on visualising ocean data. Prior to this research, researchers would schedule batch calculation jobs of ocean circulation data on a Cray C90 supercomputer. The supercomputer would save the results and the oceanographers would then analyse this data. If any changes had to be made to this data, then another batch job had to be scheduled. The authors decided to improve this iterative process by writing a visualisation tool that would display the data real-time as it was calculated by the supercomputer.

The data generated by the Cray C90 is time varying data of ocean height and velocity. This makes it very suitable for real time visualisation. At each time increment, the user can see the data, and if problems arise, then the user can either abort the visualisation or make minor changes to the model used to calculate the data. This shortens the cyclic approach of simulation, analysis and model modification.

The authors present a three-stage pipeline. At each time increment, data from the supercomputer is sent to a remote computer that performs an extraction task. This extraction task analyses the data and highlights important features for future analysis. This data is then sent to the Visualisation task, which displays the data in a virtual environment. Within a 3D environment, the user is quickly able to notice current movements, velocities and wave heights. Thus the user is immediately able to make minor modifications to the model, and test out various effects that these will have on the data.

Thus with the Cray C90 polling the Visualisation workstation for model modifications, the authors have created a real-time data analysis tool that considerably improves the ease and speed with which new oceanography theories can be tested out.

[Lehner 1997] describe how Caterpillar, the earth moving and construction vehicle company, have built a collaborative virtual design facility. Virtual prototyping decreases the time between the design phase and the market introduction of the new product, at the same time allowing for improved quality. By using a virtual prototype, engineers can evaluate and test the vehicle prior to building an iron prototype.

Caterpillar has used virtual prototyping in many design efforts. What the authors concentrate on in this paper is extending this system to several geographically distributed sites. Collaborating using audio and video within the shared VE helps eliminate poor designs and make interactive redesign possible.

The project uses ATM networks with IP multicast to deliver the bandwidth required for high quality audio and video between collaborating individuals. The virtual prototyping environment is based around a dynamic simulation package called Dynasty. Dynasty allows engineers to test and drive CAD designs real-time. Real-time operator inputs to the system include steering, throttle, gears, brake, clutch, lift and tilt. A virtual proving ground was also designed, and was based on Caterpillar's actual proving grounds. Other virtual environments include a mining pit where virtual gravel can be loaded into a truck, a road holding test environment, and a virtual factory.

To test the distributed collaborative aspect of this project, two remote virtual reality systems were used. The first was a Cave Automatic Virtual Environment, which is a 10 by 10 by 9-foot cube with screens on all four walls and floor, and surround sound. Two SGI Onyxes with a total of four Reality Engines were used to render the graphics for the walls and floor, and a pair of stereoscopic glasses (CrystalEyes,) were used to generate the illusion of depth of view. A head tracking system is used to provide information on where the user is within this environment.

At the other end of the collaboration was a Responsive Workbench. The graphics were rendered by a SGI Onyx with a single Reality Engine. These graphics are projected onto a reflective workbench. With a pair of stereoscopic glasses, the image is made to form a 3D solid form on the workbench. The input of the system includes head tracking, hand tracking, a CyberGlove for detecting finger movements, and a stylus.

The software system runs in a loop, getting all the tracking inputs, updated positions of the vehicle, selected options, and other variables. With each iteration this data is used to generate the view that will be seen from the user's tracked position. Rendering is maintained at 15 frames a second. Each site communicates with the other collaborating sites using multicast addresses to which any changes to the virtual environment are sent. Thus each member of the team sees what the others are doing to the environment.

For the collaboration to effectively take place, the users must be able to communicate with each other. Thus at each station, an SGI Indy workstation is used to capture audio and video of the user and to multicast it to the other users. The virtual environment then renders the video as a dynamically changing texture on a rectangular surface that is placed at the current position of the user from who the video is arriving. Thus each user can see where every other user is within the virtual environment, and are also able to teleconference with the audio and video channels to suggest changes to or make comments on the design. To add additional usefulness to the virtual environment, each user has a 3D-pointing device so that the other users know what is being referred to.

Testing out the system, the authors found that about 1Mbit/second transfer speed was required per user station. On high-speed fibre optic networks, this application can be scaled to a fair size. Thus the authors conclude that their distributed virtual reality system allows remotely located engineers to collaborate in real-time. Any number of participants can share a virtual environment and see each other's video transmissions at the position and orientation of the corresponding viewpoint. Audio transmission also provides natural communication.

In a series of papers on applications of the Responsive Workbench, four different uses for this technology are described. [Wesche 1997] propose a method of data visualisation using the Responsive Workbench and 3D texture maps. The authors begin by stating that currently all the powerful data visualisation systems are desktop orientated, relying on a mouse and keyboard for input. The authors intend to describe how a tailor made interactive visualisation application running on a Responsive Workbench provides an ideal workspace for engineering applications, filling the gap between immersive and desktop environments.

The Responsive Workbench operates by projecting a computer-generated, stereoscopic image off a mirror and through a table surface. Using stereoscopic shutter glasses, users observe a 3D image displayed above the tabletop. Typical methods for interacting with virtual objects on the workbench include speech recognition, gesture recognition, a simulated laser pointer, and head and hand position and orientation tracking.

In the visualisation tool developed by the authors, the data to be visualised is converted into a regular grid structure that can be loaded into texture memory. The simulation results appear as the vertices of finite volume elements. These are difficult to visualise, and the application interpolates these values to the closest 3D grid points using a scan conversion algorithm. This 3D texture can then be rendered in hardware by an SGI Onyx. Iso-surfaces that are generated allow a deeper insight into the structure of the scalar data.

The SGI Onyx's 16Mbyte texture memory allows for a 256 x 256 x 256 x 8bit 3D cube of points that is rendered without noticeable latency. Cutting planes, which are very useful for understanding 3D volumetric data are thus easy to generate, as only a portion of the 3D texture can be rendered, resulting with the cross-section being visible at the edge where rendering of the texture ended.

Daimler-Benz has used this visualisation tool in the visualisation of fluid dynamics inside a cylinder during the injection process, for the visualisation of crash test sequences, and for airbag simulations.

Another application of the Responsive Workbench is presented by [Rosenblum 1997] where the authors describe how it is used to aid in military planning, and troop co-ordination. The authors attempt to provide situational awareness for the logistical task of directing the movement of US Marines and material over rugged terrain, day and night, in uncertain weather.

The authors describe how currently the US Marines still undertake command and control predominantly with paper maps and acetate overlays. This is a cumbersome, and time consuming process, considering that detailed maps and overlays can take several hours to print and distribute. This is why the authors have developed an alternative using the Responsive Workbench.

The authors developed map-quality 3D terrain images of an area in California (62 by 72 kilometers). Using clip-mapping techniques, the authors texture the terrain with line-drawing maps. Objects to be placed on the terrain were represented by 3D models or standard icons used by the marines. The authors used the simulated laser pointer (wand) input device of the responsive workbench for interaction with the system.

When the laser intersects the landscape and the user moves their hand, the terrain will scroll with the motion, as if the landscape was attached to the laser pointer. Rotating the wand rotates the map, and moving the user's hand towards or away from the map, zooms it in and out respectively. Certain modes of the wand could also be used to pick up and move objects, to query objects, and to measure distances and heading amongst other task common to military planning.

In [Frohlich 1997] the authors describe an application where the Responsive Workbench is used for scheduling tasks. When scheduling the construction of a building or the assembly of a car, planners must always step through the construction or assembly sequence in their minds, imagining the transformation of space over time. Furthermore, planning and scheduling require tight collaboration among multiple participants. The Responsive Workbench displays CAD models in 3D and allows their assembly and disassembly. This functionality provides a good basis for the development of a virtual-production, modelling environment.

Typically, bar charts or network diagrams are used to represent construction schedules, and static 3D modules to represent buildings. However the economy of construction and its design entwine. A particular design determines how it can be built, and available construction methods affect the design. The authors propose a 4D model (space + time) that combines the 3D model and the schedule. Changes to either are reflected immediately in the 4D model.

The 4D model consists of work packages, where each work package consists of an activity and it's corresponding 3D components. It knows the duration required to complete the activity and which tasks must be completed before it can be started. Users schedule a work package by clicking on it. The system then inserts the item on a bar chart and the locations of the affected components are shown in the 3D model. Throughout the scheduling process, the workbench always displays the 3D model's corresponding state, giving the schedulers visual cues of the planned work's spatial complexity.

Users can model different scenarios in the 4D environment by lengthening, shortening or rescheduling activities in the bar chart. At any time, users can visualise the flow of production for a schedule in the 3D model by moving the time slider in the bar chart back and forth.

In [Bryson 1997] an explanation of how the Responsive Workbench can be used as a wind tunnel simulation tool is shown. A model of the aircraft is placed within the virtual wind tunnel, and appears as a scaled model on the Responsive Workbench. The aircraft and its airflow visualisation appear to sit above the surface of the Virtual Workbench in 3D. The display quality is very high, allowing examination of fine detail in the airflow.

One of the virtual wind tunnel's most important capabilities lets one reach into the simulation and interactively manipulate various visualisation tools. This interaction feels natural on the Responsive Workbench, as one can see one's hand and the cursor moving together to pick up and manipulate the various visualisations. This level of natural interactivity facilitates intuitive exploration of the flow simulation.

A literature summary of various attempts of using Virtual Reality for therapy is provided in [Potel 1997]. The paper starts off with a brief description of SpiderWorld. SpiderWorld is an immersive routine environment, like a home kitchen, that introduces realistic looking spiders that the patient can observe, manipulate, or even squash as part of exposure therapy.

Flight simulators and other VR training systems have shown that a user interacting with a well designed virtual environment is capable of gaining new skills and knowledge. As the user interacts with the environment, the environment acts back on the user. Therapeutic applications can harness this transformative potential to help therapists retrain patients whose anxiety over certain situations or things has become disabling.

Two of the original pioneering applications of Virtual Reality therapy were for aerophobia and acrophobia. Patients who were afraid of heights spend time in a virtual environment that simulates situations such as being in an elevator or on a bridge, or looking out a window on an upper floor of a building. For the fear of flying patients spend time in a flight simulator of sorts, which replicates the various stages of an air flight until the patient feels comfortable with flying.

Research has shown that even if the images rendered look cartoonish, systems that allow the user to change their point of view with normal head movements compensate to some degree for low visual quality. VR relies on a multisensory engagement, not just in compelling graphics. In SpiderWorld, the spider must not only look like a real spider, but also must scurry or crawl like a real spider too. Being able to feel the spider with a haptic interface would make the experience even more real. Currently physical stand-in objects are used. For example in SpiderWorld, the spiders physical presence is kept by a spider toy that has a tracking device attached. Thus the user can pick it up and throw it down and see it flying to the floor.

Some VR applications provide physical props, such as chairs and guard rails that add to the realism of the experience. Depending on the therapy, sound and animation are also of considerable help for Virtual Reality therapy. For therapy to be effective, the therapist must also be involved. The therapist must be able to in some manner experience part of what the user is experiencing. By doing so, they can verbally communicate their suggestions and encouragement to the patients. The therapist must also be able to control the virtual therapy session. In SpiderWorld, the therapist can trigger the spider to crawl and hop, and to move the spider around the virtual environment, moving the spiders closer to the patient as the therapy sessions progress.

Besides phobias, Virtual Reality is used also for other disorders. VR is being used to treat distorted body image, a major cause of diseases such as anorexia nervosa and bulimia. Two main therapy methods are integrated together. Cognitive-behavioural therapy addresses the patient's dissatisfaction with their bodily appearance, and visual-motor therapy is geared to shift the patient's conceptions and awareness of his or her body from negative to positive.

Pain is both a physical and an emotional health issue. A project at a Burns Center is showing how VR is helping patients cope a bit better, especially during particularly painful burn dressing changes. One child reported that he had "forgotten" his pain while manipulating objects and travelling in the virtual environment.

Patients with neurological disorders can also benefit from VR. Patients with Parkinson's disease suffer from a disorder called akinesia, where the patient suddenly looses the ability to walk. This is overcome by placing an obstacle in the person's path, which triggers the walking response. Augmented Reality glasses can be used to constantly scroll obstacles in the user's view, when the patient looks down to help the patient walk.

VR is also being used to help disabled patients understand and better cope with their situation. It blurs the line between ability and disability by providing new ways for them to act and interact. Blind people can navigate or find and manipulate objects using audio or tactile cues. Paraplegic and quadriplegic patients can be taught to use a wheel chair in a VR simulator.

Despite all these advantages and uses of VR, the technology is still expensive, not really making it suitable for commercial therapy. The negative psychological effects of VR exposure therapy also need to be evaluated before mass-market use of the technology takes place. Psychologists and psychiatrists have been involved in therapeutic VR research and development since its inception, and they are likely to be even more involved as the technology branches out and ethical issues become part of research agendas.

Numerous other applications to virtual reality have been written about in literature, having described a few in detail, I will now summarise a few more to give a bit more breadth to the research. [Allison 1997] describes research into creating a virtual reality gorilla exhibit to let zoo visitors experience life as a gorilla while still maintaining the privacy of the animals there. Visitors assume a gorilla's persona and enter a virtual gorilla exhibit where they are accepted as one of the group of gorillas. The human visitor then has to obey all the social dynamics of the community or have the virtual gorillas respond adversely as they would in real life. Visitors to the zoo thus are able to learn so much more about the animals by experiencing a small part of their lives than any number of information boards could provide.

[Johnson 1998] demonstrate a test bed multi-user virtual environment. In an attempt to demonstrate multi-user virtual environments for the Supercomputing 97 conference, the authors set up a virtual garden on an isolated virtual island, where the user's were able to tend the garden and experiment with different cultivation techniques. The ultimate goal was to demonstrate a collaborative effort by users from different countries working on a joint project, in this case, weeding a virtual garden. Numerous research groups from across USA and Europe took part in this joint exercise which culminated in all the groups doing the hokey pokey in the centre of the virtual island. This demonstrated the effects of high volume data traffic and intercontinental information lag on collaborative effort in a virtual environment.

A currently commercial use of Virtual Reality technology is in the motion capture of human figures for computer animation. [Delaney 1998] describes the use of magnetic and optical tracking devices for tracking the figure motion of subjects. The joint and limb positions of a subject are captured and are then used to create realistic movement in movie special effects. The author describes the MoCap motion capture system which has been commercially used for animation in a number of TV commercials, the people on the decks in the movie "Titanic", the Moxie character on MTV, computer games such as Madden NFL 99 and Knockout Kings, movies like "Lost in Space", and "Terminator 2," and many other Hollywood productions.

[Schulz 1998] describe a number of applications of Virtual Reality for engineering simulations. They demonstrate how BMW has used VR in evaluation of impact dynamics; the residual stress and distribution of thickness in sheet metal forming; the vibration analysis of motor vehicles; and acoustic visualisations to predict the noise levels within the motor vehicle due to its design and structure. The authors also discuss the techniques that had to be developed to enable real-time visualisation of these high–precision designs without loosing simulation accuracy to maintain a real-time simulation.

An interesting study is performed and described in [Tromp 1998] where the authors research the behaviour of groups of humans in virtual environments. They use researchers from across the world and place them together into a virtual world and analyse how they react. The authors check if social dynamics such as rank, leadership and politeness are preserved in the anonymous virtual environment. They also evaluate the role that the image of an avatar plays on the social dynamics of the group. Does a more detailed and humanoid avatar indicate superiority of the corresponding person? The authors cover many interesting questions of how people interact together in collaborative virtual environments.

1.2.4 Review Evaluation

Having covered 33 different papers, numerous interesting applications have been covered in the literature review. The different techniques and technologies discussed show what is available to be used for Virtual Reality research and how it can be used to its optimum. The applications present novel ways of using existing hardware and software for practical and useful applications. From these one can gather information on how to perform Virtual Reality research as well as highlighting areas in which Virtual Reality still has severe shortcomings. Although there were no journal papers describing research into mimicking control rooms in a virtual environment, the problems encountered and solutions used in the above papers are relevant to this research too.

Chapter 2
Budget

Below is a rough estimate of the budget for this project. When equipment procurement began in April 1997, the total budget for the project was R30 000. Due to fluctuations of the Rand / Dollar exchange rates from R5/dollar to R6/dollar during July 1997, a project that started within budget ended up over budget. The main fault being that all equipment had to be purchased in dollar amounts. Also an unexpected repair on the Head Mounted Display further increased the expenditure.

The project was supposed to have been totally funded by the money received from Mintek, however due to the above circumstances a fair portion of the costs were covered by the University of Cape Town. As with all forms of computer equipment, costs constantly fluctuate thus current figures need to be checked regularly. These figures should thus only be used for relative comparisons between the individual items.

Expenditure

Item	Price
General Reality Company CyberEye 200W	$ 2 500
Polhemus InsideTrak	$ 1 500
Additional receiver for Polhemus InsideTrak	$ 250
5DT Data Glove	$ 500
Diamond FireGL 1000 Pro	$ 250
Shipping	$ 200
Total $	$ 5 200
Total R	R 31 200
Import Duty and Tax (25%)	R 7 800
Courier for Repair of CyberEye 200W	R 1 500
Total	R 40 500

Income

Source	Amount
Mintek	R 30 000
University of Cape Town	R 10 500
Total	R 40 500

Chapter 3
Project Hardware

3.1 Hardware Overview

Three basic entities form part of a Virtual Reality system. The first is the display system. This is responsible for fooling the primary sense involved in the Virtual Reality concept, namely sight. The second is the collection of components used by the user to manipulate the environment. Examples of these would be position trackers, data gloves, wands, and other pointers. Finally, the third entity consists of components that provide auxiliary stimulation responses to the user, for example force-feedback devices that can restrict the motion of the user.

The primary one of these three entities is the virtual display. It is the most important component in convincing the user of the reality of the virtual environment. Many aspects affect the level of reality that is achievable with the virtual display system. The resolution and quality of the graphics makes the difference between a believable environment, and an unrealistic one. Other factors that can influence the perceived reality include field of view, comfort and weight, frame refresh rate, and colour depth. For this project a head mounted display manufactured by "General Reality Company," namely the CyberEye 200W, was purchased. This is a near low-end display with two, colour liquid crystal display (LCD) panels mounted in a helmet like structure.

The next entity in a VR system, consisting of the collection of components that are used by the user to manipulate the environment, is there to be able to interface with the environment and to affect changes to it. The basic requirement is to have the view change with motion of the users head. If the user moves his / her head, then the view should change to display the area of the virtual environment that the user is currently looking at. This can be achieved with a three degree of freedom orientation sensor. By knowing the elevation, tilt and azimuth of the head it is possible to tell exactly where the user is looking at. However, to be able to move within the environment (towards or away from an object) a six-degree-of-freedom sensor that can also measure the distances along the three axes X, Y and Z, from some reference point is required.

Thus, a six-degree-of-freedom tracking solution is required. The cheapest, and the one purchased for this project, is the Polhemus InsideTrak tracking solution. With a limited range and accuracy, it gives the position and orientation of a receiver with respect to a transmitter module. To be able to interact with, and modify, the environment in some manner, an additional receiver is required. This second receiver then acts as a pointing device inside the virtual environment. Additionally for this project, an inexpensive data glove with finger flexure sensors was acquired. By mounting the second receiver module of the tracking system on the data glove, a virtual pointer that can move with respect to both the virtual environment and the user's viewpoint, namely the head, is created.

The final entity in forming a virtual reality system is there to provide additional feedback (other than visual) to the user. This would consist of, for example, pressure on the fingertips when pressing a virtual button, or motion restriction when the user's hand touches a wall. Although systems for motion and force feedback do exist, they are currently expensive, cumbersome, and difficult to utilise. However mainly because of the costs involved, no additional feedback devices were acquired to form part of this project.

3.2 Data Glove

3.2.1 Overview of Operation

Shown alongside, the 5DT Data Glove measures finger flexure and the orientation (roll and pitch) of a user's hand. It achieves this by using five fibre-optic cables (whose transmissive optics change as the cable is bent) and two tilt sensors.

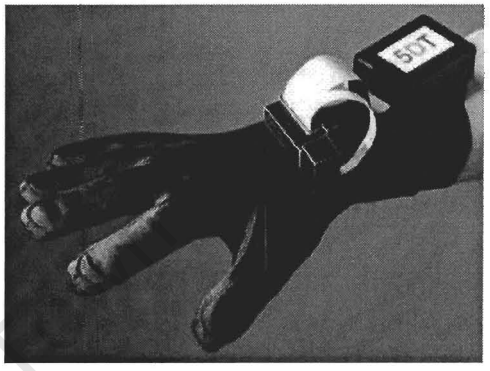


Figure 3.1: Image of glove

The data glove features an on-glove controller with which one can communicate using the computer's COM port. The glove repeatedly samples the signal in the fibre optic cables and stores 8-bit values that indicate by how much a finger has been bent. These values can then be retrieved by the user by sending a command to the controller of the data glove.

The glove operates using a single fibre-optic cable per finger that runs along the outside of the finger. A transmitter is connected to one end of the fibre optic cable, and a receiver is connected to the other. The amount of light returning is sampled by the on glove micro-controller. This value is then made linear so that when the finger is straight it returns an 8-bit value of zero, and when it is bent by about 270 degrees, it returns a value of 255.

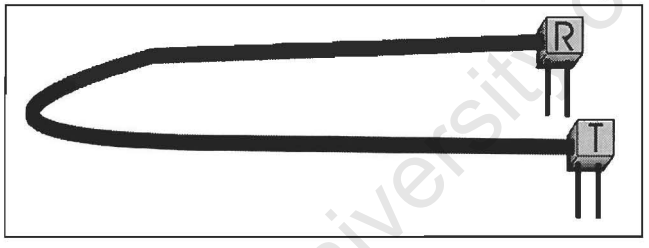


Figure 3.2: Diagram of fibre optic cable in the open hand position

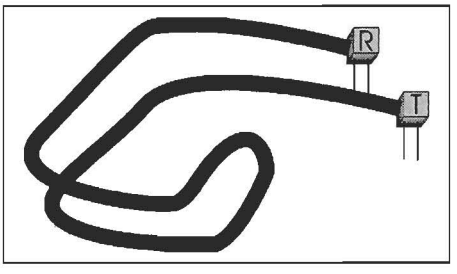


Figure 3.3: Bent fibre optic cable

3.2.2 Computer Interfacing

The data glove has an on-glove controller that communicates to the PC using a serial link that is connected to the PC's COM port. The controller's communication protocol consists of waiting for the PC to send it a single byte command. Depending on what this byte is, the controller will either wait for further instructions from the PC or reply to the request. Thus, communication takes place through the transfer of strings of 8-bit characters.

Although not all of the functionality provided by the glove is being used by my project, the table below gives a short description of the capabilities of the data glove. Note that all values are given in hexadecimal.

Command String	Returns	Description
41	55	Resets Glove to Command Mode
42 XX	XX	Used to test serial IO
43	<data>	Set to Report Mode[1]
44	<data>	Set to Continuous Data Mode[2]
45	-	Set to Mouse Emulation Mode[3]
46	-	Set to High Speed Mouse
47	<info>	Request Glove Info[4]
48<data>	55	Upload gestures[5]
49	<data>	Download gestures[6]

During Report Mode (43)		
41	55	Return to Command Mode
Else	<data>	Request the Newest Data

During Modes 44, 45, 46		
41	55	Return to Command Mode
else		Ignored

[1] The data string returned is as follows:
SY F1 F2 F3 F4 F5 TP TR CS
Where : SY = 80h is the leading header
: Fn are the raw values of the corresponding finger, where (n = 1) is the thumb. (n = 5) is the little finger.
: TP is the raw pitch value (centre = 128)
: TR is the raw roll value (centre = 128)
: CS is the checksum value: the XORed result of F1 to F5, TP and TR

[2] The data string is the same as in [1], but is emitted continuously.

[3] All commands should still be sent at 19200 baud, even though the device transmits data at 1200 baud.

4 The info structure looks as follows:

42 45 : Header
<Version Major> : as the 1 in 1.04
<Version Minor> : as the 04 in 1.04
<Capability Word> : use two bytes together (little endian), where
Bit 12 = Hand pitch and roll angles available.
Bit 6 = Simple left hand finger curvature.
Bit 0 = Simple right hand finger curvature.

5 Allow 1ms delay between finger values for the device to write the new values in non-volatile memory. Input buffer overrun will therefore not occur.

6 The data string is composed as follows

F0 F1 F2 F3 F4 : Finger threshold (thumb first, on right hand)
00 : Reserved byte set to zero
<Mask LB><Match LB> : Left Button Down gesture
<Mask DLB><Match DLB> : Left Double Click gesture
<Mask RB><Match RB> : Right Button Down gesture
<Mask DRB><Match DRB> : Right Double Click gesture

with <Match> and <Mask> set as follows :

Bit 0 : little finger
Bit 1 : ring finger
Bit 2 : middle finger
Bit 3 : index finder
Bit 4 : Thumb
Bits 5-7 : Unused

While in Mouse Mode, the device compares the actual finger value with the stored threshold value (F0 to F4). The result is ANDed with the corresponding bit in the Mask, and then XORed with the corresponding bit in the Match. If the result is zero for all five fingers, the gesture is matched, and the corresponding mouse command is transmitted, i.e.:

Mask	Match	Action
0	0	This finger always matches
0	1	This finger never matches
1	0	Match when finger is open
1	1	Match when finger is closed

3.2.3 Source Code Implementation

To ease the re-use of any code written for the data glove, all of the functions required for retrieving data from the data glove are encapsulate as member functions of a data glove class. The class is called CGlove, with the class definition residing in the file "Glove.h", and the implementation in "Glove.cpp". The class consists of one member variable and five member functions, which are defined as follows:

```
class CGlove : public CObject
{
public:
   CString Write(char Out[], DWORD Length);
   CString Read(char *Out, DWORD Length);
   CString Open();
   CGlove();
   virtual ~CGlove();

protected:
   HANDLE hndFile;
};
```

CGlove::CGlove()

The constructor is empty because there is nothing that needs to be done when an instance of the class is initialised. Setting up communications with the data glove from within the constructor is undesirable because we want objects of this class to be created without any need for error reporting to take place in the constructor. If communication with the glove commences through another member function, then it is possible to have better error reporting features, as the object will already be in existence.

```
CGlove::CGlove()
{

}
```

CString CGlove::Open()

This function is the first member function that a user of this class will call. This function sets up a handle to the COM port, through which the user will communicate to the data glove. Using a handle to the COM port has two advantages. Firstly, it is the only way of accessing the COM port under Windows NT because NT does not allow direct hardware access. Thus, all command to the data glove must go through the serial port device driver. Secondly, it allows for the use of the Windows ReadFile and WriteFile functions that simplify serial communication. The file handle is stored so that it becomes accessible to other functions, which are then able to use this handle when they need to communicate with the glove through the COM port

The next action performed by this function is to set up the timeouts on the COM port. The timeouts have been changed from the default values because the glove should be able to respond to requests within 5ms and so a longer timeout value in unnecessary. Thus to avoid long delays if the glove is faulty or incorrectly connected the timeouts have been set to 100ms.

If this function fails, then it reports the error as a string. Numeric error codes are not used within this project to facilitate continued development. Future development should not force the developer to have to learn a new set of error codes to be able to translate them to an error message. Passing error messages, although less efficient, is however more user friendly.

```
CString CGlove::Open()
{
   BOOL Result;
   CString Return;
   COMMTIMEOUTS CommTimeouts;

   hndFile = CreateFile(    "\\\\.\\COM2",           // Open the Device "file"
                    GENERIC_WRITE | GENERIC_READ,
                    FILE_SHARE_WRITE | FILE_SHARE_READ,
                    NULL,
                    OPEN_EXISTING,
                    0,
                    NULL);

    if (hndFile == INVALID_HANDLE_VALUE)
     return "Failed to Connect to COM Port 2";

   CommTimeouts.ReadIntervalTimeout = 100;
    CommTimeouts.ReadTotalTimeoutMultiplier = 100;
    CommTimeouts.ReadTotalTimeoutConstant = 100;
    CommTimeouts.WriteTotalTimeoutMultiplier = 100;
    CommTimeouts.WriteTotalTimeoutConstant = 100;

   Result = SetCommTimeouts(hndFile, &CommTimeouts);
   if (!Result)
   {
      Return.Format("Error setting timeouts on COM port 2, Error Code = %d",
GetLastError());
      return Return;
   }

   return "";
}
```


CGlove::~CGlove()

When an object of this class is no longer needed, and is deleted or goes out of scope, then the destructor will be called. Because the object will no longer exist, the user will no longer be able to communicate with the data glove, as this is the only class through which communication can take place. The handle to the COM port is therefore closed and is returned back to the operating system's collection of available resources.

```
CGlove::~CGlove()
{
   CloseHandle(hndFile);
}
```

CString CGlove::Read(char *Out, DWORD Length)

This is the first of two functions that are the main work horse functions of this class. This function takes two parameters. The first is a pointer to a block of memory, and the second is an indicator of the number of bytes that the caller is expecting to read.

This function uses the aforementioned ReadFile function to read data from the serial port. The ReadFile function checks to see if there is any data waiting on the serial port. If there is, it reads in this data and waits for the next set to arrive. It continues waiting for more data until the requested amount of bytes have been read in (specified in the variable Length) or a timeout occurs, which indicates that it read in less characters than the caller asked for. Either way it reports the number of bytes that it read in, in the BytesRead variable.

Next, there is a check to see if the ReadFile function was able to complete and if it read in the requested amount of characters. If either of the two tests fail it returns an error string stating what the problem is. If no errors occurred then it returns an empty string.

```
CString CGlove::Read(char *Out, DWORD Length)
{
   BOOL Result;
   CString Return;
   DWORD BytesRead;

   Result = ReadFile(hndFile, Out, (DWORD) Length, &BytesRead, NULL);

   if (!Result)
   {
      Return.Format("Error reading from COM port 2, Error Code = %d",
GetLastError());
      return Return;
   }

   if (BytesRead != Length)
   {
      Return.Format("Bytes read, %d, is not equal to the number of bytes
requested", BytesRead);
      return Return;
   }

   return "";
}
```


CString CGlove::Write(char Out [ ], DWORD Length)

This function is similar to the Read function. Given a buffer of characters and an indicator of how many values there are in the buffer, this function uses the WriteFile function to output these values to the data glove using the serial port. Similarly to the Read function, if the WriteFile operation fails, or the function writes less characters onto the serial port than it was asked to, it returns an error string to that effect.

```
CString CGlove::Write(char Out [ ], DWORD Length)
{
   BOOL Result;
   CString Return;
   DWORD BytesWritten;

   Result = WriteFile(hndFile, &Out[0], (DWORD) Length, &BytesWritten, NULL);

   if (!Result)
   {
      Return.Format("Error writing to COM port 2, Error Code = %d",
GetLastError());
      return Return;
   }

  if (BytesWritten != Length) return "Bytes written is not equal to the
number of bytes sent";

  return "";
}
```

A class that can be used to communicate with the data glove is now available. No higher level functionality is provided inside this class to retain the generality that this class provides. Other classes sit on top of this one and use the member functions provided to set up the data glove and get the finger positions.

3.3 Tracking System

3.3.1 Overview of Operation

The tracking system used in this project is the 'Polhemus InsideTRAK' tracking system with an additional receiver. It is a system consisting of five components.

- A PC card that plugs into a standard ISA connector on the PC motherboard,
- A Transmitter Frequency Module,
- Two receivers units, and
- One transmitter unit.

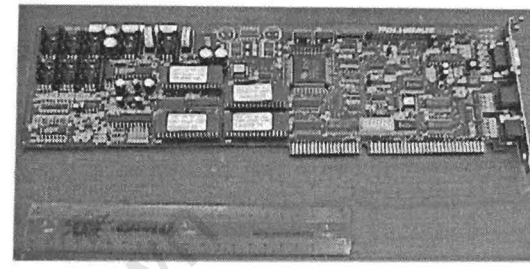


Figure 3.4 : Polhemus InsideTrak PC Card

The PC card is where all of the intelligence of the system resides. The user communicates with the card using a user selectable base port address (selected using DIP switches), and predefined offset addresses. The PC boards sends out electromagnetic pulses through the transmitter module, monitors the field strength on the receivers and from this data, is able to compute the position and orientation of the sensors with respect to the transmitter.

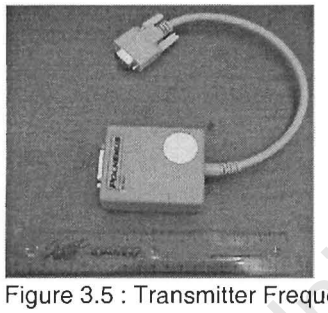


Figure 3.5 : Transmitter Frequency Module

The Transmitter Frequency Module is a separate unit that provides the unique carrier frequency for the tracking system. There are eight different such modules which enable the use of up to eight transmitters within close proximity of each other with minimal interference between the signals.

Both the transmitter and the receiver contain three coils at right angles to each other, enclosed in a plastic casing. By sending a pulse in each coil in the transmitter sequentially, and checking the strength of the resultant fields in the three coils of the receiver, one can calculate both the orientation and the position of the receiver.

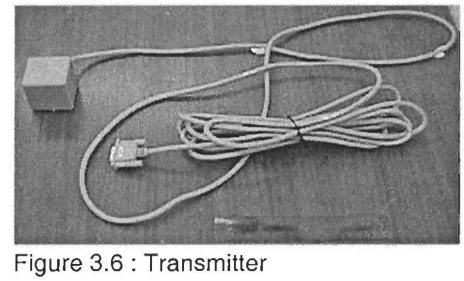


Figure 3.6 : Transmitter

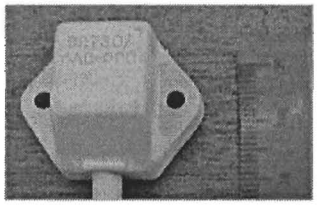


Figure 3.7 : Receiver

Large metallic objects must be kept away from both the transmitter and the receiver because the card uses electromagnetic fields, which it sends out from the transmitter and picks up in the receiver, to measure position and orientation. Metal object too close will distort the magnetic field to give inaccurate readings. The documentation supplied with the system recommends that all large metallic objects be kept at least three times the separation distance of the receiver and transmitter away from both of them.

3.3.2 Tracker Specification

Range

The Polhemus InsideTrak system provides the specified accuracy when the receivers are located within a radius of 76.2 cm from the transmitter. Operation of up to 152.4 cm is possible, with reduced accuracy.

Angular Coverage

The receivers are all-attitude.

Static Accuracy

1.3cm RMS for the X,Y or Z receiver position, and
2.0° for the receiver orientation.

Resolution

0.0003 cm / cm of range.
0.03 ° / ° of range

Latency

12 ms unfiltered from centre of receiver measurement period to beginning of transfer from output port.

Output

Output is software selectable including extended precision. Cartesian co-ordinates of position and Euler orientation angles are standard. Direction cosines and quaternions are selectable. English or metric units are also selectable.

Update Rate

One Receiver : 60 updates / second / receiver
Two Receivers : 30 updates / second / receiver

Interface

ISA Bus. 16 bit wide FIFO for output and 8 bit wide FIFO for input.

Carrier Frequency

The InsideTRAK may be configured with any one of eight discrete carrier frequencies to allow simultaneous operation of up to eight transmitters – receiver pairs in close proximity. The carrier frequencies are selected by using different Transmitter Frequency Modules, and can be any of the following:

- 8013 Hz
- 10016 Hz
- 12019 Hz
- 14022 Hz
- 18027 Hz
- 20032 Hz
- 24039 Hz
- 26042 Hz

Operating Temperature

10 °C to 40 °C at a relative humidity of 10% to 95% non-condensing.

Physical Characteristics

PC Card
10.7cm * 33.8 cm * 1.6 cm
1.13 kg

Transmitter Frequency Module
6.4 cm * 7.6 cm * 2.54 cm
105 g

Transmitter
5.3 cm * 5.3 cm * 5.8 cm
270 g

Receiver
2.79 cm * 2.29 cm * 1.52 cm
17 g

Power Consumption

15 W continuous total.
2.2 A from PC's +5V DC supply
20 mA from PC's –5V DC supply

3.3.3 Computer Interfacing

Registers

The user can communicate with the Polhemus InsideTRAK card using four consecutive address registers starting with the base address. The base address is specified using DIP switches on the card and can range from 0x0000 to 0x03ff in steps of four bytes. The registers also behave differently depending on whether the card is being written to or read from. They are defined as follows:

Read Registers

Offset 0:
This is the offset of the output buffer. Use 16 bit wide read operation to get data off the FIFO output buffer, otherwise the hardware FIFO may not remain synchronised across both bytes.

Offset 1:
This is an 8 bit Status Register. If bit 0 is clear then it indicates that the output FIFO buffer is empty, and if bit 0 is set then there is data on the output FIFO buffer that can be read in by the user. If bit 1 is clear then it indicates that input FIFO buffer (the buffer receiving commands from the user) is full and can no longer accept any more commands. Otherwise, if bit 1 is set indicates that there still is space on the input FIFO buffer.

Write Registers

Offset 0:
This is the offset of the input buffer that accepts user commands. To modify the operation of the InsideTRAK card, 8-bit wide write operations must be used.

Offset 1:
Request immediate data from station A. Write a dummy value of 0 to this address, and the InsideTRAK card will write the data associated with station A onto the output FIFO buffer from which the user can read it in. Make sure that data from previous requests has been removed from the output FIFO buffer before sending this request.

Offset 2:
Request immediate data from station B. Write a dummy value of 0 to this address, and the InsideTRAK card will write the data associated with station B onto the output FIFO buffer from which the user can read it in. Make sure that data from previous requests has been removed from the output FIFO buffer before sending this request.

Offset 3:
Send software synchronisation pulse. To write a software synchronisation pulse to the InsideTRAK card, first send a software sync mode command "y2<cr>" (all the commands available to the user are discussed in Appendix A) to the InsideTRAK input FIFO buffer, and then send a dummy byte of 0 to base address + 3.

3.3.4 Windows NT Device Driver

Having dealt with the hardware side of interfacing to the Polhemus InsideTRAK tracking system, the software now needs to be able to access it from within Windows NT. A device driver is however required because Windows NT does not allow the user to directly access the hardware. A device driver was not supplied with the tracking system, and thus one had to be written.

Why Windows NT?

Considering all the added complexity involved in writing Windows NT programs, one answer that needs to be provided is why Windows NT was chosen. The reason for choosing Windows NT can be best explained by the following points.

1. It is much easier to write a graphical user interface in a Windowing Environment, and so a DOS program would have been considerably more complicated.
2. The author is more familiar with Microsoft Operating Systems than with any others like SunOS and IRIX.
3. Windows run on relatively inexpensive desktops in comparison to other operating systems.
4. Linux is complicated to setup.
5. Most of the hardware that I was purchasing came with either example programs or source code written to run under a Microsoft Operating System.

This leaves me with a choice between Windows 95, Windows 3.1, Windows 98 and Windows NT.

1. Windows 98 was not yet released when this project commenced.
2. OpenGL (the graphics library used to draw the virtual environment) is not supported on Windows 3.11.
3. A stable version of OpenGL is supplied free with Windows NT.
4. Windows NT does not crash as often, or as severely as Windows 95 when utilised as a development environment.
5. Windows NT supports a true multi-threading environment. This is required to retrieve information efficiently and reliably from slow items of hardware (e.g. Tracking System and Data Glove).

How the Driver Is Written

When writing device drivers it is very difficult to debug the software without additional expensive software. This is because the device driver runs inside the kernel of the operating system. This means that it has the same authority as the operating system, and can thus do considerable damage if not written correctly. This also makes debugging difficult, because no user-mode debugger is allowed to interfere with the running of the operating system kernel, and thus also the device driver.

For this reason the device driver is made to be as simple as possible. The less that is implemented in the device driver, the less chance there is of it failing. Also, considering the relatively slow update rate of the tracking system, there is no need to have the added speed advantage that would be gained by implementing more functionality within the device driver.

All that is required is a driver that can act a gateway to the hardware. All it needs to do is accept a port address and a value, and write out this value to the given port address. A read operation just requires the port address and it returns the value found at this address. A simple driver like this, called GenPort, already does exist as an example device driver that comes as part of the Device Driver Kit for Windows NT.

GenPort supports six operations, three read operations (8 bit read, 16 bit read, 32 bit read) and three write operations (8 bit write, 16 bit write, 32 bit write). The read operations require two values, a port offset address from a pre-specified base address, and a buffer in which to return the read in value. The write operation also requires two values, a port offset address from a pre-specified base address, and a value to write to this port address. A brief explanation on how the device driver is written is provided in Appendix B.

Compiling and loading the Device Driver

Below is a step by step guide on how to compile and install a device driver under Windows NT. Due to a lack of stronger market forces a more user-friendly approach does not exist and this procedure must be followed through.

Step 1: Software Installation

- Install Visual C++
- Install the Windows NT DDK (Device Driver Kit)
- Install the Windows NT SDK (Software Developers Kit)

Step 2: Environment Variables Correction

When Visual C++ installs it creates a batch file that sets up all the environment variables that are needed for DOS based compilations, e.g. LIB and Include directories. Edit the \DevStudio\VC\bin\VCVars32.bat file and make sure that the paths specified under each section are correct.

Step 3: Environment Variables Setup

- Execute \DevStudio\VC\bin\VCVars32.bat
- Execute c:\SDK\setenv c:\SDK (where c:\SDK is the location where the Windows NT SDK was installed)
- Execute c:\DDK\bin\setenv c:\DDK (where c:\DDK is the location where the Windows NT DDK was installed)

Step 4: Source Code Compilation

At the root directory of the device driver project, type "build" which will compile the device driver and link in all the required libraries. If any errors are reported, they need to be corrected until the project compiles successfully. The device driver can then be found in \DDK\Lib\I386\Free or \DDK\Lib\I386\Checked, depending on whether the device driver was compiled without or with debug information.

Step 5: Device Driver Transfer

Copy the device driver, i.e. the '.sys' file to the \WinNT\System32\Drivers directory, where all of the device drivers are kept.

Step 6: INI File Execution

An INI file contains the information that will be written into the registry so that the computer knows were to find the device driver and how to load it. It also contains parameter values for use by the device driver like the Base Address and the Port Range. Below is an example INI file that is used by GenPort.

```
\Registry\Machine\System\CurrentControlSet\Services\GenPort
    Type = REG_DWORD 0x00000001
    Start = REG_DWORD 0x00000002
    Group = Extended Base
    ErrorControl = REG_DWORD 0x00000001
    Parameters
        IoPortAddress = REG_DWORD 0x00000300
        IoPortCount = REG_DWORD 0x00000020
```

The above information tells the operating system that the device driver is a SERVICE_KERNEL_DRIVER (Type = 1), that it is started by the Service Control Manager during operating system start-up, and that it belongs to the Extended Base Device Driver Group. It also sets the base port address to 300 hexadecimal and the port range to 32 bytes. These two values will be accessed from within the DriverEntry routine every time the device driver starts up.

To execute this INI file and load its contents to the registry enter the following command at the DOS prompt , "regini genport.ini". After the first time that this is done, the computer needs to be rebooted so that the system realises that there is a new device driver present. If changes are hereafter made and the regini command used again, reboots are not required. It is only the first time when a new device driver is entered into the registry that a reboot is required.

Step 7: Device Driver Start and Stop

To stop a device driver, the "net stop" command is used followed by the device name, and to start the device driver the "net start" command is used. Both of these need to be executed at the DOS prompt.

E.g. net start genport
net stop genport

If the device driver fails to start, or hangs the system once started, the source code needs to be edited to remove the error.

3.3.5 Tracker Class Implementation

Having a simple device driver, the next step is to get it to function from within Visual C++. As with CGlove, the intention was to write a self-contained class that communicates with the tracking system. By being self-contained, other classes that might need to get data from the tracker do not need to know anything about how the transfer takes place. All the information that is required is how the corresponding member function of this class works. The list of possible commands supported by the InsideTrak, together with descriptions on how to use them, are available in Appendix A.

General Operation

```
class CTrack : public CObject
{
public:
   CString GetActiveStation(BOOL * One, BOOL * Two);
   CString SetActiveStation(BOOL One, BOOL Two);
   CString SetCont(BOOL Yes);
   CString SendSync(void);
   CString GetPos(short * Ch1, short *Ch2);
   CTrack();
   virtual ~CTrack();
protected:
   CString _ReadW(short * Dat);
   CString _WriteB(UINT addr, char Dat);
   CString _ReadB(USHORT Addr, char * Dat);
   CString Read(SHORT * Dat, SHORT * Length);
   CString Write(char * Dat, SHORT Length);

   HANDLE hndFile;

};
```


The class has twelve member functions and only one member variable. Of these, only seven member functions are accessible to the calling system. These are the functions that will be used to provide a simpler and transparent interface to the device.

- GetActiveStation reports which of the two stations are currently active.
- SetActiveStation switches both stations either on or off independently.
- The function SetCont tells the tracking system to continuously report the position and orientation.
- The SendSync function causes the tracking system to send a single set of data to the output port on the InsideTRAK PC card.
- The GetPos function returns the position and orientation for both stations.
- CTrack is the constructor that initialises the class when an instance thereof is created.
- ~CTrack is the destructor that cleans up when the calling entity is finished using an object of this class.

The remaining five member functions are there as helper functions used within the class to simplify its implementation.

- _ReadW, reads a 16 bit value from the only port capable of doing so, i.e. Port 0
- _ReadB, reads an 8-bit value from the specified port offset.
- _WriteB, writes the given value to the provided offset from the base address.
- Read, reads a whole string of values from the output buffer using 16-bit reads.
- Write, writes a whole string of characters to the input buffer at Port 0.

Lastly, hndFile is the one and only member variable. It gets filled in the constructor with a pointer to the GenPort device after it has been opened. The handle is then used by other parts of the class to communicate with the device.

CTrack::CTrack()

This is the constructor of the class. It starts off by trying to create a handle to the GenPort device using its DOS symbolic link, namely GpdDev. If this fails, it returns a message to that effect. It then attempts to clear the output buffer on the InsideTRAK card. This is done by switching off continuous update mode and reading in up to the maximum number of values that there could be. This needs to be done twice to make sure that all the possible data has been removed and that the buffer is empty. The card could have just started its data update to the output buffer when the request came to stop the continuous updates. So although we read off the maximum number of bytes, the tracking system could still be outputting its final set of data, thus requiring the buffer to be re-emptied.

```
CTrack::CTrack()
{
   short Tmp;

   hndFile = CreateFile(
                    "\\\\.\\GpdDev",           // Open the Device "file"
                    GENERIC_WRITE | GENERIC_READ,
                    FILE_SHARE_WRITE | FILE_SHARE_READ,
                    NULL,
                    OPEN_EXISTING,
                    0,
                    NULL);

     if (hndFile == INVALID_HANDLE_VALUE)        // Was the device opened?
     {
         ::MessageBox(NULL,"Failed to create handle to Tracker Device",0,0);
     }

   SetCont(FALSE);
   Sleep(40);
   for (int i = 0; i < 512; i++) _ReadW(&Tmp);

   SetCont(FALSE);
   Sleep(40);
   for (i = 0; i < 512; i++) _ReadW(&Tmp);

}
```

CTrack::~CTrack()

The destructor is empty, except for a single call to close the handle to the device that was created in the constructor, making the device available again to other programs.

```
CTrack::~CTrack()
{
   CloseHandle(hndFile);
}
```

CString CTrack::_ReadW(short * Dat)

To isolate the interaction with the device driver the _ReadW, _ReadB, _WriteB functions were written. The _ReadW function exists to read in a 16bit value from the output buffer. The reason for this function not requiring a port offset, is because there is only one port offset for which 16-bit reads would have any logical interpretation.

The only 16 bit read port is at offset 0, and is the gateway from the output buffer on the InsideTRAK card to the host computer. It is 16 bits wide to allow 16-bit precision reads of the position and orientation angles, i.e. only a single 16-bit word is required per X, Y, Z position or orientations angle.

This function starts off by creating some temporary values. Two of them are the input and output buffers, namely PortNumber, and DataBuffer respectively. The input buffer is filled with 0 (the port offset of 16 bit read register). The lengths of both the input and output buffers are set to two in the DataLength and ReturnedLength variables. The control code is also set up to represent the operation that the device driver must perform.

Finally the device driver is called using the DeviceIoControl function. This function generates a major function code of CONTROL, and the device driver used the minor function code of IoctlCode to differentiate between which operation to perform. To call the device driver we need to pass

- the handle to the device driver (to know which device to communicate with),
- the IoctlCode (to know which operation to perform),
- the address of the input buffer,
- the size of the input buffer,
- the address of the output buffer,
- the size of the output buffer,
- a pointer to a variable that will contain the actual size of the returned buffer that the device driver filled. This is needed because the device driver might not always fill the output buffer completely,
- a pointer to a function that should be called once the operation is complete. If this pointer is NULL then the function will block until it is complete.

Because the created output buffer is the same size as what is expect from the device driver, an error has occurred if the expected output buffer length is not equal to the actual buffer length returned. Finally if this call succeeds, the returned value is converted from an unsigned integer to a signed one and copied into the Dat buffer created by the calling function.

```
CString CTrack::_ReadW(short * Dat)
{
   BOOL           IoctlResult;

   // The following parameters are used in the IOCTL call
   LONG                IoctlCode;
   ULONG               PortNumber;
   ULONG               DataLength;
   ULONG               ReturnedLength; // Number of bytes returned in output
   USHORT              DataBuffer;
   long                Temp;

   IoctlCode = IOCTL_GPD_READ_PORT_USHORT;
   DataLength = 2;
   PortNumber = (ULONG) 0;
   ReturnedLength = 2;

   IoctlResult = DeviceIoControl(
                           hndFile,            // Handle to device
                           IoctlCode,          // IO Control code for Read
                           &PortNumber,        // Buffer to driver.
                           sizeof(PortNumber), // Length of buffer in bytes.
                           &DataBuffer,        // Buffer from driver.
                           DataLength,         // Length of buffer in bytes.
                           &ReturnedLength,    // Bytes placed in DataBuffer
                           NULL                // NULL = wait till op. ends
                           );

    if (IoctlResult)                           // Did the IOCTL succeed?
    {
       if (ReturnedLength != DataLength) return "Returned length is not a
USHORT";

       Temp = 32767 - DataBuffer;
      *Dat = DataBuffer;//(short)Temp;
      return "";
    }

   return "Error Reading Word from port";
}
```

CString CTrack::_ReadB(USHORT Addr, char * Dat)

The _ReadB function shown below is very similar to the _ReadW function with only a few minor differences:

- The caller supplies the offset address instead of assuming that it is 0,
- The IoctlCode is different, and
- The Input and Output buffers are of length 1 not 2.

```
CString CTrack::_ReadB(USHORT Addr, char * Dat)
{
   BOOL           IoctlResult;

   // The following parameters are used in the IOCTL call
   LONG           IoctlCode;
   ULONG          PortNumber;
   ULONG          DataLength;
   ULONG          ReturnedLength; // Number of bytes returned in output buff
   UCHAR          DataBuffer;
   long           Temp;

   IoctlCode = IOCTL_GPD_READ_PORT_UCHAR;
   DataLength = 1;
   PortNumber = (ULONG) Addr;
   ReturnedLength = 1;

   IoctlResult = DeviceIoControl(
                          hndFile,           // Handle to device
                          IoctlCode,         // IO Control code for Read
                          &PortNumber,       // Buffer to driver.
                          sizeof(PortNumber), // Length of buffer in bytes.
                          &DataBuffer,       // Buffer from driver.
                          DataLength,        // Length of buffer in bytes.
                          &ReturnedLength,   // Bytes placed in DataBuffer
                          NULL               // NULL=wait till op. ends
                          );

    if (IoctlResult)                         // Did the IOCTL succeed?
    {
       if (ReturnedLength != DataLength) return "Returned length is not a
USHORT";

       Temp = (long)DataBuffer - 128;
      *Dat = (char)Temp;
      return "";
    }

   return "Error Reading Byte from port";
}
```

CString CTrack::_WriteB(UINT addr, char Dat)

The _WriteB function shown below is in turn very similar to its corresponding _ReadB function. Only a few differences exist, namely:

- The IoctlCode is different.
- The output buffer is of zero length.
- The input buffer consists of a structure containing the port offset and the value to write to this offset.
- No conversion needs to take place of the returned data.

```
CString CTrack::_WriteB(UINT addr, char Dat)
{
   BOOL                 IoctlResult;
   GENPORT_WRITE_INPUT  InputBuffer;    // Input buffer for DeviceIoControl
   LONG                 IoctlCode;
   ULONG                DataLength;
   ULONG                ReturnedLength; // Number of bytes retrnd in output buf

   IoctlCode = IOCTL_GPD_WRITE_PORT_UCHAR;
   InputBuffer.CharData = (UCHAR)Dat;
   InputBuffer.PortNumber = addr;
   DataLength = offsetof(GENPORT_WRITE_INPUT, CharData) +
                                        sizeof(InputBuffer.CharData);

   IoctlResult = DeviceIoControl(
                  hndFile,              // Handle to device
                  IoctlCode,            // IO Control code for Write
                  &InputBuffer,         // Buff to driver. Holds port & dat
                  DataLength,           // Length of buffer in bytes.
                  NULL,                 // Buffer from driver.   Not used.
                  0,                    // Length of buffer in bytes.
                  &ReturnedLength,      // Bytes placed in outbuf. = 0.
                  NULL                  // NULL = wait till I/O completes.
                  );


   if (!IoctlResult)                          // Did the IOCTL succeed?
   {
      return "Error writing byte to Tracker Card";
   }
   return "";
}
```

CString CTrack::Read(SHORT * Dat, SHORT * Length)

The Read function uses the _ReadB and ReadW functions discussed on the previous pages, to read in the whole output buffer on the InsideTRAK card. It first attempts to read the control register. If it fails to read this then it returns the error. If no error occurs then it goes into a while loop that reads continuously until its read the maximum number of bytes, namely 512, or there are no more bytes left to be read. It knows that there are still bytes left to be read if Bit 0 of the control register (offset 1) is one.

Inside the loop it tries to read in a 16-bit word, if successful, it stores it in the provided buffer and checks the control register again. Once there is no more data to be read, it completes and returns the number of values that it successfully read in the Length buffer.

```
CString CTrack::Read(SHORT * Dat, SHORT * Length)
{
   int i;
   char Bits;
   short Tmp;
   CString Result;

   i = 0;
   *Length = 0;

   Result = _ReadB(1, &Bits);
   if (Result.GetLength() != 0) return Result;

   while ((Bits & 0x01) && (i < 512))
   {
      Result = _ReadW(&Tmp);
      if (Result.GetLength() != 0) return Result;
      Dat[i] = Tmp;
      i++;
      Result = _ReadB(1, &Bits);
      if (Result.GetLength() != 0) return Result;
   }

   *Length = i;

   return "";
}
```


CString CTrack::Write(char * Dat, SHORT Length)

The Write function is even simpler. All it has to do is sit in a loop of length equal to the supplied size of the output buffer. Inside the loop, it checks bit 1 of the control register to see if there is room for another incoming byte. If there is no space then the function just sleeps in 10ms intervals until space becomes available again.

```
CString CTrack::Write(char * Dat, SHORT Length)
{
   int i;
   char Bits;
   CString  Result;

   for (i = 0; i < Length ; i++)
   {
      Result = _ReadB(1, &Bits);
      if (Result.GetLength() != 0) return Result;
      while ((Bits & 0x02) == 0)
      {
         Sleep(10);
         Result = _ReadB(1, &Bits);
         if (Result.GetLength() != 0) return Result;
      }

      Result = _WriteB(0, Dat[i]);
   }
   return "";
}
```

CString CTrack::SendSync()

This function sets bit 2 of the first register. This tells the InsideTrak to get the current position and orientation values and to write them into the output buffer so that the computer can read them out. This function can only be used if continuous mode is switched off.

```
CString CTrack::SendSync()
{
   CString Result;
   Result = _WriteB(0,0x4);
   return Result;
}
```


CString CTrack::SetCont(BOOL Yes)

This function either enables or disables continuous mode by writing a 'C' or 'c' respectively. These two commands are explained in Appendix A of this report.

```
CString CTrack::SetCont(BOOL Yes)
{
   CString Result;

   if (Yes) Result = _WriteB(0,'C');
   else Result = _WriteB(0,'c');

   return Result;
}
```

CString CTrack::SetActiveStation(BOOL One, BOOL Two)

This function controls which of the stations, if not both, are active. It creates a string and writes this string out to the InsideTRAK card using the Write function. The command sequence is as follows:

- To make station 1 active write "l1,1"<cr><lf>
- To make station 1 inactive write "l1,0" <cr><lf>
- To make station 2 active write "l2,1" <cr><lf>
- To make station 2 inactive write "l2,0" <cr><lf>

```
CString CTrack::SetActiveStation(BOOL One, BOOL Two)
{

   char Tmp[6];
   CString Result;

   Tmp[0] = 0x6c; // l
   Tmp[1] = 0x31; // 1
   Tmp[2] = 0x2c; // ,
   Tmp[3] = 0x30; // 0
   Tmp[4] = 0x0d; // Carriage Return
   Tmp[5] = 0x0a; // Line Feed

   if (One) Tmp[3] = 0x31; // 1

   Result = Write(&Tmp[0],6);

   if (Result.GetLength() != 0) return Result;

   Sleep(40);

   Tmp[1] = 0x32; // 2
   Tmp[3] = 0x30; // 0

   if (Two) Tmp[3] = 0x31; // 1

   Result = Write(&Tmp[0],6);

   if (Result.GetLength() != 0) return Result;

   Sleep(40);

   return "";
}
```

CString CTrack::GetActiveStation(BOOL * One, BOOL * Two)

This function is used to retrieve the currently active stations. First this function reads in bytes from the output buffer on the InsideTRAK to make sure that it is empty. To request that the InsideTrak send the active stations, it then sends the string "l1"<cr><lf>. It then waits for the InsideTRAK to process the request and afterwards it reads in all the data from the output buffer.

If the output string starts off with "21l" then it knows that if the next byte read in is a '1' then station 1 is on, and if the byte read in is a '0' then station 1 is off. The same applies to the following byte, in that it will specify whether station two is active or not.

```
CString CTrack::GetActiveStation(BOOL * One, BOOL * Two)
{
   char Tmp[4];
   short Ret[512];
   SHORT Len;
   CString Result;

   Tmp[0] = 0x6c;
   Tmp[1] = 0x31;
   Tmp[2] = 0x0d;
   Tmp[3] = 0x0a;

   Read(&Ret[0], &Len); // Dummy read to clear buffer

   Result = Write(&Tmp[0],4);

   if (Result.GetLength() != 0)
      return Result;

   Sleep(40);

   Result = Read(&Ret[0], &Len);

   if (Result.GetLength() != 0)
      return Result;

   if ((Ret[0] == 0x3132) && ((Ret[1] & 0xff) == 0x6c))
   {
      if ((Ret[1] & 0xff00) == 0x3100) *One = TRUE;
      else *One = FALSE;

      if ((Ret[2] & 0xff) == 0x31) *Two = TRUE;
      else *Two = FALSE;
   }
   else return "Error in the data that was returned";

   return "";
}
```

CString CTrack::GetPos(short * Ch1, short * Ch2)

This is the most important function of this class. It forms the reason for the creation of this whole class. This function is used to read in the position and orientation data from the InsideTRAK. It starts off by writing a dummy byte of 0 to the register at offset 1. This signals to the InsideTRAK that the computer now wants the data for the station 1.

The function waits for the InsideTRAK to pulse and sense the magnetic waves, and to perform any necessary calculations, and write out the required data to its output buffer. From this output buffer this function reads in the data into its own buffer.

The same thing is done for station 2. By writing a dummy byte of 0 to the register at offset 2, the InsideTRAK will send the data for station 2 to its output buffer from which this function can read it in. Once it has received both buffers, the program trims off the first 16-bit word from the front, keeps the next six 16-bit words, and trims off the rest. The two sets of six words kept are transferred into the buffers supplied by the calling function. These values represent the X, Y, Z, azimuth, elevation and roll values for each station.

CTrack thus becomes a class from which calling functions can get the position and orientation of both the user's head and hand, without needing any knowledge on how to communicate with the device driver.


```
CString CTrack::GetPos(short * Ch1, short * Ch2)
{
   short Tmp1[512];
   short Tmp2[512];
   int i;
   CString Result;
   SHORT Len1, Len2;

   Result = _WriteB(1,0);
   if (Result.GetLength() != 0) return Result;
   Sleep(40);
   Result = Read(&Tmp1[0], &Len1);
   if (Result.GetLength() != 0) return Result;

   Result = _WriteB(2,0);
   if (Result.GetLength() != 0) return Result;
   Sleep(40);
   Result = Read(&Tmp2[0], &Len2);
   if (Result.GetLength() != 0) return Result;

   for (i = 0 ; i < 6; i++)
   {
      Ch1[i] = Tmp1[i+1];
      Ch2[i] = Tmp2[i+1];
   }
   return "";
}
```

3.4 Video Accelerator Card

The Video Accelerator Card forms an important part of any graphical application. Because all of the graphical calculations are performed in either the graphics card or in the CPU of the computer, the graphical complexity, speed and response rate are limited by the performance of the video card. Cost, availability and performance lead to the purchase of the Diamond FireGL 1000 Pro graphics accelerator card that uses the 3D Labs Permedia 2 graphical engine.

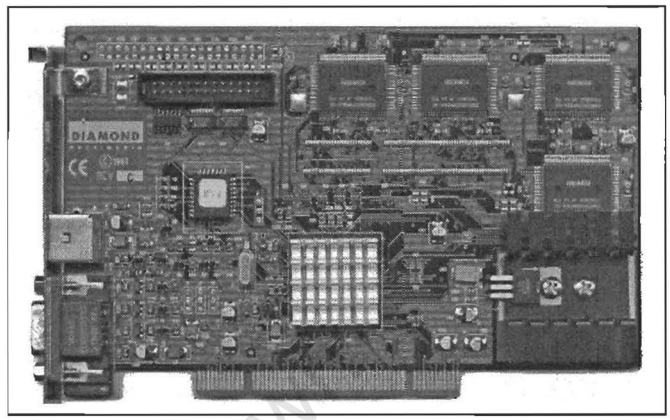


Figure 3.8 : Image of Diamond FireGL 1000 Pro

3.4.1 Capabilities

The Permedia 2 makes high performance 3D graphics pervasive on the mainstream desktop by integrating 2D acceleration, MPEG-2 video acceleration, SVGA and robust 3D acceleration into one package designed for the corporate, entry level professional and performance conscious consumer markets.

- PCI and AGP interface
- Highly pipelined 2D/3D graphics core
- Integral 230 MHz RAMDAC
- MPEG2 video playback acceleration
- VideoStreams bus for simultaneous external video I/O
- Integral SVGA
- 64-bit SDRAM or SGRAM
- 8 Mbytes
- 100+ WinMarks
- 1 Million 3D polygons/sec - textured
- 83 Million pixels/sec textured, bilinear filtered, perspective
- 42 Million pixels/sec textured, bilinear filtered, perspective, Z
- Packed 8, 16, 24 and 32-bit acceleration
- Multi-monitor support
- Highly pipelined 3D graphics core
- Integrated geometry set-up processor
- OpenGL and Direct3D compatible
- 8-bit to 32-bit 3D
- 16-bit Z-buffer
- Purchase Price = R1500

Figure 3.9 : Image of accelerator chip

3.4.2 OpenGL Extensions

'3D Labs' has implemented numerous extensions to the OpenGL standard, however this section will only discuss the few that have been used within this dissertation. Although OpenGL is a well thought out and designed application programming interface, hardware manufacturers often find that they can offer superior performance in areas that were not considered or implemented in the original OpenGL specification but which could still prove useful to the developer.

The OpenGL review board has provided the capability for the OpenGL interface to be extended through the use of extensions. These are function calls that have been implemented by the manufacturer and can be used by any developer using that manufacturer's hardware. The use of OpenGL extensions is highly discouraged because the application instantly becomes tied to a specific piece of hardware, however in some application, this loss of portability is worth the increase in performance.

3D Labs' most notable extensions are to do with texture mapping. Traditional OpenGL programs are required to load the texture map into the current state of the OpenGL engine before the texture can be used. This is reasonable, but the problem arises when the application switches between large textures on a periodic basis. This switching is accompanied by a large performance penalty.

3D Labs have designed their Permedia 2 chip to be able to store the texture maps inside the memory on the graphics card. This means that at the start of the application, it is able to load all of its textures onto the graphics card. If a texture map needs to be used, then it is already on the graphics card and does not need to be copied across. Thus, the three important extensions that are provided by 3D Labs are there to load, to select and to delete the texture maps on the graphic card's memory. These functions are:

fpglGenTextureEXT
fpglBindTextureEXT
fpglDeleteTextureEXT

3.4.3 Other Graphical Cards

Other graphical cards exist, however, often they are heavily priced to offset the development costs over a small "performance graphics" market group. In this section a few alternate graphics cards are presented, together with an evaluation on how they rate amongst each other. Rather than list the individual cards, it is better to organise them by the chip manufacturers, as these are fewer and because there is minimal performance or cost variation between cards from different card manufacturers, but that use the same chip set.

GLINT GMX – 3D Labs

- 4.4M out-of-view lit meshed triangles/sec
- 4.1M backface culled lit meshed triangles/sec
- 3.3M 50% culled lit meshed triangles/sec
- 2.6M visible lit meshed triangles/sec
- 66M pixels/sec - filtered, mip-mapped (GMX 2000)
- 33M pixels/sec - filtered, mip-mapped (GMX 1000)
- 8Gbytes/sec pixel fill (GMX 2000)
- 4Gbytes/sec pixel fill (GMX 1000)
- 3.0M 10 pixel 3D vectors/sec
- Memory on card 96M
- Price $1400

Intense 3D Wildcat - Intergraph

- 16 MB SDRAM frame buffer
- 32 MB texture memory
- Performs 3 billion floating-point operations per second
- 32-bit Z-buffer
- Supports complex fog and atmospheric effects with hardware accelerated per-pixel fog
- 6 M Lit Gouraud-shaded triangles, 25-pixel, Z-buffered (tri/sec)
- 12 M 3D vectors, 10-pixel, solid-color, Z-buffered (vec/sec)
- 90 M Textured Gouraud-shaded fill, 32-bit (RGBA) texels, trilinear interpolated, Z-buffered (pixels/sec)
- Price = $ 3000

Voodoo 2 – 3Dfx

- 90 Mpixels/sec sustained fill rate for bi-linear textures, with LOD
- MIP-mapping, Z-buffering, alpha-blending and fogging enabled
- 180 Mpixel/sec with scan line interleaved configurations
- 3M traingle/sec for filtered, LOD MIP-mapped, Z-buffered, alpha-blended, fogged, textured triangles
- Full hardware triangle setup (independent strips & fans)
- Anti-aliasing
- Depth buffering (16-bit linear, 22-bit effective)
- Alpha blending
- Per-pixel special effects: fog, transparency, translucency
- Texture compositing, morphing, animation
- Price = $ 150

For compartive reasons I have included the test results for 6 popular tests used in performance evaluations.

Card	Awadvs-01	CDRS-03	DRV-04	DX-03	Light-01	Quake 2	OpenGL	Price
Permedia 2		50		8.4	0.99	12	Yes	$ 200
Glint GMX	22.86	132.1	13.35	20.84	2.587		Yes	$ 1400
WildCat	62.9	209.6	23.98	31.9	2.96		Yes	$ 3000
Voodoo 2						66	No	$ 150

These tests are advanced rendering tests that test different aspects of video card performance. Different applications place different needs on a graphical system, thus the above six tests encompass six typical graphically intensive applications. The values listed are in frames per second, however they should not be used as absolute values, as small changes in resolution or colour depth could change them dramatically. They are more useful for use as comparative figures between graphical cards.

Quake 2 is one of the most graphically impressive computer games available at the time of this writing. Thus for gaming applications this would be an important figure. Light-01 is a test in drawing a very complex ray-traced image of a room with many objects in it. This would be important for computer generated special effects in movies. Where a number of years ago this test had results in frames per hour, dramatic improvements have reduced this figure to a few frames per second.

The last two columns show whether the card supports OpenGL and the price tag attached to the card

3.5 Head Mounted Display

For this project the CyberEye 200W head mounted display was acquired. It is from within the lower end of the head mounted display price spectrum. There is very little differentiating the head mounted display in the "for gaming" price category, however this one seemed to have the most sturdy and hardy structural design. As it was built for arcade game type environments, I thought that it would be the most suitable for an industrial environment.

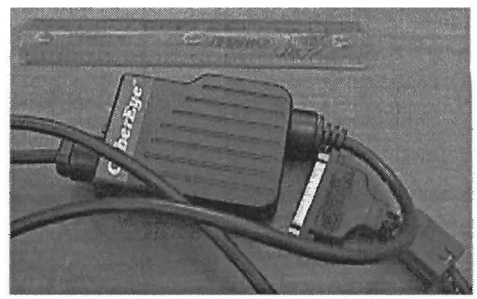


Figure 3.10 : Image of Belt Pack

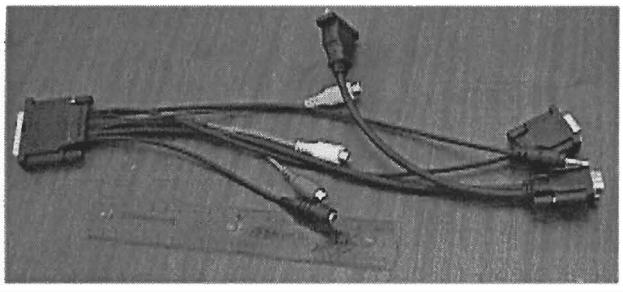


Figure 3.11 : Image of Cable Assembly

The head mounted display (HMD) consists of three physical components. The helmet like structure where the LCD displays are mounted, a belt pack which contains the video format conversion circuitry, and a cable assembly that connects the connectors for all the possible sources of audio and video signals to the belt pack.

Figure 3.12 : Images of the Head Mounted Display CyberEye 200W

3.5.1 Specifications and Tolerances

Display Type

- Dual Active Matrix LCD (AMLCD)
- Contrast Ratio : 100 : 1
- Resolution : 60333 Triads per eye
- Field of view : 45 degrees H * 34 degrees V
- Pixel Size : 4 arc minutes

Interpupillary Range : 55 – 75 mm

Signal Input : NTSC (RS-170), or
VGA

Interface

- Video : 15 pin D-connector (for VGA)
  RCA connectors (for NTSC)
- Audio Input : RCA left / right stereo connectors
- Playback : Stenheiser stereo earphones
- Frequency Response : 20Hz – 20kHz
- Distortion : < 1% T.H.D.
- Impedance : 52 ohms

3.5.2 Stereo Imaging

The stereo format that the head mounted display accepts is line sequential stereo. This means that every even line of the image on the screen goes to the one eye, and every odd line of the image is displayed to the other eye. Thus, to view a stereo image, two views are generates independently. Drawing alternate lines of each image to the display combines these two views into a stereoscopic image. The three images below show how this is achieved. First are the left and right eye images, both slightly different. The final image is how the combined stereo image looks on a non stereo format (i.e. paper).

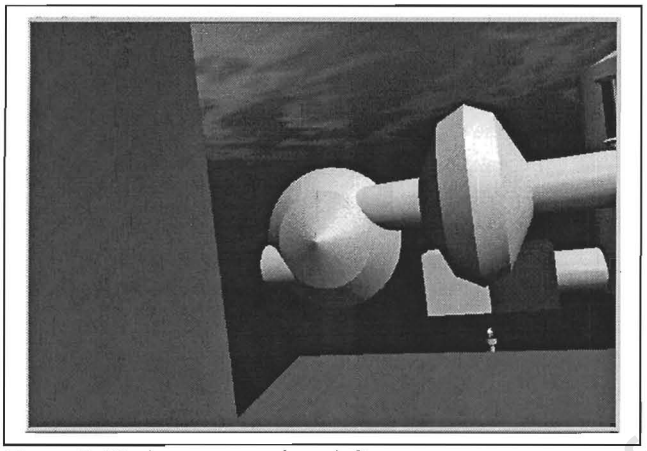


Figure 3.13 : Image seen from left eye

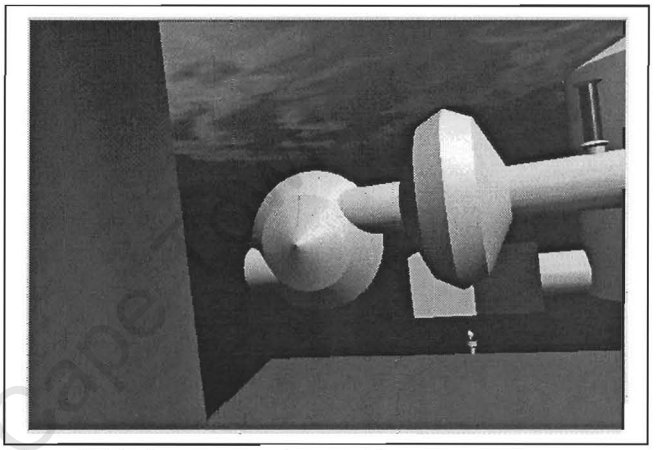


Figure 3.14 : Image seen from right eye

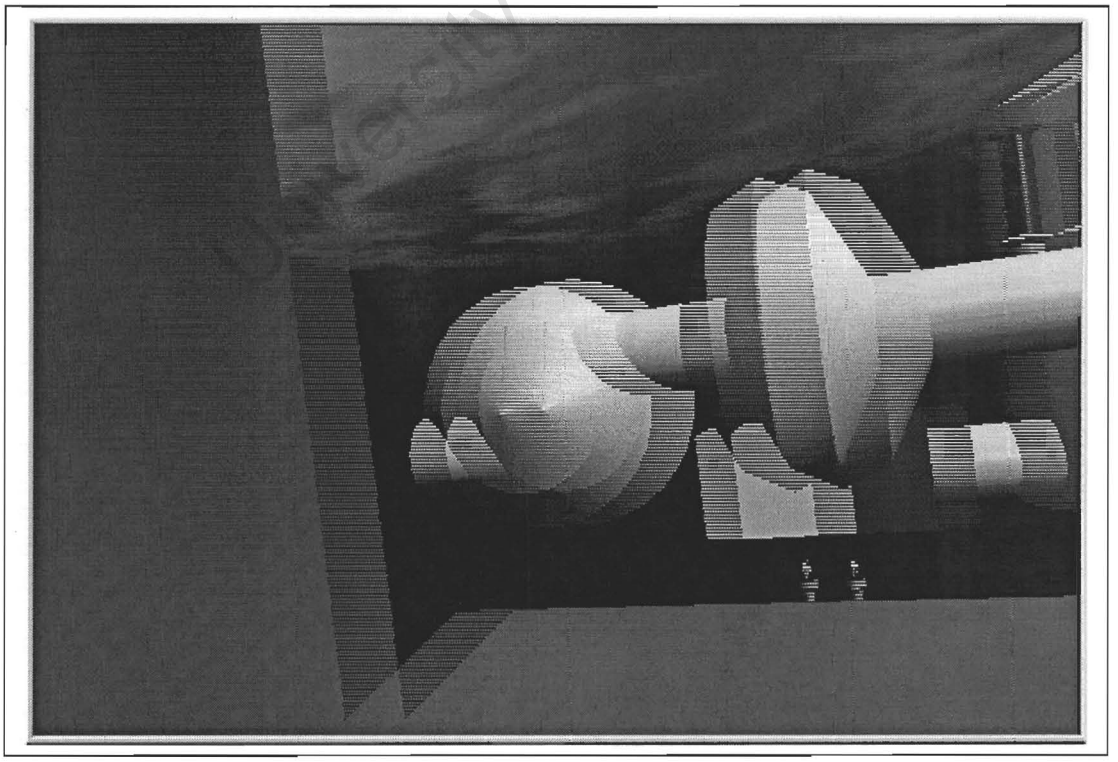


Figure 3.15 : 3D line sequential image as would be seen on a 2D format (e.g. paper or monitor)

Generation of the two independent images for each eye is done by first translating the viewpoint from the receiver's position to where the left eye would be. This view is then rendered. Next, the viewpoint is translated to where the right eye would be and the view rendered again. Finally, these two images are displayed in line sequential format as described in on the previous page. The HMD takes this line sequential image and draws all the even lines in the one eye's LCD display, and all the odd lines in the others eye's LCD display.

E.g.

- Get position of receiver mounted on head.
- Translate the viewpoint to this position.
- Translate the viewpoint down about 10cm (from top of helmet to middle of head)
- Translate the viewpoint left about 5cm (to left eye)
- Render view to left frame buffer
- Translate the viewpoint right about 10cm (to right eye)
- Render view to right frame buffer
- Display in line sequential format.

Figure 3.16 : Image of HMD on head with the tracking system's receiver mounted on top

Chapter 4
The Virtual World

Chapter 3 discussed all the individual items of hardware that are used by this project, now Chapter 4 will present the programming the unifies all the hardware into a single interactive 3D virtual environment. This virtual environment is written in a C graphics library, called OpenGL. Numerous 3D development tools currently exist, but they are all limited to a small amount of hardware that they can support. The only development tools that could have been of use are 3D design environments from whose data OpenGL source code can be generated, for example Autocad, 3D Studio Max, and World Toolkit. These packages are however more expensive than the hardware that was used in this research project.

4.1 Setting up the OpenGL Framework

The setting up of the OpenGL graphical framework has been implemented in three classes. These are the view class, namely CMy3DVRView, and two service classes, CTextures and COpenGLInit. CTextures is a class that is responsible for the loading, maintenance and disposal of textures that are used throughout the Virtual World. COpenGLInit is a class that was written to modularise and self contain some of the initialisation routines. Modularity is essential in developing large projects, thus all logical sub-areas are self-contained in individual classes

The view class uses the functionality provided by the other two classes to create a 3D environment in what is essentially a 2D Windows interface. Windows, and thus Visual C++, come with included functionality to help improve, and ease the design of the user interface for programs. They provide unified ways of drawing and interacting with objects that make it easier for the end user to move from one application to another. However, none of these added extras were created with 3D in mind. Menus, scrollbars, progress indicators, and copy and paste functionality has no logical significance in a 3D environment. Even the traditional mouse is of little use once depth is included.

For the above reasons, the traditional features of Windows programming need to be disabled. A new windowing environment needs to be installed with all relevant messages being routed from the Windows framework to this new windowing environment. The next section provides a description on how all the initialisation is combined together in the View class. This will give the reader a brief overview of the initialisation process before the two sections thereafter that discuss the two helper classes.

4.1.1 CMy3DVRView Class

The first step in the view class is to capture all the commands and messages that might be generated by Windows NT, which might affect the display system. These events will need to handled, or at least disabled so that no visual artefacts or disturbances occur.

The OnCreate function is captured so that an OpenGL display area can be created to replace the default Windows view. OnDestroy thus also needs to be captured, so that all the initialisation that took place in OnCreate can be cleaned up. OnEraseBkgnd is captured next and disabled so that Windows is unable to clear the OpenGL display area. OnSize needs to be captured so that OpenGL knows to resize its view when the display area changes. Finally, WM_TIMER messages are trapped to generate a periodic frame refresh.

The next step in the class is to create some member variables that will be used by the member functions. There are three member variables, all of which are defined as 'protected', as parent classes need not have access to them. The first variable defined is 'TimeOuts'. This variable is an integer that counts the number of timer messages that have been generated. This count is kept because different events need to take place at different timeout periods.

ConR is an object of class CControlRoom. This is the class that is responsible for drawing the whole control room, and will be discussed in its own section (4.2 Drawing the Room). The last member variable, OGLInit is an object of class COpenGLInit and exits to enable the view class to access the functionality provided by the class COpenGLInit.

CMy3DVRView, ~CMy3DVRView

The constructor and destructor are the first two functions that form the basis of any class. The destructor would normally contain cleanup code, but in our case is however empty. The constructor is the place where member variables are initialised. In the source below, only two of the three variables are setup. The 'TimeOuts' variable is set to zero, and a variable pointer inside the COpenGLInit object is initialised to point to the view class. Now that the COpenGLInit class has a pointer to the View class, it will be able to access all the public member variables and member functions that form part of the view class.

```
CMy3DVRView::CMy3DVRView()
{
   OGLInit.Vw = (CView *) this;
   TimeOuts = 0;
}

CMy3DVRView::~CMy3DVRView()
{
}
```

PreCreateWindow

This function is called before the window is set up and is used to modify the style that it will come up with when it does first open. Here the style is modified to include WS_CLIPSIBLINGS and WS_CLIPCHILDREN to prevent OpenGL from trying to draw into any other windows.

```
BOOL CMy3DVRView::PreCreateWindow(CREATESTRUCT& cs)
{
   cs.style |= WS_CLIPSIBLINGS | WS_CLIPCHILDREN;
   cs.cx = 640;
   cs.cy = 480;

   return CView::PreCreateWindow(cs);
}
```

OnCreate

The OnCreate function starts by calling the base class' OnCreate, which creates the view on the screen. Next the timer is set to generate WM_TIMER events every 20ms. These timer events are used to generate screen updates. This effectively generates the frame rate of the environment.

The last two function calls are to two initialisation routines. The first one is a member function of the COpenGLInit class and it sets up the display for OpenGL. The second is a member function of the CControlRoom class and it is called to set up the variables that will form part of the 3D environment, and need to be initialised before anything is drawn.

```
int CMy3DVRView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CView::OnCreate(lpCreateStruct) == -1)
      return -1;

   SetTimer(1, 20, NULL);

   OGLInit.InitPFD();   // initialize OpenGL
   ConR.InitEnv();

   return 0;
}
```

OnDestroy

The OnDestroy does almost the opposite of what the OnCreate function did. It stops the timer from generating any more events. It calls the OnDestroy member function of the CControlRoom class, which frees up everything that it had created. It finally calls the OnDestroy member function of the COpenGLInit class to remove OpenGL's hold on the view.

```
void CMy3DVRView::OnDestroy()
{
   CView::OnDestroy();

   KillTimer(1);

   glDeleteLists(1000, 256);

   ConR.OnDestroy();
   OGLInit.OnDestroy();
}
```

OnSize

The OnSize member function call the corresponding member function of the COpenGLInit class after calling the base class' OnSize. The base class will change the size of the window, and COpenGLInit.OnSize will scale the 3D view to fit the new area.

```
void CMy3DVRView::OnSize(UINT nType, int cx, int cy)
{
   CView::OnSize(nType, cx, cy);

   OGLInit.OnSize(cx, cy);
}
```

OnEraseBkgnd

The OnEraseBkgnd function does nothing, with the exception of returning TRUE. This indicates to Windows that the screen has been cleared. This is done to make sure that Windows has no way of clearing the screen, apart from when the virtual environment decides that it is time to clear the screen. This needs to be done because when the screen is cleared, then the depth, stencil, accumulator and other buffers also need to be cleared. OpenGL's clear screen function call is also implemented in hardware, making it faster than the Windows screen clear.

```
BOOL CMy3DVRView::OnEraseBkgnd(CDC* pDC)
{
   return TRUE;
}
```

OnTimer

The OnTimer function is called every time that a timer interrupt occurs, which as initialised previously, is every 20ms. The first action is to invalidate the screen. This tells the windowing system to force a redraw by calling the OnDraw function. The FALSE indicates that Windows must not clear the screen because it will be done by the virtual environment.

The next step is to remove all the spurious WM_TIMER messages from the message buffer. If the display frame rate is less than 50 frames per second, then there will be a build up of timer messages that will be generated faster than they can be serviced. This way I remove all the WM_TIMER messages that still have not been serviced.

Finally, after every 50 timer events, I call the ControlLoop function from the CControlRoom class. This updates the data in the simulated control loops. This means that sample period of the control system will be one second.

```
void CMy3DVRView::OnTimer(UINT nIDEvent)
{
   Invalidate(FALSE);

   CView::OnTimer(nIDEvent);

   // Eat spurious WM_TIMER messages
   MSG msg;
   TimeOuts++;
   while(::PeekMessage(&msg, m_hWnd, WM_TIMER, WM_TIMER, PM_REMOVE))
   {
      TimeOuts++;
   }

   if (TimeOuts > 50)
   {
      TimeOuts = TimeOuts - 50;
      ConR.ControlLoop();
   }
}
```


OnDraw

The final member function of this class is the OnDraw function. This function is called every time a screen redraw needs to occur. A redraw is necessary when a timer initiates an invalidate command, or when the user performs some action that modifies the viewing area, for example the resizing or moving of a window.

```
void CMy3DVRView::OnDraw(CDC* pDC)
{
   CMy3DVRDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   ConR.DrawControlRoom();
}
```

4.1.2 COpenGLInit Class

This class provides member function support to the View class. It implements essential initialisation routines that are required to enable the use of OpenGL within the Windows windowing environment.

This class has three member variables. The first, namely oldRect, is an object of the class CRect. It is used to store the current viewing rectangle. This information is required to check if the window really did change size when an OnSize function call is received.

The other two member variables are both pointers. The first variable points to the instance of the view class. This first pointer is used to acquire the second pointer, which points to the Device Context. The Device Context is an abstraction of the window. It is used to provide a unified device independent interface for the developer that is the same irrespective of whether the user is drawing to the screen, a printer, or a plotter. Where it provides an advantage in traditional Windows programming, it makes OpenGL programming, which is very device dependent, more complicated. OpenGL is designed to only work when used to display graphics on the monitor.

COpenGLInit, and ~ COpenGLInit

As in the view class, the destructor is empty. The constructor does the variable initialisation, and initialises the oldRect variable to zeros, i.e. the display area is of zero size before the first OnSize call. It also sets the two pointers to NULL, so that they cannot be used until they are properly initialised.

```
COpenGLInit::COpenGLInit()
{
   oldRect.right = 0;
   oldRect.bottom = 0;
   Vw = NULL;
   pDC = NULL;
}

COpenGLInit::~COpenGLInit()
{

}
```

InitPFD

The InitPFD starts by getting a pointer to the Device Context that is created by the View class. Once it has this pointer, it can create and setup a True Type font that will be used while in OpenGL

```
void COpenGLInit::InitPFD()
{
    PIXELFORMATDESCRIPTOR pfd;
    int         n;
    HGLRC     hrc;

    pDC = new CClientDC(Vw);

    LOGFONT logFont;
    logFont.lfHeight = 8;
    logFont.lfWidth = 0;
    logFont.lfEscapement = 0;
    logFont.lfOrientation = 600;
    logFont.lfWeight = FW_NORMAL;
    logFont.lfItalic = 0;
    logFont.lfUnderline = 0;
    logFont.lfStrikeOut = 1;
    logFont.lfCharSet = ANSI_CHARSET;
    logFont.lfOutPrecision = OUT_DEFAULT_PRECIS;
    logFont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
    logFont.lfQuality = PROOF_QUALITY;
    logFont.lfPitchAndFamily = VARIABLE_PITCH | FF_ROMAN;
    strcpy(logFont.lfFaceName, "Times New Roman");

    logFont.lfHeight = 16;
    CFont font;
    font.CreateFontIndirect(&logFont);
    pDC->SelectObject(&font);

    ASSERT(pDC != NULL);
```


The next step is to setup the Pixel Format Descriptor. This function will set up the device capabilities of the video display hardware, e.g. pixel depth, Z-buffer depth, type of buffering, and other functionality. Pixel formats are the translation layer between OpenGL calls and the rendering operations that Windows performs. For example, an OpenGL call to draw a pixel with a RGB triad value of [128,120,135] might be drawn by windows with a translated value of [128,128,128]. The pixel format that is selected describes how colours are displayed, the colour depth, the resolution and additional capabilities that are supported by the created rendering context.

```
    if (!bSetupPixelFormat()) return;
```

Next a data structure is filled with the format descriptor that was created by the above function call. It is now possible to check the hardware capabilities that have been provided.

```
    n = ::GetPixelFormat(pDC->GetSafeHdc());
    ::DescribePixelFormat(pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
```

Now that OpenGL is set up, a new blank Context (framework) is created. This context will become the one in which OpenGL will operate. This new context is selected to be the current one.

```
hrc = wglCreateContext(pDC->GetSafeHdc());
wglMakeCurrent(pDC->GetSafeHdc(), hrc);
```

The final part of this function is used to create a 3D font from the currently selected True Type font. The function wglUseFontOutlines will convert each character of the font into a display list. By calling each of these display lists, OpenGL will draw the corresponding letter.

To execute the same sequence of OpenGL commands repeatedly, it is possible to create and store a display list. A display list is a cached sequence of commands what can be repeated with minimal overhead. The reason why a display list has a smaller overhead is because all of the vertices, lighting calculations, texture, and matrix operations are stored in the list. All once off calculations are only performed when the list is created, and thus need not be recalculated when the list is replayed.

How this applies to text, is that OpenGL works out all the commands that are required to generate each character that is part of a font. OpenGL caches these operations inside a separate display list for each letter. Thus when text has to be written to the virtual environment, then the corresponding display list for each letter, in the string that needs to be written, is called.

```
if (wglUseFontOutlines(pDC->GetSafeHdc(), 0, 255, 1000, 0.0f, 0.5f,
WGL_FONT_POLYGONS, NULL) == FALSE)
   {
      DWORD err = GetLastError();
      Beep(1000,100);
      char s[100];
      sprintf(s,"Err = %d",err);
      ::MessageBox(NULL,s,0,0);
   }
   glListBase(1000);

}
```

bSetupPixelFormat

This function sets up what the OpenGL framework is going to be like. The viewing environment is set here to be double buffered. With double buffering, drawing is done to the back buffer, and when the image is complete, then the back buffer is swapped with the front buffer, and a fully drawn picture becomes visible. The environment is also set have 24 colour bits. The Z-buffer is chosen to be 16-bit deep because that is the size of the hardware Z-buffer.

The requested device framework is however not always available. There is a limit to the number of modes in which the physical hardware, namely the video card and the monitor, will work in. For example, the card might not be able to display 24-bit colour in the current resolution, or the Z-buffer might be too large for the chosen colour depth. For this reason, when a call to ChoosePixelFormat occurs, then the graphical hardware responds with a mode that is closest in capability to the one that was requested. This is why in the function InitPFD there is a check to see the capabilities that were received. Once a specific mode number is available, OpenGL is set into this mode using the function call SetPixelFormat.

```
BOOL COpenGLInit::bSetupPixelFormat()
{
    static PIXELFORMATDESCRIPTOR pfd =
  {
       sizeof(PIXELFORMATDESCRIPTOR),  // size of this pfd
       1,                              // version number
       PFD_DRAW_TO_WINDOW |            // support window
         PFD_SUPPORT_OPENGL |          // support OpenGL
      PFD_GENERIC_ACCELERATED |
         PFD_DOUBLEBUFFER,             // double buffered
       PFD_TYPE_RGBA,                  // RGBA type
       24,                             // 24-bit color depth
       0, 0, 0, 0, 0, 0,               // color bits ignored
       0,                              // no alpha buffer
       0,                              // shift bit ignored
       0,                              // no accumulation buffer
       0, 0, 0, 0,                     // accum bits ignored
       16,                             // 16-bit z-buffer
       1,                              // single bit stencil buffer
       0,                              // no auxiliary buffer
       PFD_MAIN_PLANE,                 // main layer
       0,                              // reserved
       0, 0, 0                         // layer masks ignored
    };
    int pixelformat;

    if ( (pixelformat = ChoosePixelFormat(pDC->GetSafeHdc(), &pfd)) == 0 )
    {
        ::MessageBox(NULL, "ChoosePixelFormat failed", 0, 0 );
        return FALSE;
    }

    if (SetPixelFormat(pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)
    {
        ::MessageBox(NULL, "SetPixelFormat failed", 0, 0);
        return FALSE;
    }
return TRUE;
}
```

OnDestroy

To remove OpenGL as the current Device Context, when the application is shut down, the current OpenGL context must be deleted. Before doing so, the current context must be set back to normal. All the display lists that were created for each letter of the current font also need to be deleted.

```
void COpenGLInit::OnDestroy()
{
   HGLRC hrc;

   hrc = ::wglGetCurrentContext();
     ::wglMakeCurrent(NULL,  NULL);
   glDeleteLists(1000, 256) ;

     if (hrc) ::wglDeleteContext(hrc);

     if (pDC) delete pDC;
}
```

OnSize

The final member function of this class is the OnSize function. It starts by setting the OpenGL view port to the current size. It then checks to see if the window has become smaller in either the X, or Y direction. If it has then it has uncovered areas previously drawn over by OpenGL. The function thus needs to inform all the windows underneath it, to update their views. This is done using the member function RedrawWindow of the CView class.

The last section sets up a 3D perspective view with a viewing angle of 45 degrees, the current aspect ratio, a near clipping plane of 0.1 and a far clipping plane at 40. The clipping planes will disable the drawing of any objects that are closer than 10cm and that are further than 40m away. This distance will also form the range into which the 16 bits of the Z-buffer need to be subdivided. The Z-buffer is used to indicate depth of each pixel and is used to aid in drawing occluded objects. If the object currently being drawn has a depth larger than the Z-buffer value at the current pixel position, then we know that the object is occluded by what has been already drawn. If the Z-buffer value is larger than the depth of the current object, then this new object is in front of the previously drawn object and thus will be rendered.

```
void COpenGLInit::OnSize(int cx, int cy)
{
    if(cy > 0)
    {
        glViewport(0, 0, cx, cy);

        if((oldRect.right > cx) || (oldRect.bottom > cy)) Vw->RedrawWindow();

        oldRect.right = cx;
        oldRect.bottom = cy;

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45.0f, (GLdouble)cx/cy, 0.1f, 40.0f);
        glMatrixMode(GL_MODELVIEW);
    }
}
```

4.1.3 CTextures Class

The CTextures class does all of the background work necessary to be able to use textures within OpenGL. Although it is not directly used within the View class, it is still a class providing initialisation functionality. It does not do any drawing itself, but provides the class owner with textures.

The OpenGL specification only defines the specification for using textures sequentially. After one texture has been used, and a new one is needed, then the old one has to be removed, and the new texture loaded into the OpenGL state machine. This makes for inefficient swapping between textures. To improve this, 3Dlabs, the manufacturer of the chipset (Permedia 2) used in the video card (Diamond FireGL 1000 Pro), added three OpenGL extensions. Extensions for OpenGL are functions that have been implemented by the writer of the Device Drivers and the OpenGL libraries, but that were not part of the original OpenGL specification.

The Permedia 2 chipset allows the use of some of the onboard memory as texture memory. Textures are loaded only once onto the video card. The programmer only needs to select which of the on card textures is to be used. This is a more efficient process because not only are the texture maps only loaded once, but also the card does not have to access system memory when it is using the texture maps.

The Permidia 2 provides three functions to make use of this facility. FpglDeleteTexturesEXT deletes the textures from memory, fpglBindTextureEXT selects the specified texture for current use, and fpglGenTexturesEXT loads the given texture into onboard memory.

This class has ten member variables. Three of these are used to define, and runtime load from the DLL, the above three extension functions. Six of the remaining seven variables define each of the six textures that are used inside this program. The last variable is a array of textures used to reference the 32 flame textures. Of the five member functions, two, namely the constructor and the destructor, are empty.

LoadBMP

LoadBMP, shown on the following page is a helper function that is only available from within the CTexture class. It is used to set up a texture map from bitmaps that have been defined as resources inside the Visual C++ project workspace. It is passed the Bitmap ID, which it will convert, and then load the bitmap into the OpenGL state machine as a texture map.

This function starts by loading the bitmap from the resources into a CBitmap object. It then loads the bitmap into a structure of type BITMAP. The function then creates an area of memory into which it copies the colour bit information from the BITMAP structure. Having a pointer to the raw data of the bitmap, the function now generates a 2D mipmap using an OpenGL utility library function.

Mipmaps are the original bitmap with multiple copies of itself. Each copy is a quarter of the area of the previous one. The motivation behind using mipmaps, is for its improved appearance when the texture moves away from the viewpoint and becomes smaller. Mipmaps generate high quality, scaled copies beforehand, and then make quick scaling approximations from the closest mipmap at runtime. A mipmap however does take up 33% more space than a regular bitmap.

The mipmap is stored by separating it into its component colours (red, green and blue) and storing then separately as rectangles. These three rectangles together form a bigger rectangle with one of the quarters missing. This missing quarter is subdivided into four pieces, with three being taken up by the separated colours of the next level mipmap. This continues recursively for the specified number iterations.

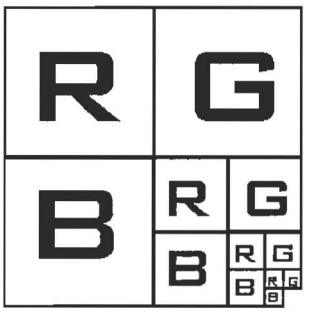


Figure 4.1 : Mipmap format

```
BOOL CTextures::LoadBMP(UINT ID)
{
   CBitmap BMP;
   BITMAP BMPdata;

   BMP.LoadBitmap(ID);
   BMP.GetBitmap(&BMPdata);
   DWORD sz = BMPdata.bmWidth*BMPdata.bmHeight*4;
   PUCHAR Ar;
   Ar = new UCHAR[1048576];
   BMP.GetBitmapBits(sz, Ar);

   gluBuild2DMipmaps(GL_TEXTURE_2D, 3, BMPdata.bmWidth, BMPdata.bmHeight,
0x80e1, GL_UNSIGNED_BYTE, Ar);

   delete Ar;
   return TRUE;
}
```

OnCreate

The OnCreate function performs the majority of the work of this entire class. It starts by defining two of the three function extensions made available by the Permedia 2, namely fpglBindTextureEXT, and fpglGenTextureExt.

It first generates and binds the texture that will be used for the floor inside the control room. It then opens a file that was generated containing 32 consecutive images of a flame burning. When played back one after the other, these textures will give the illusion of a burning fire. Inside the loop, while loading each of the 32 images, the parameters for what to do when the texture needs to be magnified when the object is close to the viewer, or made smaller when the object is far away, are set.

Multiple options for scaling exist. GL_NEAREST tell OpenGL to fill each pixel on the screen with the same colour as the closest corresponding pixel in the texture map. GL_LINEAR on the other hand would do a linear interpolation between the closest four pixels. This method is better if the image is zoomed in. Assume that for a zoomed up image, each pixel of the texture map is represented by a square of 20 by 20 pixels. Using GL_NEAREST, big blocks of 20 by 20 pixels all with the same colour as the corresponding single pixel in the texture map will be visible. With GL_LINEAR, the zoomed up image will provide an interpolation of colours for each pixel as a function of the colours and the distance away from the closest four pixels in the texture map. Being more accurate, GL_LINEAR is thus slower.

The texture maps for the floor in the factory, the sky, the wall, the panel, and the PLC controller are loaded next. The sky texture is set to repeat, so that if the block, that the texture map is used to fill, is larger than the texture map then it repeats the texture map. This is useful when the edges of the bitmap co-inside, i.e. the left edge matches the right edge, and the top matches the bottom.

```
void CTextures::OnCreate()
{
   fpglBindTextureEXT = (PFNGLBINDTEXTUREEXTPROC)
wglGetProcAddress("glBindTextureEXT");
   fpglGenTexturesEXT = (PFNGLGENTEXTURESEXTPROC)
wglGetProcAddress("glGenTexturesEXT");

   fpglGenTexturesEXT( 1, &FloorTex );
   fpglBindTextureEXT( GL_TEXTURE_2D, FloorTex);
   LoadBMP(IDB_FLOORTEX);

   CFile f("FireAll_RGBA.dat",CFile::modeRead);
   UCHAR data[524288];
   f.Read(data,524288);
   for (int i = 0; i < 32; i++)
   {
      fpglGenTexturesEXT( 1, &FireTex[i] );
      fpglBindTextureEXT( GL_TEXTURE_2D, FireTex[i]);
      gluBuild2DMipmaps(GL_TEXTURE_2D, 4, 64, 64, GL_RGBA, GL_UNSIGNED_BYTE,
&data[i*16384]);
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
      glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   }

   fpglGenTexturesEXT( 1, &Floor2Tex );
   fpglBindTextureEXT( GL_TEXTURE_2D, Floor2Tex);
   LoadBMP(IDB_FLOOR2TEX);

   fpglGenTexturesEXT( 1, &SkyTex );
   fpglBindTextureEXT( GL_TEXTURE_2D, SkyTex);
   LoadBMP(IDB_SKYTEX);

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

   fpglGenTexturesEXT( 1, &WallTex );
   fpglBindTextureEXT( GL_TEXTURE_2D, WallTex);
   LoadBMP(IDB_WALLTEX);

   fpglGenTexturesEXT( 1, &PanelTex );
   fpglBindTextureEXT( GL_TEXTURE_2D, PanelTex);
   LoadBMP(IDB_PANELTEX);

   fpglGenTexturesEXT( 1, &PLCTex );
   fpglBindTextureEXT( GL_TEXTURE_2D, PLCTex);
   LoadBMP(IDB_PLCTEX);

   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
}
```

OnDestroy

The last member function of this class is the OnDestroy function. It first creates a pointer to the function extension fpglDeleteTexturesEXT, and then uses this function to delete all of the textures that were set up in the OnCreate function, from off onboard graphical memory.

```
void CTextures::OnDestroy()
{
   fpglDeleteTexturesEXT = (PFNGLDELTEXTURESEXTPROC)
wglGetProcAddress("glDeleteTexturesEXT");
   fpglDeleteTexturesEXT(1, &PLCTex);
   fpglDeleteTexturesEXT(1, &PanelTex);
   fpglDeleteTexturesEXT(1, &WallTex);
   fpglDeleteTexturesEXT(1, &SkyTex);
   fpglDeleteTexturesEXT(1, &Floor2Tex);
   for (int i = 0; i < 32; i++) fpglDeleteTexturesEXT(1, &FireTex[i]);
   fpglDeleteTexturesEXT(1, &FloorTex);
}
```

4.2 Drawing the Room

This section covers the three classes that form the core of drawing the virtual control room. This is the largest section in Chapter 4 because drawing the room forms the basis for the 3D virtual environment. The three classes that fall under this heading are CControlRoom, CBlocks and COpenGLObject. CControlRoom is referred to back in section 4.1 in the View class. It co-ordinates the conglomeration of the virtual control room using the functionality provided by the COpenGLObject class. The COpenGLObject class in turn uses the 3D graphical building blocks provided by the class CBlocks. Together these three classes enable the following view to be drawn.

Not everything below looks exactly like it does in the physical world. Attempts to make everything look as realistic as possible were made, however there is a limit to the detail and intricacy of the graphics that can be generated. Highly complicated graphics cause the frame rate to slowdown and take a lot of time to design and program in source code.

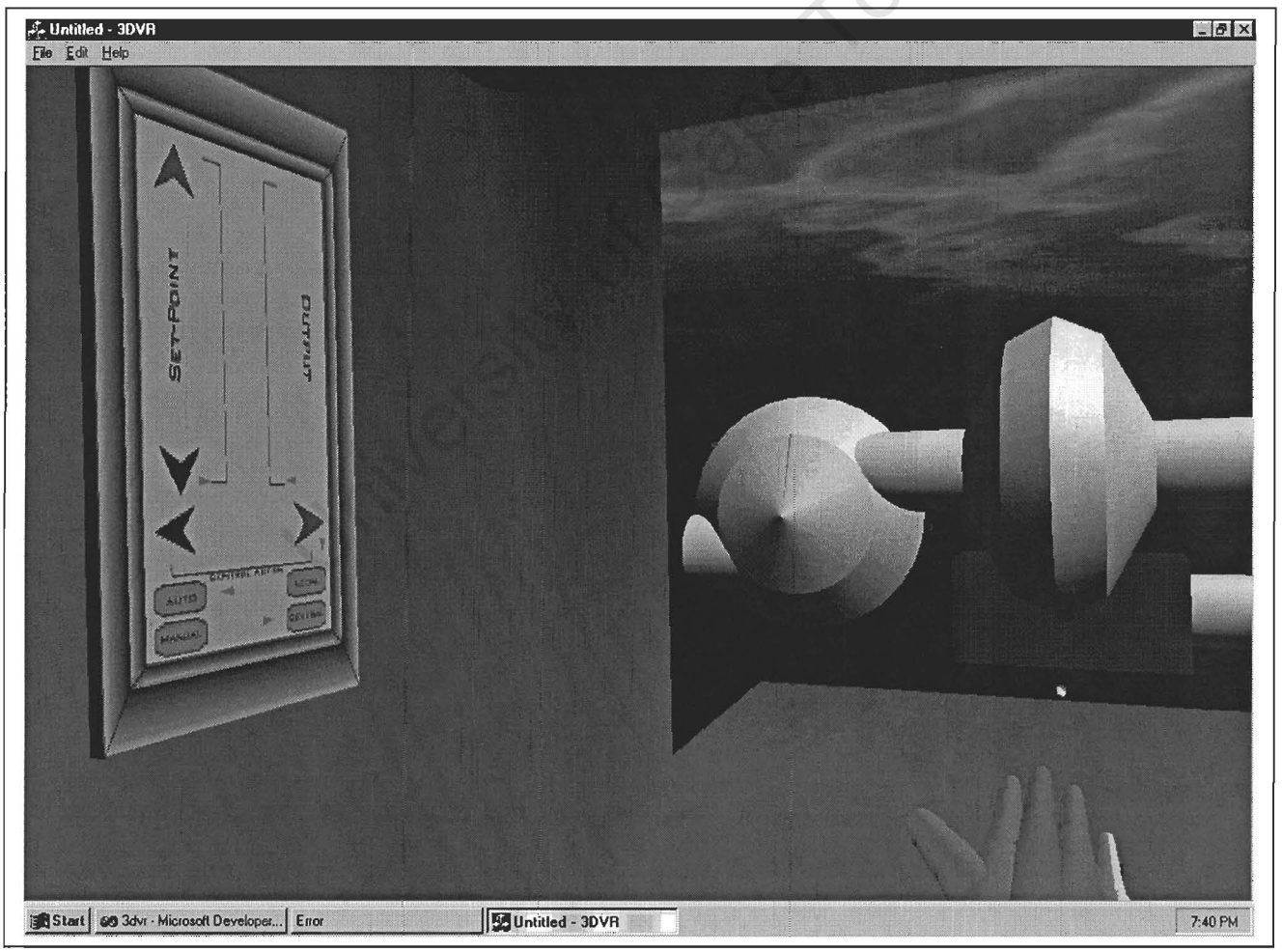


Figure 4.2 : View of the virtual environment

4.2.1 CBlocks Class

This class provides the basic and intermediate graphical building blocks that get used in drawing the control room. It has 13 member functions of which three provide internal functionality and the other ten each draw a different object on the screen. This class has four member variables CosX, SinX, pTex and Index. Index is a counter that gets incremented each time the screen is redrawn and pTex is a pointer to a CTexture object declared in the CControlRoom class. The pTex member variable enables the functions inside this class to have access to the textures defined in the CTexture class.

CosX and SinX are both arrays, and form trigonometric lookup tables. The number of calculations using cosine and sine that need to be evaluated forces the use of lookup tables to prevent a loss in performance.

CBlocks, ~ CBlocks

As before the destructor is empty, because no cleaning up needs to be done on shutdown. The constructor is however responsible in setting up the lookup tables that will be used by the other functions. The last entry is made to be the same as the first one so that no gaps are created by slightly different calculation of the sine and cosine values at 0 and at 2PI.

```
CBlocks::CBlocks()
{
   for (int i = 0 ; i < 30; i++)
   {
      CosX[i] = (GLfloat)cos((double)i/15.0f*PI);
      SinX[i] = (GLfloat)sin((double)i/15.0f*PI);
   }
   SinX[30] = SinX[0];
   CosX[30] = CosX[0];
   Index = 0;
}

CBlocks::~CBlocks()
{

}
```

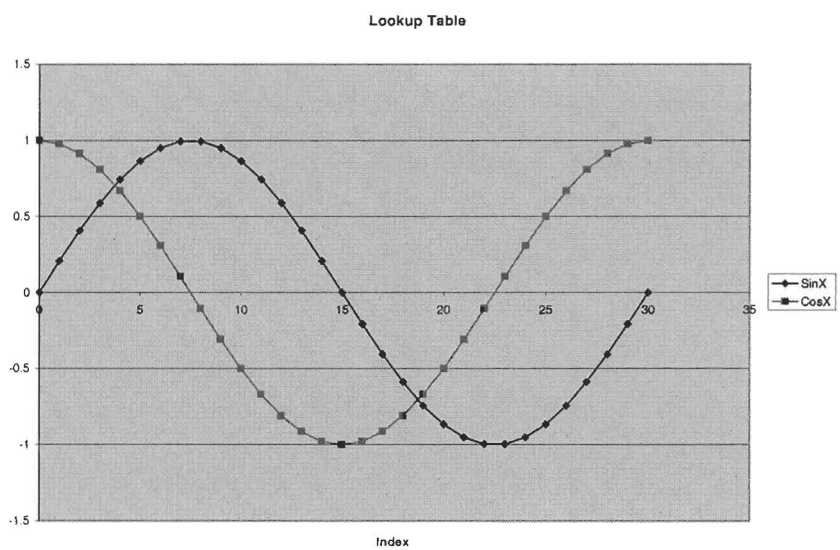


Figure 4.3 : Sine and Cosine lookup table

Inc

This is a simple function that cycles the member variable Index from 0 through to 31. This variable, namely Index, is then used to display different flame data. By gradually incrementing a counter, a different frame of a burning flame is used.

```
void CBlocks::Inc()
{
   Index ++;
   if (Index > 31) Index = 0;
}
```

Pipe

The pipe function is one of the most versatile functions in this class. It draws a hollow cylinder made up of 31 slats, and is used in the majority of member function in this class. It has a number of variables that need to be passes to it, that dictate how it will be drawn. The width variable specifies the radius of the cylinder, and the length variable specifies the distance through which the circle is extruded to generate a cylinder.

The StartX, StartY and StartZ variables give the starting point of the axis around which the cylinder is drawn.

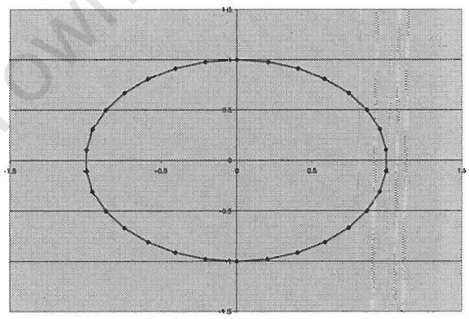


Figure 4.4 : Side view of pipe

Figure 4.5 : Wire frame picture of Pipe

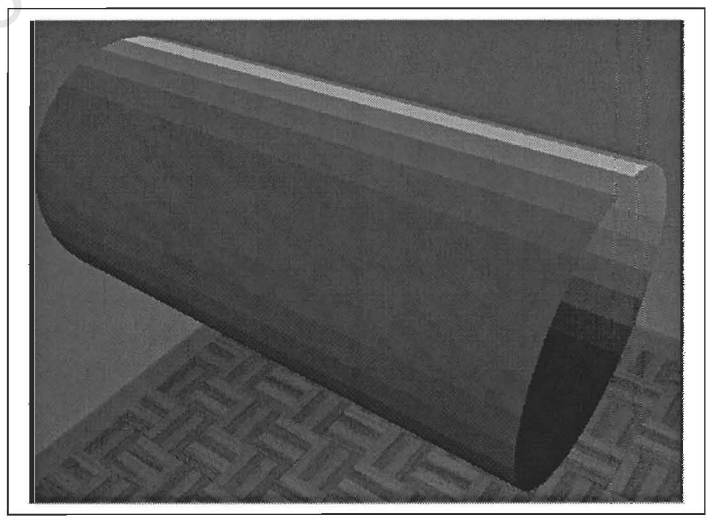


Figure 4.6 : Solid fill image of Pipe

To make the cylinder look smoother and more realistic, without needing to add in more slats, the normal vectors at the vertices must be made to point directly outwards away from the centre axis. By doing so, the two edges of each slat will have different colours that will be smoothly interpolated, making cylinder's surface look smooth.

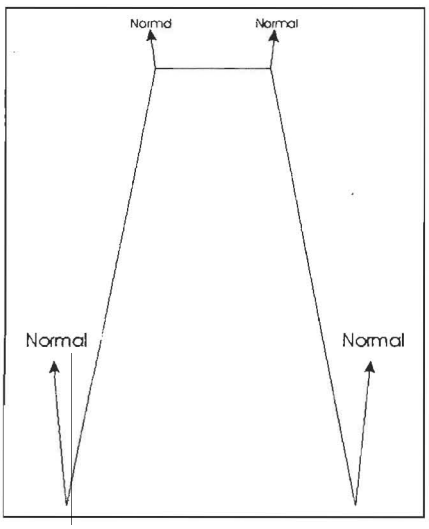


Figure 4.7 : Normals for smooth shading

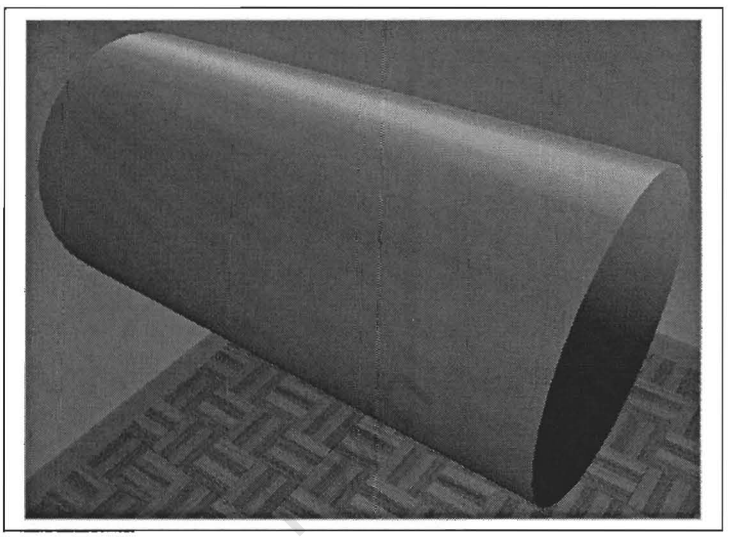


Figure 4.8 : Smooth shaded pipe

```
void CBlocks::Pipe(int Axis, GLfloat StartX, GLfloat StartY, GLfloat StartZ,
GLfloat Length, GLfloat Width )
{
  glBegin(GL_QUAD_STRIP);
     for (int i = 0 ; i < 31 ; i++)
     {  switch (Axis)
        {
        case 1 :
           glNormal3f(0.0f, SinX[i], CosX[i]);
           glVertex3f(StartX, StartY + SinX[i] * Width,
                                          StartZ + CosX[i] * Width);
           glVertex3f(StartX + Length, StartY + SinX[i] * Width,
                                          StartZ + CosX[i] * Width);
           break;
        case 2:
           glNormal3f(SinX[i], 0.0f, CosX[i]);
           glVertex3f(StartX + SinX[i] * Width, StartY,
                                          StartZ + CosX[i] * Width);
           glVertex3f(StartX + SinX[i] * Width, StartY + Length,
                                          StartZ + CosX[i] * Width);
           break;
        case 3:
           glNormal3f(CosX[i], SinX[i], 0.0f);
           glVertex3f(StartX + CosX[i] * Width, StartY + SinX[i] * Width,
                                          StartZ);
           glVertex3f(StartX + CosX[i] * Width, StartY + SinX[i] * Width,
                                          StartZ + Length );
           break;
        default :
           break;
        }     }
  glEnd();
}
```

OpenGL draws the cylinder with the use of the GL_QUAD_STRIP identifier. Once glBegin() is started with the GL_QUAD_STRIP option, each pair of vertices, except for the first pair, is drawn as a rectangle by joining with the previous pair. By displacing each new pair of points along two axes by pre-calculated amounts from the lookup tables, one can generate a 3D cylinder.

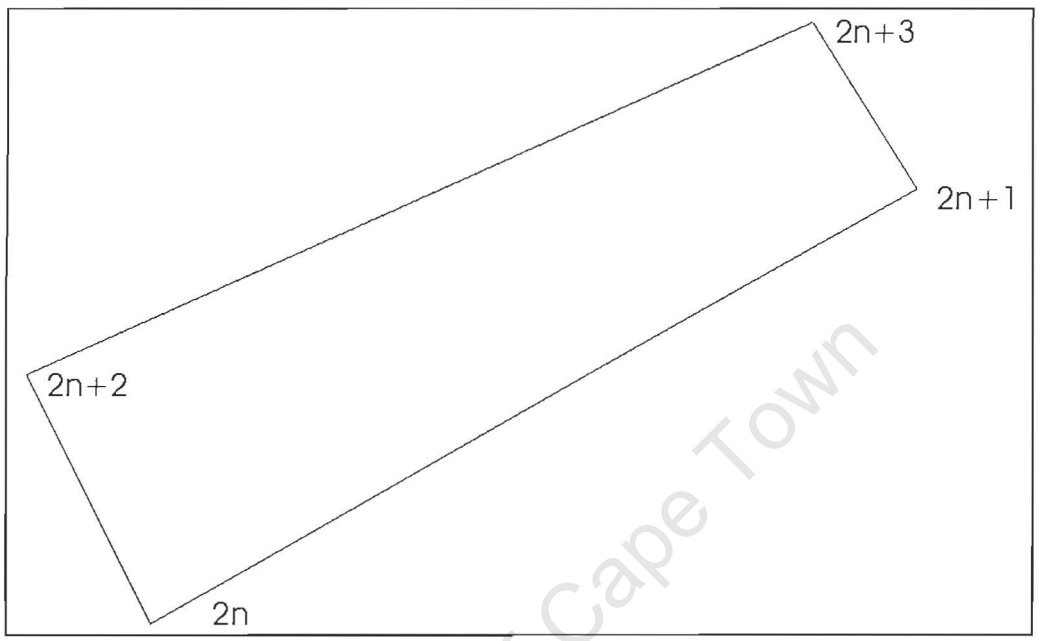


Figure 4.9 : Point sequence for GL_QUAD_STRIP

By specifying the Axis variable, I can also control along which axis the cylinder is drawn. The switch on Axis statement switches between drawing along the X, Y and Z-axis. Below is an image of three pipes. All of them have the same width and length, and all starting from the same point, but each is drawn along a different axis.

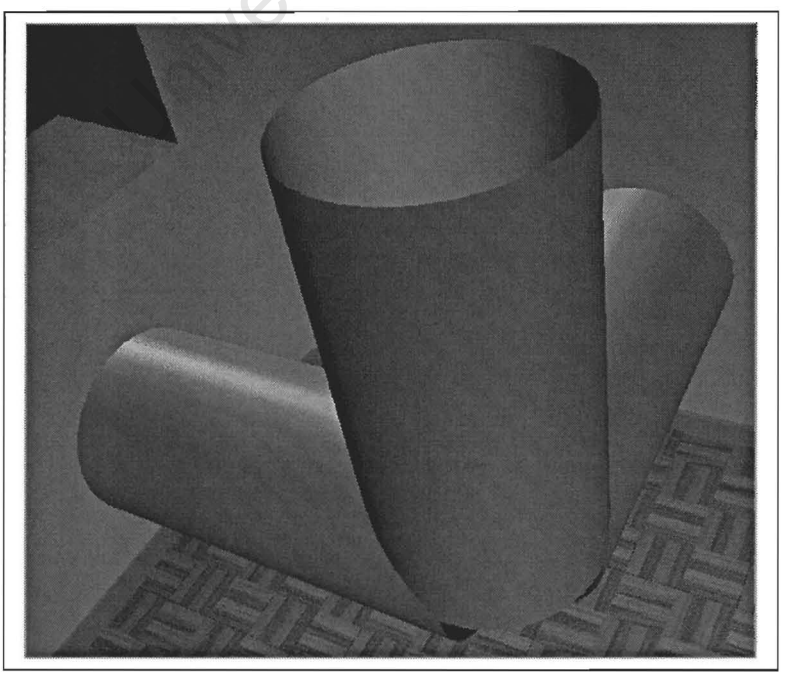


Figure 4.10 : Three pipes at right angles

ShrinkPipe

The ShrinkPipe function is almost identical in functionality to the Pipe function with the one exception that it has another width, namely a destination width. This way a pipe that constricts or expands at one end can be generated. If Width and WidthDest are equal, then this function will perform the same as the Pipe function. The Pipe function is still however used because it is a more efficient. Because both functions are used often, it is worth the effort of having two different functions.

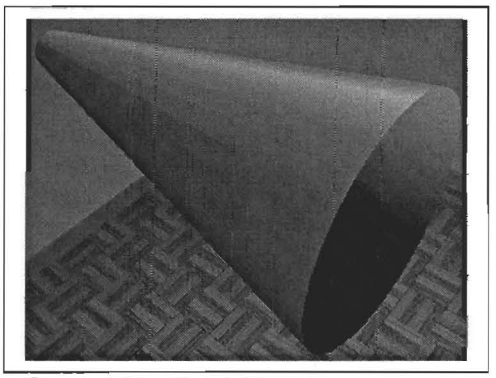


Figure 4.11 : Shrink Pipe image

```
void CBlocks::ShrinkPipe(int Axis, GLfloat StartX, GLfloat StartY, GLfloat
StartZ, GLfloat Length, GLfloat Width, GLfloat WidthDest )
{
   GLfloat tmp, norm;
   tmp = (Width - WidthDest)/Length;
   norm = (GLfloat)sqrt(1 + tmp * tmp);
   glBegin(GL_QUAD_STRIP);
      for (int i = 0 ; i < 31 ; i++)
      {
         switch (Axis)
         {
         case 1 :
            glNormal3f( tmp/norm, SinX[i]/norm, CosX[i]/norm);
            glVertex3f(StartX, StartY + SinX[i] * Width,
                                          StartZ + CosX[i] * Width);
            glVertex3f(StartX + Length, StartY + SinX[i] * WidthDest,
                                          StartZ + CosX[i] * WidthDest);
            break;
         case 2:
            glNormal3f(SinX[i]/norm, tmp/norm, CosX[i]/norm);
            glVertex3f(StartX + SinX[i] * Width, StartY,
                                          StartZ + CosX[i] * Width);
            glVertex3f(StartX + SinX[i] * WidthDest, StartY + Length,
                                          StartZ + CosX[i] * WidthDest);
            break;
         case 3:
            glNormal3f(CosX[i]/norm, SinX[i]/norm, tmp/norm);
            glVertex3f(StartX + CosX[i] * Width, StartY + SinX[i] * Width,
                                          StartZ);
            glVertex3f(StartX + CosX[i] * WidthDest,
                              StartY + SinX[i] * WidthDest, StartZ + Length );
            break;
         default :
            break;
         }
      }
   glEnd();
}
```

Knee

The knee is an edge connector that allows the connection of two pipes at right angles to each other. What this function does is join the edge of one pipe with the edge of another pipe. The reason for there being six case statements is that for each of the axes that the incoming pipe could be coming along, there are two axis along which the outgoing pipe could be leaving.

As before the edges of the slats have different normal vectors, with each end taking up the normal of the pipe that it is connecting to. This way, although the edge is triangular in profile, it appears smooth once the colours generated by the normal vectors are interpolated.

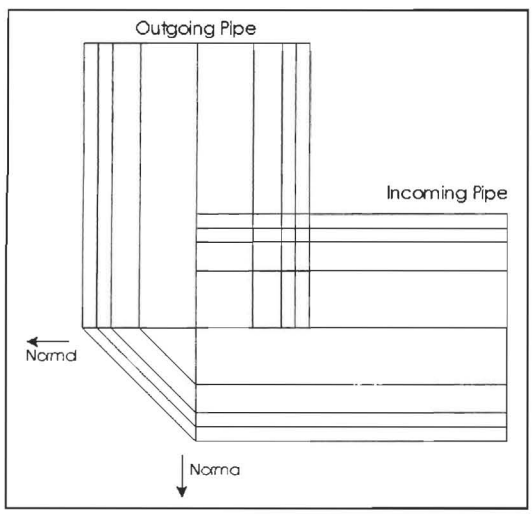


Figure 4.12 : Design diagram of Knee

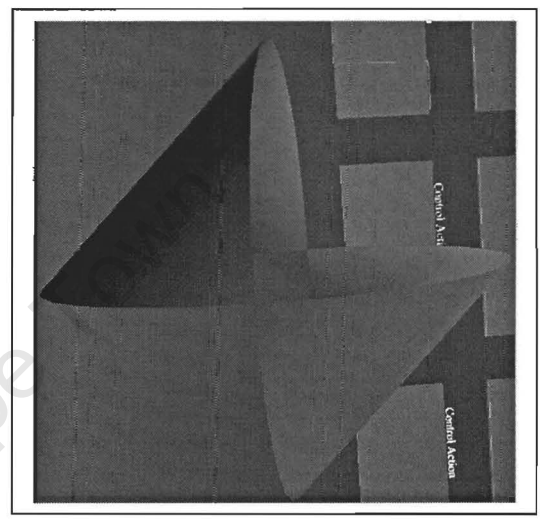


Figure 4.13 : Image of Knee

```
void CBlocks::Knee(int Axis, int AxisOut, int Sign, int SignOut, GLfloat
StartX, GLfloat StartY, GLfloat StartZ, GLfloat Width )
{
  glBegin(GL_QUAD_STRIP);
     for (int i = 0 ; i < 31 ; i++)
     {
        switch (Axis)
        {
        case 1 :
           switch (AxisOut)
           {
           case 2:
              glNormal3f(0.0f, SignOut * SinX[i], CosX[i]);
              glVertex3f(StartX, StartY + SignOut * SinX[i] * Width,
                                 StartZ + CosX[i] * Width);
              glNormal3f(- Sign * SinX[i], 0.0f, CosX[i]);
              glVertex3f(StartX - Sign * SinX[i] * Width, StartY,
                                 StartZ + CosX[i] * Width);
              break;
           case 3:
              glNormal3f(0.0f, SinX[i], SignOut * CosX[i]);
              glVertex3f(StartX, StartY + SinX[i] * Width,
                                 StartZ + SignOut * CosX[i] * Width);
              glNormal3f(-Sign * CosX[i], SinX[i], 0.0f);
              glVertex3f(StartX - Sign * CosX[i] * Width,
                                 StartY + SinX[i] * Width, StartZ);
              break;
```

```
            default :
               break;
            }
            break;
         case 2:
            switch (AxisOut)
            {
            case 1:
               glNormal3f(0.0f, - Sign * SinX[i], CosX[i]);
               glVertex3f(StartX, StartY - Sign * SinX[i] * Width,
                                  StartZ + CosX[i] * Width);
               glNormal3f(SignOut * SinX[i], 0.0f, CosX[i]);
               glVertex3f(StartX + SignOut * SinX[i] * Width, StartY,
                                  StartZ + CosX[i] * Width);
               break;
            case 3:
               glNormal3f(SinX[i], 0.0f, -SignOut * CosX[i]);
               glVertex3f(StartX + SinX[i] * Width, StartY,
                                  StartZ - SignOut * CosX[i] * Width);
               glNormal3f(SinX[i], Sign * CosX[i], 0.0f);
               glVertex3f(StartX + SinX[i] * Width,
                                  StartY + Sign * CosX[i] * Width, StartZ);
               break;
            default :
               break;
            }
            break;
         case 3:
            switch (AxisOut)
            {
            case 1:
               glNormal3f(0.0f, SinX[i], Sign * CosX[i]);
               glVertex3f(StartX, StartY + SinX[i] * Width,
                                  StartZ + Sign * CosX[i] * Width);
               glNormal3f(-SignOut * CosX[i], SinX[i], 0.0f);
               glVertex3f(StartX - SignOut * CosX[i] * Width,
                                  StartY + SinX[i] * Width, StartZ);
               break;
            case 2:
               glNormal3f(SinX[i], 0.0f, -Sign * CosX[i]);
               glVertex3f(StartX + SinX[i] * Width, StartY,
                                  StartZ - Sign * CosX[i] * Width);
               glNormal3f(SinX[i], SignOut * CosX[i], 0.0f);
               glVertex3f(StartX + SinX[i] * Width,
                                  StartY + SignOut * CosX[i] * Width, StartZ);
               break;
            default :
               break;
            }
            break;
         default :
            break;
         }
      }
   glEnd();
}
```

Cube

The Cube function draws a rectangular cube centred on the specified X, Y and Z co-ordinates with sides of size 2dX, 2dY and 2dY.

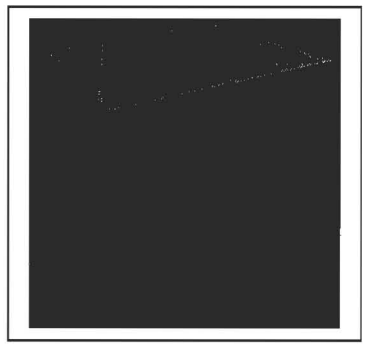


Figure 4.14 : Image of Cube

```
void CBlocks::Cube(GLfloat X, GLfloat Y, GLfloat Z, GLfloat dX, GLfloat dY,
GLfloat dZ)
{
   glBegin(GL_QUADS);
      glNormal3f(0.0f, 0.0f, -1.0f);
      glVertex3f(X + dX, Y + dY, Z - dZ);
      glVertex3f(X + dX, Y - dY, Z - dZ);
      glVertex3f(X - dX, Y - dY, Z - dZ);
      glVertex3f(X - dX, Y + dY, Z - dZ);

      glNormal3f(0.0f, 0.0f, 1.0f);
      glVertex3f(X + dX, Y + dY, Z + dZ);
      glVertex3f(X + dX, Y - dY, Z + dZ);
      glVertex3f(X - dX, Y - dY, Z + dZ);
      glVertex3f(X - dX, Y + dY, Z + dZ);

      glNormal3f(0.0f, -1.0f, 0.0f);
      glVertex3f(X + dX, Y - dY, Z + dZ);
      glVertex3f(X + dX, Y - dY, Z - dZ);
      glVertex3f(X - dX, Y - dY, Z - dZ);
      glVertex3f(X - dX, Y - dY, Z + dZ);

      glNormal3f(0.0f, 1.0f, 0.0f);
      glVertex3f(X + dX, Y + dY, Z + dZ);
      glVertex3f(X + dX, Y + dY, Z - dZ);
      glVertex3f(X - dX, Y + dY, Z - dZ);
      glVertex3f(X - dX, Y + dY, Z + dZ);

      glNormal3f(-1.0f, 0.0f, 0.0f);
      glVertex3f(X - dX, Y + dY, Z + dZ);
      glVertex3f(X - dX, Y + dY, Z - dZ);
      glVertex3f(X - dX, Y - dY, Z - dZ);
      glVertex3f(X - dX, Y - dY, Z + dZ);

      glNormal3f(1.0f, 0.0f, 0.0f);
      glVertex3f(X + dX, Y + dY, Z + dZ);
      glVertex3f(X + dX, Y + dY, Z - dZ);
      glVertex3f(X + dX, Y - dY, Z - dZ);
      glVertex3f(X + dX, Y - dY, Z + dZ);

   glEnd();
}
```

Heater

This function draws a heater virtual object that consists of a tank, a burner and a flame that changes size depending on the value of the FSize variable. By changing the FSize variable from 0 to 100, the flame size is adjusted from non-existent to full size. The other three variables provide the starting co-ordinates of the axis of the incoming pipe. The diagram below shows the structure of the Heater object.

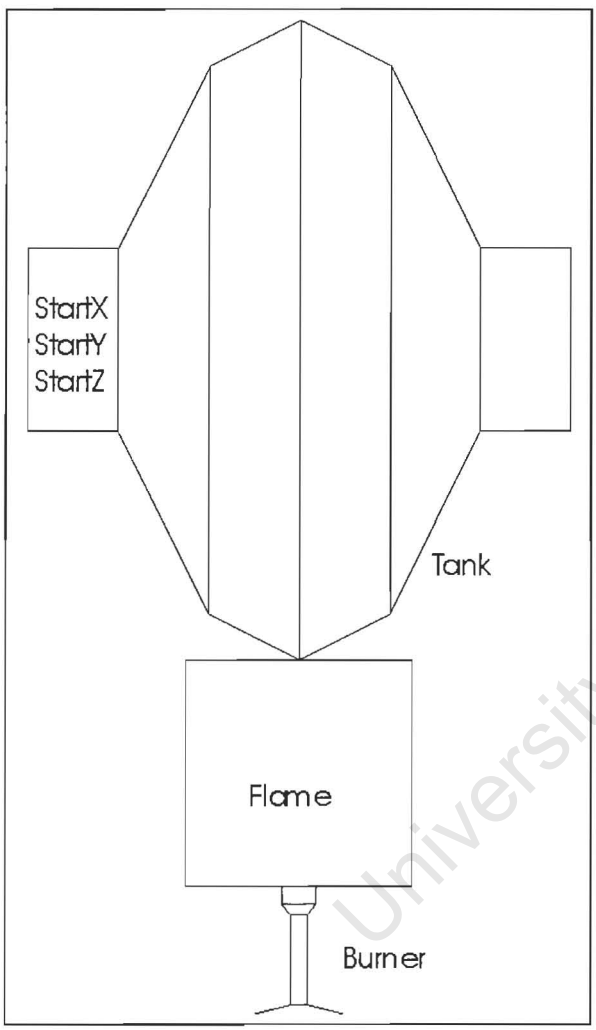


Figure 4.15 : Design of Heater

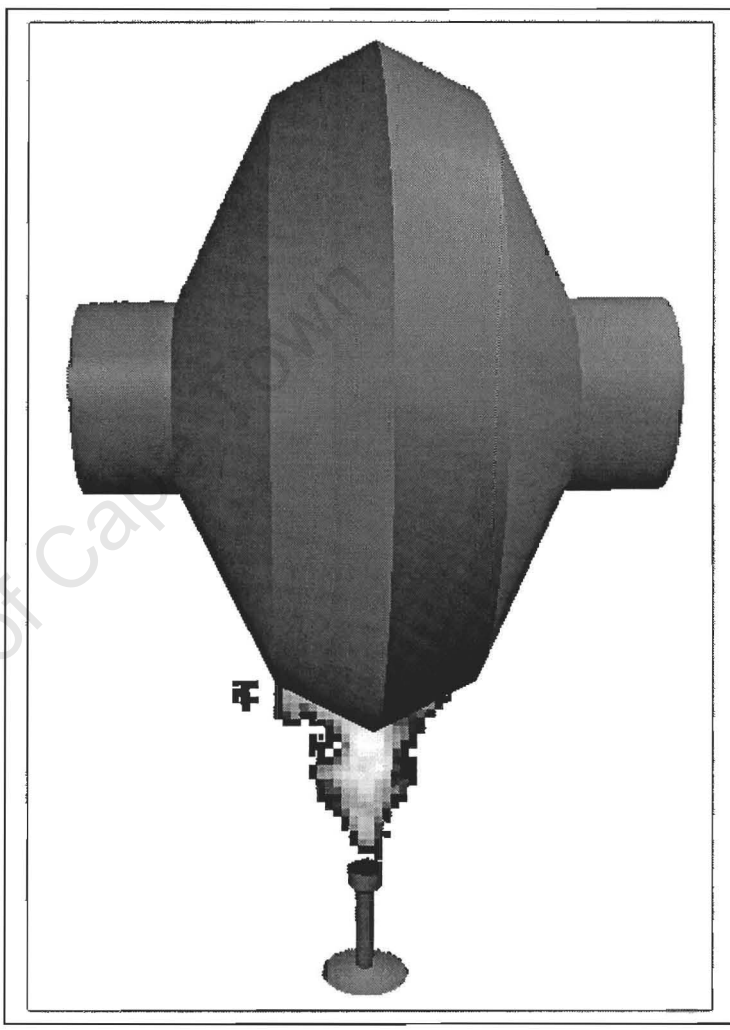


Figure 4.16 : Image of heater

This function starts by first drawing the water tank. The tank is make out of white material, and starts with a pipe of length 10cm and diameter 20cm. Next come two ShrinkPipe functions that expand the pipe first to a diameter of 60cm, and then to 70cm, both of which do it over a length of 10cm each. Next follows a mirror image with two ShrinkPipe and one Pipe function call bringing the pipe diameter back down to 10cm.

The burner is drawn in 90% red and 60% green. The burner consists of a 2cm long pipe with a 4cm diameter, a ShrinkPipe down to a diameter of 2cm, a 10cm long pipe, and finally a ShrinkPipe up to a diameter of 6cm.

The final part of the heater is the flame. The flame is drawn with the lighting calculations disabled because it generates its own light and its brightness is independent of other sources of light. When the flame data is created, the alpha value of each RGBA quadruplet is set to zero. This means that the transparency of each flame pixel is 100%, and thus the colour of each flame pixel is added to the colour of the object behind the flame. This gives the appearance of a semi-transparent flame. The Inc function call ensures that a different flame texture is used with every frame.

```
void CBlocks::Heater(GLfloat FSize, GLfloat StartX, GLfloat StartY, GLfloat
StartZ)
{
   GLfloat DiffuseNorm[] = {1.0f, 1.0f, 1.0f, 1.0f };
   GLfloat DiffuseCopper[] = {0.90f, 0.60f, 0.0f, 1.0f };
   GLfloat  Sz;

   Inc();
   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

   Pipe(3, StartX, StartY, StartZ, 0.1f, 0.1f);
   ShrinkPipe(3, StartX, StartY, StartZ + 0.1f, 0.1f, 0.1f, 0.3f);
   ShrinkPipe(3, StartX, StartY, StartZ + 0.2f, 0.1f, 0.3f, 0.35f);
   ShrinkPipe(3, StartX, StartY, StartZ + 0.3f, 0.1f, 0.35f, 0.3f);
   ShrinkPipe(3, StartX, StartY, StartZ + 0.4f, 0.1f, 0.3f, 0.1f);
   Pipe(3, StartX, StartY, StartZ, 0.1f + 0.5f, 0.1f);

   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseCopper);

   Pipe(2, StartX, StartY - 0.55f, StartZ + 0.3f, -0.02f, 0.02f);
   ShrinkPipe(2, StartX, StartY - 0.57f, StartZ + 0.3f, -0.01f, 0.02f, 0.01f);
   Pipe(2, StartX, StartY - 0.58f, StartZ + 0.3f, -0.1f, 0.01f);
   ShrinkPipe(2, StartX, StartY - 0.68f, StartZ + 0.3f, -0.01f, 0.01f, 0.05f);

   Sz = FSize / 333.3f;
   if (Sz < 0.03f) Sz = 0.03f;
   if (Sz > 0.3f) Sz = 0.3f;

   glDisable(GL_LIGHTING);
   glEnable(GL_TEXTURE_2D);
   glColor3f(1.0f, 1.0f, 1.0f);

   pTex->fpglBindTextureEXT( GL_TEXTURE_2D, pTex->FireTex[Index]);

   glBegin(GL_QUADS);
      glNormal3f(-1.0f, 0.0f, 0.0f);
      glTexCoord2f(0.0f, 0.0f);
      glVertex3f(StartX, StartY - 0.55f - Sz / 4.0f, StartZ + 0.3f - Sz);
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f(StartX, StartY - 0.55f - Sz / 4.0f, StartZ + 0.3f + Sz);
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f(StartX, StartY - 0.55f + Sz + Sz - Sz / 4.0f,
                                                  StartZ + 0.3f + Sz);
      glTexCoord2f(0.0f, 1.0f);
      glVertex3f(StartX, StartY - 0.55f + Sz + Sz - Sz / 4.0f,
                                                  StartZ + 0.3f - Sz);
   glEnd();
   glDisable(GL_TEXTURE_2D);
   glEnable(GL_LIGHTING);
}
```

TempMeter

The TempMeter function draws a virtual thermometer. The thermometer is used to indicate the temperature of the water flowing through a pipe. By increasing the FSize variable from 0 to 100, one can change the thermometer level. The StartX, StartY and StartZ variables give the position of the start of the axis along which temperature thermometer will be drawn. The diagrams below show a wire-frame and a solid colour image of the thermometer.

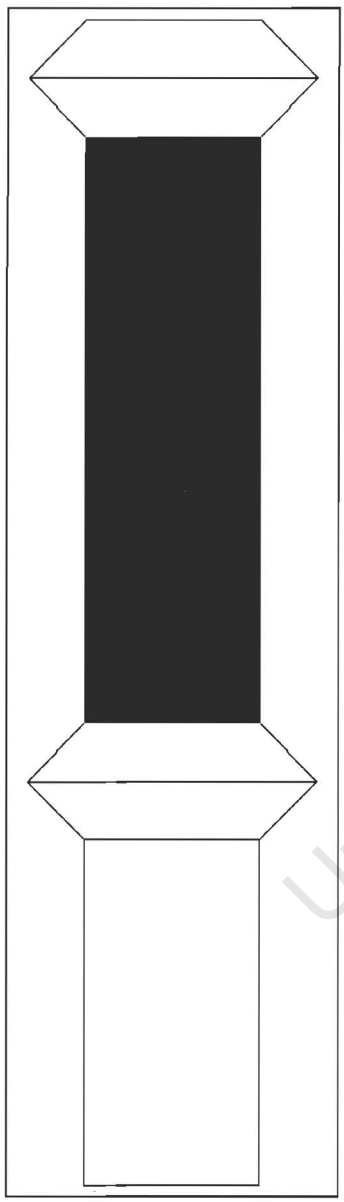


Figure 4.17 : Temperature Meter design

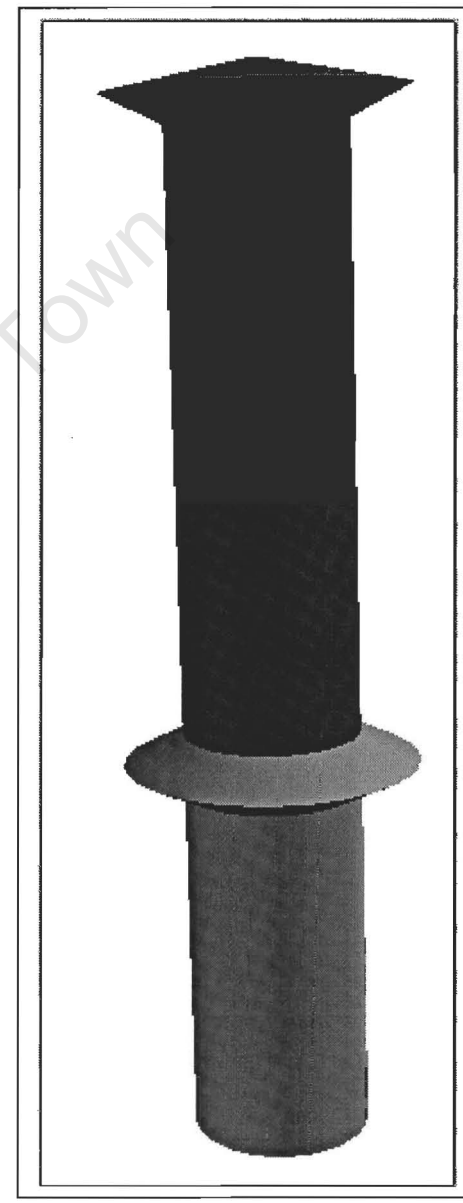


Figure 4.18 : Image of TempMeter

Drawing of the TempMeter commences with a 12cm long pipe, of 6cm in diameter, along the Y-axis. This pipe is then expanded to a diameter of 10cm, and contracted back to 6cm over a total length of 2cm. The colour is now changed from white to red, and a pipe of diameter 6cm and a length of 20cm * FSize / 100 drawn. A blue pipe is then drawn for the remaining length to make up 20 cm, i.e. 20cm – 20cm * FSize / 100. The function ends by expanding a white pipe to a diameter of 10cm, and then back to 6cm over 2cm

```
void CBlocks::TempMeter(GLfloat FSize, GLfloat StartX, GLfloat StartY,
GLfloat StartZ)
{
   GLfloat DiffuseBlue[] = {0.0f, 0.0f, 1.0f, 1.0f };
   GLfloat DiffuseRed[] = {1.0f, 0.0f, 0.0f, 1.0f };
   GLfloat DiffuseNorm[] = {1.0f, 1.0f, 1.0f, 1.0f };
   GLfloat  Sz;

   Sz = FSize / 500.0f;
   if (Sz < 0.0f) Sz = 0.0f;
   if (Sz > 0.2f) Sz = 0.2f;

   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

   Pipe(2, StartX, StartY, StartZ, 0.12f, 0.03f);
   ShrinkPipe(2, StartX, StartY + 0.12f , StartZ, 0.01f, 0.03f, 0.05f);
   ShrinkPipe(2, StartX, StartY + 0.13f, StartZ, 0.01f, 0.05f, 0.03f);

   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseRed);

   Pipe(2, StartX, StartY + 0.14f, StartZ, Sz, 0.03f);

   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseBlue);

   Pipe(2, StartX, StartY + Sz + 0.14f, StartZ, 0.2f - Sz, 0.03f);

   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

   ShrinkPipe(2, StartX, StartY + 0.34f, StartZ, 0.01f, 0.03f, 0.05f);
   ShrinkPipe(2, StartX, StartY + 0.35f, StartZ, 0.01f, 0.05f, 0.001f);

}
```

FlowMeter

The FlowMeter function draws a flow rate meter. It is used to indicate how fast water is flowing through a pipe. In appearance, it is very similar to the TempMeter function with the exception that it lies parallel to the pipe onto which it attaches. It also has both ends connected to the underlying pipe whose flow rate it is intending to measure. This virtual object can be drawn in all three axes. Below is a wire-frame and solid colour image of what the FlowMeter looks like.

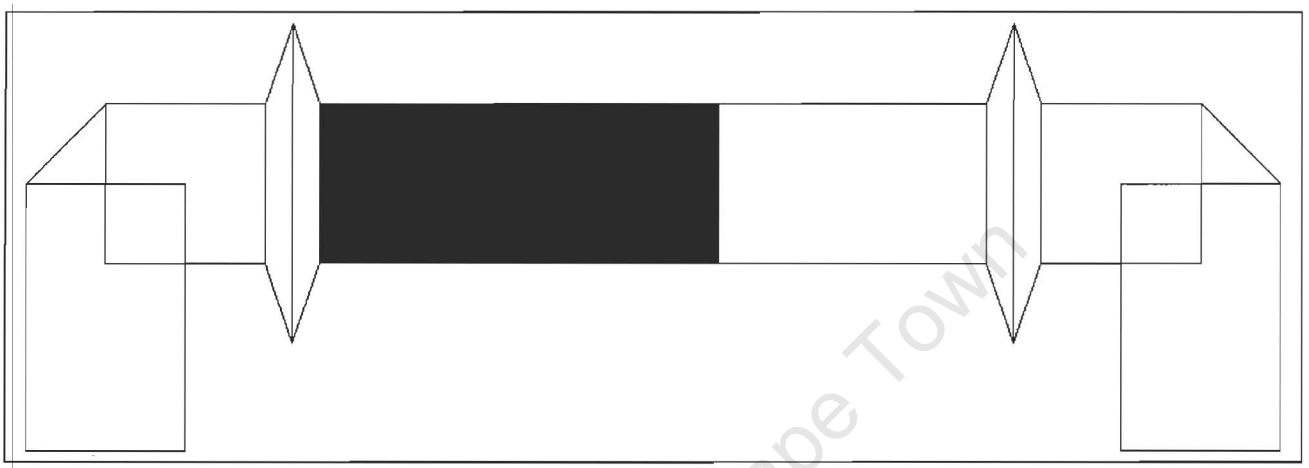


Figure 4.19 : Design of the flow meter

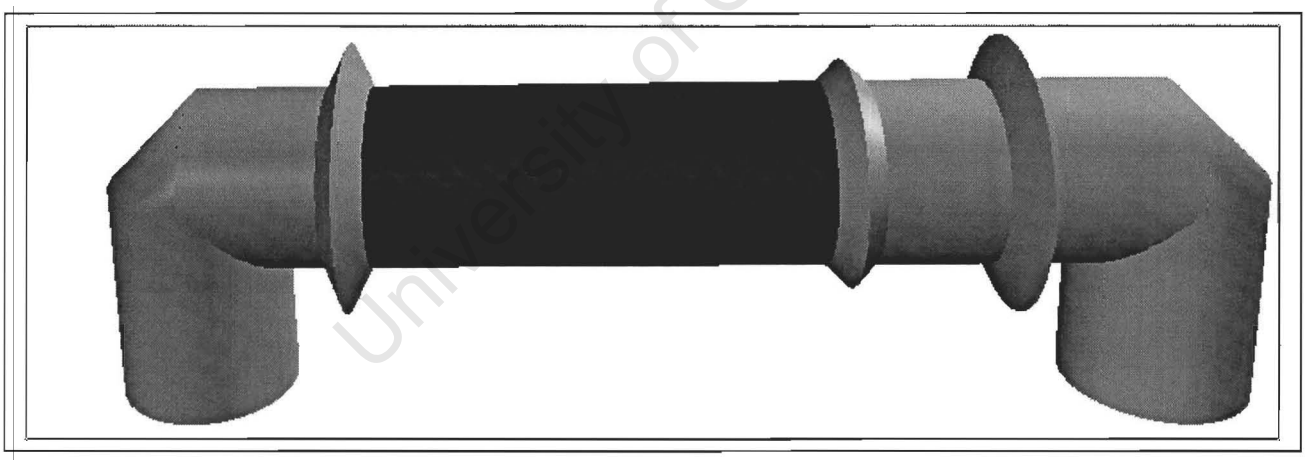


Figure 4.20 : Image of flow meter

Assuming the pipe whose flow rate is to be measured to lie along the X axis, the StartX, StartY and StartZ variables give the point on the surface of this pipe through which the axis of the in-flow pipe to the flow meter will be drawn. This in-flow pipe starts 4cm deep inside the main pipe and is 8cm in diameter and 10cm long.

The in-flow pipe is followed by a knee joint that brings the direction of the TempMeter back along the X-axis. The FlowMeter continues with a 6cm long pipe and two ShrinkPipe commands that expand the diameter to 12cm and contract it back to 8cm, both over a length of 1cm each. This forms the minimum flow rate marker.

The colour is now changed from white to water coloured, where the colour of the water is dependent on the temperature of the water. If the water is cold then the colour will be blue, and if the water is hot, then the colour will be red. In this new colour a pipe of length 25cm * Speed / 100 is drawn. The colour is then changed back to white and the pipe expanded to 10cm and contracted back to 8cm. The remaining length of the pipe to make up a total length of 25cm, (i.e. 25cm – 25cm * Speed / 100) is then drawn.

To end off the flow meter, a maximum flow marker is drawn in the same manner as the minimum flow marker. The out-flow pipe is now drawn consisting of a 6cm pipe, a knee back to along the Y axis, and a 10cm pipe that should embed itself back inside the main pipe.

```
void CBlocks::FlowMeter(int Axis, GLfloat Temp, GLfloat Speed, GLfloat StartX,
GLfloat StartY, GLfloat StartZ)
{
   GLfloat DiffuseBlue[] = {0.0f, 0.0f, 1.0f, 1.0f };
   GLfloat DiffuseNorm[] = {1.0f, 1.0f, 1.0f, 1.0f };
   DiffuseBlue[0] = Temp / 100.0f;
   DiffuseBlue[2] = 1.0f - DiffuseBlue[0];

   GLfloat Length = Speed / 400.0f;
   GLfloat LengthRes = 0.25f - Length;
   switch (Axis)
   {
   case 1:
      Pipe(2, StartX, StartY - 0.04f, StartZ, 0.1f, 0.04f);
      Knee(2, 1, 1, 1, StartX, StartY + 0.06f, StartZ, 0.04f);
      Pipe(1, StartX, StartY + 0.06f, StartZ, 0.06f, 0.04f);
      ShrinkPipe(1, StartX + 0.06f, StartY + 0.06f, StartZ, 0.01f, 0.04f, 0.06f);
      ShrinkPipe(1, StartX + 0.07f, StartY + 0.06f, StartZ, 0.01f, 0.06f, 0.04f);

      glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseBlue);
      Pipe(1, StartX + 0.08f, StartY + 0.06f, StartZ, Length, 0.04f);
      glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

      ShrinkPipe(1, StartX + 0.08f + Length, StartY + 0.06f, StartZ, 0.01f,
                                                          0.04f, 0.05f);
      ShrinkPipe(1, StartX + 0.09f + Length, StartY + 0.06f, StartZ, 0.01f,
                                                          0.05f, 0.04f);

      Pipe(1, StartX + 0.1f + Length, StartY + 0.06f, StartZ, LengthRes, 0.04f);
      ShrinkPipe(1, StartX + 0.35f, StartY + 0.06f, StartZ, 0.01f, 0.04f, 0.06f);
      ShrinkPipe(1, StartX + 0.36f, StartY + 0.06f, StartZ, 0.01f, 0.06f, 0.04f);
      Pipe(1, StartX + 0.37f, StartY + 0.06f, StartZ, 0.06f, 0.04f);

      Knee(1, 2, 1, -1, StartX + 0.43f, StartY + 0.06f, StartZ, 0.04f);
      Pipe(2, StartX + 0.43f, StartY + 0.06f, StartZ, -0.1f, 0.04f);

      break;
```

```
   case 2:
      Pipe(1, StartX + 0.04f, StartY, StartZ, - 0.1f, 0.04f);
      Knee(1, 2, -1, 1, StartX - 0.06f, StartY, StartZ, 0.04f);

      Pipe(2, StartX - 0.06f, StartY, StartZ, 0.06f, 0.04f);
      ShrinkPipe(2, StartX - 0.06f, StartY + 0.06f, StartZ, 0.01f, 0.04f, 0.06f);
      ShrinkPipe(2, StartX - 0.06f, StartY + 0.07f, StartZ, 0.01f, 0.06f, 0.04f);

      glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseBlue);
      Pipe(2, StartX - 0.06f, StartY + 0.08f, StartZ, Length, 0.04f);
      glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

      ShrinkPipe(2, StartX - 0.06f, StartY + 0.08f + Length, StartZ, 0.01f,
                                                  0.04f, 0.05f);
      ShrinkPipe(2, StartX - 0.06f, StartY + 0.09f + Length, StartZ, 0.01f,
                                                  0.05f, 0.04f);

      Pipe(2, StartX - 0.06f, StartY + 0.1f + Length, StartZ, LengthRes, 0.04f);

      ShrinkPipe(2, StartX - 0.06f, StartY + 0.35f, StartZ, 0.01f, 0.04f, 0.06f);
      ShrinkPipe(2, StartX - 0.06f, StartY + 0.36f, StartZ, 0.01f, 0.06f, 0.04f);
      Pipe(2, StartX - 0.06f, StartY + 0.37f, StartZ, 0.06f, 0.04f);

      Knee(2, 1, 1, 1, StartX - 0.06f, StartY + 0.43f, StartZ, 0.04f);
      Pipe(1, StartX - 0.06f, StartY + 0.43f, StartZ, 0.1f, 0.04f);

      break;
   case 3:
      Pipe(2, StartX, StartY - 0.04f, StartZ, 0.1f, 0.04f);
      Knee(2, 3, 1, 1, StartX, StartY + 0.06f, StartZ, 0.04f);

      Pipe(3, StartX, StartY + 0.06f, StartZ, 0.06f, 0.04f);
      ShrinkPipe(3, StartX, StartY + 0.06f, StartZ + 0.06f, 0.01f, 0.04f, 0.06f);
      ShrinkPipe(3, StartX, StartY + 0.06f, StartZ + 0.07f, 0.01f, 0.06f, 0.04f);

      glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseBlue);
      Pipe(3, StartX, StartY + 0.06f, StartZ + 0.08f, Length, 0.04f);
      glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

      ShrinkPipe(3, StartX, StartY + 0.06f, StartZ + 0.08f + Length, 0.01f,
                                                  0.04f, 0.05f);
      ShrinkPipe(3, StartX, StartY + 0.06f, StartZ + 0.09f + Length, 0.01f,
                                                  0.05f, 0.04f);

      Pipe(3, StartX, StartY + 0.06f, StartZ + 0.1f + Length, LengthRes, 0.04f);

      ShrinkPipe(3, StartX, StartY + 0.06f, StartZ + 0.35f, 0.01f, 0.04f, 0.06f);
      ShrinkPipe(3, StartX, StartY + 0.06f, StartZ + 0.36f, 0.01f, 0.06f, 0.04f);
      Pipe(3, StartX, StartY + 0.06f, StartZ + 0.37f, 0.06f, 0.04f);
      Knee(3, 2, 1, -1, StartX, StartY + 0.06f, StartZ + 0.43f, 0.04f);
      Pipe(2, StartX, StartY + 0.06f, StartZ + 0.43f, -0.1f, 0.04f);

      break;
   default:
      break;
   }
}
```

Pump

The pump function draws a virtual fluid pump. This will be used to affect the flow rate of the water in the piping system. The pump is supposed to look like a centrifugal pump whose speed can be controlled. It is possible to draw the pump along all three axes. Below is a wire frame and a shaded solid view of the pump.

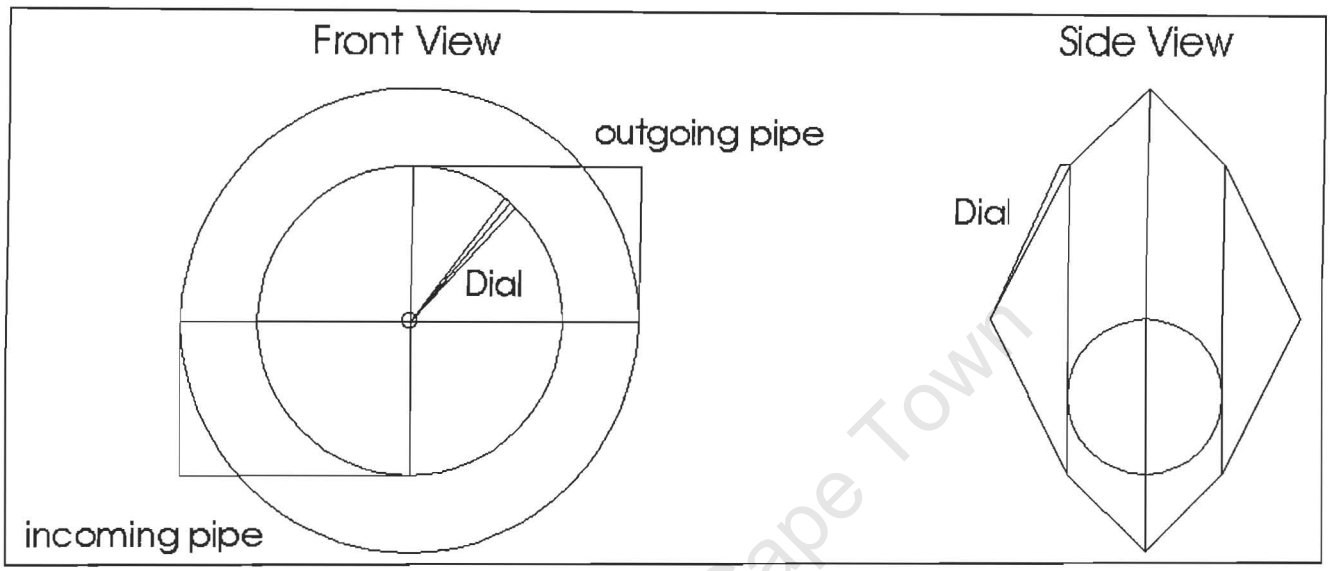


Figure 4.21 : Design of the fluid pump

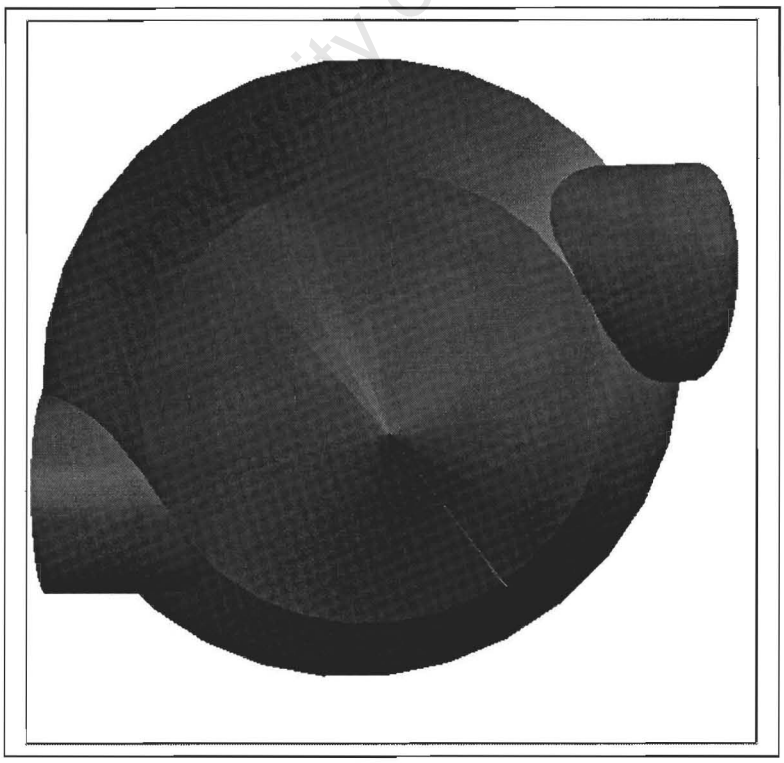


Figure 4.22 : Image of pump

Starting off, the StartX, StartY and StartZ variables give the starting point of the axis around which the incoming pipe will be drawn. Describing for the case when the pump is drawn along the X-axis, the drawing starts with an incoming pipe of length 30cm and 20cm in diameter, and a similar outgoing pipe 20cm higher up.

Next, follow four calls to the ShrinkPipe function, all of which are drawn along the Z-axis. These start from a diameter of 2mm, increasing to a diameter of 40cm over a length of 10cm, and up to 60cm over another 10cm. The remaining two ShrinkPipe function calls shrink the pipe first down to 40cm and then back down to 2mm.

The last thing to draw is a dial. This dial will rotate around the centre of the pump and will be used as an indicator of the speed of the centrifugal pump. By increasing the size of the jumps, in proportion to the Speed variable, the appearance of motion is generated. The dial is a pair of triangles drawn from the centre of the first shrink pipe, outwards to a length of 20cm

```
void CBlocks::Pump(int Axis, GLfloat Speed, GLfloat StartX, GLfloat StartY,
GLfloat StartZ)
{
  GLfloat DiffuseNorm[] = {1.0f, 0.9f, 0.8f, 1.0f };
  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

  int Index = (int)((GLfloat)Speed / 200.0f * 31.0f);
  GLfloat x1, y1, x2, y2, x3, y3;

  x1 = SinX[Index] * 0.2f;
  y1 = CosX[Index] * 0.2f;

  x2 = x1 - y1 / 64.0f;
  y2 = y1 - x1 / 64.0f;

  x3 = x1 + y1 / 64.0f;
  y3 = y1 + x1 / 64.0f;

  switch (Axis)
  {
  case 1:
     Pipe(1, StartX, StartY, StartZ, 0.3f, 0.1f);
     Pipe(1, StartX + 0.3f, StartY + 0.2f, StartZ, 0.3f, 0.1f);
     ShrinkPipe(3, StartX + 0.3f, StartY + 0.1f, StartZ - 0.2f, 0.1f, 0.001f,
                                                          0.2f);
     ShrinkPipe(3, StartX + 0.3f, StartY + 0.1f, StartZ - 0.1f, 0.1f, 0.2f,
                                                          0.3f);
     ShrinkPipe(3, StartX + 0.3f , StartY + 0.1f, StartZ, 0.1f, 0.3f, 0.2f);
     ShrinkPipe(3, StartX + 0.3f, StartY + 0.1f, StartZ + 0.1f, 0.1f, 0.2f,
                                                          0.001f);

     glNormal3f(0.0f, 0.0f, 1.0f);
     glBegin(GL_TRIANGLE_FAN);
        glVertex3f(StartX + 0.3f, StartY + 0.1f, StartZ + 0.2f);
        glVertex3f(StartX + 0.3f + x2, StartY + 0.1f + y2, StartZ + 0.11f);
        glVertex3f(StartX + 0.3f + x1, StartY + 0.1f + y1, StartZ + 0.11f);
        glVertex3f(StartX + 0.3f + x3, StartY + 0.1f + y3, StartZ + 0.11f);
     glEnd();

     break;
```

```
    case 2:

       Pipe(2, StartX, StartY, StartZ, 0.3f, 0.1f);
       Pipe(2, StartX, StartY + 0.3f, StartZ + 0.2f, 0.3f, 0.1f);
       ShrinkPipe(1, StartX - 0.2f, StartY + 0.3f, StartZ + 0.1f, 0.1f, 0.001f,
                                                                  0.2f);
       ShrinkPipe(1, StartX - 0.1f, StartY + 0.3f, StartZ + 0.1f, 0.1f, 0.2f,
                                                                  0.3f);
       ShrinkPipe(1, StartX , StartY + 0.3f, StartZ + 0.1f, 0.1f, 0.3f, 0.2f);
       ShrinkPipe(1, StartX + 0.1f, StartY + 0.3f, StartZ + 0.1f, 0.1f, 0.2f,
                                                                  0.001f);

       glNormal3f(-1.0f, 0.0f, 0.0f);
       glBegin(GL_TRIANGLE_FAN);
          glVertex3f(StartX - 0.2f, StartY + 0.3f, StartZ + 0.1f);
          glVertex3f(StartX - 0.11f, StartY + 0.3f + y2, StartZ + 0.1f + x2);
          glVertex3f(StartX - 0.11f, StartY + 0.3f + y1, StartZ + 0.1f + x1);
          glVertex3f(StartX - 0.11f, StartY + 0.3f + y3, StartZ + 0.1f + x3);
       glEnd();


       break;

    case 3:

       Pipe(3, StartX, StartY, StartZ, 0.3f, 0.1f);
       Pipe(3, StartX, StartY + 0.2f, StartZ + 0.3f, 0.3f, 0.1f);
       ShrinkPipe(1, StartX - 0.2f, StartY + 0.1f, StartZ + 0.3f, 0.1f, 0.001f,
                                                                  0.2f);
       ShrinkPipe(1, StartX - 0.1f, StartY + 0.1f, StartZ + 0.3f, 0.1f, 0.2f,
                                                                  0.3f);
       ShrinkPipe(1, StartX , StartY + 0.1f, StartZ + 0.3f, 0.1f, 0.3f, 0.2f);
       ShrinkPipe(1, StartX + 0.1f, StartY + 0.1f, StartZ + 0.3f, 0.1f, 0.2f,
                                                                  0.001f);

       glNormal3f(-1.0f, 0.0f, 0.0f);
       glBegin(GL_TRIANGLE_FAN);
          glVertex3f(StartX - 0.2f, StartY + 0.1f, StartZ + 0.3f);
          glVertex3f(StartX - 0.11f, StartY + 0.1f + y2, StartZ + 0.3f + x2);
          glVertex3f(StartX - 0.11f, StartY + 0.1f + y1, StartZ + 0.3f + x1);
          glVertex3f(StartX - 0.11f, StartY + 0.1f + y3, StartZ + 0.3f + x3);
       glEnd();

       break;

    default:
       break;

    }
}
```

Tank

The Tank function draws a virtual storage tank. Water flows into and out from the tank, and depending on the net flux of water, the water level will change. It is both a functional virtual object and a virtual indicator, because it indicates its own level.

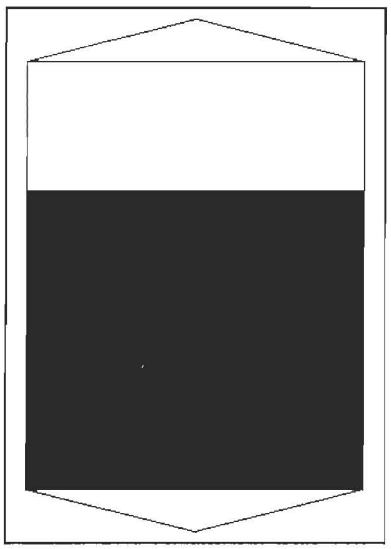


Figure 4.23 : Design of liquid tank

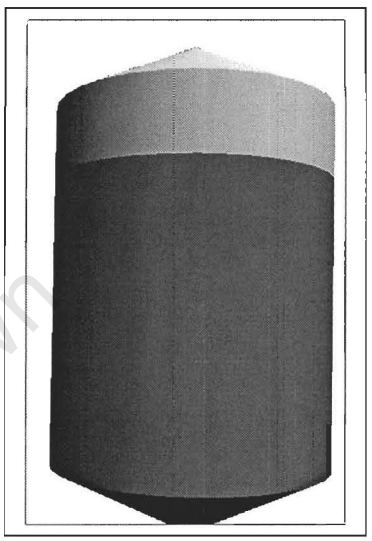


Figure 4.24 : Image of tank

This function starts by setting the drawing colour to the current water colour. The water colour is dependent on the water temperature, where the temperature is derived from the Temp variable. The Full variable tells the function how full is should draw the tank. The level indication is done by drawing a percentage of the tank in the water colour, and the remainder of the tank in the default colour of white.

The StartX, StartY and StartZ variables form a point 30cm above the bottom tip of the tank. The drawing of the tank is done by two ShrinkPipe and two Pipe function calls. A ShrinkPipe call expands a pipe from 2mm to 80cm in diameter over a length of 20cm. Next comes the water coloured pipe which is drawn to a length of 100cm * Full / 100, with the remainder of the 1m length being drawn in white. The function ends off by the last ShrinkPipe function call, which contracts the pipe back to a diameter of 2mm.

```
void CBlocks::Tank(GLfloat Full, GLfloat Temp, GLfloat StartX, GLfloat
StartY, GLfloat StartZ)
{
   GLfloat DiffuseBlue[] = {0.0f, 0.0f, 1.0f, 1.0f };
   GLfloat DiffuseNorm[] = {1.0f, 1.0f, 1.0f, 1.0f };

   DiffuseBlue[0] = Temp / 100.0f;
   DiffuseBlue[2] = 1.0f - DiffuseBlue[0];

   ShrinkPipe(2, StartX, StartY - 0.3f, StartZ, 0.2f, 0.001f, 0.4f);
   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseBlue);
   Pipe(2, StartX, StartY - 0.1f, StartZ, Full / 100.0f, 0.4f);
   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);
   Pipe(2, StartX, StartY - 0.1f + Full / 100.0f, StartZ, 1.0f - Full /
                                          100.0f, 0.4f);
   ShrinkPipe(2, StartX, StartY + 0.9f, StartZ, 0.2f, 0.4f, 0.001f);
}
```

SlushGate

Due to the difficulty in drawing a 3D tap-valve, that can indicate how much it is open, a simpler way to indicate how much water is being taped off is going to be used. This function draws two cubes, with one being separated into two parts whose size depends on the Open variable. The difference in height of the water level in the two cubes is an indication of how much the valve is open.

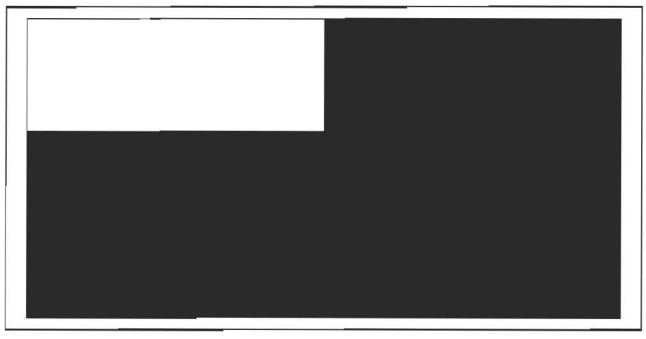


Figure 4.25 : Design of Slush Gate

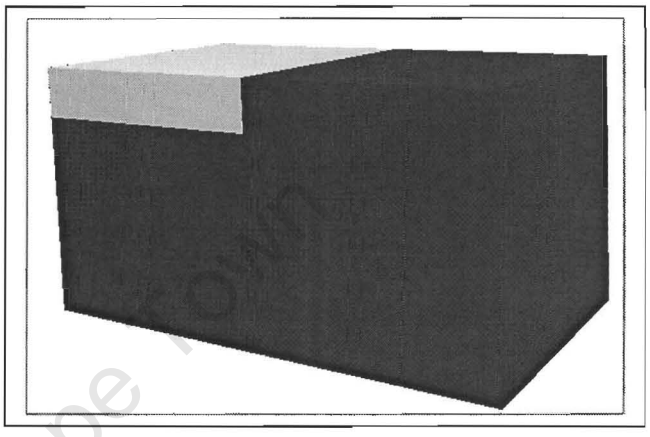


Figure 4.26 : Image of Slush Gate

The first step is to set the drawing colour to the temperature dependent water colour, and to draw a Cube. Next to this cube, a second cube is drawn, with its bottom lining up to the previous cube. The second cube's height is however somewhere between 0 and the height of the previous cube (30cm) depending on the value of the Open variable. The drawing colour is next changed back to white and a third cube is drawn on top of the second cube. The size of the third cube is adjusted such that the height of both the second and third cube combined is the same as that of the first cube. This is the last member function of my CBlocks class and forms the end of the basic building blocks in the virtual environment.

```
void CBlocks::SlushGate(GLfloat Open, GLfloat Temp, GLfloat StartX, GLfloat
StartY, GLfloat StartZ)
{
  GLfloat DiffuseBlue[] = {0.0f, 0.0f, 1.0f, 1.0f };
  GLfloat DiffuseNorm[] = {1.0f, 1.0f, 1.0f, 1.0f };

  GLfloat sz = 0.15f * Open / 100.0f;
  GLfloat left = 0.15f - sz;

  DiffuseBlue[0] = Temp / 100.0f;
  DiffuseBlue[2] = 1.0f - DiffuseBlue[0];

  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseBlue);
  Cube(StartX, StartY, StartZ - 0.15f, 0.15f, 0.15f, 0.15f);
  Cube(StartX, StartY - 0.15f + sz, StartZ - 0.45f, 0.15f, sz, 0.15f);

  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);
  Cube(StartX, StartY + 0.15f - left, StartZ - 0.45f, 0.15f, left, 0.15f);
}
```

4.2.2 COpenGLObject Class

The COpenGLObject Class provides a slightly higher level of graphical building blocks than was exported by the CBlocks class. Inside this class there are functions that draw the room, the factory, and position the lights. The class is build up of seven member variables, of which only two are available to the class object owner, with the others being declared protected. There are also twelve member functions that form the functionality that is exported by this class.

```
class COpenGLObject : public CObject
{
public:
   BOOL Alarm;
   void SetupPos(GLfloat Sens[], int frm);
   void Action(void);
   void FloorSky(void);
   void FloorCeilingPanel(void);
   void DemoMode(void);
   void Skirting(void);
   void Walls(void);
   void CLS(void);
   void SetupLighting(void);
   void Lights(void);
   COpenGLObject();
   virtual ~COpenGLObject();
   CTextures *pTex;

protected:
   GLfloat Direc;
   GLfloat dSky;
   GLfloat X;
   GLfloat Y;
   GLfloat Z;
};
```


Starting with the protected member variables: The floating point variables X, Y and Z are the current angles around the respective axis. They are used when the application is placed in demo mode, where the whole virtual environment rotates around each axis with different increments. The dSky variable is used for a similar purpose. When the sky is drawn, the texture map starting point is incremented by a small amount that is stored in the dSky variable. This gives the appearance of moving clouds in the sky.

The Direc variable is used to store the direction in which the alarm light is pointing. The alarm light is a red spotlight that rotates from the ceiling around the Y-axis that runs through the centre of the room. It is normally off, but goes on when the temperature of the water exceeds a certain value.

There are two public member variables. The first pTex is a pointer to the object of class CTexture that is created and initialised inside the CControlRoom class. The CControlRoom class, which also created the object of type COpenGLObject, copies a pointer to its CTexture object to the pTex variable so that textures can be used by the member functions of the COpenGLObject class. The Alarm variable is also set from within the CControlRoom class and is an indicator to the COpenGLObject class to whether or not it should draw the alarm light.

Moving onto the member functions, the constructor initialising some member variables, while the destructor does not do anything. The Lights function draws the lights. The lights are two spotlights drawn in opposite corners of the control room. CLS is a simple function that clears the screen. The Walls function draws the rounded walls of the control room, and the Skirting function draws a skirting that connects the floor to the walls. DemoMode is a function that rotates the view around the origin.

The FloorCeilingPanel function draws the floor inside the control room, the ceiling, and a small panel that faces out onto the factory floor from within the control room. The FloorSky function draws the floor in the factory as well as the sky above. Lastly, the action function increments the texture offset when drawing the sky to create the impression of moving clouds.

```
void SetupPos(GLfloat Sens[]);
void Action(void);
void FloorSky(void);
void FloorCeilingPanel(void);
void DemoMode(void);
void Skirting(void);
void Walls(void);
void CLS(void);
void SetupLighting(void);
void Lights(void);
COpenGLObject();
virtual ~COpenGLObject();
```

COpenGLObject, ~ COpenGLObject

The destructor does not do any processing, and the constructor initialises some variables to zero.

```
COpenGLObject::COpenGLObject()
{
   X = 0;
   Y = 0;
   Z = 0;

   Direc = 0.0f;
   dSky = 0;
}

COpenGLObject::~COpenGLObject()
{

}
```

Lights

The Lights function draws two light sources at opposite ends of the room. It consists of four sets of commands, each between a pair of glBegin and glEnd statements. The first two draw the light sockets for both lights, and the second two draw the light bulb for both. The light sockets are drawn in light grey, and are in the shape of a rounded pyramid at whose base the bulb is drawn. The bulb is drawn as a slightly yellow polygon the fits into the shape of the light socket. The diagrams below show the wire frame and a solid shaded view of a single light.

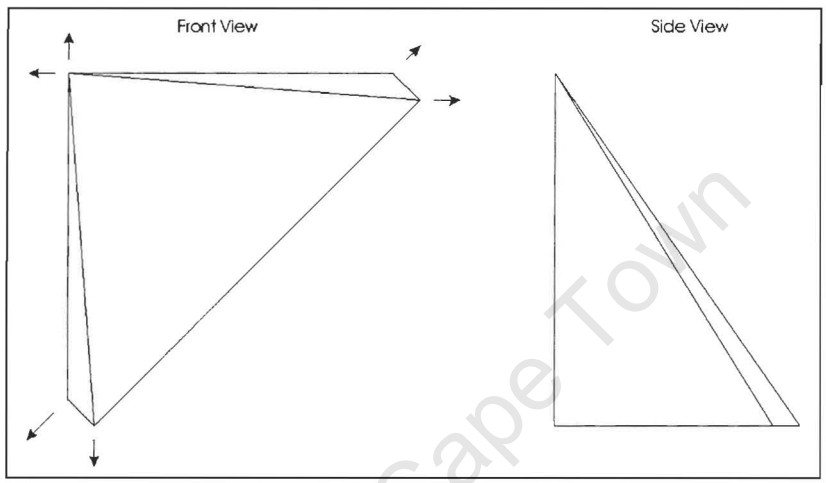


Figure 4.27 : Design diagram of light fittings

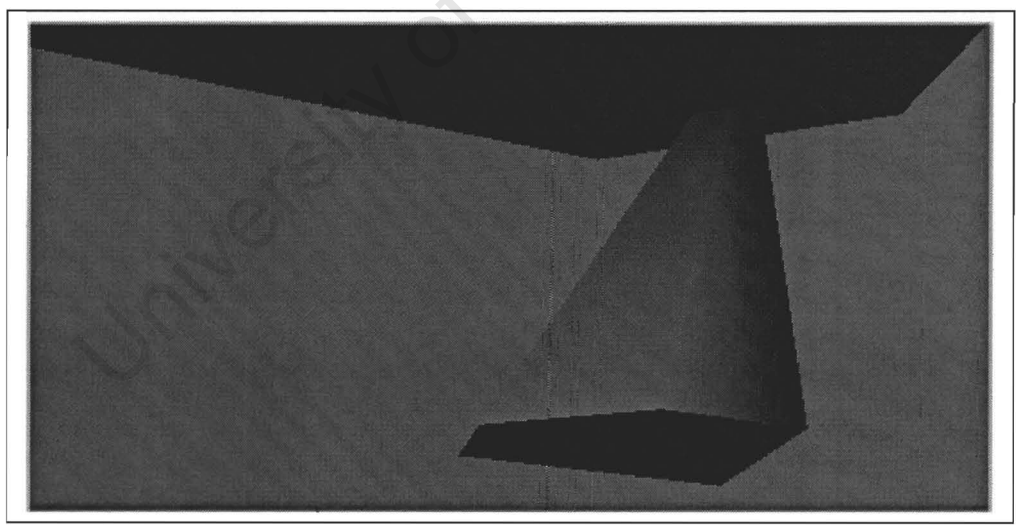


Figure 4.28 : Image of light fitting

```
void COpenGLObject::Lights()
{
  GLfloat YellowLight[] = { 1.0f, 0.9f, 0.8f, 1.0f };
  GLfloat LightDirect[] = { +0.5f/1.5f, -1.0f/1.5f, 0.5f/1.5f };
  GLfloat Light2Direct[] = { -1.0f/3.0f, -1.0f/3.0f, -1.0f/3.0f };
  GLfloat NoLight[] = { 0.0f, 0.0f, 0.0f, 1.0f };
  GLfloat DiffuseNorm[] = {0.8f, 0.8f, 0.8f, 1.0f };

  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);
```

```
    glBegin(GL_TRIANGLE_FAN);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(-1.05f, 1.05f, -1.05f);
       glNormal3f(-0.707f, 0.0f, -0.707f);
       glVertex3f(-1.05f, 0.85f, -1.05f);
       glNormal3f(-1.0f, 0.0f, 0.0f);
       glVertex3f(-1.05f, 0.85f, -0.95f);
       glNormal3f(0.0f, 0.0f, 1.0f);
       glVertex3f(-1.0f, 0.85f, -0.95f);
       glNormal3f(1.0f, 0.0f, 0.0f);
       glVertex3f(-0.95f, 0.85f, -1.0f);
       glNormal3f(0.0f, 0.0f, -1.0f);
       glVertex3f(-0.95f, 0.85f, -1.05f);
       glNormal3f(-0.707f, 0.0f, -0.707f);
       glVertex3f(-1.05f, 0.85f, -1.05f);
    glEnd();

    glBegin(GL_TRIANGLE_FAN);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(1.05f, 1.05f, 1.05f);
       glNormal3f(0.707f, 0.0f, 0.707f);
       glVertex3f(1.05f, 0.85f, 1.05f);
       glNormal3f(1.0f, 0.0f, 0.0f);
       glVertex3f(1.05f, 0.85f, 0.85f);
       glNormal3f(0.0f, 0.0f, -1.0f);
       glVertex3f(0.95f, 0.85f, 0.85f);
       glNormal3f(-1.0f, 0.0f, 0.0f);
       glVertex3f(0.85f, 0.85f, 0.95f);
       glNormal3f(0.0f, 0.0f, 1.0f);
       glVertex3f(0.85f, 0.85f, 1.05f);
       glNormal3f(0.707f, 0.0f, 0.707f);
       glVertex3f(1.05f, 0.85f, 1.05f);
    glEnd();

    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, YellowLight);
    glBegin(GL_TRIANGLE_FAN);
        glNormal3fv(LightDirect);
        glVertex3f(-1.05f, 0.95f, -1.05f);
        glVertex3f(-1.05f, 0.95f, -1.0f);
        glVertex3f(-1.025f, 0.95f, -1.0f);
        glVertex3f(-1.0f, 0.95f, -1.025f);
        glVertex3f(-1.0f, 0.95f, -1.05f);
        glVertex3f(-1.05f, 0.95f, -1.05f);
    glEnd();
    glBegin(GL_TRIANGLE_FAN);
        glNormal3fv(Light2Direct);
        glVertex3f(1.05f, 0.95f, 1.05f);
        glVertex3f(1.05f, 0.95f, 0.95f);
        glVertex3f(1.0f, 0.95f, 0.95f);
        glVertex3f(0.95f, 0.95f, 1.0f);
        glVertex3f(0.95f, 0.95f, 1.05f);
        glVertex3f(1.05f, 0.95f, 1.05f);
    glEnd();
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, NoLight);

}
```

SetupLighting

This function is responsible for setting up the lights for the whole virtual environment. There are four light sources, of which one is selectively on. Two of the lights are inside the control room, one is the sunlight falling outside on the factory, and the fourth light is the alarm light which rotates around the centre of the control room when an alarm condition exits.

The first two lights that are inside the control room are both spotlights with a cut off angle of 90 degree. The diagrams below show the position and directions of these two lights as well as their colour. The diffused colour is the colour that a white piece of paper would look like under this light source from any angle. The specular colour is the colour that is added to the diffuse colour when the viewing angle nearly corresponds to the angle of incidence of the light beams. I.e. it is the colour that is visible due to the mirror effect in objects where you can see the reflection of the light source in the object.

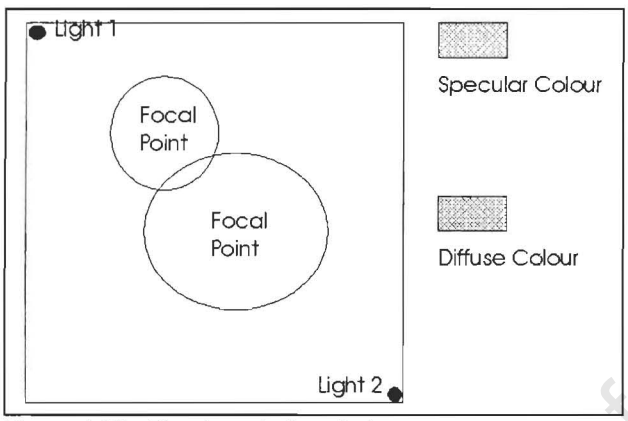


Figure 4.29 : Plan layout of control room

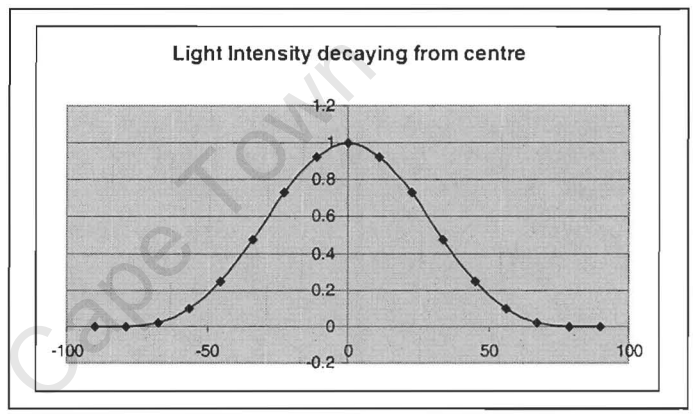


Figure 4.30 : Graph of light decay with angle away from axis

The third light source is the one that will be shining onto the factory, and should be similar to sunlight. For this reason it is not defined a spotlight, it has no spot direction, spot exponent, or spot cut off. It is only given a specular and diffused colour and a position, which is interpreted as a direction vector because of a zero w component in the position vector. The specular colour is the same as for the above two light sources and the diffuse colour is much less yellow and looks more like the specular colour shown above.

The last light source is the alarm / emergency light. This light has similar settings to the first two lights, with a few minor exceptions. Firstly, the diffuse colour is pure red. Secondly the position vector is in the middle of the room, and lastly, the direction vector changes with every frame to rotate around the room. Below is a image capture with the emergency light on.

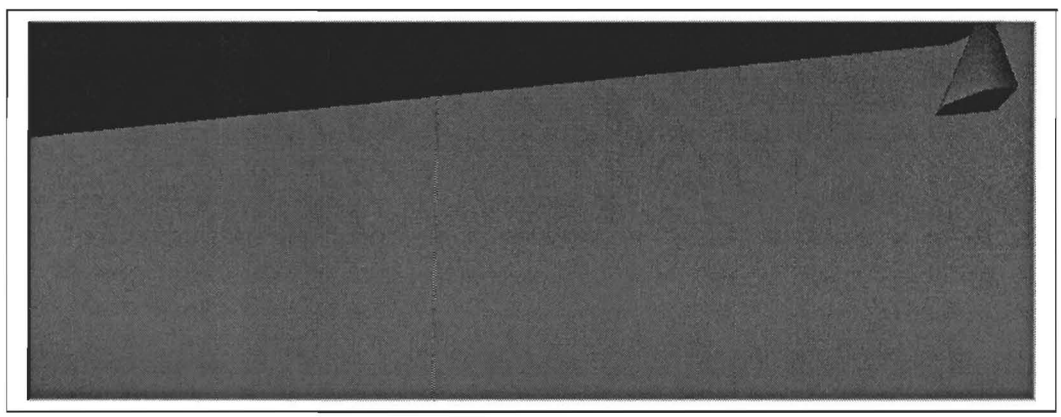


Figure 4.31 : Image of effect of alarm light on the colour of the wall

```
void COpenGLObject::SetupLighting()
{
   GLfloat YellowLight[] = { 1.0f, 0.9f, 0.8f, 1.0f };
   GLfloat LightDirect[] = { +0.5f/1.5f, -1.0f/1.5f, 0.5f/1.5f };
   GLfloat Light2Direct[] = { -1.0f/3.0f, -1.0f/3.0f, -1.0f/3.0f };
   GLfloat Light4Direct[] = { 0.0f, 0.0f, 0.0f};
   GLfloat LightPos[] = { -1.0f, 0.9f, -1.0f, 1.0f };
   GLfloat Light2Pos[] = { 1.0f, 0.9f, 1.0f, 1.0f };
   GLfloat Light3Pos[] = { -0.5f, 0.707f, 0.5f, 0.0f };
   GLfloat Light4Pos[] = { 0.0f, 0.9f, 0.0f };
   GLfloat BrownSpecular[] = { 1.0f, 0.7f, 0.2f, 1.0f };
   GLfloat DiffLight[] = { 1.0f, 0.9f, 0.2f, 1.0f };
   GLfloat SunLight[] = { 1.0f, 0.9f, 0.7f, 1.0f };
   GLfloat RedLight[] = {1.0f, 0.0f, 0.0f, 1.0f};

   Direc += 0.1f;
   if (Direc > 6.28) Direc -= 6.28f;

   if (Alarm)
   {
      Beep(1000,10);
      glEnable(GL_LIGHT3);
   }
   else glDisable(GL_LIGHT3);

   Light4Direct[0] = (GLfloat)cos(Direc);
   Light4Direct[2] = (GLfloat)sin(Direc);

   glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 4);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, DiffLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, YellowLight);
   glLightfv(GL_LIGHT0, GL_POSITION, LightPos);
   glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, LightDirect);
   glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 90.0f);

   glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 4);
   glLightfv(GL_LIGHT1, GL_DIFFUSE, DiffLight);
   glLightfv(GL_LIGHT1, GL_SPECULAR, YellowLight);
   glLightfv(GL_LIGHT1, GL_POSITION, Light2Pos);
   glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, Light2Direct);
   glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 90.0f);

   glLightfv(GL_LIGHT2, GL_DIFFUSE, SunLight);
   glLightfv(GL_LIGHT2, GL_SPECULAR, YellowLight);
   glLightfv(GL_LIGHT2, GL_POSITION, Light3Pos);

   glLightf(GL_LIGHT3, GL_SPOT_EXPONENT, 4);
   glLightfv(GL_LIGHT3, GL_DIFFUSE, RedLight);
   glLightfv(GL_LIGHT3, GL_SPECULAR, YellowLight);
   glLightfv(GL_LIGHT3, GL_POSITION, Light4Pos);
   glLightfv(GL_LIGHT3, GL_SPOT_DIRECTION, Light4Direct);
   glLightf(GL_LIGHT3, GL_SPOT_CUTOFF, 50.0f);

   glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, BrownSpecular);
   glMaterialf(GL_FRONT, GL_SHININESS, 100.0);
}
```

CLS

This function sets up the clearing colour to black, and clears both the colour and the Z-buffer. The Z-buffer is cleared to enable it to be used for hidden surface removal. Z-buffering, and how it works is described in section 4.1.

```
void COpenGLObject::CLS()
{
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

Walls

The Walls function is responsible for drawing the four walls of the control room. The walls are made up of eight sections, which can be seen in the wire frame picture below. By using normal vectors to affect the colours of the walls at the edges I was able to create a smooth looking transition between the walls, and have the room look like the corners have been rounded off.

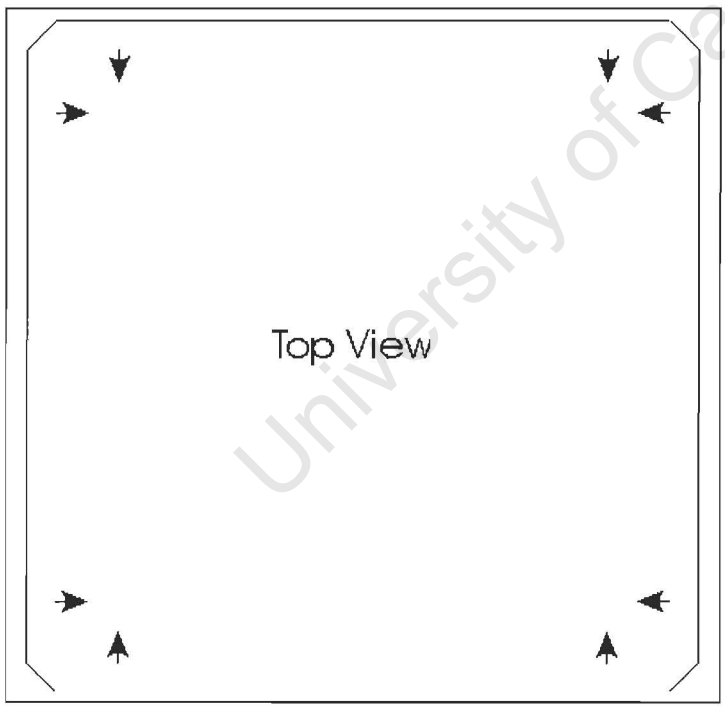


Figure 4.32 : Plan view of normals of the edges of the walls

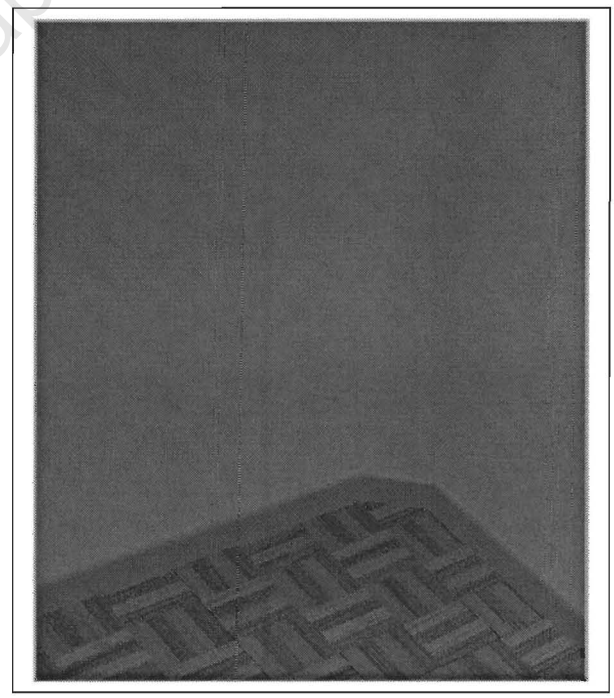


Figure 4.33 : Image of corner in wall

```
void COpenGLObject::Walls()
{
   GLfloat DiffuseNorm[] = {0.8f, 0.8f, 0.8f, 1.0f };

   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

   glBegin(GL_QUAD_STRIP);
      glNormal3f(0.0f, 0.0f, -1.0f);
      glVertex3f(-1.0f, 1.0f, 1.1f);
      glVertex3f(-1.0f, -1.0f, 1.1f);
      glVertex3f(1.0f, 1.0f, 1.1f);
      glVertex3f(1.0f, -1.0f, 1.1f);
      glNormal3f(-1.0f, 0.0f, 0.0f);
      glVertex3f(1.1f, 1.0f, 1.0f);
      glVertex3f(1.1f, -1.0f, 1.0f);
   glEnd();

   glBegin(GL_QUAD_STRIP);
      glNormal3f(-1.0f, 0.0f, 0.0f);
      glVertex3f(1.1f, 1.0f, -1.0f);
      glVertex3f(1.1f, -1.0f, -1.0f);
      glNormal3f(0.0f, 0.0f, 1.0f);
      glVertex3f(1.0f, 1.0f, -1.1f);
      glVertex3f(1.0f, -1.0f, -1.1f);
      glVertex3f(-1.0f, 1.0f, -1.1f);
      glVertex3f(-1.0f, -1.0f, -1.1f);
      glNormal3f(1.0f, 0.0f, 0.0f);
      glVertex3f(-1.1f, 1.0f, -1.0f);
      glVertex3f(-1.1f, -1.0f, -1.0f);
      glVertex3f(-1.1f, 1.0f, 1.0f);
      glVertex3f(-1.1f, -1.0f, 1.0f);
      glNormal3f(0.0f, 0.0f, -1.0f);
      glVertex3f(-1.0f, 1.0f, 1.1f);
      glVertex3f(-1.0f, -1.0f, 1.1f);
   glEnd();

   glBegin(GL_QUADS);
      glNormal3f(-1.0f, 0.0f, 0.0f);
      glVertex3f(1.1f, 0.9f, 0.8f);
      glVertex3f(1.1f, 1.0f, 1.0f);
      glVertex3f(1.1f, 1.0f, -1.0f);
      glVertex3f(1.1f, 0.9f, -0.8f);
      glVertex3f(1.1f, 0.9f, 0.8f);
      glVertex3f(1.1f, -0.2f, 0.8f);
      glVertex3f(1.1f, -1.0f, 1.0f);
      glVertex3f(1.1f, 1.0f, 1.0f);
      glVertex3f(1.1f, 0.9f, -0.8f);
      glVertex3f(1.1f, 1.0f, -1.0f);
      glVertex3f(1.1f, -1.0f, -1.0f);
      glVertex3f(1.1f, -0.2f, -0.8f);
      glVertex3f(1.1f, -1.0f, 1.0f);
      glVertex3f(1.1f, -0.2f, 0.8f);
      glVertex3f(1.1f, -0.2f, -0.8f);
      glVertex3f(1.1f, -1.0f, -1.0f);
   glEnd();

}
```

Skirting

The skirting is a layer that joins the walls and the floor together. This function draws two planes at right angles to each other along the inside of the wall in the control room. The diagrams below show how the normal vectors have been arranged to give the impression of a rounded edge on the skirting.

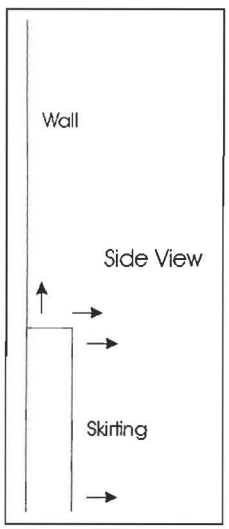


Figure 4.34 : Skirting Design

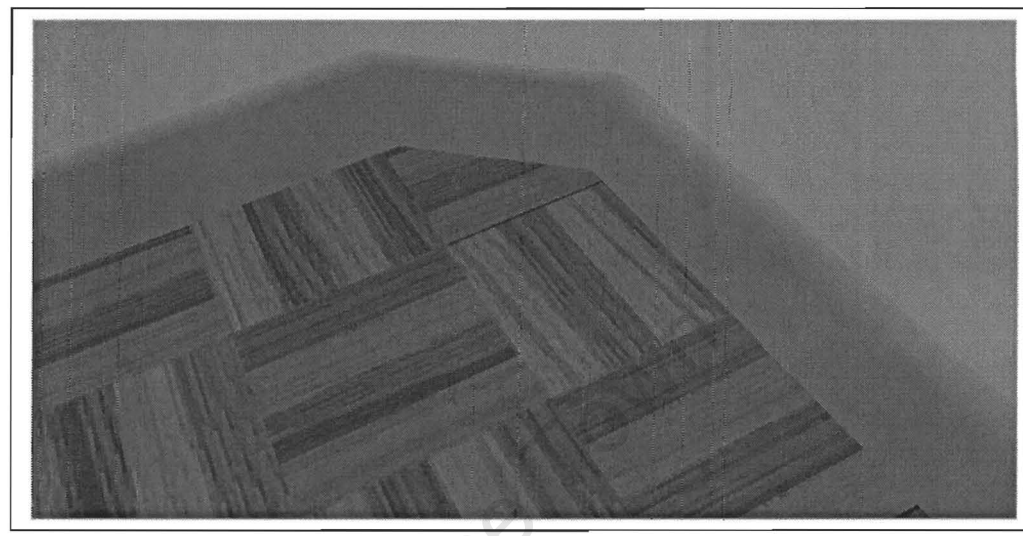


Figure 4.35 : Image of Skirting

```
void COpenGLObject::Skirting()
{
   GLfloat TanDiffuse[] = { 0.7f, 0.4f, 0.1f, 1.0f };

   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , TanDiffuse);

   glBegin(GL_QUAD_STRIP);
      glNormal3f(0.0f, 0.0f, -1.0f);
      glVertex3f(-1.0f, -0.95f, 1.08f);
      glVertex3f(-1.0f, -1.0f, 1.08f);
      glVertex3f(1.0f, -0.95f, 1.08f);
      glVertex3f(1.0f, -1.0f, 1.08f);
      glNormal3f(-1.0f, 0.0f, 0.0f);
      glVertex3f(1.08f, -0.95f, 1.0f);
      glVertex3f(1.08f, -1.0f, 1.0f);
      glVertex3f(1.08f, -0.95f, -1.0f);
      glVertex3f(1.08f, -1.0f, -1.0f);
      glNormal3f(0.0f, 0.0f, 1.0f);
      glVertex3f(1.0f, -0.95f, -1.08f);
      glVertex3f(1.0f, -1.0f, -1.08f);
      glVertex3f(-1.0f, -0.95f, -1.08f);
      glVertex3f(-1.0f, -1.0f, -1.08f);
      glNormal3f(1.0f, 0.0f, 0.0f);
      glVertex3f(-1.08f, -0.95f, -1.0f);
      glVertex3f(-1.08f, -1.0f, -1.0f);
      glVertex3f(-1.08f, -0.95f, 1.0f);
      glVertex3f(-1.08f, -1.0f, 1.0f);
      glNormal3f(0.0f, 0.0f, -1.0f);
      glVertex3f(-1.0f, -0.95f, 1.08f);
      glVertex3f(-1.0f, -1.0f, 1.08f);
   glEnd();
```

```
    glBegin(GL_QUAD_STRIP);
       glNormal3f(0.0f, 0.0f, -1.0f);
       glVertex3f(-1.0f, -0.95f, 1.08f);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(-1.0f, -0.95f, 1.10f);
       glNormal3f(0.0f, 0.0f, -1.0f);
       glVertex3f(1.0f, -0.95f, 1.08f);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(1.0f, -0.95f, 1.10f);
       glNormal3f(-1.0f, 0.0f, 0.0f);
       glVertex3f(1.08f, -0.95f, 1.0f);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(1.10f, -0.95f, 1.0f);
       glNormal3f(-1.0f, 0.0f, 0.0f);
       glVertex3f(1.08f, -0.95f, -1.0f);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(1.10f, -0.95f, -1.0f);
       glNormal3f(0.0f, 0.0f, 1.0f);
       glVertex3f(1.0f, -0.95f, -1.08f);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(1.0f, -0.95f, -1.10f);
       glNormal3f(0.0f, 0.0f, 1.0f);
       glVertex3f(-1.0f, -0.95f, -1.08f);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(-1.0f, -0.95f, -1.10f);
       glNormal3f(1.0f, 0.0f, 0.0f);
       glVertex3f(-1.08f, -0.95f, -1.0f);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(-1.10f, -0.95f, -1.0f);
       glNormal3f(1.0f, 0.0f, 0.0f);
       glVertex3f(-1.08f, -0.95f, 1.0f);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(-1.10f, -0.95f, 1.0f);
       glNormal3f(0.0f, 0.0f, -1.0f);
       glVertex3f(-1.0f, -0.95f, 1.08f);
       glNormal3f(0.0f, 1.0f, 0.0f);
       glVertex3f(-1.0f, -0.95f, 1.10f);
    glEnd();
}
```


DemoMode

The DemoMode function increments the X, Y and Z variables, and rotate the whole virtual environment by these values around their corresponding axis. This generates a rotating flythrough of the environment.

```
void COpenGLObject::DemoMode()
{
   X -= 2.0f;
   Y += 1.0f;
   Z += 0.50f;
   dSky += 0.01f;
   glRotatef(X, 1.0f, 0.0f, 0.0f);
   glRotatef(Y, 0.0f, 1.0f, 0.0f);
   glRotatef(Z, 0.0f, 0.0f, 1.0f);
}
```

FloorCeilingPanel

This function draws the floor and ceiling of the control room and a small panel that forms a windowsill type effect on the window looking out onto the factory. All three of these objects are texture mapped to give them added realism. The layout of the virtual environment can be seen in the diagram below.

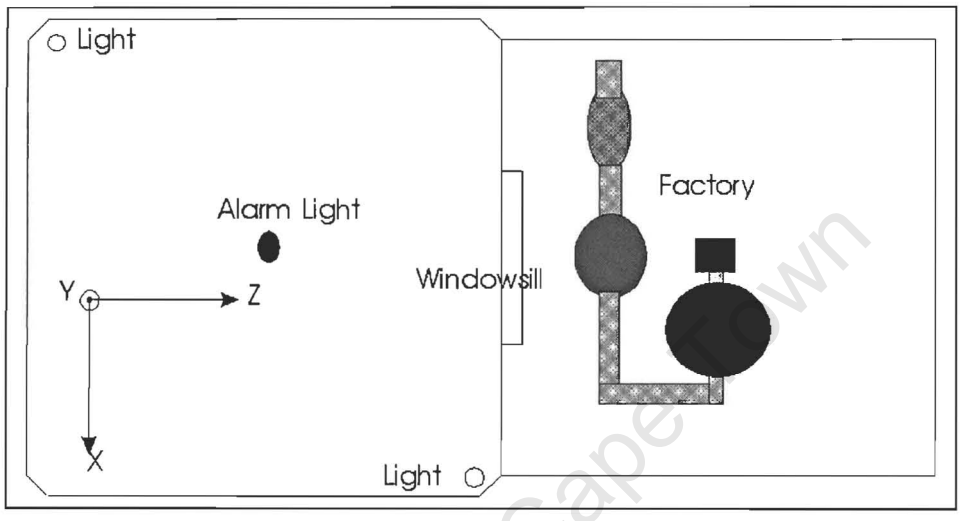


Figure 4.36 : Plan layout of Factory and Control Room floor

The floor and ceiling are both made up of a grid of blocks instead of a single rectangle. The reason for splitting them both into a mesh of squares is because the colour of a textured rectangle is affected by the colour of the underlying rectangle. This is in turn affected by the colour of the vertices. OpenGL only calculates the lighting calculations at the vertices of a rectangle and interpolates the colours over the surface of the plane. If there is only a single large plane, then it is not possible to see the visual affect of a light shining in the middle of the plane, because it would affect each corner equally.

Because the surface colours are interpolated from the corner colours, it is impossible for the centre of a plane to be any darker or any lighter than the darkest / lightest vertex respectively making up the plane. By having a grid of blocks, some will be closer than others to the light's focal point and will thus be brighter than the others. This will give the appearance of light spreading outwards from the focal point. The image below shows this effect occurring.

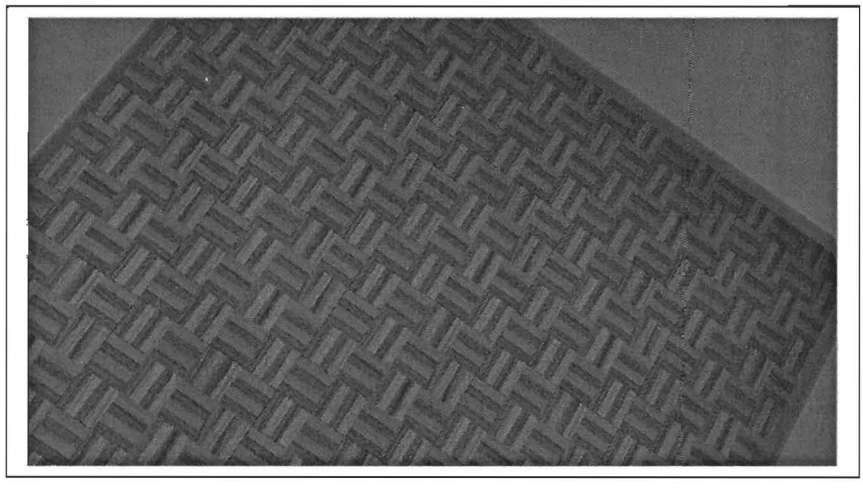


Figure 4.37 : Image showing light intensity variation across floor

```
void COpenGLObject::FloorCeilingPanel()
{
   GLfloat DiffuseNorm[] = {0.8f, 0.8f, 0.8f, 1.0f };

   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

   glEnable(GL_TEXTURE_2D);

   pTex->fpglBindTextureEXT( GL_TEXTURE_2D, pTex->FloorTex);
   for (int i = 0 ; i < 5; i++)
      for (int j = 0 ; j < 5; j++)
      {
         glBegin(GL_QUADS);
            glTexCoord2f(0.0f, 0.0f);
            glVertex3f(-1.1f+0.44f*i, - 1.0f, -1.1f+0.44f*j);
            glTexCoord2f(1.0f, 0.0f);
            glVertex3f(-0.66f+0.44f*i, - 1.0f, -1.1f+0.44f*j);
            glTexCoord2f(1.0f, 1.0f);
            glVertex3f(-0.66f+0.44f*i, - 1.0f, -0.66f+0.44f*j);
            glTexCoord2f(0.0f, 1.0f);
            glVertex3f(-1.1f+0.44f*i, - 1.0f, -0.66f+0.44f*j);
         glEnd();
      }

   pTex->fpglBindTextureEXT( GL_TEXTURE_2D, pTex->WallTex);
   for ( i = 0 ; i < 5; i++)
      for (int j = 0 ; j < 5; j++)
      {
         glBegin(GL_QUADS);
            glNormal3f(0.0f, -1.0f, 0.0f);
            glTexCoord2f(0.0f, 0.0f);
            glVertex3f(-1.1f+0.44f*i, 1.0f, -1.1f+0.44f*j);
            glTexCoord2f(1.0f, 0.0f);
            glVertex3f(-0.66f+0.44f*i, 1.0f, -1.1f+0.44f*j);
            glTexCoord2f(1.0f, 1.0f);
            glVertex3f(-0.66f+0.44f*i, 1.0f, -0.66f+0.44f*j);
            glTexCoord2f(0.0f, 1.0f);
            glVertex3f(-1.1f+0.44f*i, 1.0f, -0.66f+0.44f*j);
         glEnd();
      }

   pTex->fpglBindTextureEXT( GL_TEXTURE_2D, pTex->PanelTex);
   glBegin(GL_QUADS);
      glNormal3f(-0.447f, 0.894f, 0.0f);
      glTexCoord2f(0.0f, 0.0f);
      glVertex3f(1.3f, -0.1f, 0.7f);
      glTexCoord2f(0.0f, 1.0f);
      glVertex3f(1.1f, -0.2f, 0.8f);
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f(1.1f, -0.2f, -0.8f);
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f(1.3f, -0.1f, -0.7f);
   glEnd();

   glDisable(GL_TEXTURE_2D);
}
```

FloorSky

The FloorSky function draws the floor and the sky in the factory area. Both are texture mapped for additional realism. The sky's texture map, as was discussed earlier, has its starting point incremented with each frame. As inside the control room, the floor is made up of a matrix of panels. The two pictures below show the cement like floor and cloud filled sky.

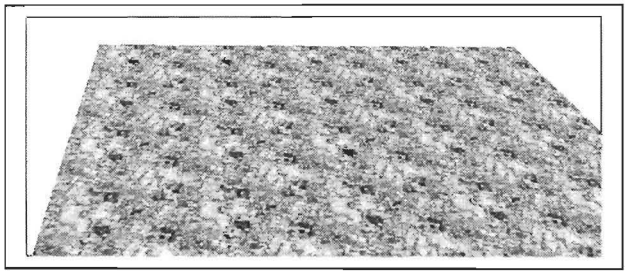


Figure 4.38 : Image of factory floor

Figure 4.39 : Image of Sky

```
void COpenGLObject::FloorSky()
{
   glDisable(GL_LIGHTING);
   glEnable(GL_TEXTURE_2D);

   pTex->fpglBindTextureEXT( GL_TEXTURE_2D, pTex->Floor2Tex);
   for (int i = 0 ; i < 5; i++)
      for (int j = 0 ; j < 5; j++)
      {
         glBegin(GL_QUADS);
            glTexCoord2f(0.0f, 0.0f);
            glVertex3f(1.1f+2.0f/5.0f*i, -1.0f, -1.0f+2.0f/5.0f*j);
            glTexCoord2f(0.0f, 1.0f);
            glVertex3f(1.1f+2.0f/5.0f*i, -1.0f, -0.6f+2.0f/5.0f*j);
            glTexCoord2f(1.0f, 1.0f);
            glVertex3f(1.5f+2.0f/5.0f*i, -1.0f, -0.6f+2.0f/5.0f*j);
            glTexCoord2f(1.0f, 0.0f);
            glVertex3f(1.5f+2.0f/5.0f*i, -1.0f, -1.0f+2.0f/5.0f*j);
         glEnd();
      }

   pTex->fpglBindTextureEXT( GL_TEXTURE_2D, pTex->SkyTex);
   glBegin(GL_QUADS);
      glColor3f(1.0f, 1.0f, 1.0f);
      glTexCoord2f(0.0f - dSky / 3, 0.0f + dSky);
      glVertex3f(0.0f, 4.0f, -30.0f);
      glTexCoord2f(0.0f - dSky / 3, 2.0f + dSky);
      glVertex3f(0.0f, 4.0f, 30.0f);

      glColor3f(0.0f, 0.0f, 0.0f);
      glTexCoord2f(1.0f - dSky / 3, 2.0f + dSky);
      glVertex3f(30.0f, 4.0f, 30.0f);
      glTexCoord2f(1.0f - dSky / 3, 0.0f + dSky);
      glVertex3f(30.0f, 4.0f, -30.0f);
   glEnd();

   glEnable(GL_LIGHTING);
   glDisable(GL_TEXTURE_2D);
}
```

Action

The Action function is used when the user is not in demo mode. Because in normal mode we still want the clouds to move, this function, every time it is called, increments the dSky variable. The dSky variable in turn effects the position from which the sky texture map starts, creating the illusion of clouds in motion.

```
void COpenGLObject::Action()
{
   dSky += 0.01f;
}
```

SetupPos

Although a member function of this class, this function is discussed later in section 4.4 under the discussion of, 'incorporating motion into the virtual environment'. This ends the discussion of the COpenGLObject Class, leaving the CControlRoom Class as the next sub-section that will show how all these functions are linked together.

4.2.3 CControlRoom Class

The CControlRoom Class forms the final class of this section on 'Drawing the Room'. This class operates by incorporating the functionality provided by the CBlocks and COpenGLObject classes into a more unified virtual environment. It takes the building blocks provided by the above two classes and builds a virtual environment with them.

```
class CControlRoom : public CObject
{
public:
   void ControlLoop(void);
   CBlocks Blk;
   void DrawHand(void);
   BOOL Running;
   void OnDestroy(void);
   void DrawControlRoom(void);
   CControlRoom();
   virtual ~CControlRoom();
   void InitEnv(void);
   GLfloat  Sens1[6];
   GLfloat  Sens2[6];
   CTrack Trk;
   CHandGL Hand;

protected:
   GLfloat State[10];
   GLfloat State2[10];
   GLfloat State3[10];
   CPLC Plc, Plc2, Plc3;
   COpenGLObject Obj;
   CTextures Tex;
   void DrawRoom(void);
   void DrawFactory(void);
};
```


This class has numerous member functions and member variables, having 9 and 14 of each respectively. Starting with the constructor and destructor, these are responsible for setting up and removing any bits of data that need to be dealt with at object creation and destruction time respectively.

The ControlLoop function is responsible for implementing the control related environment where the sensors react in some defined manner to the values of the actuators, and whose results are controlled and displayed by the virtual Programmable Logic Controllers (PLCs).

DrawFactory and DrawRoom draw the two sections of the virtual environment. These two functions are used internally (they are thus declared protected) by the function DrawControlRoom. DrawHand is responsible for drawing the user's virtual hand in the environment to enable interaction.

The last two member functions are OnDestroy, and InitEnv. InitEnv is called after the class is created, but before it is used for the first time. This is so that a second stage initialisation process is possible. OnDestroy forms the cleaning up counterpart of InitEnv.

Moving onto the protected member variables, Tex is an object of class CTexture, and Obj is an object of class COpenGLObject. Both are present to enable the CControlRoom class to use the functionality provided by these two classes.

Plc, Plc2 and Plc3 are three instances of the CPLC class, which will be discussed in the last section of this chapter. Three instances of this one class are required because there are three independent virtual control loops within the virtual environment that control how the elements in the factory interact. State, State2 and State3 are three arrays that assist in the functioning of the PLC objects. They keep a record of the past inputs into each system. They thus allow for the creation of multiple order control loops with delays.

Amongst the public member variables, Blk, Hand and Trk are object instances of the classes CBlocks, CHandGL and CTrack respectively. These instances enable the CControlRoom to use the functionality provided by these classes in the form of member functions and member variables to perform its operations.

Sens1 and Sens2 are two arrays that store the current positions and orientations of the two tracking system receivers with respect to the transmitter. Finally, the Running boolean is there to indicate to the two background threads setup by this class when to stop execution.

CControlRoom, ~ CControlRoom

The destructor is empty, whereas the constructor is responsible for a number of tasks. Firstly it sets up both receivers of the tracking systems to be activated and start acquiring data. It zeros the state variables, to make sure that all the variables were zero before the program was started. It then sets the running variable to TRUE so that the threads know to continue execution, and finally, it creates the two separate threads. The first is the tracking system thread, and the second is the data glove thread.

These threads get the data from their corresponding bits of hardware. By using threads, if the thread needs to wait for the hardware, the thread just suspends, and the main thread can carry on processing. This enables no processor time to have to be wasted while waiting for the hardware to respond.

```
CControlRoom::CControlRoom()
{
   CString Result = Trk.SetActiveStation(TRUE, TRUE);
   if (Result.GetLength() != 0) ::MessageBox(NULL, (LPCTSTR) Result, 0, 0);

   for (int i = 0; i < 10; i++)
   {
      State[i] = 0.0f;
      State2[i] = 0.0f;
      State3[i] = 0.0f;
   }

   Running = TRUE;
   AfxBeginThread(GetTracker, this, THREAD_PRIORITY_NORMAL);
   AfxBeginThread(GetFingers, this, THREAD_PRIORITY_NORMAL);
}

CControlRoom::~CControlRoom()
{
}
```

InitEnv

This function is called after the constructor, and after the view class has initialised a display area for OpenGL, but before any OpenGL graphics are drawn. It starts off by calling the OnCreate function of the CTexture class. This loads up all the textures. Next it passes a pointer to the CTexture object to the instances of COpenGLObject, CPLC and CBlocks, all of which need to use textures and thus also the functionality provided from within the CTexture object.

Next Z-buffering is enabled to simplify hidden surface removal. 3D smooth shading and the lighting calculations are also enabled. The above three enables, make the image appear more realistic, but also make the refresh rate slower. The front face is defined as the face around which the vertices are created in a counter clockwise direction. The diagram below should explain how a front and back face is determined.

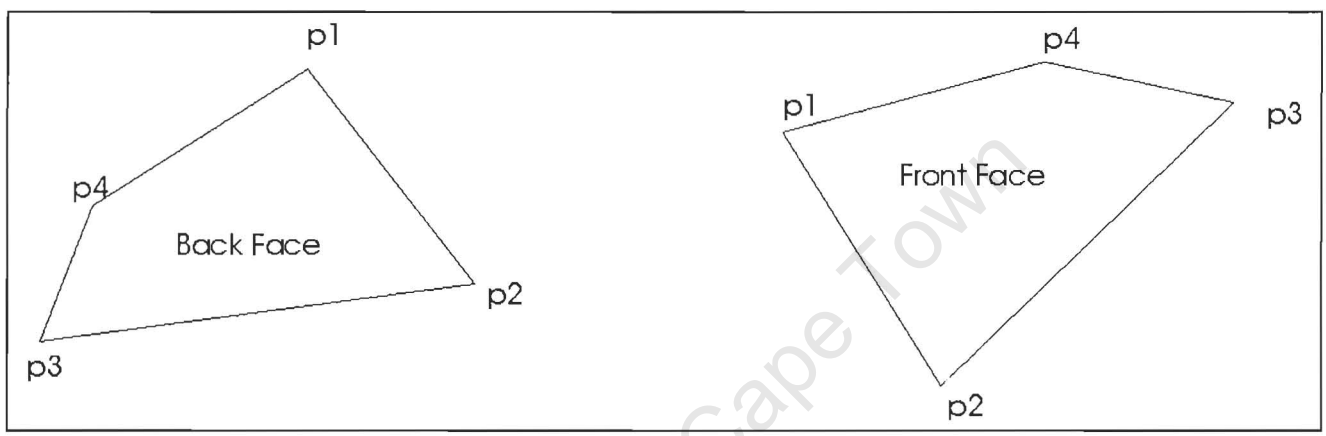


Figure 4.40 : Vertex order used to decide which side is front or back facing

```
void CControlRoom::InitEnv()
{
   Tex.OnCreate();
   Obj.pTex = &Tex;
   Plc.pTex = &Tex;
   Plc2.pTex = &Tex;
   Plc3.pTex = &Tex;
   Blk.pTex = &Tex;

   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_SMOOTH);
   glEnable(GL_LIGHTING);

   glFrontFace(GL_CCW);
   glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glEnable(GL_BLEND);
   glEnable(GL_STENCIL_TEST);
   glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
}
```

In the last five lines I set the texture environment to GL_MODULATE which means that the texture maps are combined with the lighting calculations. I then set the blend function to enable the textures to be transparent in areas where the Alpha value of the RGBA quadruplet is less than one, and then enable the blending. I end off by enabling the stencil test.

OnDestroy

This function first clears the boolean variable 'Running' so that the two threads collecting the data from the hardware know that they can stop doing so, and then calls the OnDestroy function in the object of class CTexture which removes the texture maps from the graphics card's memory.

```
void CControlRoom::OnDestroy()
{
   Running = FALSE;
   Tex.OnDestroy();
}
```

DrawControlRoom

This is the central function which pulls everything together to draw the virtual environment. The variable frm maintains the number of the current frame being drawn. This variable is used to ascertain whether the frame is an even or an odd frame. The stencil buffer is a single bit deep buffer that is set up with even lines equal to zero and odd lines equal to one. The stencil function is set such that on even pages only the pixels where the stencil buffer is equal to zero are drawn, and on odd pages only the pixels where the stencil buffer is set to one are drawn.

With the stencil buffer set up thus, the even pages are drawn in the even lines and the odd pages are drawn in the odd lines. A line sequential stereo-graphic image is generated in this way. The display area is cleared only on even frames, as both an even frame and an odd frame need to be drawn before the image is displayed. Once the odd frame is drawn, the back buffer is swapped and the image displayed. At the start of the next even frame, the back buffer is cleared for the next pair of images.

The next set of commands is placed between the glPushMatrix and glPopMatrix function calls. These functions return the state of the OpenGL state machine back to its original state upon completion of this routine. This is done so that all the initialisation information stays but any modifications to the state done in drawing the scene does not remain.

Depending on the value of the Demo boolean variable, either the COpenGLObject's DemoMode function or its Action function is called. The viewing position and direction are next to be set up with a call to the SetupPos function, using the position and orientation values from the first receiver. The DrawHand function draws the operator's virtual hand based on information from the second tracking receiver and from the data from the data glove.

Finally the light sources are set to shine at the correct parts of the virtual environment. With calls to DrawFactory and DrawRoom these two parts of the virtual environment are drawn. The SwapBuffers function swaps the front and back buffer so that the currently drawn scene can now be displayed, but this is done only once the odd frame has been drawn.

```
static int frm = 0;
BOOL Demo = FALSE;

if ((frm % 2) == 0) glStencilFunc(GL_EQUAL, 0, 1);
else glStencilFunc(GL_EQUAL, 1, 1);

if (frm == 0)
{
   frm++;

   ::glClear(GL_STENCIL_BUFFER_BIT);
   for (int i = 0; i < 480; i++)
   {
      if ( (i%2) == 0 ) for (int j = 0; j < 640; j++) *(p+i*640+j) = 0;
      else for (int j = 0; j < 640; j++) *(p+i*640+j) = 255;
   }
   ::glDrawPixels(640, 480, GL_STENCIL_INDEX, GL_UNSIGNED_BYTE, p);
}

if ( (frm%2) == 0) Obj.CLS();

glPushMatrix();
   if (Demo) Obj.DemoMode();
   else Obj.Action();

   Obj.SetupPos(&Sens1[0], frm);
   DrawHand();

   Obj.SetupLighting();
   ViewSpecial();
   DrawFactory();
   DrawRoom();

glPopMatrix();

if ( (frm%2) == 1) SwapBuffers(wglGetCurrentDC());

frm++;
```


DrawFactory

This function is responsible for drawing the control room. It starts off by switching on light 2 (sun light). This will be the only light source at the moment, as the other three lights, which are inside the control room, should have no affect on the outside factory.

The Floor and Sky are drawn next, followed by which are the components making up the factory, namely a pump, a temperature thermometer, a flow meter, a flow off valve, a heater, a water tank, three knee joints, and five pipes. The heater is drawn last because the flames of the fire have been made transparent. Thus to be able to see the scene behind the flames, everything else needs to have been drawn already.

```
void CControlRoom::DrawFactory()
{
   GLfloat DiffuseNorm[] = {1.0f, 1.0f, 1.0f, 1.0f };

   glEnable(GL_LIGHT2);
   Obj.FloorSky();

   Blk.Pump(3, Plc2.SumActPoint, 2.0f, 0.1f, -1.1f);

   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);
   Blk.Pipe(3, 2.0f, 0.3f, 0.1f, 1.2f, 0.1f);

   Blk.TempMeter(Plc.OutPoint, 2.0f, 0.3f, 0.3f);
   Blk.FlowMeter(3, Plc.OutPoint, Plc2.OutPoint, 2.0f, 0.4f, 0.5f);

   Blk.Knee(3, 2, 1, 1, 2.0f, 0.3f, 1.3f, 0.1f);
   Blk.Pipe(2, 2.0f, 0.3f, 1.3f, 0.4f, 0.1f);
   Blk.Knee(2, 1, 1, 1, 2.0f, 0.7f, 1.3f, 0.1f);
   Blk.Pipe(1, 2.0f, 0.7f, 1.3f, 0.6f, 0.1f);
   Blk.Knee(1, 3, 1, -1, 2.6f, 0.7f, 1.3f, 0.1f);
   Blk.Pipe(3, 2.6f, 0.7f, 1.3f, -0.4f, 0.1f);
   Blk.Pipe(3, 2.6f, -0.1f, 0.4f, -0.3f, 0.1f);

   Blk.SlushGate(Plc3.ActPoint, Plc.OutPoint, 2.6f, -0.1f, 0.1f);

   Blk.Tank(Plc3.OutPoint, Plc.OutPoint, 2.6f, -0.1f, 0.7f);

// Last : Flame Transparent
   Blk.Heater(Plc.ActPoint, 2.0f, 0.3f, -0.50f);

   glDisable(GL_LIGHT2);
}
```

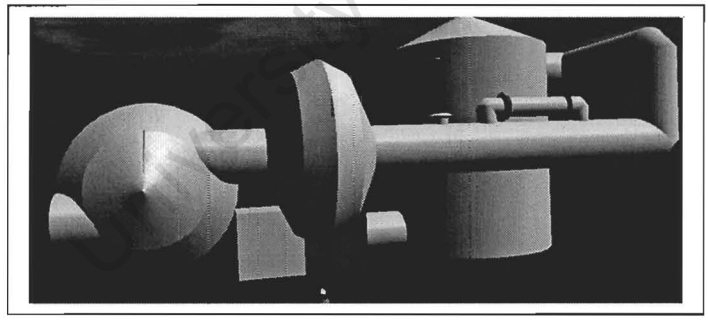


Figure 4.41 : Image of the factory

DrawRoom

Here lights 0 and 1, the two normal lights inside the control room, are enabled. Next the light fittings, the skirting, the floor, the ceiling, the walls, and the windowsill panel are drawn. The three sections that follow thereafter draw the three PLCs.

The PLCs are drawn by first writing their name above them, then there is a check to see if the user is making contact with any of the buttons. If contact is made, the program can show an acknowledgement of the user's interaction while drawing the virtual PLC. Finally the two light sources used herein are disabled, as they will not be needed outside of this function.

```
void CControlRoom::DrawRoom()
{
  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHT1);

  Obj.Lights();
  Obj.Skirting();
  Obj.FloorCeilingPanel();
  Obj.Walls();

  Plc.SetPos(-0.8f, 0.0f, -1.098f, "Temperature");
  Plc.CheckContact(this, &Sens2[0]);
  Plc.DrawPLC();

  Plc2.SetPos(-0.3f, 0.0f, -1.098f, "Flow Rate");
  Plc2.CheckContact(this, &Sens2[0]);
  Plc2.DrawPLC();

  Plc3.SetPos(0.2f, 0.0f, -1.098f, "Level");
  Plc3.CheckContact(this, &Sens2[0]);
  Plc3.DrawPLC();

  glDisable(GL_LIGHT1);
  glDisable(GL_LIGHT0);
}
```

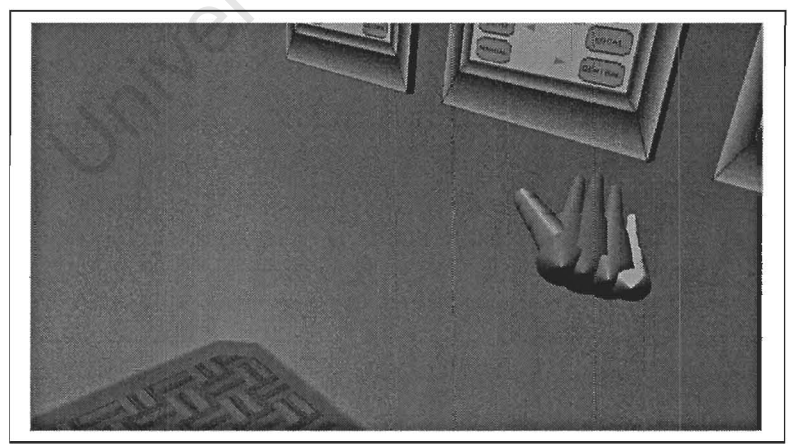


Figure 4.42 : Image of the control room

This ends off the discussion of the CControlRoom class. The two remaining functions, DrawHand and ControlLoop will be discussed under their respective sections of this chapter, namely '4.3 Drawing the Hand' and '4.5 PLC controls' respectively.

4.3 Drawing the Hand

This section follows a discussion on how the virtual hand is drawn. The virtual hand forms the only method of interaction that the user has with his / her virtual environment. The hand is drawn based on two sets of data. The first set of data will be from the data glove that will provide the finger flexures. The second set of data will come from the second receiver of the tracking system. Together these will be able to give the position and orientation of the data glove and the amount of bending taking place by each finger.

There are only two sub-sections in this section. The first discusses the CHandGL class, which is responsible for drawing the visual parts of the hand. The second sub-section will discuss the two other functions that form part of drawing the virtual hand, but that are included in other classes due to the resources they require.

4.3.1 CHandGL Class

This class is responsible for drawing the virtual hand. It does so with the use of six member function and five member variables. Starting with the member variables there are SinX and CosX, both of which are arrays that constitute sine lookup tables to help increase the rendering speed of the hand. Next is AngData, which is an array that stores the 8-bit values of how much each finger has been bent.

'glv' is a object instance of the class CGlove which was discussed in chapter 3. This instance is used to retrieve the data from the data glove using the member functions provided by the CGlove class. The final member variable is dat, which is of type STATESTRUCT. This variable is used to store the current state of the drawing in progress.

```
typedef struct StateStruct
{
   GLfloat StartX;
   GLfloat StartY;
   GLfloat StartZ;
   GLfloat EndX;
   GLfloat EndY;
   GLfloat EndZ;
   GLfloat R;
   GLfloat G;
   GLfloat B;
   GLfloat A;
   int Sign;
   int Axis;
   int SignOut;
   int AxisOut;
   GLfloat Width;
   GLfloat WidthDest;
   GLfloat Length;
} STATESTRUCT;
```

How this class draws the hand is by separating it into joints, with each joint being drawn separately. In the declaration above, the STATESTRUCT structure contains variables to store all the necessary information required to carry on the next joint from the previous one. As each joint is drawn it updates the dat variable to indicate the new starting and ending co-ordinates. Other important variables, for example the current width and length are also updated.

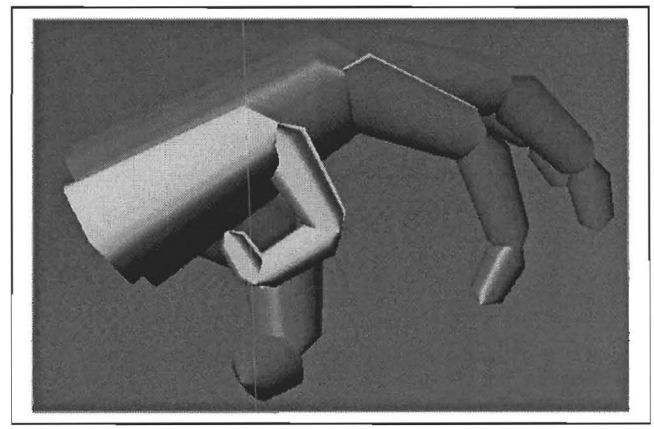


Figure 4.43 : Image 1 of virtual glove

```
class CHandGL : public CObject
{
public:
   void DrawHand();
   void GetAng();
   CHandGL();
   virtual ~CHandGL();

protected:
   CGlove glv;
   GLfloat AngData[7];
   void Finger(GLfloat sz, GLfloat ang, BOOL Thumb);
   STATESTRUCT dat;
   STATESTRUCT SzConvert(STATESTRUCT dat);
   GLfloat SinX[31];
   GLfloat CosX[31];

};
```

Moving onto the member functions, the constructor and destructor that do the initialisation and cleanup of the class respectively. The SzConvert is a function similar in functionality to the ShrinkPipe function described in the CBlocks class. One difference is that it uses the data in the STATESTRUCT variable passed to it. It then updates this data to point to where the pipe ends, so that the new joint knows where to start.

The Finger function uses the SzConvert to draw a whole finder made up of joints, using the angle data for that finger which is retrieved with the GetAng function. Finally the DrawHand function puts all of these together by drawing five fingers.

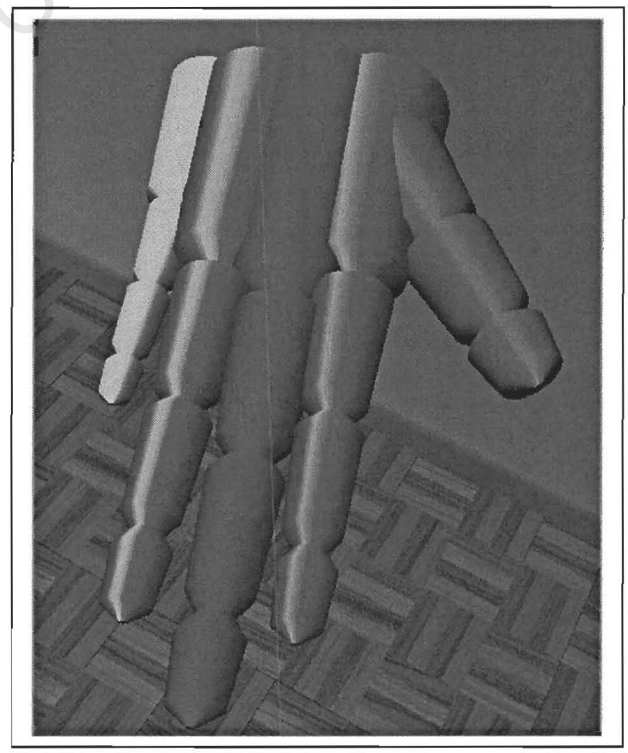


Figure 4.44 : Image 2 of virtual glove

CHandGL, ~ CHandGL

The destructor is empty, while the constructor does some initialisation work. It starts off by setting up the two trigonometric tables that will be used in drawing the fingers. Then it zeros the angular data so that it has a known default state before any data arrives from the hardware. The last part opens the connection to the data glove, sends a reset request to it, and receives a reset confirmation from the glove. This puts the data glove into a known state.

```
CHandGL::CHandGL()
{
   double PI = 3.14159265358979323846;

   for (int i = 0 ; i < 30; i++)
   {
      CosX[i] = (GLfloat)cos((double)i/15.0f*PI);
      SinX[i] = (GLfloat)sin((double)i/15.0f*PI);
   }
   SinX[30] = SinX[0];
   CosX[30] = CosX[0];

   for ( i = 0 ; i < 7; i++) AngData[i] = 0;

   CString Result;
   char  Wr[10];
   char  R;

   Result = glv.Open();
   if (Result.GetLength() != 0) ::MessageBox(NULL, (LPCTSTR) Result, 0, 0);

   Wr[0] = 0x41;
   Result = glv.Write(Wr, 1);
   if (Result.GetLength() != 0) ::MessageBox(NULL, (LPCTSTR) Result, 0, 0);

   Result = glv.Read(&R, 1);
   if (Result.GetLength() != 0) ::MessageBox(NULL, (LPCTSTR) Result, 0, 0);
}

CHandGL::~CHandGL()
{
}
```

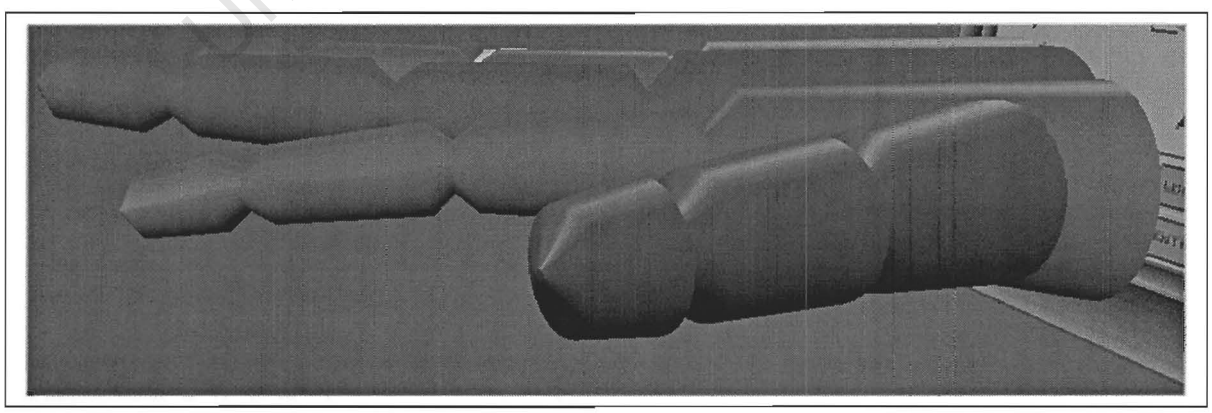


Figure 4.45 : Image 4 of virtual glove

SzConvert

This function is almost identical to the ShrinkPipe function that was explained in the CBlocks class. It however updates the STATESTRUCT variable it is provided with and then returns it once it has completed drawing the object.

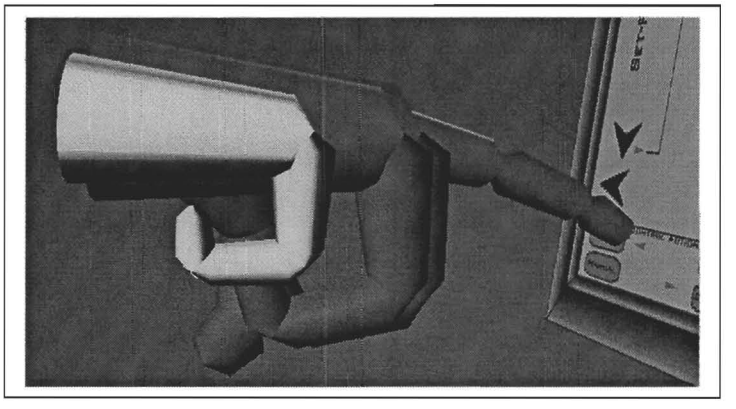


Figure 4.46 : Image 4 of virtual glove

```
STATESTRUCT CHandGL::SzConvert(STATESTRUCT dat)
{
   glBegin(GL_QUAD_STRIP);
      for (int i = 0 ; i < 31 ; i++)
      {
         switch (dat.Axis)
         {
         case 1 :
            glNormal3f(0.0f, SinX[i], CosX[i]);
            glVertex3f(dat.StartX, dat.StartY + SinX[i] * dat.Width,
                                 dat.StartZ + CosX[i] * dat.Width);
            glVertex3f(dat.StartX + dat.Length, dat.StartY + SinX[i] *
                     dat.WidthDest, dat.StartZ + CosX[i] * dat.WidthDest);
            if (i == 30) dat.StartX += dat.Length;
            break;
         case 2:
            glNormal3f(SinX[i], 0.0f, CosX[i]);
            glVertex3f(dat.StartX + SinX[i] * dat.Width, dat.StartY,
                                 dat.StartZ + CosX[i] * dat.Width);
            glVertex3f(dat.StartX + SinX[i] * dat.WidthDest, dat.StartY +
                     dat.Length, dat.StartZ + CosX[i] * dat.WidthDest);
            if (i == 30) dat.StartY += dat.Length;
            break;
         case 3:
            glNormal3f(CosX[i], SinX[i], 0.0f);
            glVertex3f(dat.StartX + CosX[i] * dat.Width,
                     dat.StartY + SinX[i] * dat.Width, dat.StartZ);
            glVertex3f(dat.StartX + CosX[i] * dat.WidthDest, dat.StartY +
                     SinX[i] * dat.WidthDest, dat.StartZ + dat.Length );
            if (i == 30) dat.StartZ += dat.Length;
            break;
         default :
            break;
         }
      }
   glEnd();
   dat.Width = dat.WidthDest;
   return dat;
}
```

GetAng

This function is responsible for getting the angular finger bend data from the data glove hardware. It starts off by writing a request to the hardware to send some data, then it waits to receive the output from the data glove hardware. Once it has the data, it removes the header and stores the data as angles in degrees by converting it from an 8-bit value to an angle with a range of 0 to 90 degrees. 90 degrees will be the largest angle possible between two joints in each finger.

```
void CHandGL::GetAng()
{
   CString Result;
   char  Wr[10];
   UCHAR Fingers[9];

   Wr[0] = 0x43;
   Result = glv.Write(Wr, 1);
   if (Result.GetLength() != 0) ::MessageBox(NULL, (LPCTSTR) Result, 0, 0);

   Result = glv.Read((char *)(&Fingers[0]), 9);
   if (Result.GetLength() != 0) ::MessageBox(NULL, (LPCTSTR) Result, 0, 0);

   for (int i = 0 ; i < 5; i++) AngData[i] = (GLfloat)Fingers[1+i] /
                                                    255.0f * 90.0f;
   AngData[5] = -((GLfloat)Fingers[6] - 128.0f)/128.0f*90.0f;
   AngData[6] = ((GLfloat)Fingers[7] - 128.0f)/128.0f*90.0f;
}
```

Finger

The Finger function builds up the finger from many pipes that form the joints of the finger. The wire frame sketch below presents a side view of a single unbent finger. It gives a rough idea of how the finger is built up.

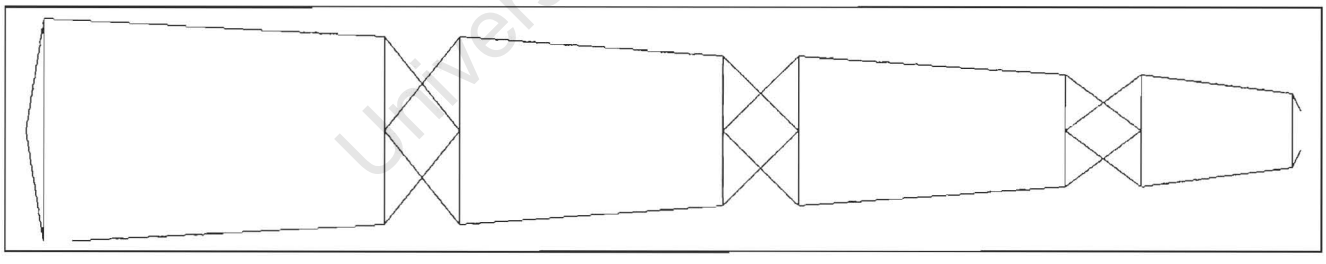


Figure 4.47 : Design of a finger of the virtual glove

The variable sz that is passed to the function can change the size of the finger, to enable the drawing of different sized fingers. The first joint is optional, and can be selected using the Thumb boolean variable. The motivation behind this is to enable the use of this function for all five fingers, where the thumb has one less metacarpal.

After each segment is drawn, this is a rotation around the Z-axis (axis coming out of page in above diagram). This rotation needs to be preceded and followed by a translation so that the point of rotation passes through the centre of the joint between the two fingers.

```
void CHandGL::Finger(GLfloat sz, GLfloat ang, BOOL Thumb)
{
  glPushMatrix();

  dat.StartX = 0.0f;
  dat.StartY = 0.0f;
  dat.StartZ = 0.0f;
  dat.Axis = 1;
  dat.AxisOut = 2;
  dat.Sign = -1;
  dat.SignOut = 1;
  GLfloat f;

  if (!Thumb)
  {
     f = 0.8f;
     dat.Length = 0.005f;
     dat.Width = 0.001f;
     dat.WidthDest = 0.03f*f;
     glTranslatef(0.005f, 0.0f, 0.0f);
     SzConvert(dat);

     dat.Length = 0.08f;
     dat.Width = 0.03f*f;
     dat.WidthDest = 0.020f*f;
     glTranslatef(0.005f, 0.0f, 0.0f);
     SzConvert(dat);

     dat.Length = 0.01f;
     dat.Width = 0.02f*f;
     dat.WidthDest = 0.001f;
     glTranslatef(0.08f, 0.0f, 0.0f);
     SzConvert(dat);

     dat.Length = 0.01f;
     dat.Width = 0.001f;
     dat.WidthDest = 0.016f*f;
     glTranslatef(0.005f, 0.0f, 0.0f);
     glRotatef(-ang, 0.0f, 0.0f, 1.0f);
     glTranslatef(-0.005f, 0.0f, 0.0f);
     SzConvert(dat);
  }
  else
  {
     f = 1.3f;
     dat.Length = 0.005f;
     dat.Width = 0.001f;
     dat.WidthDest = 0.016f*f;
     glRotatef(-ang, 0.0f, 0.0f, 1.0f);
     SzConvert(dat);
     glTranslatef(-0.005f, 0.0f, 0.0f);
  }
```

```
    dat.Length = 0.04f;
    dat.Width = 0.016f*f;
    dat.WidthDest = 0.012f*f;
    glTranslatef(0.01f, 0.0f, 0.0f);
    SzConvert(dat);

    dat.Length = 0.01f;
    dat.Width = 0.012f*f;
    dat.WidthDest = 0.001f;
    glTranslatef(0.04f, 0.0f, 0.0f);
    SzConvert(dat);

    dat.Length = 0.01f;
    dat.Width = 0.001f;
    dat.WidthDest = 0.012f*f;
    glTranslatef(0.005f, 0.0f, 0.0f);
    glRotatef(-ang, 0.0f, 0.0f, 1.0f);
    glTranslatef(-0.005f, 0.0f, 0.0f);
    SzConvert(dat);

    dat.Length = 0.03f;
    dat.Width = 0.012f*f;
    dat.WidthDest = 0.01f*f;
    glTranslatef(0.01f, 0.0f, 0.0f);
    SzConvert(dat);

    dat.Length = 0.01f;
    dat.Width = 0.01f*f;
    dat.WidthDest = 0.001f;
    glTranslatef(0.03f, 0.0f, 0.0f);
    SzConvert(dat);

    dat.Length = 0.01f;
    dat.Width = 0.001f;
    dat.WidthDest = 0.01f*f;
    glTranslatef(0.005f, 0.0f, 0.0f);
    glRotatef(-ang, 0.0f, 0.0f, 1.0f);
    glTranslatef(-0.005f, 0.0f, 0.0f);
    SzConvert(dat);

    dat.Length = 0.015f;
    dat.Width = 0.01f*f;
    dat.WidthDest = 0.008f*f;
    glTranslatef(0.01f, 0.0f, 0.0f);
    SzConvert(dat);

    dat.Length = 0.004f;
    dat.Width = 0.008f*f;
    dat.WidthDest = 0.001f;
    glTranslatef(0.015f, 0.0f, 0.0f);
    SzConvert(dat);

    glPopMatrix();
}
```

DrawHand

As the last function in the class, it uses the finger function to draw the four fingers and the thumb. It uses the angular data in the AngData array to decide the angle between metacarpals and also changes the scale of the drawing so that each finger appear a different sizes. The translation between each finger is to draw each finger at a different offset (i.e. next to each other) and the initial translation is there to account for the receiver, from the tracking system, being at the base of the wrist on dorsal part of the hand.

```
void CHandGL::DrawHand()
{
  glPushMatrix();
  glTranslatef(-0.03f, 0.03f, -0.02f);
  Finger(1.0f, AngData[1],FALSE);

  glTranslatef(0.0f, 0.0f, 0.02f);
  glScalef(1.1f, 1.1f, 1.1f);
  Finger(1.0f, AngData[2],FALSE);

  glTranslatef(0.0f, 0.0f, 0.02f);
  glScalef(0.9f, 0.9f, 0.9f);
  Finger(1.0f, AngData[3],FALSE);

  glTranslatef(0.0f, 0.0f, 0.02f);
  glScalef(0.7f, 0.7f, 0.7f);
  Finger(1.0f, AngData[4],FALSE);
  glPopMatrix();

  glPushMatrix();
  glTranslatef(0.00f, 0.03f, -0.02f);
  glRotatef(30.0f, 0.0f, 1.0f, 0.0f);
  Finger(0.5f, AngData[0],TRUE);
  glPopMatrix();
}
```

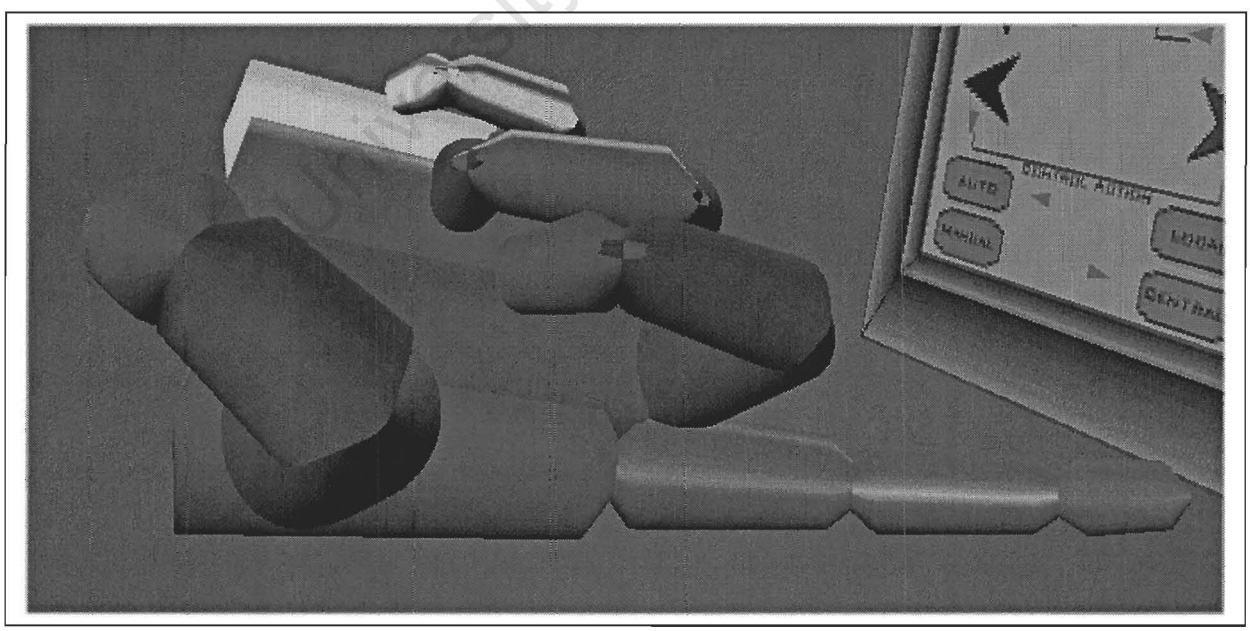


Figure 4.48 : Image 5 of virtual glove

4.3.2 Helper functions

There are two helper functions that assist in the drawing of the hand but are not part of the CHandGL class. These are the GetFingers function, and the DrawHand function.

GetFingers

This function does not belong to any class, but is a global function that forms part of the application. The reason for its definition is that it is used to create an independent thread of execution that continuously cycles round getting the data from the data glove hardware. The reason for a need for an independent thread is to prevent any CPU time being wasted due to waiting on the data glove to respond to requests. This way, if the thread has to wait for the data glove to respond, the Windows NT kernel just switches it out, and another thread can carry on processing, (i.e. drawing the scene).

As was shown earlier when the threads were created, it was passed a pointer to the object that created them, namely an object of class CControlRoom. This pointer is received in the param variable and is properly type cast to its correct form. Now the thread can access the resources of the CControlRoom class. Now it waits in a loop, waiting for the Running boolean variable to go false. Once this happens, the thread knows that it can terminate.

Inside the loop it gets the finger angles from the data glove every 20ms using the GetAng member function of the CHandGL class. This class can be accessed through the object of its type that is a member variable of the CControlRoom class. Although the data glove is able to provide up to 200 updates a second, there is no reason for such a high update rate to be used. Because the frame rate is lower, there is no need to waste processing time getting data that will never be used.

```
UINT GetFingers(LPVOID param)
{
   CControlRoom * Vw;
   Vw = (CControlRoom *)param;
   CString Result;

   while (Vw->Running)
   {
      Vw->Hand.GetAng();
      Sleep(20);
   }

   return 0;
}
```

DrawHand

The DrawHand function, a member function of the CControlRoom class, combines the drawing of the hand using angular data got from the data glove, with the position and orientation data that was collected from the second receiver of the positioning system. Not only must the fingers move, but also the whole hand must move and change orientation with respect to the transmitter and the other receiver.

All these statements are placed within a pair of glPushMatrix and glPopMatrix commands which ensure that any translations or rotations do not affect anything that needs to be drawn after the hand is drawn. Next there is a translation of the virtual environment origin to the negative of the position of the hand (i.e. anything drawn at the new origin will be equivalent to being drawn at the position of the second receiver). Finally I rotate around each axis so that the orientation that the virtual hand will be drawn at, will be the same as the orientation of the second receiver of the tracking system. The function ends by drawing the hand using the DrawHand member function of CHandGL.

```
void CControlRoom::DrawHand()
{
  glPushMatrix();
    glTranslatef(-Sens2[0], -Sens2[1], -Sens2[2]);
    glRotatef(90.0f-Sens2[4], 0.0f, 1.0f, 0.0f);
    glRotatef(Sens2[5], 1.0f, 0.0f, 0.0f);
    glRotatef(-Sens2[3], 0.0f, 0.0f, 1.0f);
  Hand.DrawHand();
  glPopMatrix();
}
```

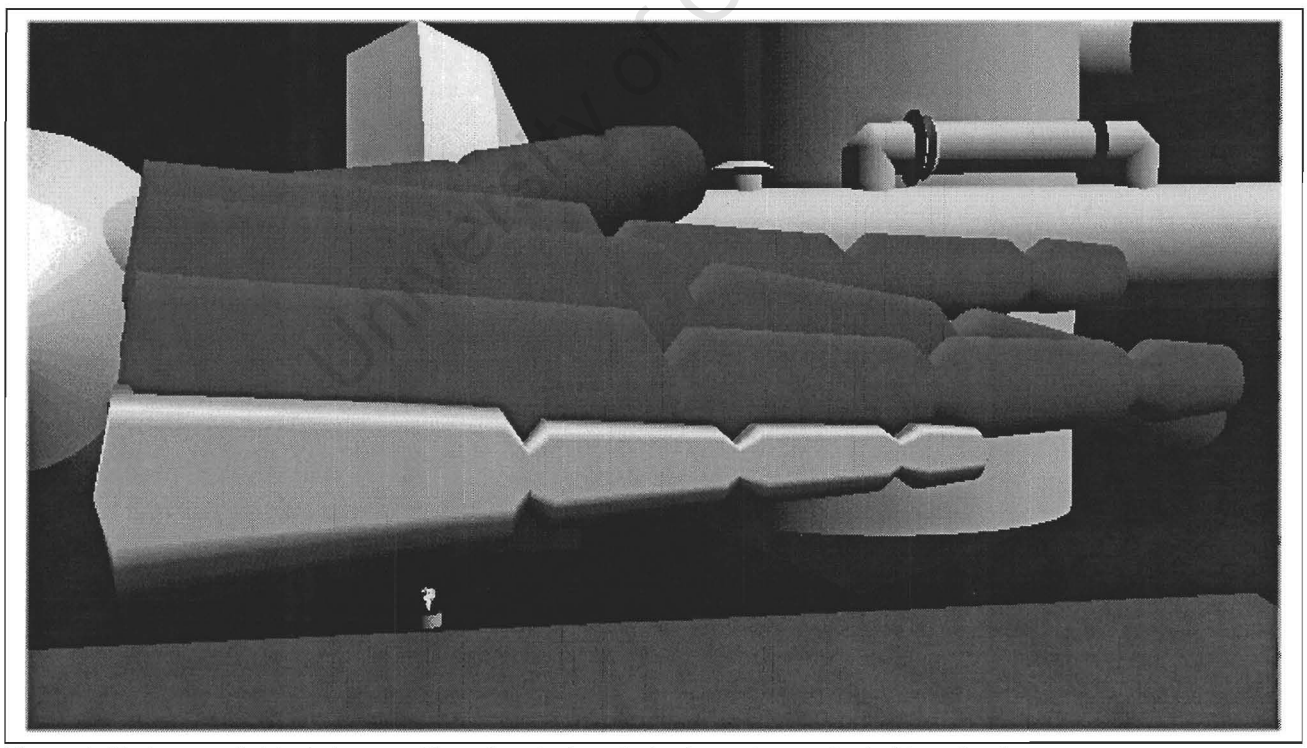



Figure 4.49 : Image of virtual glove positioned and orientated using values collected from tracker

4.4 Incorporating Motion

This section will describe how motion has been incorporating into the 3D virtual environment. Two functions are discussed, the first is the GetTracker function which forms an independent thread collecting position and orientation information. The second is SetupPos that rotates and translates the environment so that the virtual environment follows the movement of the user's head with respect to some fixed point.

As discussed in the GetFingers function, GetTracker also forms an independent thread of execution. As before it is also passed a pointer to the object of class CControlRoom. With this pointer, this thread becomes able to use all the functionality and resources provided by the CControlRoom class.

It starts off by getting the data from the tracking hardware using the member variable Trk of the CControlRoom class. This variable Trk, is an object of class CTracker, and thus this thread can call the member function GetPos of the CTracker class to get the orientation and position of the two receivers.

```
UINT GetTracker(LPVOID param)
{
   CControlRoom * Vw;
   Vw = (CControlRoom *)param;
   CString Result;
   short TmpA[6];
   short TmpB[6];

   while (Vw->Running)
   {
      Result = Vw->Trk.GetPos(&TmpA[0], &TmpB[0]);

      Vw->Sens1[0] = -(GLfloat)TmpA[1]/32768.0f*3.0f;
      Vw->Sens1[1] = -(GLfloat)TmpA[2]/32768.0f*3.0f;
      Vw->Sens1[2] = (GLfloat)TmpA[0]/32768.0f*3.0f;
      Vw->Sens1[3] = -(GLfloat)TmpA[4]/32768.0f*180.0f;
      Vw->Sens1[4] = 180 + (GLfloat)TmpA[3]/32768.0f*180.0f;
      Vw->Sens1[5] = (GLfloat)TmpA[5]/32768.0f*180.0f;

      Vw->Sens2[0] = -(GLfloat)TmpB[1]/32768.0f*3.0f;
      Vw->Sens2[1] = -(GLfloat)TmpB[2]/32768.0f*3.0f;
      Vw->Sens2[2] = (GLfloat)TmpB[0]/32768.0f*3.0f;
      Vw->Sens2[3] = -(GLfloat)TmpB[4]/32768.0f*180.0f;
      Vw->Sens2[4] = 180 + (GLfloat)TmpB[3]/32768.0f*180.0f;
      Vw->Sens2[5] = (GLfloat)TmpB[5]/32768.0f*180.0f;

      if (Result.GetLength() != 0)
      {
         ::MessageBox(NULL, (LPCTSTR)Result, 0, 0);
         return 1;
      }
      Sleep(200);
   }

   return 0;
}
```

The largest section of the above thread is responsible for converting the values got from the tracking system into OpenGL virtual distances and orientation angles. The data arrives as a series of six signed 16 bit integers for each receiver. For the distances, a value of 0 corresponds to 0 meters and a value of 32768 corresponds to 3 meters, so the values have to be divided by 32768 and multiplied by 3.

The angular measurements have a range of –180 to 180 degrees, and so must be divided by 32768 and multiplied by 180 to give angular measurements.

There is no direct correlation between the X, Y and Z co-ordinates of the tracking system and that used by OpenGL. This is because both use different frames of reference. As is seen from the diagram below, the X, Y and Z co-ordinates of the tracking system correspond to the Z, -X and -Y co-ordinates in the OpenGL framwork.

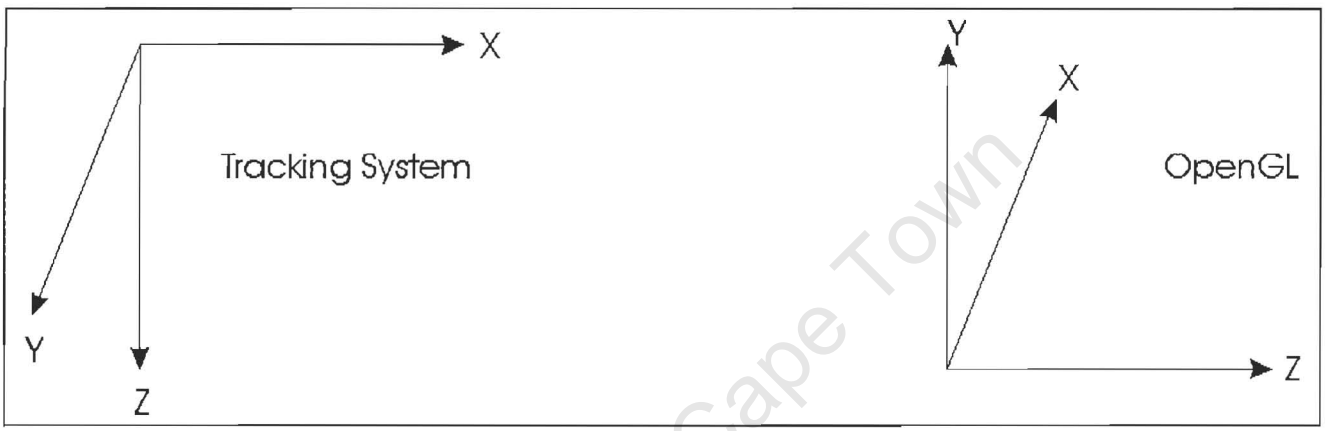


Figure 4.50 : Figure showing axis orientations of the OpenGL and Tracking system environments

Another point to note is that in the tracking system, the azimuth value is around the Z-axis, the elevation is around the Y-axis, and the roll is around the X-axis. Whereas in the OpenGL framework, azimuth is around the Y-axis, elevation is around the Z-axis, and roll is around the X-axis. Thus the mapping from the tracker to OpenGL is:

OpenGL azimuth = Tracker elevation
OpenGL elevation = Tracker azimuth
OpenGL roll = Tracker roll.

The sampling period is only 40ms to avoid wasting time collecting data that will be never be used because of a slow screen refresh rate.

SetupPos

Because the orientation angles that are returned by the tracking system are Euler angles, implementing the rotations is fairly involved. The reason for this is that OpenGL implements rotations around a specified axis by some given angle. After a rotation in OpenGL around the Y-axis (azimuth) by 90 degrees, the Y and Z-axis become swapped. It is no longer possible to rotate around Z-axis with the elevation angle, because it must now be done around the negative X-axis.

If the azimuth angle is not a multiple of 90 then it becomes more complicated. As then the new X and Z-axis are now functions of the azimuth angle. To implement the roll orientation next, there is another problem because the X-axis is now a function of the both the azimuth and the elevation angles.

Rotation is however still possible. OpenGL is a state machine, thus all rotations and translations operate on a transformation matrix. This transformation matrix is then used to multiply every vertex in every object that is rendered. OpenGL implements a function call that can retrieve this matrix.

Thus after rotating around the Y-axis by the azimuth angle, the new direction that the old X-axis would be pointing to needs to be calculated. This direction vector will become the axis around which the elevation angle will be rotated. This new direction vector can be found by multiplying the translation matrix by the old X-axis.

$$\text{Direction of old X axis} = \begin{bmatrix} m0 & m1 & m2 & m3 \\ m4 & m5 & m7 & m8 \\ m8 & m9 & m10 & m11 \\ m12 & m13 & m14 & m15 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} m0 \\ m1 \\ m2 \\ m3 \end{bmatrix}$$

Similarly to get the direction of the old Z-axis, the transformation matrix after both the azimuth and elevation rotations have been performed is used to multiply the old Z-axis. Only the X, Y and Z values need to be used in the transformation, as the last value, namely W, is not used in rotations, and can thus be safely ignored.

$$\text{Direction of old Z axis} = \begin{bmatrix} m0 & m1 & m2 & m3 \\ m4 & m5 & m7 & m8 \\ m8 & m9 & m10 & m11 \\ m12 & m13 & m14 & m15 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} m8 \\ m9 \\ m10 \\ m11 \end{bmatrix}$$

The changes in sign that occur on some variables is to account for the fact that OpenGL and the tracking system don't both have the same direction of positive angle increase. To finish, the translation of the virtual environment's origin is implemented. At the start of this function the frm variable is used to effect a horizontal translation. This translation is to implement stereo by shifting the viewpoint left for the left eye's image and shifting it right for the right eye's image.

4.5 PLC Controls

This section discusses how the PLC controls have been implemented. There are two sub-sections. The first one will discuss the CPLC class, which is responsible for the graphics, and the second sub-section will discuss the ControlLoop function where the implementation and simulation of the control laws is performed.

4.5.1 CPLC Class

The CPLC class is responsible for drawing and co-ordinating the Programmable Logic Controllers. This is done with the use of 21 member variables and nine member functions. Although being long, this class is simple to understand.

```
class CPLC : public CObject
{
public:
   BOOL First;
   void Update(void);
   GLfloat SumActPoint;
   void CheckContact(void *CR, GLfloat Sens[]);
   void DrawPLC();
   void SetPos(GLfloat tX, GLfloat tY, GLfloat tZ, CString Name);
   CPLC();
   virtual ~CPLC();
   CTextures *pTex;
   GLfloat SetPointIn;
   GLfloat OutPointIn;
   GLfloat ActPointIn;
   GLfloat ActPoint;
   GLfloat SetPoint;
   GLfloat OutPoint;

protected:
   GLuint List;
   GLfloat OutputArray[100];
   GLfloat SetPointArray[100];
   GLfloat ControlActionArray[100];
   void DrawData(void);
   void DrawGraphs(void);
   void GetEqu(int Btn, int Idx, GLdouble Equ[]);
   BOOL Button[8];
   GLuint selectBuf[256];
   BOOL  Auto;
   BOOL  Local;
   GLfloat X;
   GLfloat Y;
   GLfloat Z;
   CString PLCName;
};
```

```
void COpenGLObject::SetupPos(GLfloat Sens [ ], int frm)
{
  GLfloat t1, t2, t3;
  GLfloat m[16];

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  if ( (frm%2) == 0) glTranslatef(- 0.05f, 0.0f, 0.0f);
  else glTranslatef( 0.05f, 0.0f, 0.0f);

  glRotatef(Sens[4], 0.0f, 1.0f, 0.0f);
  glGetFloatv(GL_MODELVIEW_MATRIX, &m[0]);

  t1 = m[0];
  t2 = m[1];
  t3 = -m[2];

  glRotatef(Sens[3], t1, t2, t3);
  glGetFloatv(GL_MODELVIEW_MATRIX, &m[0]);

  t1 = -m[8];
  t2 = m[9];
  t3 = m[10];

  glRotatef(Sens[5], t1, t2, t3);

  glGetFloatv(GL_MODELVIEW_MATRIX, &m[0]);
  glLoadIdentity();
  glMultMatrixf(&m[0]);

  glTranslatef(Sens[0], Sens[1], Sens[2]);


}
```

Starting with the member variables, there are:

- X, Y and Z: These variables store the position of the co-ordinates of the bottom left hand corner of where the picture of the PLC will be placed.
- PLCName: This CString variable stores the name that appears above the PLC.
- Local: This boolean variable corresponds directly to the two buttons on the PLC, namely 'Local' and 'Central' and is used to indicate whether the setpoint is going to be set locally on the PLC, or whether it will be received from a remote controller.
- Auto: This boolean variable corresponds to the two buttons 'Auto' and 'Manual' on the PLC image. In auto mode, the control action is calculated by the PLC as a function of the setpoint, input and past values. In Manual mode, the user can use the controls on the PLC to set the control action.
- selectBuf: This is an array of 256 unsigned integers that will contain the selected objects. It is used internally by the class to find out which button is being pressed by the user.
- Button: This is an array of 8 booleans. Each boolean if TRUE specifies that a particular button on the PLC is being pressed. The relation between the variables is as follows :
  - Button[0] : Switch into manual mode.
  - Button[1] : Switch into auto mode.
  - Button[2] : Switch into Central control mode.
  - Button[3] : Switch into Local control mode.
  - Button[4] : Decrease the control action.
  - Button[5] : Increase the control action.
  - Button[6] : Decrease the Setpoint.
  - Button[7] : Increase the Setpoint.
- OutputArray, SetPointArray and ControlActionArray are three arrays that contain the last 100 values of the process output, the setpoint to the PLC and the control action to the actuator respectively. These values are used to plot graphs of these values on the one wall of the control room. This gives a past history of the three variables for the PLCs.
- List is an internal variable which is used in conjunction with selectBuf to aid in selection of which button is being pressed.
- ActPoint, SetPoint and OutPoint are variables that contain the current values of the Actuator control action, the setpoint set on the PLC, and the process output respectively.
- ActPointIn, SetPointIn and OutPointIn are variables that are used to set the values of the previously discussed three variables. These values are used by an external function that updates each value based on some control law.
- pTex is a variable that contains the pointer to the object of class CTexture which keeps the texture map that is used to display the PLC.
- First: This is a boolean variable that is used to identify whether this is the first time that the PLC is drawn. If it is then it is loaded into a display list. Next time it is used (First = FALSE) the display list will be called instead of redrawing a whole section of the PLC.
- SumActPoint: This is a floating point variable that is used to sum up the actuator control action values. The pump uses SumActPoint to draw a rotating blade.

The nine member functions are:

- Constructor and destructor that do the initialisation and cleanup of the class.
- DrawGraphs draws the three graphs and labels them.
- DrawData draws the data plots on the graph axes.
- GetEqu: This function returns the clipping plane for each side of a cube for each of the eight buttons. These clipping planes surround each button that can be pressed. How it is used will be explained later.
- SetPos sets the X, Y and Z values of the PLC image.
- DrawPLC draws the PLC virtual object.
- Update is called every time period and is used by the PLC to update the input, output and setpoint, and their respective arrays.
- CheckContact goes systematically through each button and checks if the glove is making contact with that button.

CPLC, ~ CPLC

The destructor is empty, and the constructor sets almost all of the 22 member variables to zero. It also sets the default settings to central and automatic control.

```
CPLC::~CPLC()
{

}

CPLC::CPLC()
{
   X = 0;
   Y = 0;
   Z = 0;

   SetPoint = 0.0f;
   OutPoint = 0.0f;
   ActPoint = 0.0f;
   SetPointIn = 0.0f;
   OutPointIn = 0.0f;
   ActPointIn = 0.0f;
   SumActPoint = 0.0f;

   for (int i = 0; i < 100; i++)
   {
      OutputArray[i] = 0.0f;
      SetPointArray[i] = 0.0f;
      ControlActionArray[i] = 0.0f;
   }

   Local = FALSE;
   Auto = TRUE;

   First = TRUE;
}
```

SetPos

This function sets up the X, Y and Z values of the bottom left hand corner of where the PLC virtual object is to be drawn. It also sets up the name that will be used tc draw a title above the PLC.

```
void CPLC::SetPos(GLfloat tX, GLfloat tY, GLfloat tZ, CString Name)
{
   X = tX;
   Y = tY;
   Z = tZ;
   PLCName = Name;
}
```

DrawPLC

By far the longest function that I have discussed thus far, this function is responsible for drawing the PLC virtual object. It starts off by setting up a few vectors that will be used to set the properties of materials used in drawing the PLC virtual object.

```
void CPLC::DrawPLC()
{
   GLfloat YellowLight[] = { 1.0f, 0.1f, 0.1f, 1.0f };
   GLfloat NoLight[] = { 0.0f, 0.0f, 0.0f, 1.0f };
   GLfloat DiffuseNorm[] = {0.8f, 0.8f, 0.8f, 1.0f };
   GLfloat DiffuseTrans[] = {0.8f, 0.8f, 0.8f, 0.5f };
   GLfloat DiffuseBlue[] = {0.5f, 0.5f, 1.2f, 1.0f };
   GLfloat DiffuseRed[] = {1.0f, 0.2f, 0.2f, 1.0f };
   GLfloat x,y,z,tmp;
```

Next it sets the colour to red and prints out the name of the PLC above the image of the PLC.

```
   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseRed);

   glPushMatrix();
      glTranslatef(X,Y + 0.8f,Z + 0.05f);
      glScalef(0.05f, 0.05f, 0.02f);
      glCallLists(PLCName.GetLength(), GL_UNSIGNED_BYTE, (LPCTSTR)PLCName);
   glPopMatrix();
```

Figure 4.51 : Title above virtual PLC

Next the function sets the colour to blue and draws the rim going round the PLC panel, making it look like it is displaced away from the wall.

```
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseBlue);

glBegin(GL_QUAD_STRIP);
   glVertex3f(X, Y, Z - 0.01f);
   glVertex3f(X, Y, Z + 0.02f);
   glVertex3f(X + 0.408f, Y, Z - 0.01f);
   glVertex3f(X + 0.408f, Y, Z + 0.02f);
   glVertex3f(X + 0.408f, Y + 0.8f, Z - 0.01f);
   glVertex3f(X + 0.408f, Y + 0.8f, Z + 0.02f);
   glVertex3f(X, Y + 0.8f, Z - 0.01f);
   glVertex3f(X, Y + 0.8f, Z + 0.02f);
   glVertex3f(X, Y, Z - 0.01f);
   glVertex3f(X, Y, Z + 0.02f);
glEnd();
```

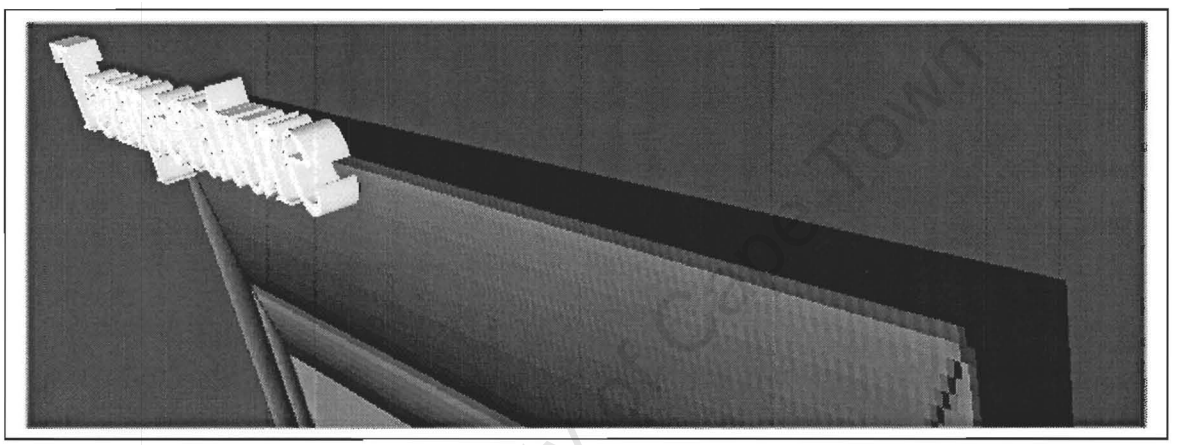


Figure 4.52 : Image showing 3D nature of the PLC control panel

Now the function will draw the texture map image of the PLC control panel. It starts off by disabling the lighting calculations to make the PLC texture map appear clearer. Next it enables texture mapping and sets the current colour to light grey. It then draws a rectangular plane that is filled with a texture map image of a PLC control panel. Once done, texture mapping is switched off and the lighting calculations turned back on. A temporary set of x, y and z variables are created that point to a point slightly raised away from the upper left hand corner of the PLC control panel image. These variables will be used in drawing the following set of images.

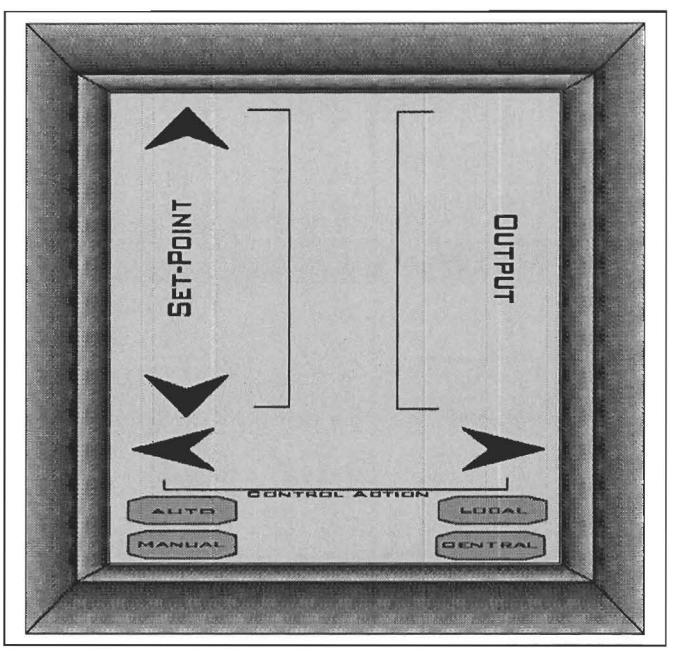


Figure 4.53 : Virtual PLC control panel

```
glDisable(GL_LIGHTING);
glEnable(GL_TEXTURE_2D);
glColor3f(0.7f, 0.7f, 0.7f);

pTex->fpglBindTextureEXT( GL_TEXTURE_2D, pTex->PLCTex);

glBegin(GL_QUADS);
   glTexCoord2f(0.0f, 1.0f);
   glVertex3f(X,Y,Z+0.02f);
   glTexCoord2f(0.0f, 0.0f);
   glVertex3f(X, Y + 0.8f, Z+0.02f);
   glTexCoord2f(1.0f, 0.0f);
   glVertex3f(X + 0.408f,Y + 0.8f,Z+0.02f);
   glTexCoord2f(1.0f, 1.0f);
   glVertex3f(X + 0.408f,Y,Z+ 0.02f);
glEnd();

x = X;
y = Y + 0.8f;
z = Z + 0.022f;

glDisable(GL_TEXTURE_2D);
glEnable(GL_LIGHTING);
```

In this part, the colour is set to a light grey, and the emissive colour is set to yellow. These colours are used to draw two triangles to indicate whether the PLC is in Auto or Manual mode, or whether it is under Local or Central control.

```
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, YellowLight);

if (Auto)
{
   glBegin(GL_TRIANGLES);
      glVertex3f(x + 0.16f, y - 0.636f, z);
      glVertex3f(x + 0.18f, y - 0.630f, z);
      glVertex3f(x + 0.18f, y - 0.642f, z);
   glEnd();
}
else
{
   glBegin(GL_TRIANGLES);
      glVertex3f(x + 0.16f, y - 0.680f, z);
      glVertex3f(x + 0.18f, y - 0.674f, z);
      glVertex3f(x + 0.18f, y - 0.686f, z);
   glEnd();
}
```

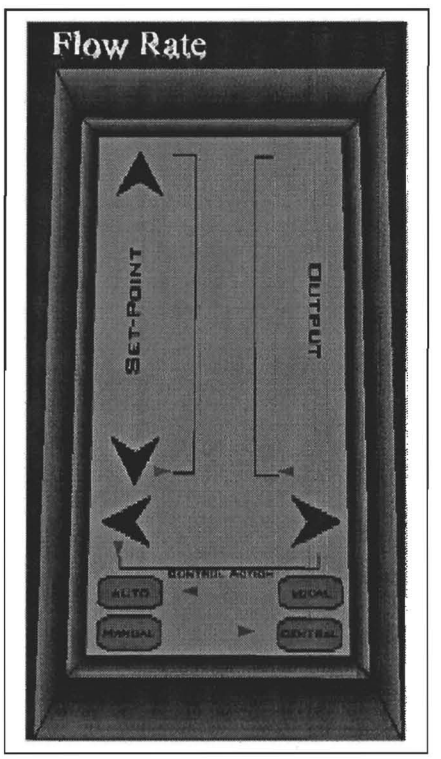


Figure 4.54 : Image showing Mode Indicators

```
if (Local)
{
   glBegin(GL_TRIANGLES);
      glVertex3f(x + 0.246f, y - 0.636f, z);
      glVertex3f(x + 0.226f, y - 0.630f, z);
      glVertex3f(x + 0.226f, y - 0.642f, z);
   glEnd();
}
else
{
   glBegin(GL_TRIANGLES);
      glVertex3f(x + 0.246f, y - 0.680f, z);
      glVertex3f(x + 0.226f, y - 0.674f, z);
      glVertex3f(x + 0.226f, y - 0.686f, z);
   glEnd();
}
```

The colour is now set to a 50% transparent grey, and rectangular boxes are drawn above the buttons that are currently in contact with the virtual image of the data glove. This is to provide visual feedback to the user to inform him / her that a button is busy being pressed. This feature can be substituted with a data glove that can provide force feedback to the user.

```
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseTrans);

if (Button[0])
{
   glBegin(GL_QUADS);
      glVertex3f(x + 0.068f, y - 0.664f, z + 0.002f);
      glVertex3f(x + 0.068f, y - 0.700f, z + 0.002f);
      glVertex3f(x + 0.140f, y - 0.700f, z + 0.002f);
      glVertex3f(x + 0.140f, y - 0.664f, z + 0.002f);
   glEnd();
}
if (Button[1])
{
   glBegin(GL_QUADS);
      glVertex3f(x + 0.068f, y - 0.618f, z + 0.002f);
      glVertex3f(x + 0.068f, y - 0.656f, z + 0.002f);
      glVertex3f(x + 0.140f, y - 0.656f, z + 0.002f);
      glVertex3f(x + 0.140f, y - 0.618f, z + 0.002f);
   glEnd();
}
if (Button[2])
{
   glBegin(GL_QUADS);
      glVertex3f(x + 0.27f, y - 0.664f, z + 0.002f);
      glVertex3f(x + 0.27f, y - 0.700f, z + 0.002f);
      glVertex3f(x + 0.34f, y - 0.700f, z + 0.002f);
      glVertex3f(x + 0.34f, y - 0.664f, z + 0.002f);
   glEnd();
}
```

```
if (Button[3])
{
   glBegin(GL_QUADS);
      glVertex3f(x + 0.27f, y - 0.620f, z + 0.002f);
      glVertex3f(x + 0.27f, y - 0.656f, z + 0.002f);
      glVertex3f(x + 0.340f, y - 0.656f, z + 0.002f);
      glVertex3f(x + 0.340f, y - 0.620f, z + 0.002f);
   glEnd();
}
if (Button[4])
{
   glBegin(GL_QUADS);
      glVertex3f(x + 0.070f, y - 0.528f, z + 0.002f);
      glVertex3f(x + 0.070f, y - 0.584f, z + 0.002f);
      glVertex3f(x + 0.126f, y - 0.584f, z + 0.002f);
      glVertex3f(x + 0.126f, y - 0.528f, z + 0.002f);
   glEnd();
}
if (Button[5])
{
   glBegin(GL_QUADS);
      glVertex3f(x + 0.286f, y - 0.53f, z + 0.002f);
      glVertex3f(x + 0.286f, y - 0.588f, z + 0.002f);
      glVertex3f(x + 0.336f, y - 0.588f, z + 0.002f);
      glVertex3f(x + 0.336f, y - 0.53f, z + 0.002f);
   glEnd();
}
if (Button[6])
{
   glBegin(GL_QUADS);
      glVertex3f(x + 0.08f, y - 0.46f, z + 0.002f);
      glVertex3f(x + 0.08f, y - 0.514f, z + 0.002f);
      glVertex3f(x + 0.134f, y - 0.514f, z + 0.002f);
      glVertex3f(x + 0.134f, y - 0.46f, z + 0.002f);
   glEnd();
}
if (Button[7])
{
   glBegin(GL_QUADS);
      glVertex3f(x + 0.076f, y - 0.11f, z + 0.002f);
      glVertex3f(x + 0.076f, y - 0.166f, z + 0.002f);
      glVertex3f(x + 0.1340f, y - 0.166f, z + 0.002f);
      glVertex3f(x + 0.1340f, y - 0.11f, z + 0.002f);
   glEnd();
}
```

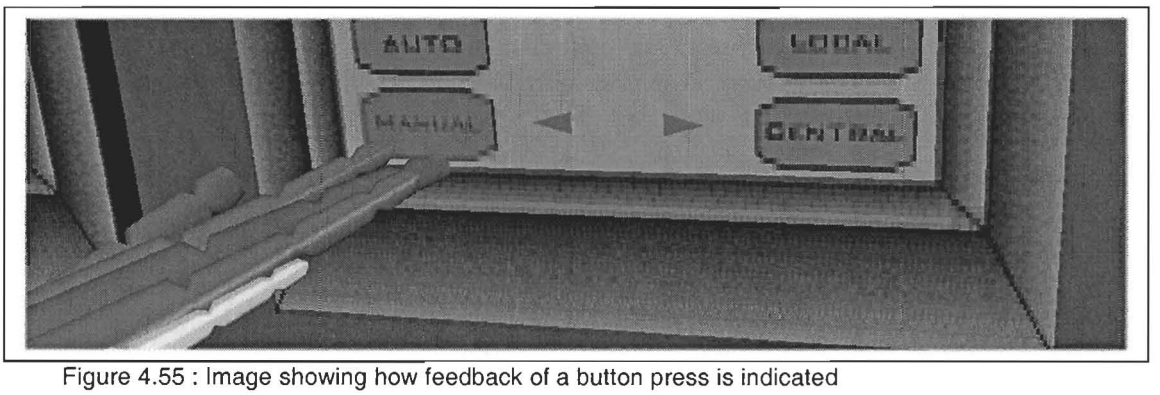


Figure 4.55 : Image showing how feedback of a button press is indicated

Next, three more triangles are drawn which indicate on sliding scales the values of the Actuator control action, the Setpoint on the PLC, and the output value from the sensor in the plant. These give live feedback of the current state of the PLC's control loop to the user. Finally it ends with calls to the DrawGraphs and DrawData functions which display the data as it has occurred over the past 100 time periods.

```
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

    tmp = X + 0.09f + (0.336f - 0.09f) / 100.0f * ActPoint;

    glBegin(GL_TRIANGLES);
       glVertex3f(tmp, y - 0.600f, z);
       glVertex3f(tmp - 0.006f, y - 0.580f, z);
       glVertex3f(tmp + 0.006f, y - 0.580f, z);
    glEnd();

    tmp = y - 0.112f - (0.5f - 0.112f) / 100.0f * (100.0f - SetPoint);

    glBegin(GL_TRIANGLES);
       glVertex3f(x + 0.15f, tmp , z);
       glVertex3f(x + 0.13f, tmp + 0.006f, z);
       glVertex3f(x + 0.13f, tmp - 0.006f, z);
    glEnd();

    tmp = y - 0.112f - (0.5f - 0.112f) / 100.0f * (100.0f - OutPoint);

    glBegin(GL_TRIANGLES);
       glVertex3f(x + 0.27f, tmp , z);
       glVertex3f(x + 0.29f, tmp + 0.006f, z);
       glVertex3f(x + 0.29f, tmp - 0.006f, z);
    glEnd();

    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, NoLight);

    DrawGraphs();
    DrawData();
}
```

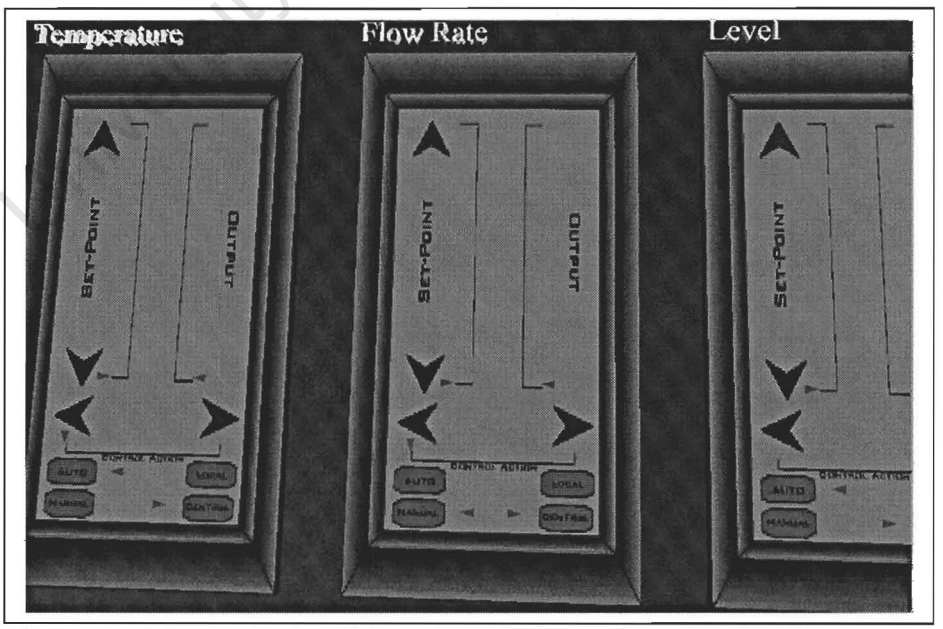


Figure 4.56 : Image showing all three virtual PLC control panels with indicators

CheckContact

This function is executed to test if the virtual glove inside the environment has come into contact with any one of the eight buttons that form part of the PLC. If contact is made, then internal member variables are adjusted to take effect of the fact that the user is pressing a button.

This function starts off by getting a pointer to the object instance of the CControlRoom class. This is needed so that it can access the DrawHand function, which is a member function of the CControlRoom class. The Button array of booleans is then cleared.

```
void CPLC::CheckContact(void * CR, GLfloat Sens[])
{
  CControlRoom *ConR;
  ConR = (CControlRoom *)CR;
  GLdouble Equ[4];

  GLfloat x, y, z, dx, dy, dz;
  int i;

  x = X + 0.204f;
  y = Y + 0.4f;
  z = Z;

  dx = x - Sens[0];
  dy = y - Sens[1];
  dz = z - Sens[2];

  if (dx < 0) dx = -dx;
  if (dy < 0) dy = -dy;
  if (dz < 0) dz = -dz;

  for (i = 0; i < 8; i++) Button[i] = FALSE;
```

This next section is repeated eight times, once for each button. The clipping planes are first enabled. The six clipping plane equations are then acquired from the GetEqu function for the current button, and set up as OpenGL clipping planes.

Having set up the clipping planes, the selection buffer is set up, the rendering context is put into selection mode and names of the objects that will be drawn are initialised and loaded. Only one name is required, as checking only needs to take place for a single object's intersection with the current clipped area.

The current state matrix is then saved, the hand is drawn, and the state matrix restored. The rendering state is set back into rendering mode, which on returning returns the value of the object that intersected with the area that was rendered. If the hand intersected this area, then the calculation that follows will return its name of '1'. If intersection did occur then corresponding boolean in the Button array can be set.

The final section of the loop disables the clipping planes.

```
for (i = 0; i < 8; i++)
{
   glEnable(GL_CLIP_PLANE0);  //Left
   glEnable(GL_CLIP_PLANE1);  //Right
   glEnable(GL_CLIP_PLANE2);  //Top
   glEnable(GL_CLIP_PLANE3);  //Bottom
   glEnable(GL_CLIP_PLANE4);  //Back
   glEnable(GL_CLIP_PLANE5);  //Front
   GetEqu(i,0,&Equ[0]);
   glClipPlane(GL_CLIP_PLANE0, Equ);
   GetEqu(i,1,&Equ[0]);
   glClipPlane(GL_CLIP_PLANE1, Equ);
   GetEqu(i,2,&Equ[0]);
   glClipPlane(GL_CLIP_PLANE2, Equ);
   GetEqu(i,3,&Equ[0]);
   glClipPlane(GL_CLIP_PLANE3, Equ);
   GetEqu(i,4,&Equ[0]);
   glClipPlane(GL_CLIP_PLANE4, Equ);
   GetEqu(i,5,&Equ[0]);
   glClipPlane(GL_CLIP_PLANE5, Equ);

   glSelectBuffer(256, selectBuf);
    glRenderMode(GL_SELECT);
   glInitNames();
   glPushName((GLuint)~0);

   glPushMatrix();
   glLoadName(1);
   ConR->DrawHand();
   glPopMatrix();

   GLint hits = glRenderMode(GL_RENDER);
   int item = selectBuf[(hits-1)*4+3];
   if ((item > 0) && (hits > 0)) Button[i] = TRUE;

   glDisable(GL_CLIP_PLANE0);
   glDisable(GL_CLIP_PLANE1);
   glDisable(GL_CLIP_PLANE2);
   glDisable(GL_CLIP_PLANE3);
   glDisable(GL_CLIP_PLANE4);
   glDisable(GL_CLIP_PLANE5);
}
```


The last part of this function implements the changes that could occur if a button had been pressed. The first two buttons change the state between either Auto to Manual control. The second two change the state between local and central control.

Buttons four and five increase or decrease the control action to the actuator if the PLC is in Manual mode. If it is in Auto mode then the control action is set from the ActPointIn variable that is set by a remote function. Buttons six and seven change the setpoint value that is used by the PLC closed loop controller if it is Local Control mode. If the PLC is in Central Control mode, the value for the setpoint is acquired from the SetPointIn variable.

Finally OutPoint, the variable containing the value of the sensor, is acquired from the OutPointIn variable. All the above three values are also limited to being between 0 and 100.

```
    if (Button[0]) Auto = FALSE;
    if (Button[1]) Auto = TRUE;

    if (Button[2]) Local = FALSE;
    if (Button[3]) Local = TRUE;

    if (Button[4] && !Auto) ActPoint = ActPoint - 1;
    if (Button[5] && !Auto) ActPoint = ActPoint + 1;

    if (Auto) ActPoint = ActPointIn;

    if (ActPoint < 0) ActPoint = 0;
    if (ActPoint > 100) ActPoint = 100;1

    SumActPoint = SumActPoint + ActPoint;
    if (SumActPoint > 200.0f) SumActPoint = SumActPoint - 200.0f;
    if (SumActPoint < 0.0f) SumActPoint = 0.0f;

    if (Button[6] && Local) SetPoint = SetPoint - 1;
    if (Button[7] && Local) SetPoint = SetPoint + 1;

    if (!Local) SetPoint = SetPointIn;

    if (SetPoint < 0) SetPoint = 0;
    if (SetPoint > 100) SetPoint = 100;

    OutPoint = OutPointIn;

    if (OutPoint < 0) OutPoint = 0;
    if (OutPoint > 100) OutPoint = 100;

}
```

GetEqu

This function returns the clipping equation for each of the six sides of each of the eight buttons. These values have been derived experimentally for the sizes and positions of the buttons. In the latter part of this function, the equations are modified to reflect that the PLC could be at different X, Y and Z co-ordinates.

```
void CPLC::GetEqu(int Btn, int Idx, GLdouble Equ [ ])
{
  int i;
  GLdouble Data[][4] = {  { 1.0f, 0.0f, 0.0f, 0.068f },
                          { -1.0f, 0.0f, 0.0f, -0.152f}, \
  { 0.0f, -1.0f, 0.0f, 0.664f }, {0.0f, 1.0f, 0.0f, -0.70f }, \
  { 0.0f, 0.0f, 1.0f, -0.01f }, {0.0f, 0.0f, -1.0f, -0.02f }, \

  { 1.0f, 0.0f, 0.0f, 0.068f }, { -1.0f, 0.0f, 0.0f, -0.152f}, \
  { 0.0f, -1.0f, 0.0f, 0.618f }, {0.0f, 1.0f, 0.0f, -0.656f }, \
  { 0.0f, 0.0f, 1.0f, -0.01f }, {0.0f, 0.0f, -1.0f, -0.02f }, \
```

```
    { 1.0f, 0.0f, 0.0f, 0.27f }, { -1.0f, 0.0f, 0.0f, -0.340f}, \
    { 0.0f, -1.0f, 0.0f, 0.664f }, {0.0f, 1.0f, 0.0f, -0.70f }, \
    { 0.0f, 0.0f, 1.0f, -0.01f }, {0.0f, 0.0f, -1.0f, -0.02f }, \

    { 1.0f, 0.0f, 0.0f, 0.27f }, { -1.0f, 0.0f, 0.0f, -0.340f}, \
    { 0.0f, -1.0f, 0.0f, 0.620f }, {0.0f, 1.0f, 0.0f, -0.656f }, \
    { 0.0f, 0.0f, 1.0f, -0.01f }, {0.0f, 0.0f, -1.0f, -0.02f }, \

    { 1.0f, 0.0f, 0.0f, 0.07f }, { -1.0f, 0.0f, 0.0f, -0.126f}, \
    { 0.0f, -1.0f, 0.0f, 0.53f }, {0.0f, 1.0f, 0.0f, -0.588f }, \
    { 0.0f, 0.0f, 1.0f, -0.01f }, {0.0f, 0.0f, -1.0f, -0.02f }, \

    { 1.0f, 0.0f, 0.0f, 0.286f }, { -1.0f, 0.0f, 0.0f, -0.336f}, \
    { 0.0f, -1.0f, 0.0f, 0.53f }, {0.0f, 1.0f, 0.0f, -0.588f }, \
    { 0.0f, 0.0f, 1.0f, -0.01f }, {0.0f, 0.0f, -1.0f, -0.02f }, \

    { 1.0f, 0.0f, 0.0f, 0.08f }, { -1.0f, 0.0f, 0.0f, -0.134f}, \
    { 0.0f, -1.0f, 0.0f, 0.460f }, {0.0f, 1.0f, 0.0f, -0.514f }, \
    { 0.0f, 0.0f, 1.0f, -0.01f }, {0.0f, 0.0f, -1.0f, -0.02f }, \

    { 1.0f, 0.0f, 0.0f, 0.076f }, { -1.0f, 0.0f, 0.0f, -0.134f}, \
    { 0.0f, -1.0f, 0.0f, 0.11f }, {0.0f, 1.0f, 0.0f, -0.164f }, \
    { 0.0f, 0.0f, 1.0f, -0.01f }, {0.0f, 0.0f, -1.0f, -0.02f }, \
    };

    i = Idx + Btn * 6;

    Equ[0] = Data[i][0];
    Equ[1] = Data[i][1];
    Equ[2] = Data[i][2];
    switch (Idx)
    {
    case 0:
       Equ[3] = -Data[i][3] - X;
       break;
    case 1:
       Equ[3] = -Data[i][3] + X;
       break;
    case 2:
       Equ[3] = Y + 0.8f - Data[i][3];
       break;
    case 3:
       Equ[3] = - Data[i][3] - Y - 0.8f;
       break;
    case 4:
       Equ[3] = - Data[i][3] - Z;
       break;
    case 5:
       Equ[3] = - Data[i][3] + Z;
       break;
    default:
       break;

    }

}
```

DrawGraphs

This function is responsible for drawing the graphs, i.e. the boxes and labels. The data will be drawn by a separate function. If this is the first time that this function is called, then the whole image is drawn into a display list. Henceforth, the display list will be called instead of going through all the individual calls.

The function starts by drawing the three rectangles on which the data will be drawn. Once drawn, the colour is changed to red, and the four sets of textual output, namely the three graph titles as well as a section title to identify the PLC to which the graphs belong, are rendered.

```
void CPLC::DrawGraphs()
{
  GLfloat DiffuseNorm[] = {1.0f, 1.0f, 1.0f, 1.0f };
  GLfloat DiffuseRed[] = {1.0f, 0.2f, 0.2f, 1.0f };

  if (First)
  {
     List = glGenLists(1);
     glNewList( List, GL_COMPILE_AND_EXECUTE );

     glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseNorm);

     glBegin(GL_QUADS);
        glNormal3f(0.0f, 0.0f, -1.0f);
        glVertex3f(X, Y, -Z);
        glVertex3f(X + 0.4f, Y, -Z);
        glVertex3f(X + 0.4f, Y + 0.2f, -Z);
        glVertex3f(X, Y + 0.2f, -Z);
        glVertex3f(X, Y + 0.3f, -Z);
        glVertex3f(X + 0.4f, Y + 0.3f, -Z);
        glVertex3f(X + 0.4f, Y + 0.5f, -Z);
        glVertex3f(X, Y + 0.5f, -Z);
        glVertex3f(X, Y + 0.6f, -Z);
        glVertex3f(X + 0.4f, Y + 0.6f, -Z);
        glVertex3f(X + 0.4f, Y + 0.8f, -Z);
        glVertex3f(X, Y + 0.8f, -Z);
     glEnd();

     glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE , DiffuseRed);

     glPushMatrix();
        glTranslatef(X + 0.35f,Y + 0.9f,- Z - 0.02f);
        glScalef(-0.05f, 0.05f, 0.02f);
        glCallLists(PLCName.GetLength(), GL_UNSIGNED_BYTE, (LPCTSTR)PLCName);
     glPopMatrix();

     glPushMatrix();
        glTranslatef(X + 0.35f,Y + 0.2f,- Z - 0.02f);
        glScalef(-0.03f, 0.03f, 0.01f);
        glCallLists(14, GL_UNSIGNED_BYTE, "Control Action");
     glPopMatrix();
```

```
        glPushMatrix();
           glTranslatef(X + 0.35f,Y + 0.5f,- Z - 0.02f);
           glScalef(-0.03f, 0.03f, 0.01f);
           glCallLists(6, GL_UNSIGNED_BYTE, "Output");
        glPopMatrix();

        glPushMatrix();
           glTranslatef(X + 0.35f,Y + 0.8f,- Z - 0.02f);
           glScalef(-0.03f, 0.03f, 0.01f);
           glCallLists(8, GL_UNSIGNED_BYTE, "Setpoint");
        glPopMatrix();

        glEndList();
        First = FALSE;
     }
     else
     {
        glCallList(List);
     }
}
```

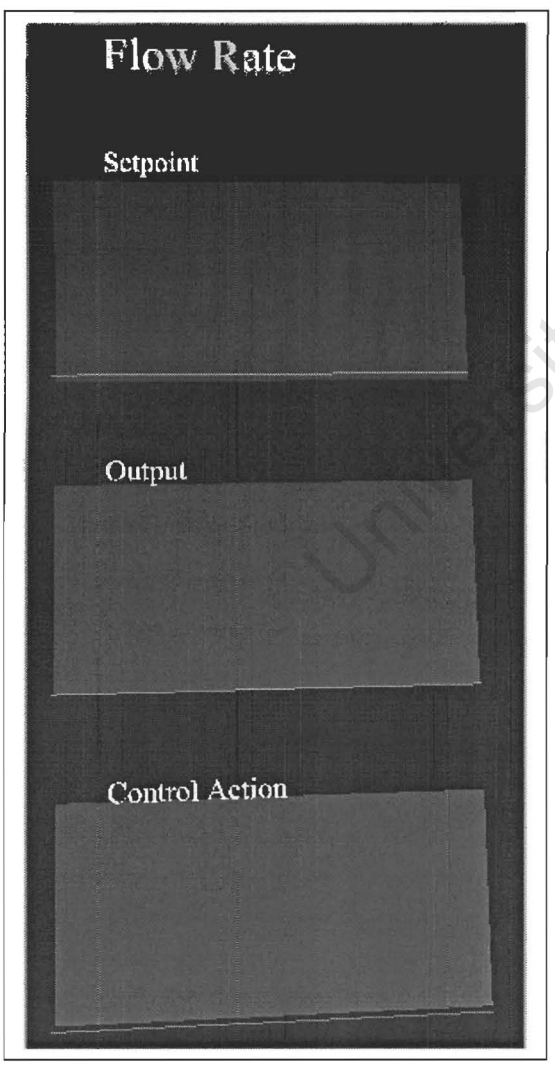


Figure 4.57 : Image of graphs with titles

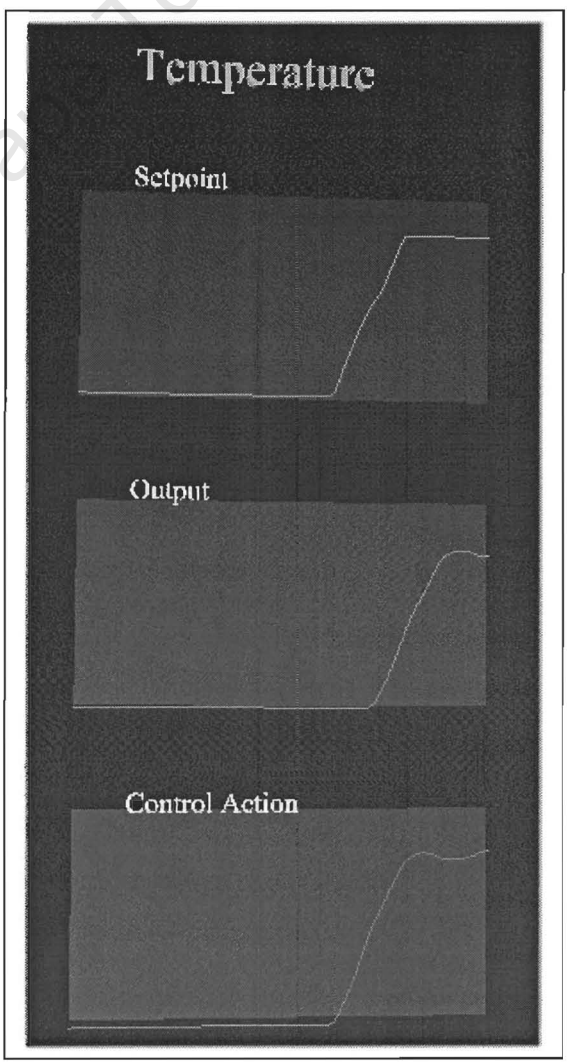


Figure 4.58 : Image of graphs displaying values over time

DrawData

This function draws the data stored in the arrays of past data for the control action, setpoint, and sensor output variables. It draws three lines, each line made up of a 100 green line segments.

```
void CPLC::DrawData()
{
   glDisable(GL_LIGHTING);
   glColor3f(0.1f, 0.7f, 0.0f);
   int i;

   glBegin(GL_LINE_STRIP);
      for (i = 0; i < 100; i++) glVertex3f(X + 0.4f - 0.004f * i,
                  Y + ControlActionArray[i] * 0.002f, -Z - 0.025f);
   glEnd();

   glBegin(GL_LINE_STRIP);
      for (i = 0; i < 100; i++) glVertex3f(X + 0.4f - 0.004f * i,
                  Y + 0.3f + OutputArray[i] * 0.002f, -Z - 0.025f);
   glEnd();

   glBegin(GL_LINE_STRIP);
      for (i = 0; i < 100; i++) glVertex3f(X + 0.4f - 0.004f * i,
            Y + 0.6f + SetPointArray[i] * 0.002f, -Z - 0.025f);
   glEnd();
   glEnable(GL_LIGHTING);
}
```

Update

This function is used to update the history data with the new values. Updating the data is done by shifting each element to one position lower and filling the last variable with the latest data, i.e. it is a shifting graph with the latest value on the farthest right of the graph.

```
void CPLC::Update()
{
   for (int i = 0; i < 99; i++)
   {
      ControlActionArray[i] = ControlActionArray[i + 1];
      OutputArray[i] = OutputArray[i + 1];
      SetPointArray[i] = SetPointArray[i + 1];
   }

   ControlActionArray[99] = ActPoint;
   OutputArray[99] = OutPoint;
   SetPointArray[99] = SetPoint;
}
```

4.5.2 ControlLoop Function

This function defines both the control dynamics of the equipment in the virtual plant as well as the control strategies that are implemented to control these dynamics. The following section describes the control action and response dynamics graphically.

The temperature meter's response to heating has been created to have the following dynamics:

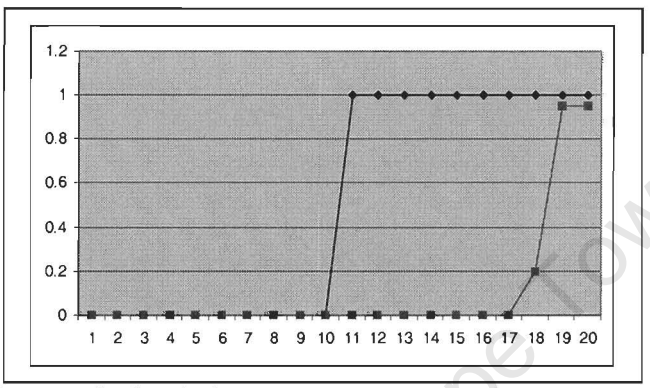


Figure 4.59 : Graph showing temperature meter's step response

There is an alarm condition that is programmed to turn on when the temperature exceeds 90% of its full range, and switch off again once the temperature has dropped to less than 75% of the full range.

The flow rate's response to a step in pump rate is shown below:

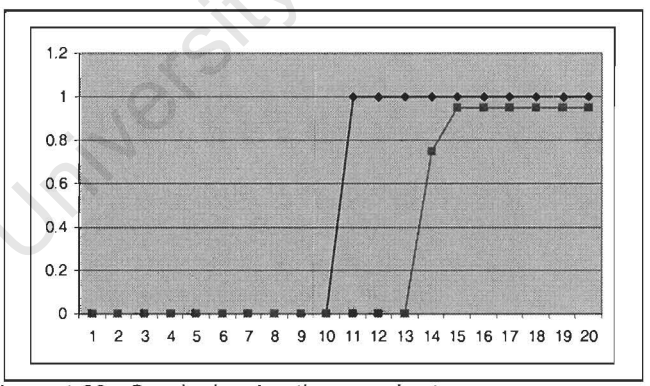


Figure 4.60 : Graph showing the pump's step response

The dynamics of the water tank level indicator are a bit more complicated because the water level depends on the inflow, and the outflow. The outflow however depends on the water level. The graph on the following page demonstrates how the changing the inflow pump and outflow valves affects the water level.

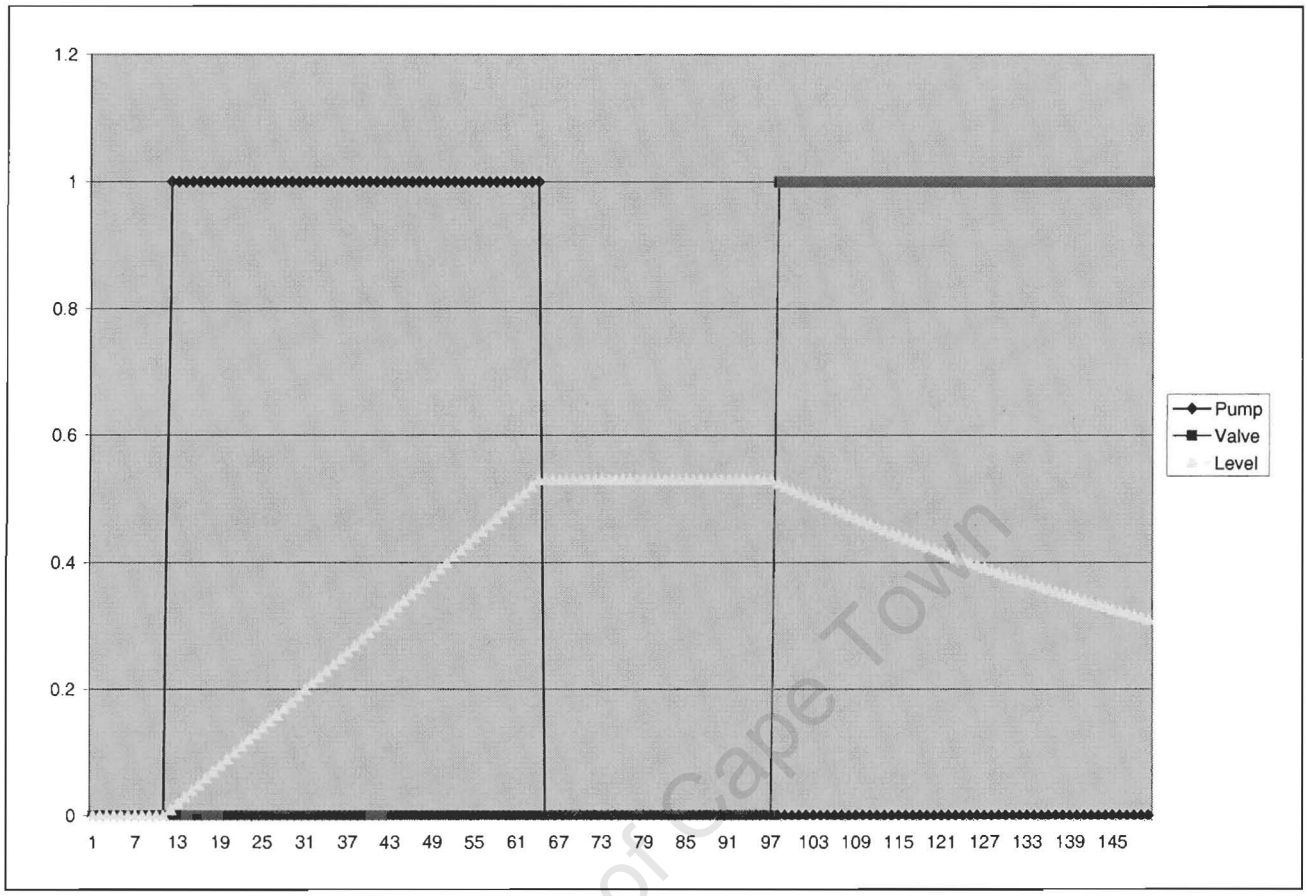


Figure 4.61 : Graph showing the response of the tank level with steps in pump and valve values.

```
void CControlRoom::ControlLoop()
{
  int i;
  GLfloat tmp;

  for (i = 8; i > 0; i--) State[i] = State[i - 1];
  for (i = 8; i > 0; i--) State2[i] = State2[i - 1];
  for (i = 8; i > 0; i--) State3[i] = State3[i - 1];

  State[0] = Plc.ActPoint;
  State2[0] = Plc2.ActPoint;

  Plc.OutPointIn = 0.2f * State[7] + 0.75f * State[8];

  if (Plc.OutPointIn > 90.0f) Obj.Alarm = TRUE;

  if (Plc.OutPointIn < 75.0f) Obj.Alarm = FALSE;

  Plc2.OutPointIn = 0.75f * State2[3] + 0.2f * State2[4];

  State3[0] = Plc2.OutPointIn;

  Plc3.OutPointIn = Plc3.OutPointIn + State3[3] * 0.01f -
                    (Plc3.OutPointIn * Plc3.ActPoint * 0.0001f);
```

```
    if (Plc3.OutPointIn < 0.0f) Plc3.OutPointIn = 0.0f;
    if (Plc3.OutPointIn > 100.0f) Plc3.OutPointIn = 100.0f;

    State[9] += 0.1f * (Plc.SetPoint - Plc.OutPointIn);
    if (State[9] > 100.0f) State[9] = 100.0f;
    if (State[9] < 0.0f) State[9] = 0.0f;

    State2[9] += 0.1f * (Plc2.SetPoint - Plc2.OutPointIn);
    if (State2[9] > 100.0f) State2[9] = 100.0f;
    if (State2[9] < 0.0f) State2[9] = 0.0f;

    State3[9] += 0.1f * (Plc3.SetPoint - Plc3.OutPointIn);
    if (State3[9] > 100.0f) State3[9] = 100.0f;
    if (State3[9] < 0.0f) State3[9] = 0.0f;


    tmp = State[9] + 0.5f * (Plc.SetPoint - Plc.OutPointIn);

    if (tmp > 100.0f) tmp = 100.0f;
    if (tmp < 0.0f) tmp = 0.0f;

    Plc.ActPointIn = tmp;
    Plc.Update();

    tmp = State2[9] + 2.5f * (Plc2.SetPoint - Plc2.OutPointIn);

    if (tmp > 100.0f) tmp = 100.0f;
    if (tmp < 0.0f) tmp = 0.0f;

    Plc2.ActPointIn = tmp;
    Plc2.Update();

    tmp = State3[9] + 2.5f * (Plc3.SetPoint - Plc3.OutPointIn);

    if (tmp > 100.0f) tmp = 100.0f;
    if (tmp < 0.0f) tmp = 0.0f;

    Plc3.ActPointIn = 100.0f - tmp;
    Plc3.Update();
}
```


All the states are limited to a range of between 0 and 100. Below are the control laws for the three Controllers. Although none of these are in any way optimal controllers for the given plants, they have been designed to show some sort of activity on the graphs in the virtual control room.

$$Output_{Heater} = 0.5 * (\text{Setpoint} - \text{Error}) + 0.1 * \sum_{-\infty}^{t} (\text{Setpoint} - \text{Error})$$

$$Output_{Pump} = 2.5 * (\text{Setpoint} - \text{Error}) + 0.1 \sum_{-\infty}^{t} (\text{Setpoint} - \text{Error})$$

$$Output_{Valve} = 2.5 * (\text{Setpoint} - \text{Error}) + 0.1 \sum_{-\infty}^{t} (\text{Setpoint} - \text{Error})$$

Chapter 5
Technical Evaluation

This chapter will be dedicated to evaluating all the pieces of hardware purchase from a technical point of view. Topics that could influence their suitability to being used in a Control Engineering environment are additionally discuss.

First will be a discussion on the performance, and the suitability of the data glove for interaction with a virtual environment. Starting with the tilt sensors on the glove, moving on through the fibre optic flexure sensors, and ending with a discussion of items that are lacking. The discussion of the tracking system focuses mainly on how well it is able to perform its function of position and orientation monitoring under non-ideal conditions. A short discussion of its suitability is also held. The visual capabilities of the head mounted display are finally evaluated and some qualitative analysis of whether these capabilities are good enough is performed.

5.1 Data Glove

This section evaluates the performance of the data glove and how it meets the suitability as a tool for user interaction with the virtual environment. It starts with a discussion of the tilt sensors that form part of the data glove. Their range, accuracy and dynamic response are evaluated. Secondly the fibre optic flexure sensors are evaluated. Their performance is tested, and also diagnosed to see if their capabilities are sufficient for effective human – VR environment interaction. This section ends off by discussing other gloves and interfaces that are currently available on the market.

5.1.1 Tilt Sensors

The "Fifth Dimension Technology's" Data Glove has two tilt sensors mounted on the glove. These are used to measure the roll and tilt of the hand with respect to Earth. I.e. they are gravitational based tilt sensors. It has already been mentioned that these will not be used as part of the virtual environment. It is only with the discussions within this section that the reasons for not using them will become fully understood.

This section starts by measuring the range of these sensors. This leads onto a discussion of the static accuracy of these sensors over their usable range. It then continues on to measure the dead-band of the sensors. Next, a few tests are performed to show how the accuracy of one sensor is affected as a function of the value of the other sensor. A brief evaluation of the system dynamics of the sensors is also performed. Finally this section ends off with a discussion of various other effects that can affect the accuracy of the readings provided by these sensors.

Range

The easiest way to measure the range is to record the values of the tilt sensors at various orientation angles. By plotting these it is possible to visually determine the usable range over which that the data glove's tilt sensors can provide orientation information. The plot of recorded readings with changing orientation is shown below.

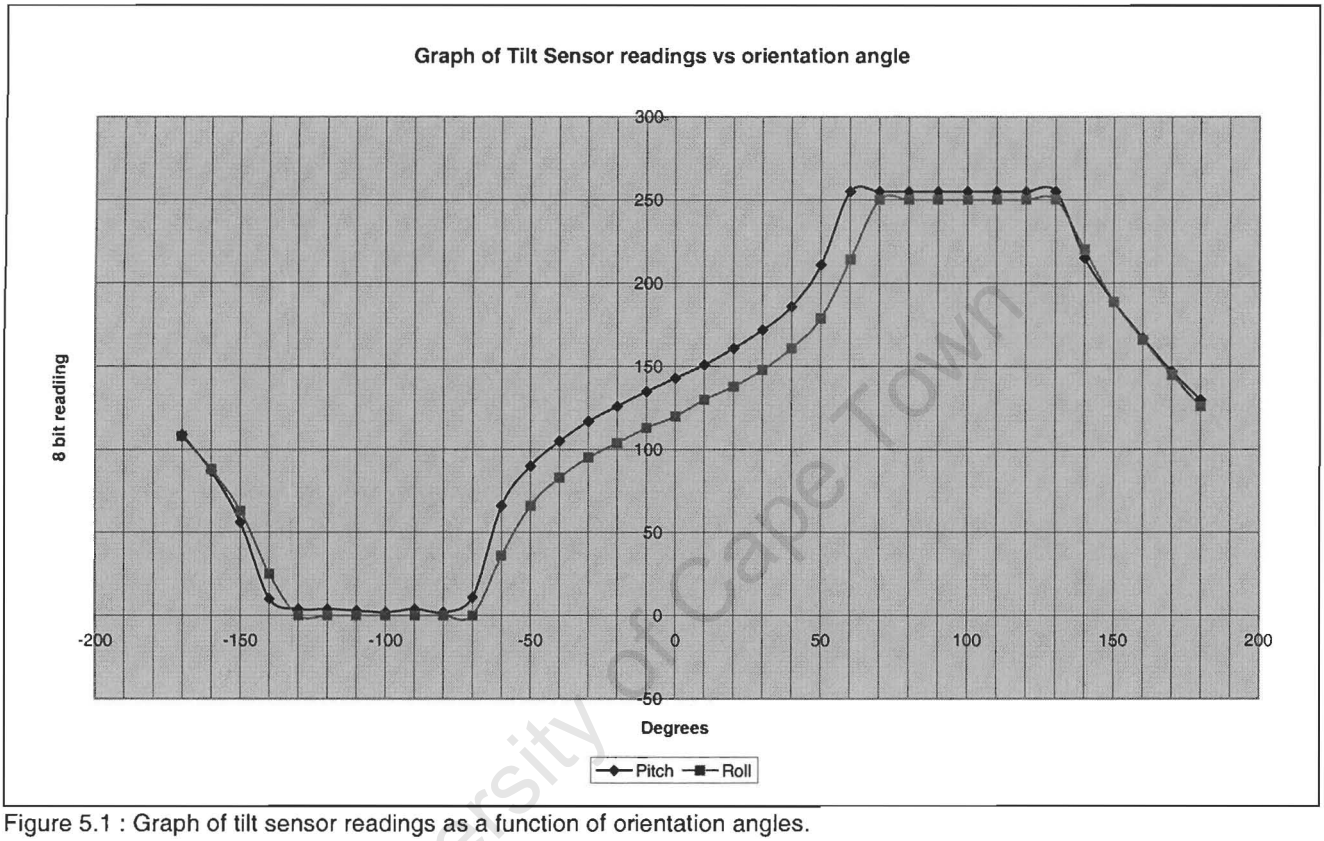


Figure 5.1 : Graph of tilt sensor readings as a function of orientation angles.

The graph above shows that the tilt sensors are also able to operate when upside down with a slightly reduced range. There is however no way of distinguishing whether the glove is upside down or not, and thus the range of values from 140 to 180 degrees and -180 to –140 degrees cannot be used together with the range from –60 to 60 degrees. Doing so would cause an ambiguity in the readings. Because the range between -60 and 60 degrees is larger, this will be the usable range of the data glove's tilt sensors.

Static Accuracy

As was already visible in the graph used to calculate the range of the data glove's tilt sensors, the values that are recorded are by no means a linear reflection of the orientation angle that they are supposed to measure. Even over the usable range the tilt sensor readings exhibit a non-linearity.

The next step is to find the straight line that would best fit the values read in from the tilt sensors. This was done in Excel by solving for the *m* and *c* in the equation "y = *m*x + *c*" and trying to minimise the mean square error between this straight line and the data readings recorded from the tilt sensors. This was done only over the earlier decided on range of –60 to 60 degrees.

For the pitch readings the best fit straight line (as shown on the graph below) was found to be "y = 1.271429x + 147.5385". The fact that the zero degrees axis intersection is at 148 and not at 127 as would be expected (half of 255 which is the maximum value represented by 8 bits) shows that the sensor was inaccurately mounted inside the plastic package. It should be reasonably expected that a perfectly vertical orientation should yield a reading of 127. The graph below also shows how the error between the best fit straight line and the tilt sensor readings increases outside the chosen range.

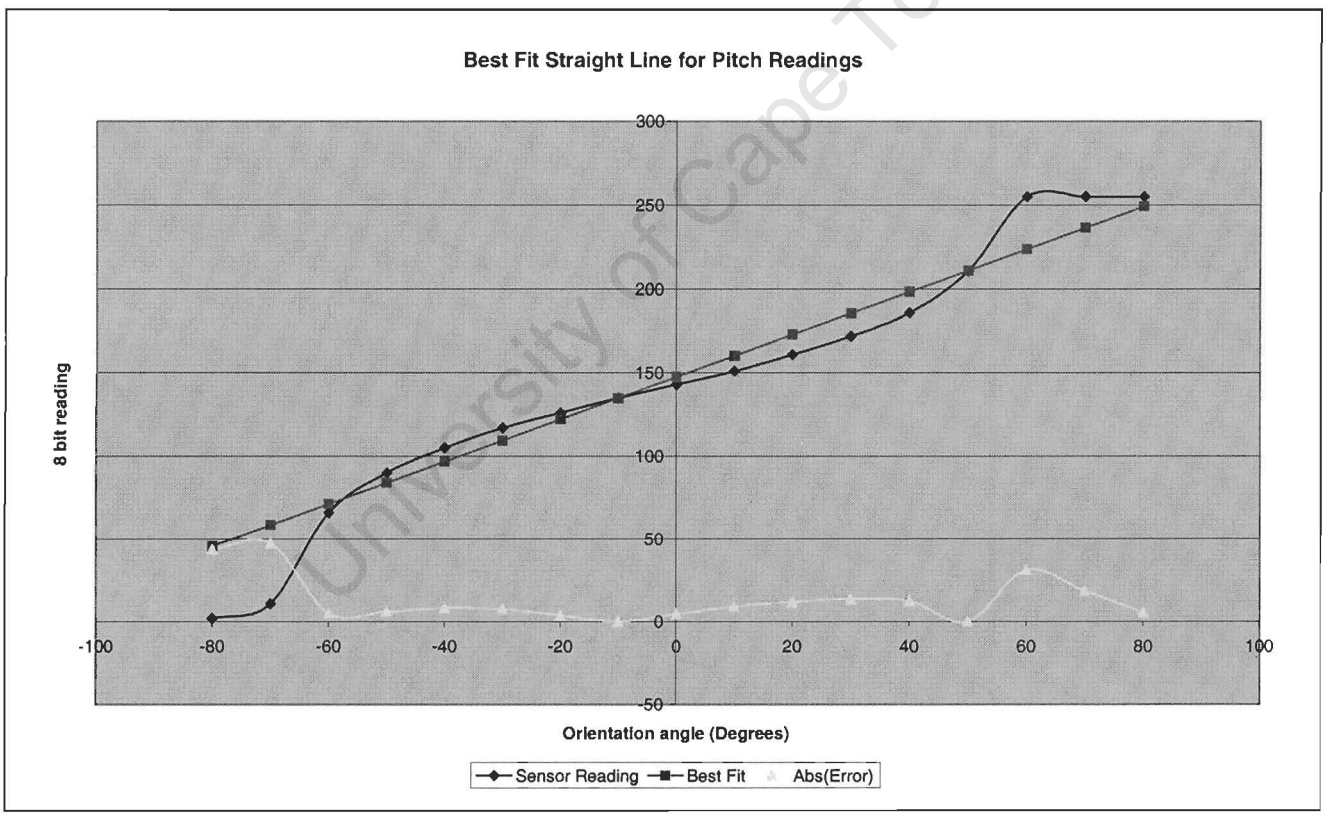


Figure 5.2 : Graph of best fit straight line to pitch readings from tilt sensor.

Next the same is done for the Roll angles. Minimising the mean square error over the range from –60 to 60 degrees, the best fist straight line equation is found to be y = 1.202747x + 122.0769. The graph for this line, as well as a graph of the absolute error over a slightly extended range is shown on the following page. As before the zero point is not on 127, but it is closer than for the pitch reading.

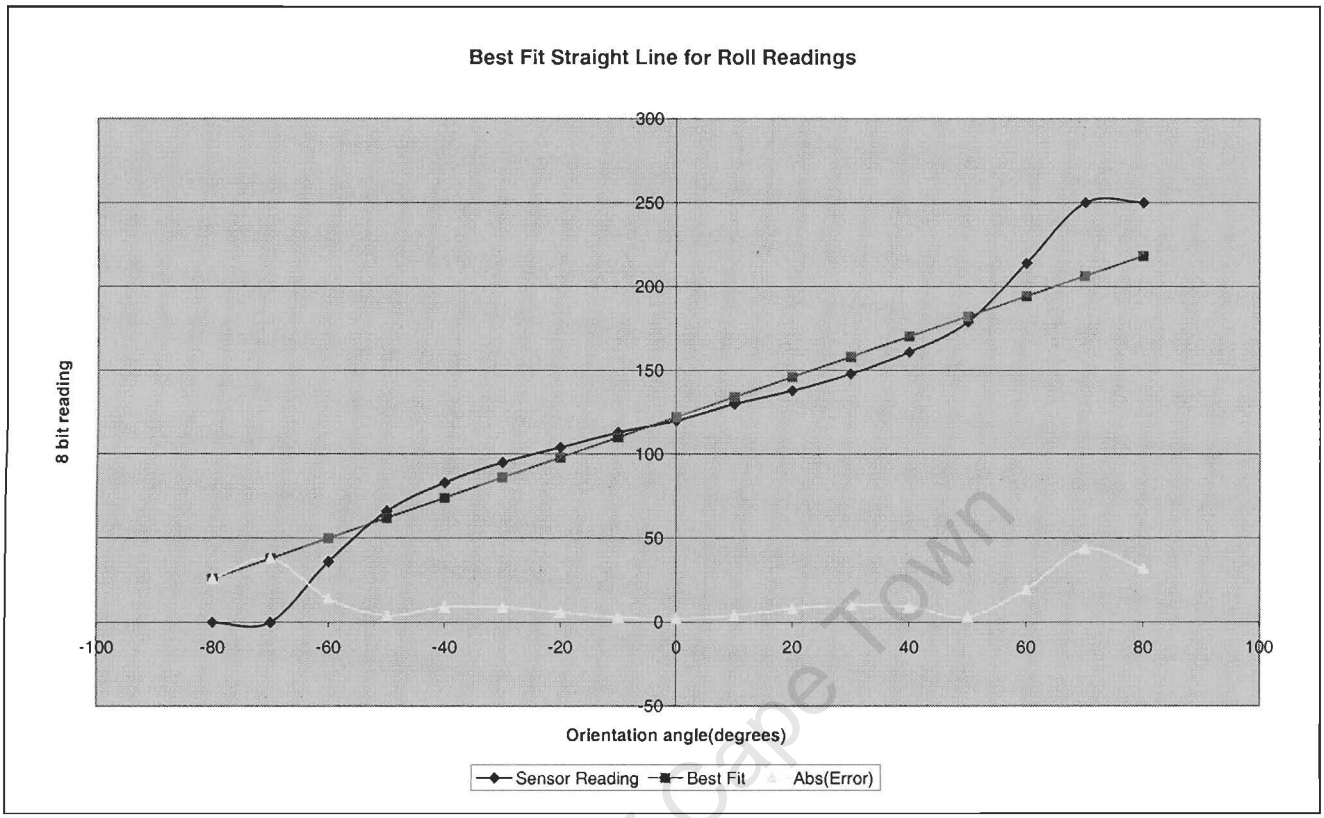


Figure 5.3 : Graph of best fit straight line to roll readings from tilt sensor.

From the two graphs presented above, the first important characteristic that needs to be noted is the sharp escalation of the error outside the range of –60 to 60 degrees. This causes a problem because there is no way to restrict the user's motion to within this range. It is possible to trigger some sort of audio or visual alarm to inform the user when the value read in fall below 50 of above 200 to tell the user that the representation that the computer is providing will no longer be accurate. This is however not a feasible solution within a virtual environment as it detracts considerably from its realism.

The second characteristic that needs to be noted is that the average error over the range is 12° for the pitch measurements and 11° for the roll readings. This limits the accuracy with which it is possible to represent the orientation of the data glove. This is not all together critical in the virtual environment because the user will never be able to compare the orientation discrepancy visually. The user is unable to see his / her hand once the HMD is put on.

This puts up another question that needs to be considered. If a person cannot see his / her hand, then how accurately can he / she tell the orientation of the hand? This could cause problems when the user sees his / her virtual hand as being totally level, when to him / her it feels as if the hand is at an angle to the pull of gravity. My hypothesis would be that for a person to pick up this discrepancy, it would have to be larger than the 12° accuracy of the tilt sensors. However this hypothesis is based purely on my personal ability to judge the orientation of my hands, and may not hold for all people. This aspect is however beyond the scope of this dissertation.

Dead-band

There is not much to say under this heading apart from that the dead band is in the order of one degree for both the roll and pitch readings. Due partly to the lack of very high accuracy angle measurement equipment, but mainly because a one or two degree dead-band is more than sufficient when compared to the low static accuracy, no attempt was made to measure the dead-band more accurately. It suffices to say that the dead-band is small enough to have a negligible, if any, negative effect of the whole virtual system.

Effects of Rotation along Perpendicular Axis

In this section an attempt will be made to find out how changes along one axis affect the readings along the another. To do this, the pitch will be maintained at some constant value, and the roll angle value measured at various orientations. This is illustrated below.

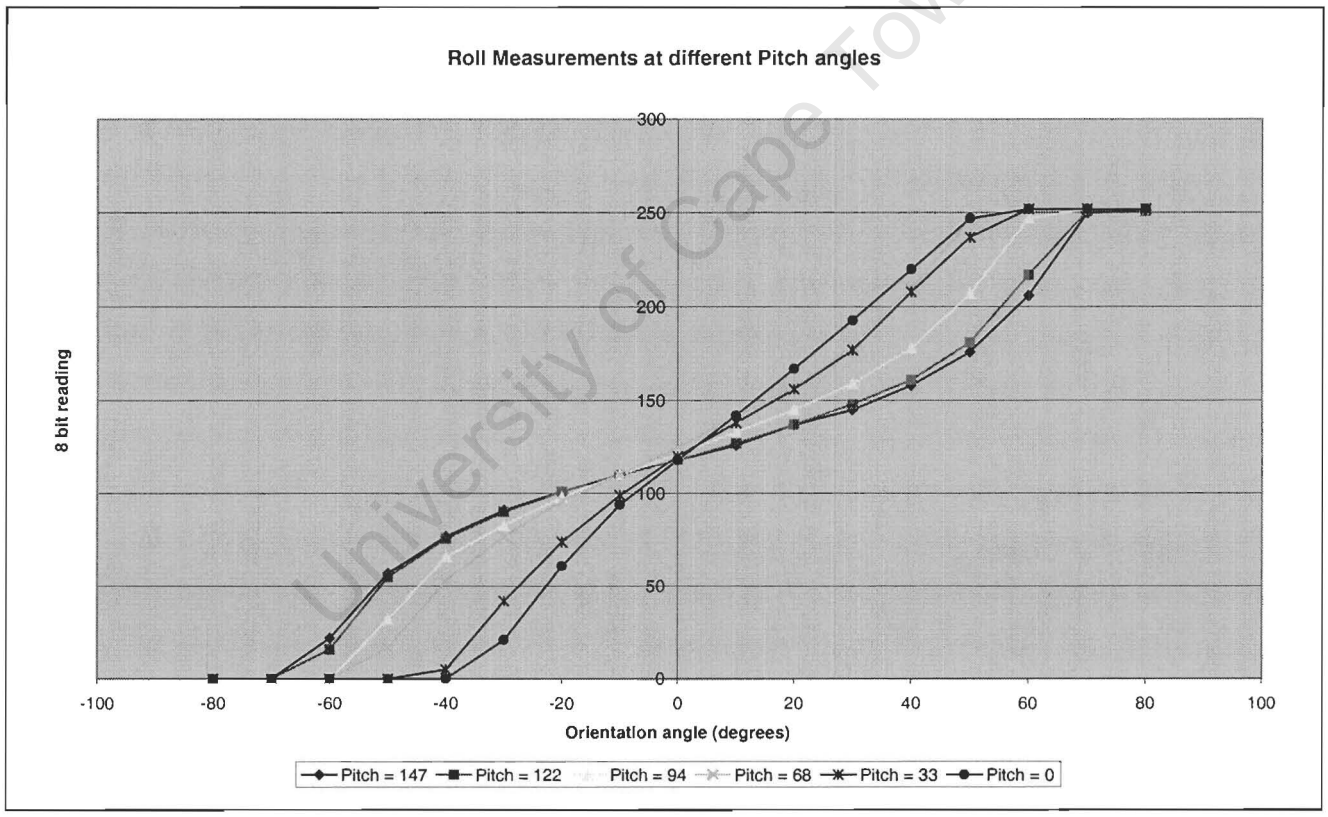


Figure 5.4 : Roll angle readings at different constant Pitch angles.

The above graph shows how the range of the roll readings decreases as the Pitch angle decreases from its zero degree value (Pitch = 147) down to its –60 degree value (Pitch = 0). The plots also become more linear over their new usable range.

Below is a plot of the usable range of the roll readings as a function of the different pitch readings.

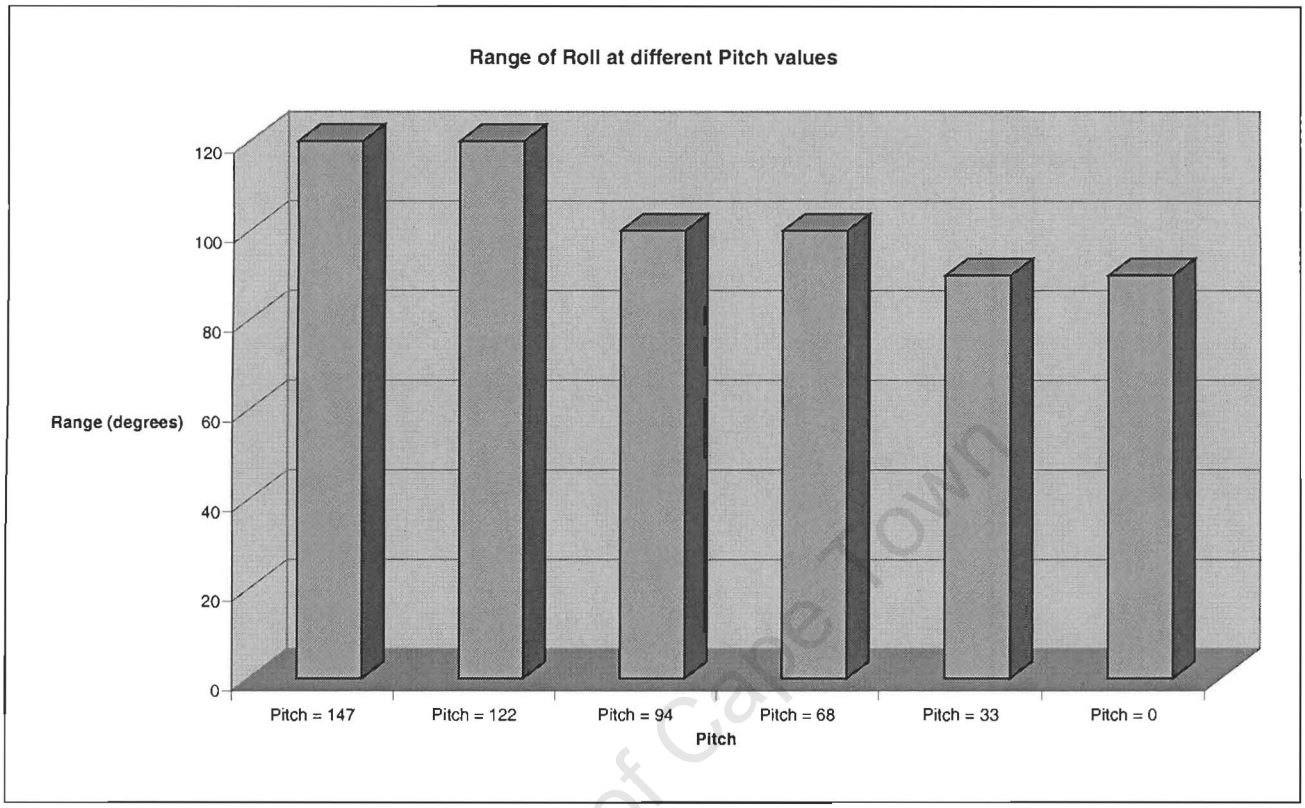


Figure 5.5 : Effect of pitch value on range of roll readings

The next check that is performed is to see what the error is at these different pitch values if the roll readings are approximated using the previously derived equation. The equation is inverted to calculate the orientation angle that the computer believes it is receiving for the roll reading returned from the glove. I.e. the 8-bit reading is converted to an orientation angle using the straight line approximation and compared to the actual orientation angle.

E.g.

Error = absolute ((reading – 122.0769) / 1.202747) – (Actual orientation angle)

The results of this calculation are shown on the following page. The straight line approximation is seen to deteriorate as the pitch angle changes. This causes a problem because the approximation starts becoming invalid. This force the use of a matrix of linearization values, where for each pair of roll and pitch readings that are collected from the data glove there is a function that is linear over a small region around that sample point to calculate the actual orientation angles.

Figure 5.7 shows the severity of the problem where the average error in degrees of the approximation over the decided range of –60 to 60 degrees is displayed.

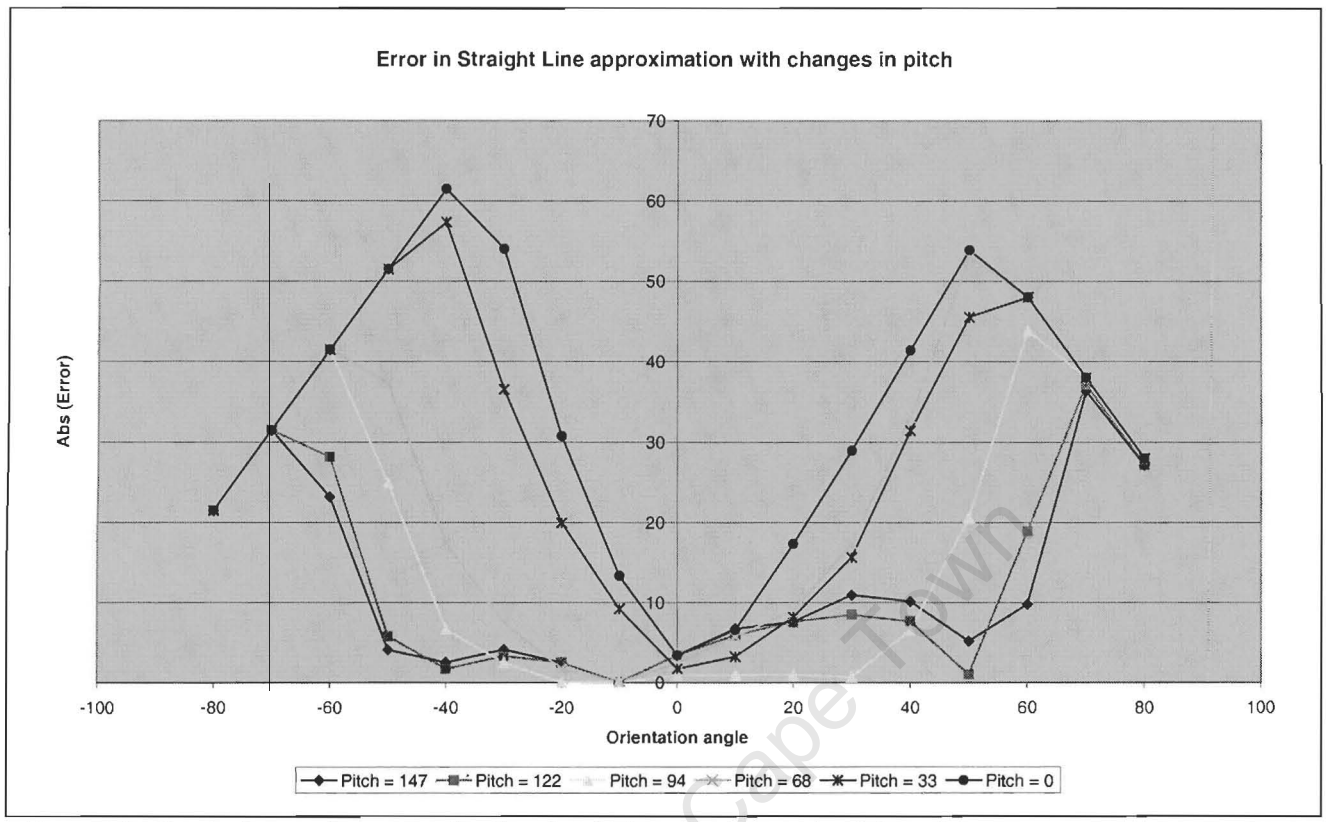


Figure 5.6 : Graph of error of my approximating straight line with changes in pitch angles

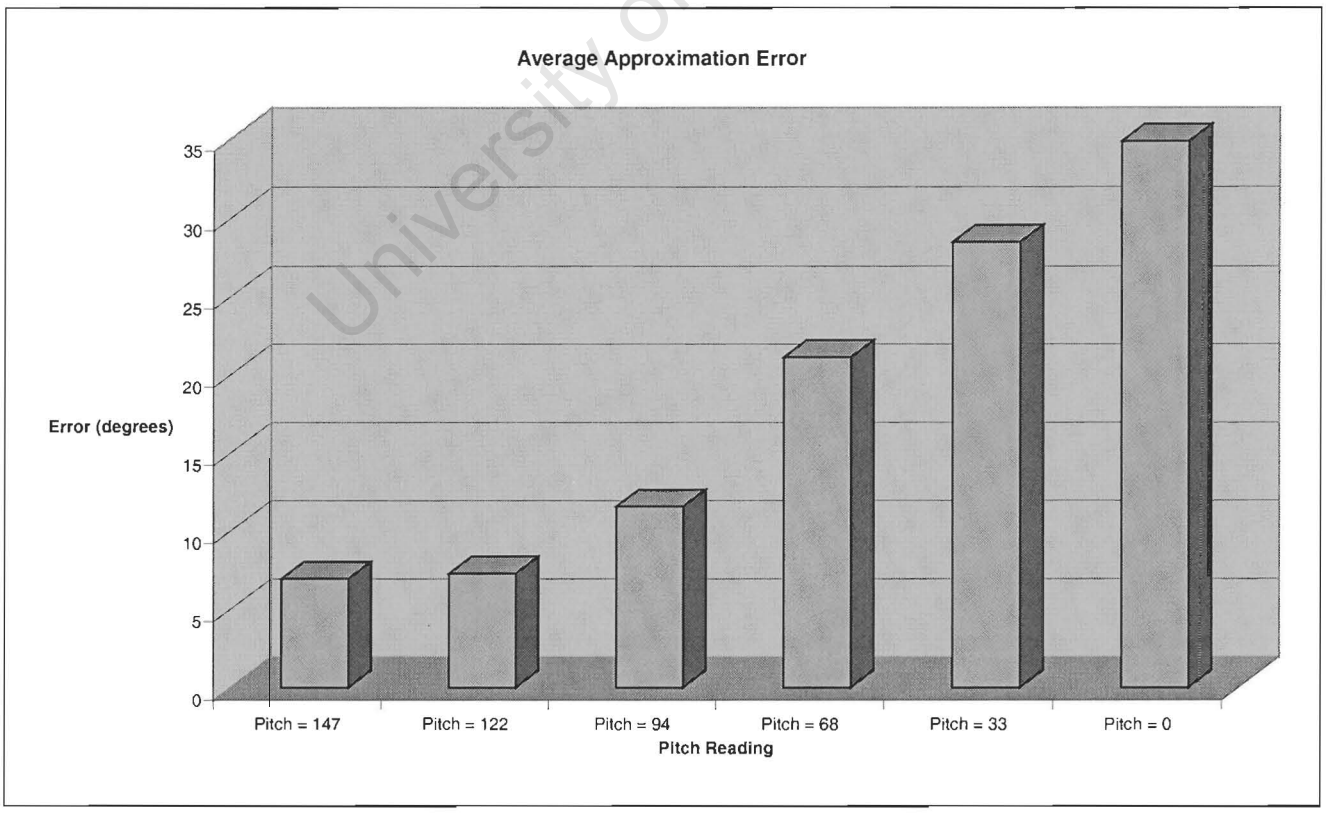


Figure 5.7 : Graph of the average approximation error in assuming a straight line approximation

Typical linearization matrixes based on the matrix of readings recorded for the calculations above look like the two below. Production matrixes would include more readings along the pitch axis, however this should suffice as an example.

M - Matrix Roll \ Pitch	-0.42354	-20.0865	-42.1089	-62.5583	-90.0864	-116.041
-80						
-70	0	0	0	0	0	0
-60	2.2	1.6	0	0	0	0
-50	3.5	3.9	3.2	1.7	0	0
-40	2	2.1	3.4	3.6	0.5	0
-30	1.4	1.4	1.7	2.4	3.7	2.1
-20	1	1.1	1.5	1.9	3.2	4
-10	0.9	0.9	1.2	1.4	2.5	3.3
0	0.8	0.8	1.1	1.5	2.1	2.4
10	0.8	0.9	1.2	1.6	1.8	2.4
20	1.1	1	1.2	1.6	1.8	2.5
30	0.8	1.1	1.4	2.2	2.1	2.6
40	1.3	1.3	1.9	3.1	3.1	2.7
50	1.8	2	2.9	3.6	2.9	2.7
60	3	3.6	4	0.6	1.5	0.5
70	4.4	3.4	0.5	0	0	0
80	0.1	0	0	0	0	0

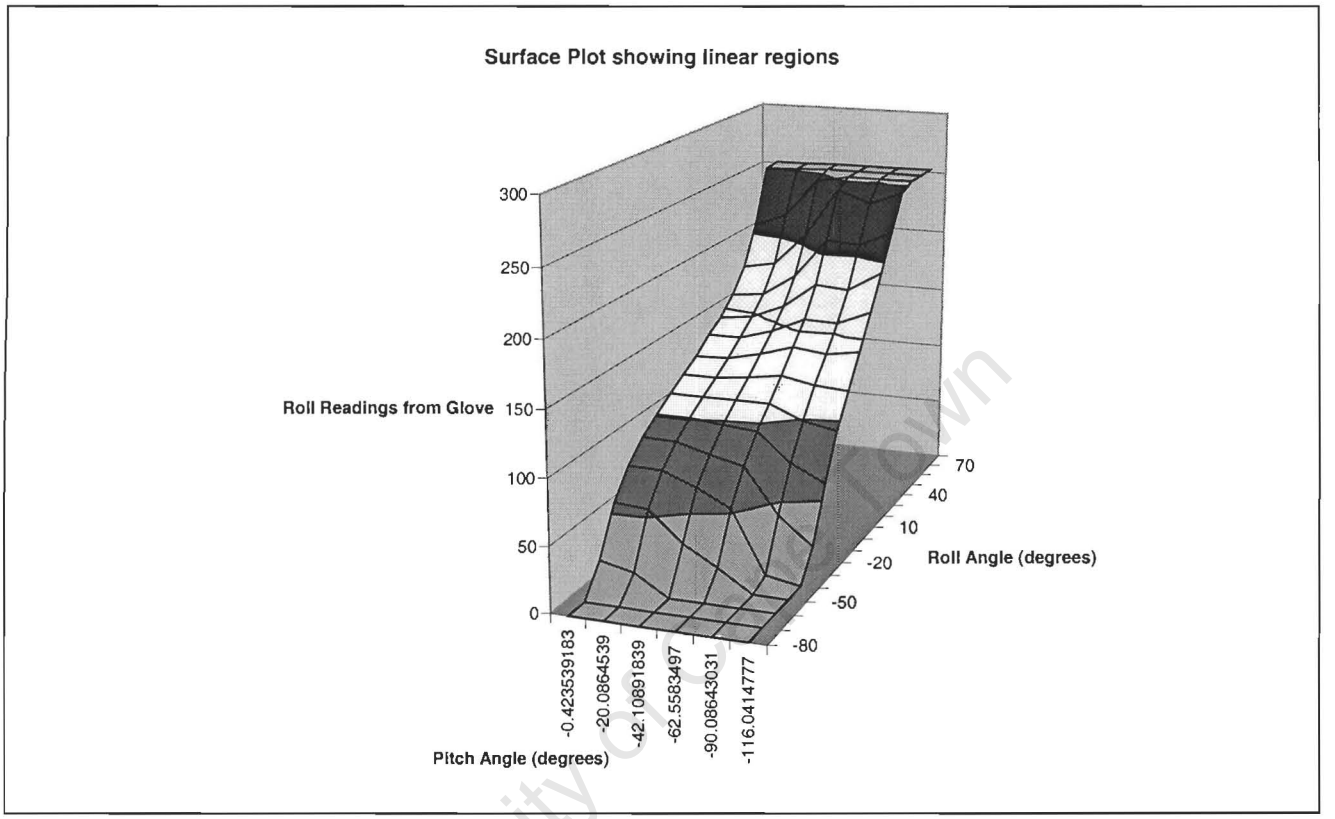
Figure 5.8 : Table of gradient values in linear regions specified by index roll and pitch values

C – Matrix Roll \ Pitch	-0.42354	-20.0865	-42.1089	-62.5583	-90.0864	-116.041
-80						
-70	0	0	0	0	0	0
-60	154	112	0	0	0	0
-50	232	250	192	102	0	0
-40	157	160	202	197	25	0
-30	133	132	134	149	153	84
-20	121	123	128	134	138	141
-10	119	119	122	124	124	127
0	118	118	121	125	120	118
10	118	118	121	125	120	118
20	115	117	121	125	120	117
30	121	115	117	113	114	115
40	106	109	102	86	84	112
50	86	81	62	66	92	112
60	26	1	7	216	162	222
70	-58	13	217	252	252	252
80	243	251	252	252	252	252

Figure 5.9 : Table of intercept values in linear regions specified by index roll and pitch values

To calculate the real roll angle one chooses the closest pitch and roll values in the above matrices and uses them as indexes into the matrices to find the m and c value that forms a linear approximation in that region. These two values are then used in conjunction with the roll reading to get the actual roll angle.

Finally the diagram below shows how the above matrices of gradient and intersection values linearize the area of possible input values into individual rectangles. Each rectangle represents an area over which the straight-line approximation is of reasonable accuracy. This process can be continued to further subdivide the area of interest into smaller regions until the desired level of accuracy is achieved.



System Dynamics

An important part in any technical analysis intended for Control Engineering is to analyse the system dynamics of all the components. This is done by recording the pitch and roll measurements at the maximum sampling frequency. While the recording is taking place, the orientation angle is suddenly changed. This way the dynamic effects to changes in orientation along the axis of measurement (for the pitch sensor) and perpendicular to the axis of measurement (for the roll sensor) are visible.

The data collected for a single run is shown below. Although it is impractical to derive system dynamics from a single run, it does however illustrate how noisy the measurement is. It shows the peaks of errors that occur due to oscillations within the sensor.

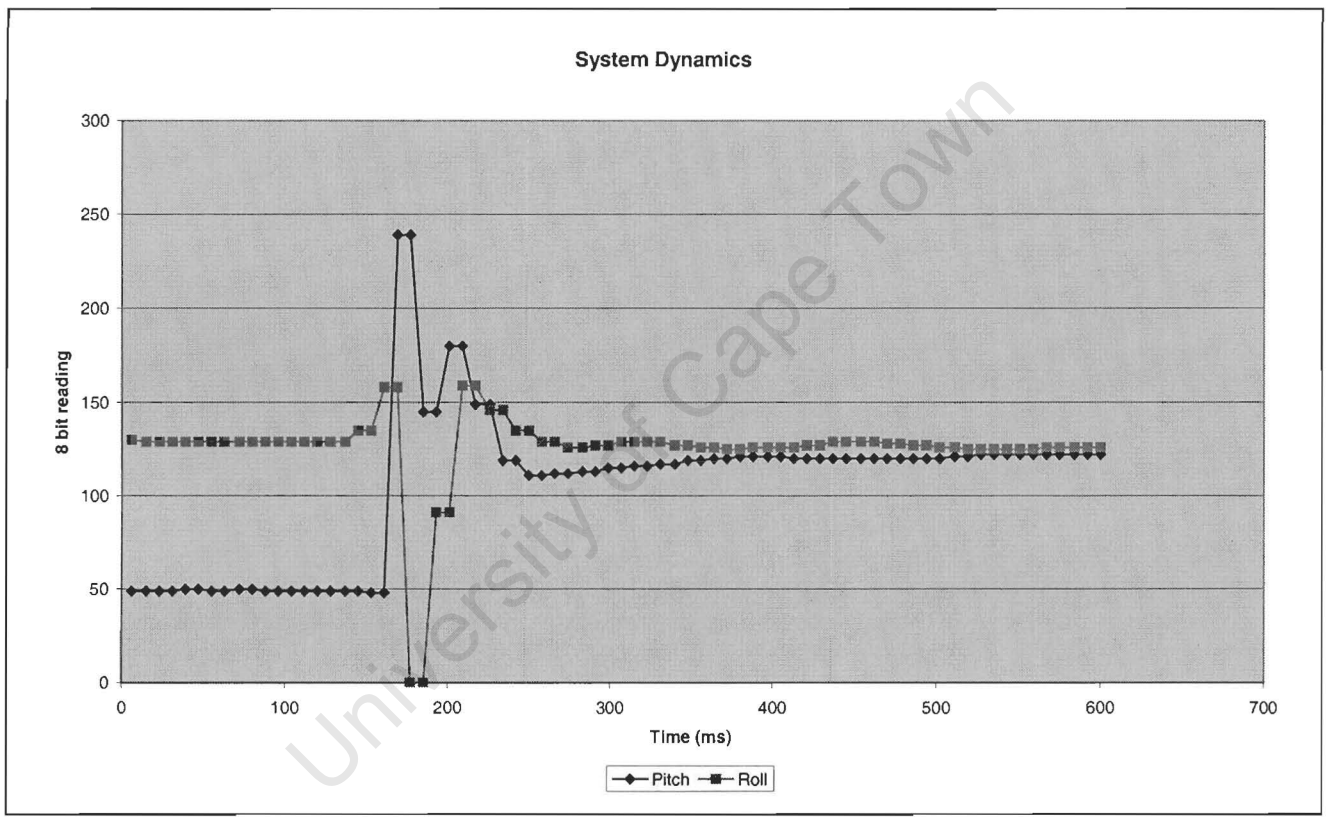


Figure 5.10 : Graph of Pitch and Roll readings for a single run with a step change in the Pitch orientation angle

To calculate a mathematical approximation for the system dynamics a number of runs need to be collected over which some approximate curve can be formed. The graph below shows how the graph shown on the previous page changes after eight runs have been approximated. Note how the peaks have been smoothed out.

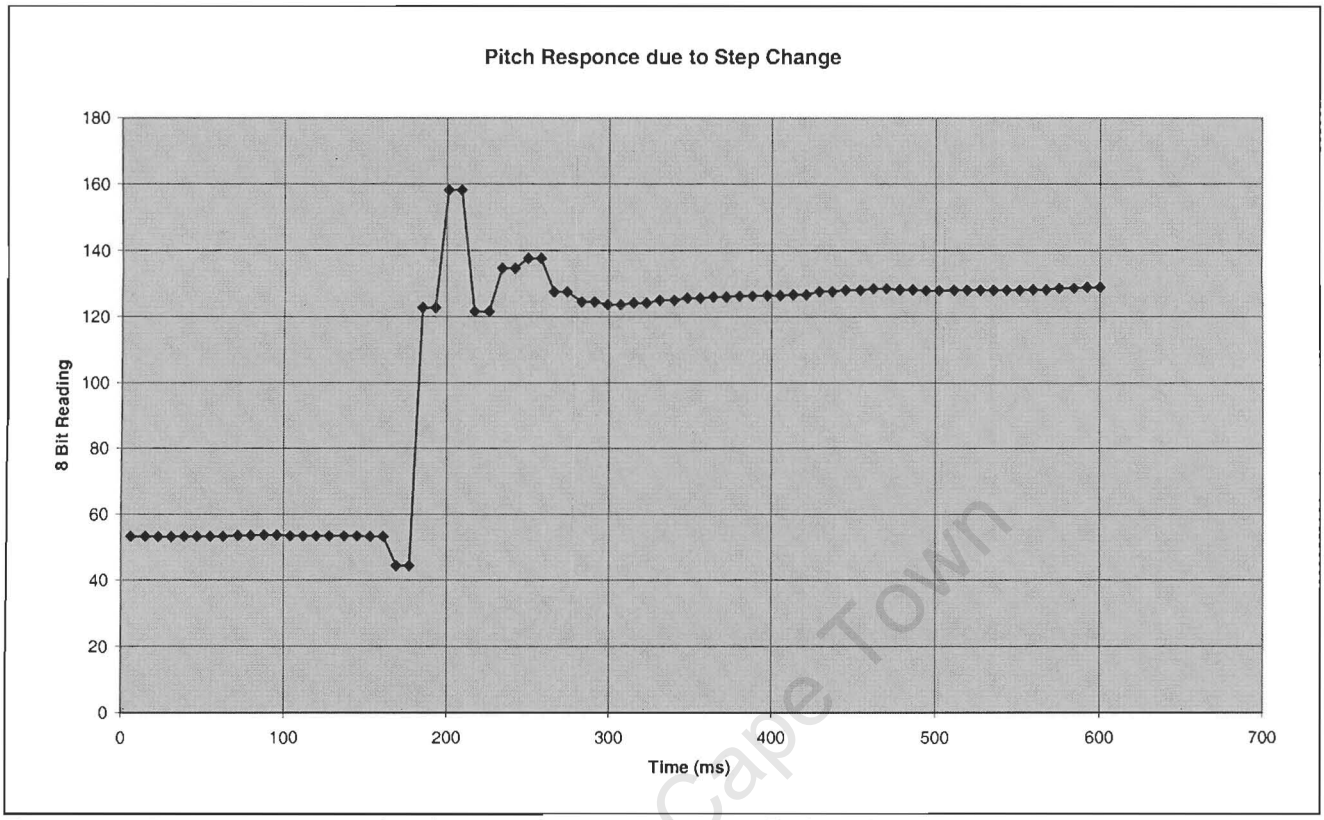


Figure 5.11 : Dynamic Response of Pitch sensor due to step change in Pitch angle

Using Matlab's FSOLVE function together with a short little program a second order approximation to the above curve is found to be :

$$G = \frac{-0.25055s^2 - 0.12575s + 0.2681}{s^2 + 0.09282s + 0.01486} = \frac{-0.25055(s + 0.6631)(s - 0.1614)}{s^2 + 0.09282s + 0.01486}$$, which gives the following

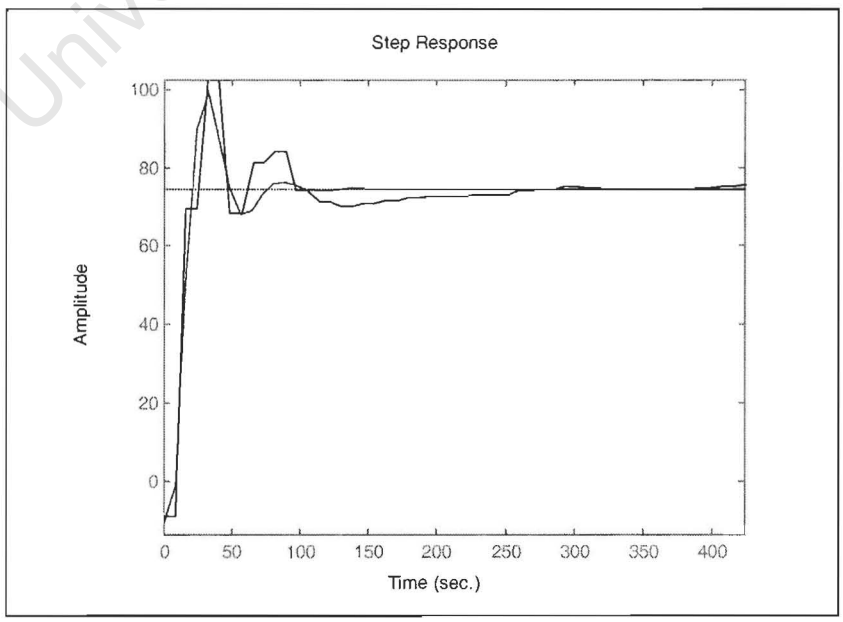


Figure 5.12 : Step Response of modelled function (red) together with original data (blue)

The roll sensor, whose axis of measurement is perpendicular to the step change in orientation, gives the following averaged response. This is a much smoother response, but some dynamic interaction is visible due to the change in Pitch angle.

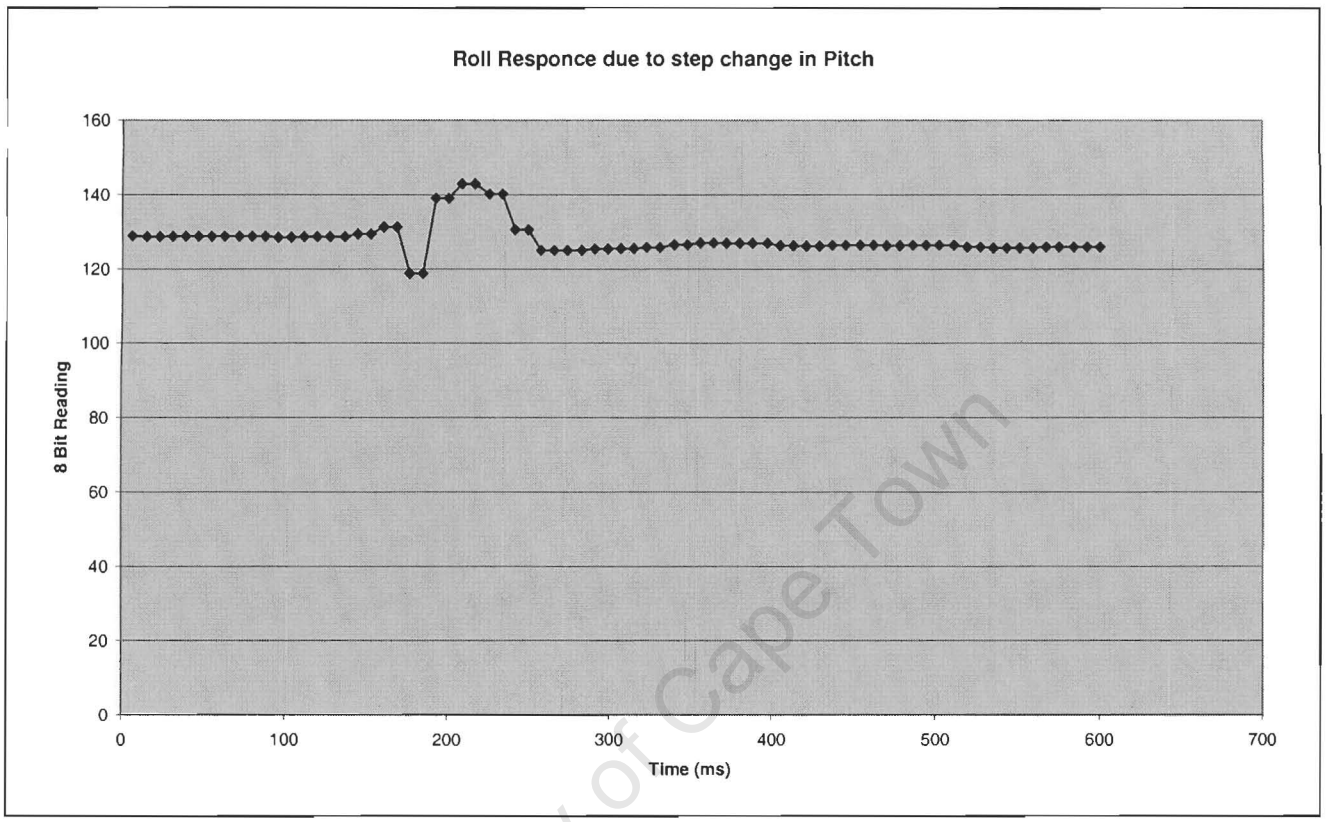


Figure 5.13 : Dynamic Response of Roll sensor due to step change in Pitch angle

Using this data in Matlab, the following transfer function is found:

$$G = \frac{-0.373s^2 + 0.02552s - 0.000183}{s^2 + 0.04326s + 0.002194} = \frac{-0.373(s - 0.06029)(s - 0.008138)}{s^2 + 0.04326s + 0.002194}$$, which gives the following

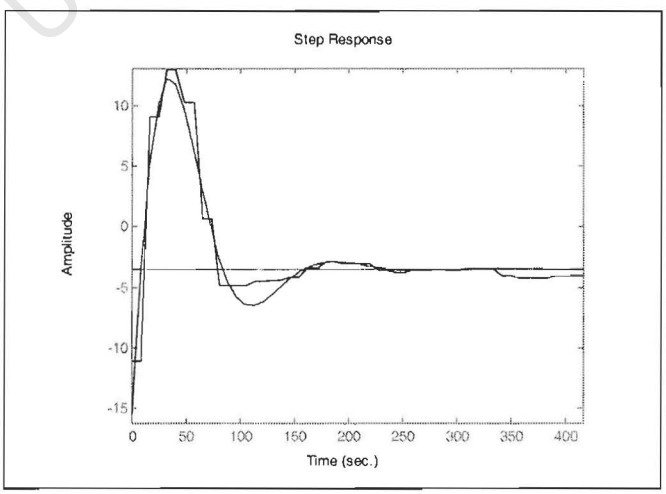


Figure 5.14 : Step Response of modelled function (red) together with original data (blue)

The most important observation that needs be interpreted from the above graphs is that whatever the dynamics are, the response settles down to its final value within 100ms. As this is of similar speed to the screen update rate, system dynamics have a marginal visual effect. Even if an incorrect image of the virtual data glove does appear for a single frame, the user should not notice it, if the frames before and after are accurate.

It can be observed that the sensors were designed for applications that required better accuracy and system dynamics than is necessitated by a virtual environment. Where the sensors are lacking in, is in their range. A user interacting with a virtual environment can justifiably expect the sensors to follow any body motion that he / she is able to undergo. With the range being small, the user immediately notices this limitation.

Effect of acceleration on the readings

The final set of tests that are performed are to see how the sensors react to changes in velocity. Because the sensors are moulded into a plastic case, and because no detailed technical specification on how the orientation sensors work is provided, the method by which the orientation sensors operate is unknown. However a problem that arises with sensors that use gravity is that they are affected by acceleration along other axes. One method to test for this effect is to shake the sensor and record the readings taken. By shaking the sensor while maintaining it orientation level, the sensor should ideally keep its readings constant as no change in tilt occurs. The shaking is done manually by moving the sensor along one axis with an amplitude of motion of one centimetre and period of motion of about five Hertz.

Motion along line of sensor's swing

The first part will test how the sensors reacts to acceleration along the axis that the sensor would swing, were it a pendulum type tilt sensor. The graph on the following page shows the measurements of one of the tilt sensors, while the sensor was being shaken. Even though the orientation angle was maintained constant, the sensor still records changes in orientation. This would indicate that some pendulum type motion is occurring in the sensor and that the sensor is swinging free in response to acceleration.

Assuming that the oscillations occur in a sinusoidal manner it is possible to perform some basic calculations on the jitter that is induced onto the tilt measurement system. The assumption of sinusoidal oscillations is permissible because when the sensor is manually moved between two points, in a repetitive motion, then it will also experience the greatest velocity at the centre between the two points, and the greatest acceleration at the two end points when it changes direction. A rough estimate of the maximum acceleration is being looked for.

Starting by differentiating the position to get the velocity, and differentiating the velocity to get the acceleration, Figure 5.16 illustrates this process that occurs. This translates into a maximum value for the acceleration of about 10 m/s^2. This is similar to the acceleration experienced due to gravitational forces. Considering that the motion, and thus the acceleration, is perpendicular to the gravitational axis, a pendulum type sensor will be at an angle of 45 degrees to both the gravitational axis and to the axis of motion. A tilt of between –45 and 45 degrees in the tilt sensor should provide a reading of between 75 and 170 (based on the values shown in Figure 5.1), which is similar to the range of readings recorded in Figure 5.15 below. This causes a potential problem in that fast motion of the hand from one position to another will be interpreted by the sensors as a rocking of the hand. I.e. the hand will tilt forwards when motion starts and then tilt backwards when motion ends.

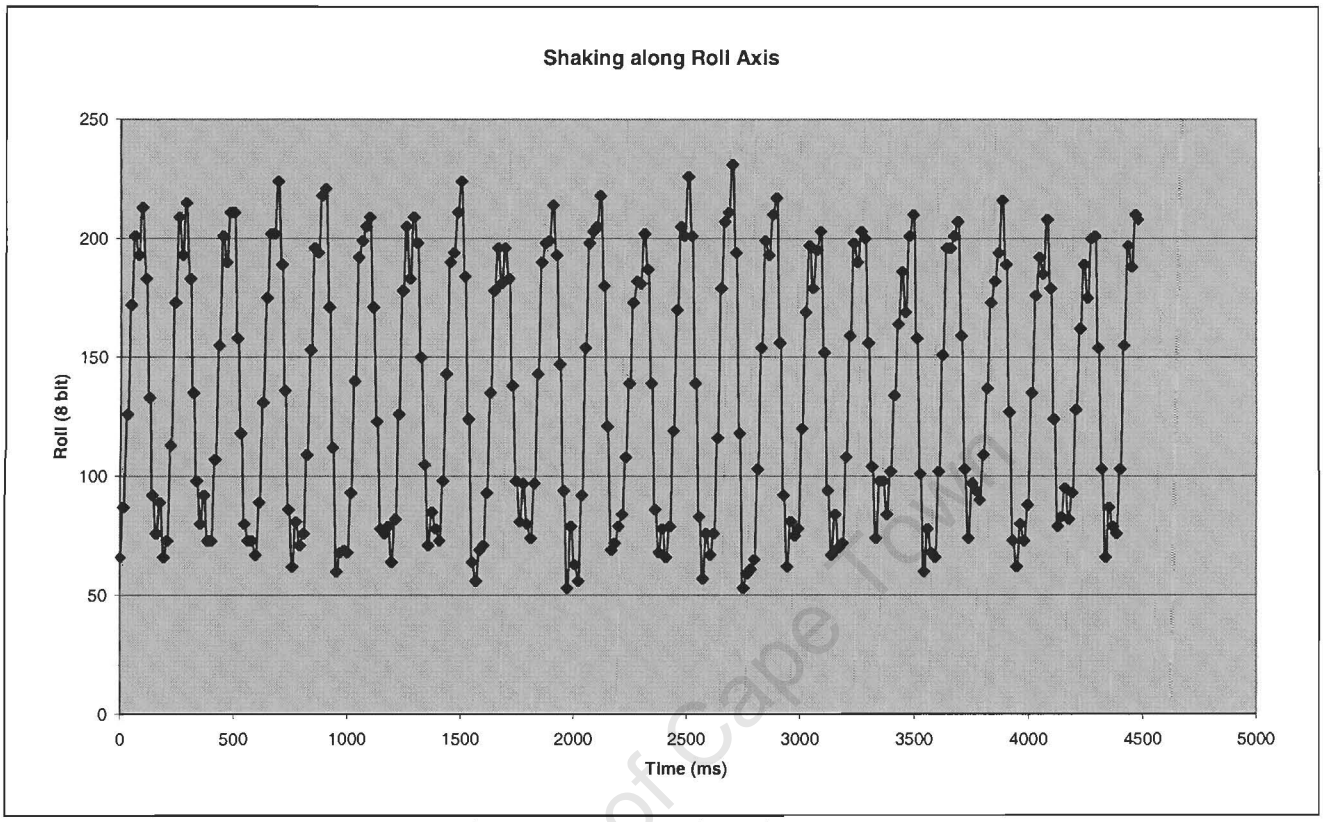


Figure 5.15 : Graph of roll readings with shaking along roll axis taking place.

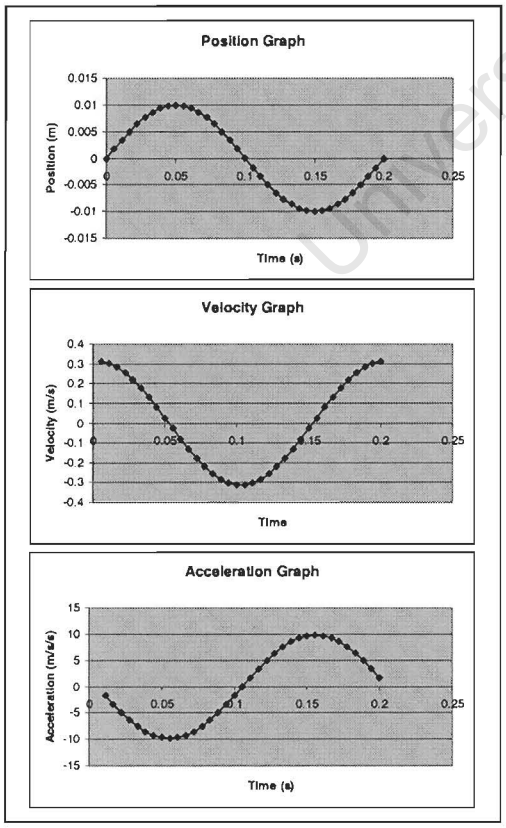


Figure 5.16 : Finding the acceleration placed on the sensor.

Motion perpendicular to line of sensor's swing

A check to see if this jittering has any effect on the other sensors reading is now performed. For the pitch sensor, the jitter occurs along an axis that it has no capability of measuring acceleration for. Thus, as shown in the graph of results below, the jitter is interpreted as noise along the pitch axis. This noise can be occurring due to a number of reasons:

- either from the sensor bumping against internal guide rails,
- or the motion of the sensor is not purely along one axis (the most likely explanation),
- or the sensors are not mounted exactly at 90 degrees to each other.

Considering that the noise has an amplitude of only 5 (i.e. less than 4%) it can be concluded that jitter occurring perpendicular to a sensors measurement axis has a marginal effect on the reading.

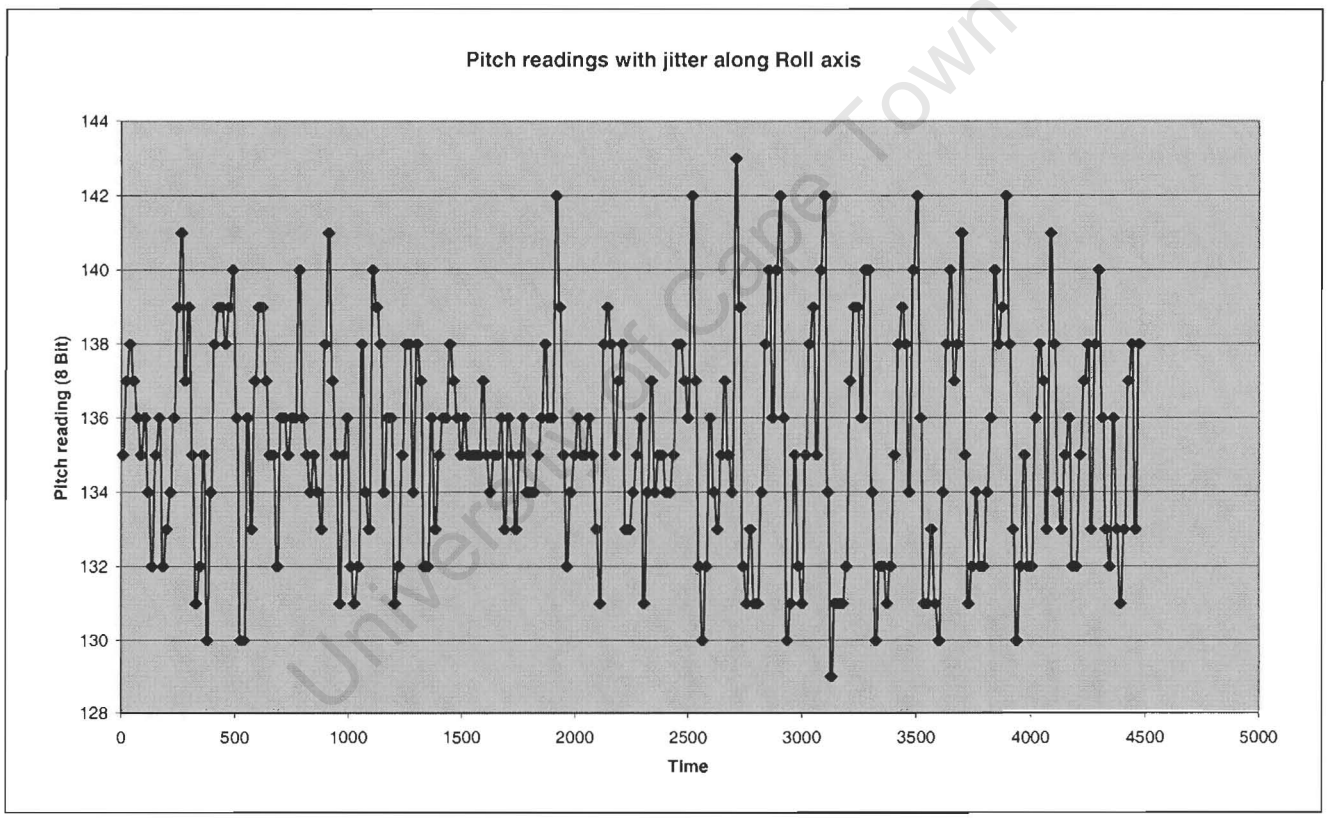


Figure 5.17 : Graph of Pitch readings with jitter taking place along the Roll axis.

5.1.2 Fibre Optic flexure sensors

The other sub-system forming part of the data glove is made up of five fibre optic flexure measurement sensors. This sub-system is responsible for measuring how much each of the four fingers and the thumb are being bent. It works using a light emitting diode (LED) that transmits light through a fibre optic cable that runs along the outside and round a finger back to a receiver at the other end of the fibre optic cable.

When the finger is bent, the fibre optic cable bends with it. In bending the transmissive optics of the fibre optic cable change, and less light returns to the receiver. The return value is digitised and linearlised by the on-glove microcontroller. The linearlisation procedure converts the digitised returning light strength to a zero when the finger is straight and 255 when the finger is fully bent. This calibration is factory performed on some standard or average finger.

This section discusses what these fibre optic sensors can, and cannot measure. It also follows a discussion on how suitable or adequate the data glove is for the intended function of interaction within a virtual environment.

Range of measurement

As the data glove measures the amount of light that returns down the fibre-optic path, anything that can change the transmissive optics of the cable will be registered as a measurement. Cuts and breaks, and dirt entering between the cable and either the transmitter and receiver can be assumed to be fault conditions and not part of daily measurements that are taken. This leaves squeezing and bending of the cable as the only other methods to reduce light transmission. The user has no real control over any squeezing of the cable, so it will have to be taken as an input disturbance that is a function of how tightly the glove fits the hand.

The amount of light that is transmitted through a fibre optic cable depends both on the amount of bending (e.g. 90 or 180 degrees), and also by how sharp the bending is (i.e. the angle of curvature). Both of these interact to give a bending value out of the data glove. The graph below displays the measured readings at different bending angles and different angles of curvature.

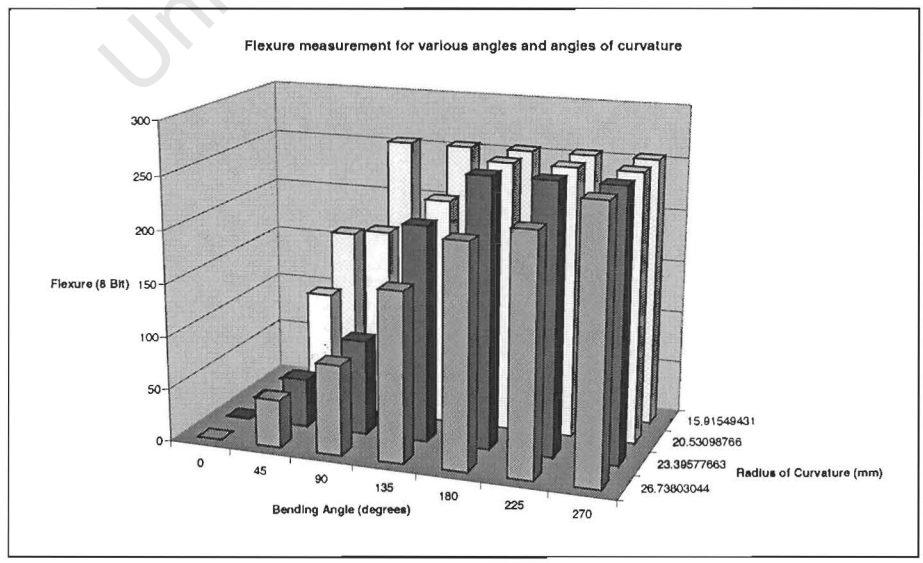


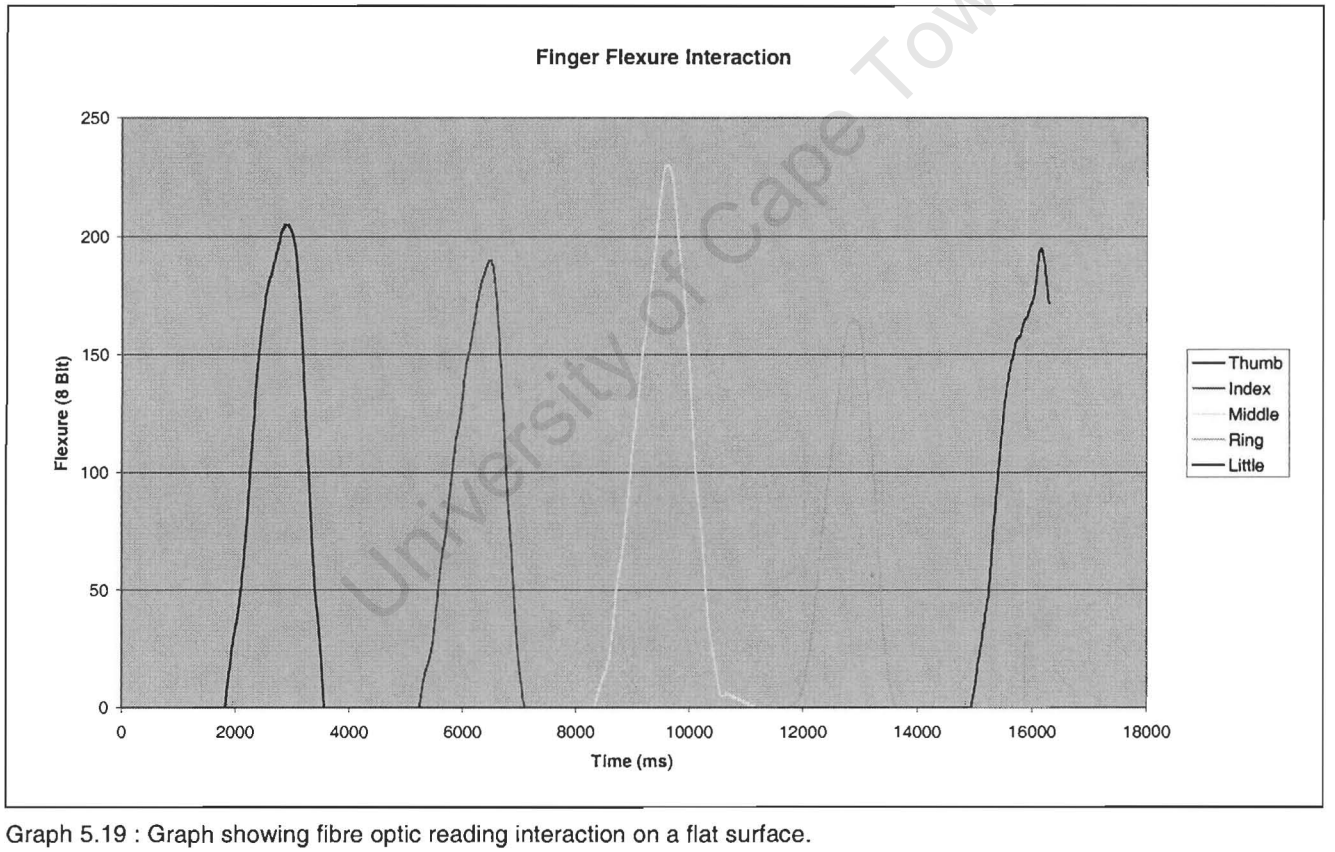
Figure 5.18 : Graph of how angle of bending and curvature radius affect flexure measurement

The graph above presents a logical view of what is expected. The more the fibre optic cable is bent, the greater the flexure measurement. Similarly, the sharper the angle (small curvature radius) around which the cable has to bend, the greater the flexure measurement for a fixed angle of bending. The effect that this has will be seen later in this section, with the illustration of how the glove reacts to different types of hands.

Interference between fingers

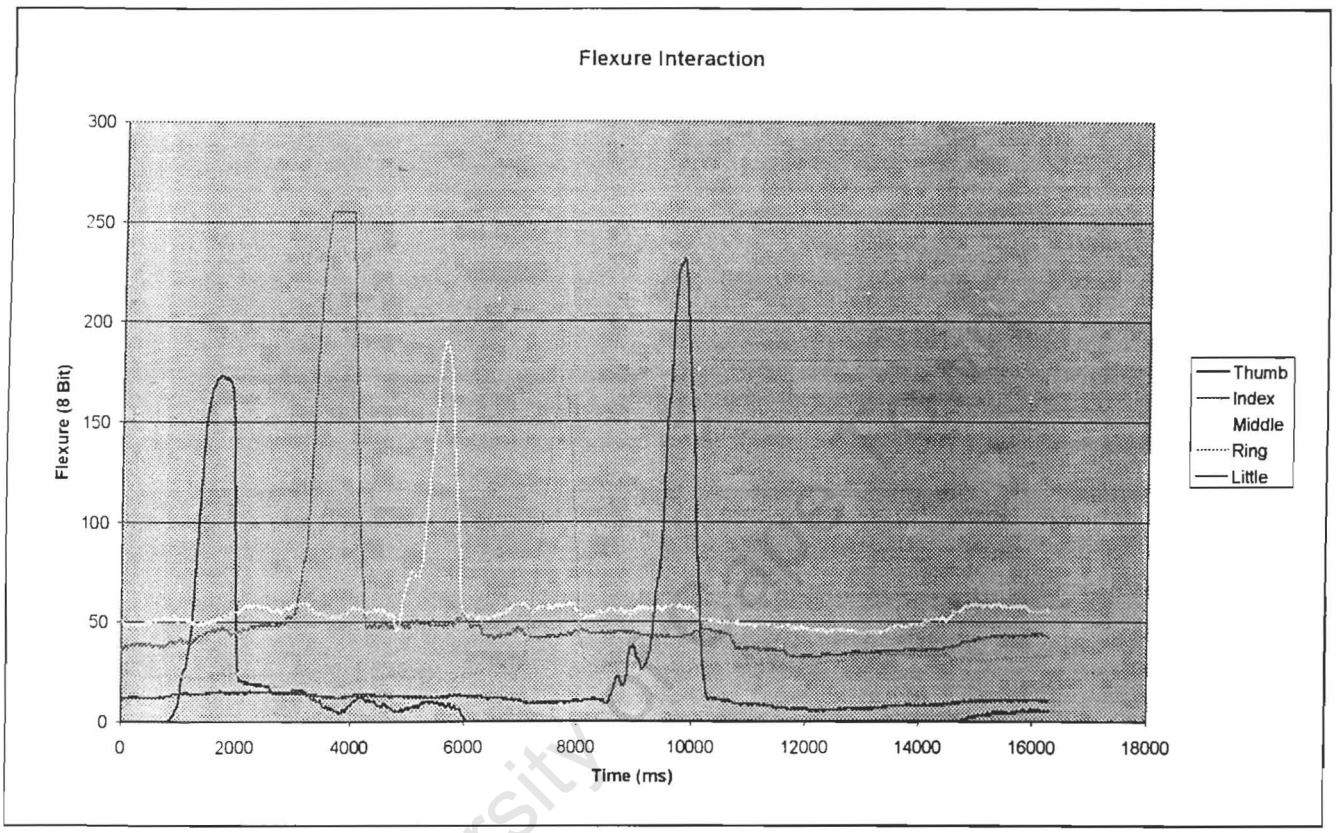
This section investigates to see if there is any cross coupling between the fingers, i.e. if a single finger is moved, is there any reading registered on any of the other fingers? This is done in a two-stage process. Firstly is there any interaction due to the design of the glove? And secondly, is there is any interaction due to the structure of the hand?

For the first case, the glove is kept lying on a flat table, and while keeping all the fingers level, each finger is individually lifted up and lower. This gives the diagram below.



Graph 5.19 : Graph showing fibre optic reading interaction on a flat surface.

In the second case the flexure measurement interaction is measured while the user is wearing the glove. These results are not as important as the previous set, as it is very rare that a person will be able to bend a single finger while keeping all the others stationary. Even if possible, there is hardly ever the need for such interaction with the virtual environment. Again there is negligible interaction between the finger flexure readings.



Graph 5.20 : Graph showing fibre optic reading interaction on a users hand

Practical Accuracy of Flexure Sensors

There is only one sensor per finger, whereas each finger has three joints along which it can bend. How then does one decide on what image to display for a specific finger flexure measurement? It is not possible to be accurate for every single possibility of joint positions, so an appropriate representation needs to be chosen.

One possible solution is to have the first 85 values correspond to the motion from 0 to 90 degrees of the first joint. From 85 to 169 the motion of the second joint is represented and the last 86 values represent the last joint. This dictates that as a finger slowly starts bending, then the image displayed will first bend at the first joint. Once the first joint reaches an angle of 90 degrees the next joint will start bending, and finally once that is also at 90 degrees the last joint will start bending.

If a person opens their hand and slowly close it, it can be seen that a more logical assumption would be that all the joints bend equally throughout. In effect the flexure measurement is taken and the full value is translated into a 0° to 90° range. This angle is the angle at which all the joints are drawn. The problem is that not everybody closes their fists in the same manner, neither do people only close and open fists with their hands. This is however the best model that can be used to represent fifteen variables with five measurements.

Now after calibrating the data glove to display an open hand when the user's hand is open, by shifting the zero offset of the readings and to display a closed hand when the user's hand is closed, by changing the gain of the readings, then will the intermediate images of the hand between fully open and fully closed still be accurate? As there is no quantitative way to evaluate the difference in closure of two hands, I will present the reader with a couple pairs of images. The images on the left are of the user's hand, the images on the right are those of the data glove. This enables a more qualitative study of the accuracy.

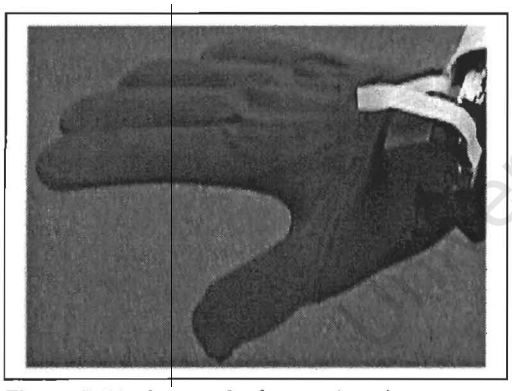


Figure 5.21 : Image A of users hand

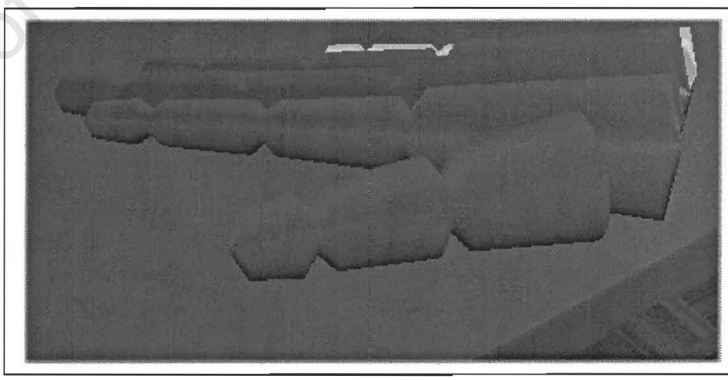


Figure 5.22 : Image of virtual glove corresponding to Image A

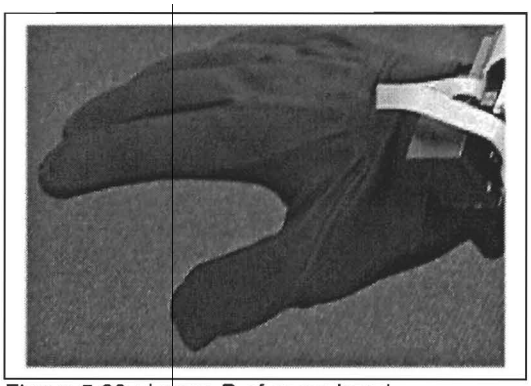


Figure 5.23 : Image B of users hand

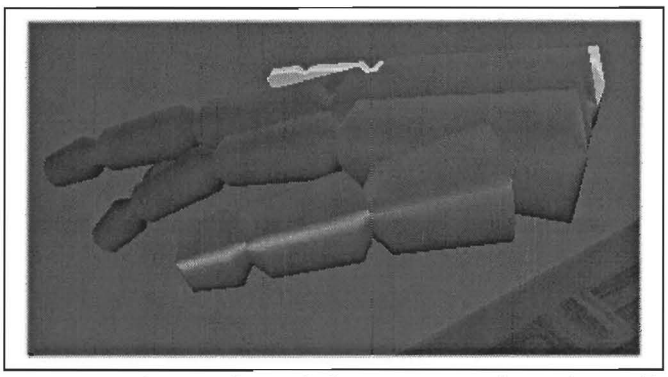


Figure 5.24 : Image of virtual glove corresponding to Image B

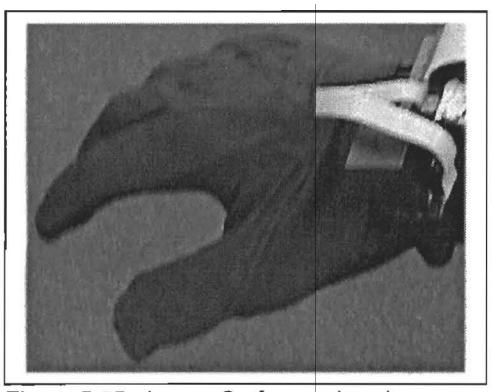


Figure 5.25 : Image C of users hand

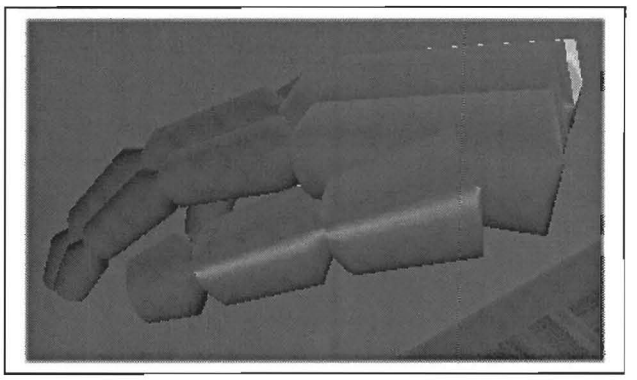


Figure 5.26 : Image of virtual glove corresponding to Image C

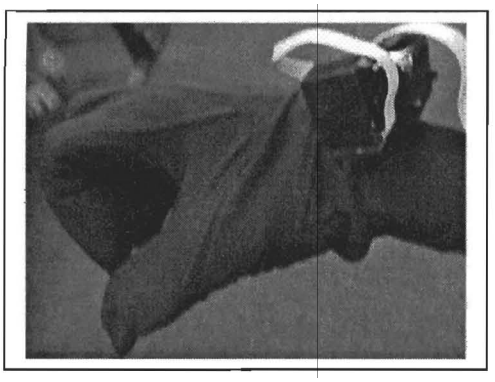


Figure 5.27 : Image D of users hand

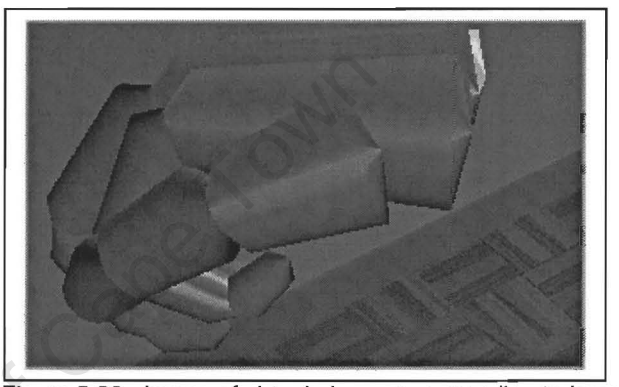


Figure 5.28 : Image of virtual glove corresponding to Image D

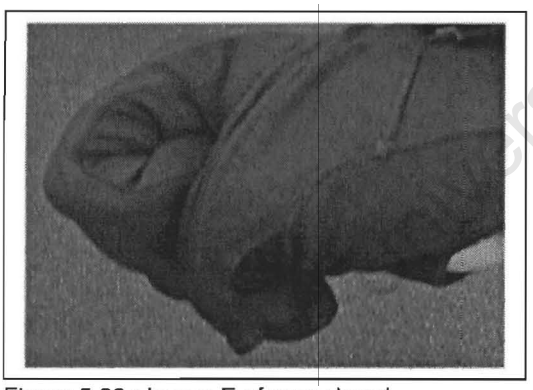


Figure 5.29 : Image E of users hand

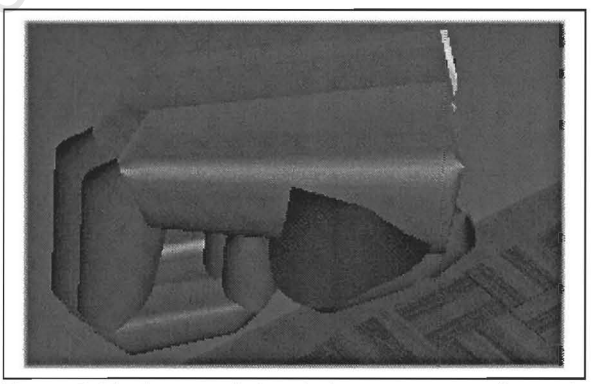


Figure 5.30 : Image of virtual glove corresponding to Image E

From the images above we see that the virtual glove is effectively able to follow the motions of the user's hand. This is however the best case scenario. It will be shown in the following section how the image of the virtual glove can deviate from the position of the user's hand inside the data glove.

Inaccuracy of finger bending assumptions

The above images were generated assuming that all the joints in a finger bend to the same angle. This is not always the case. Below are a few cases there this assumption falls apart.

- Finger push down position (only the third joint is bent)

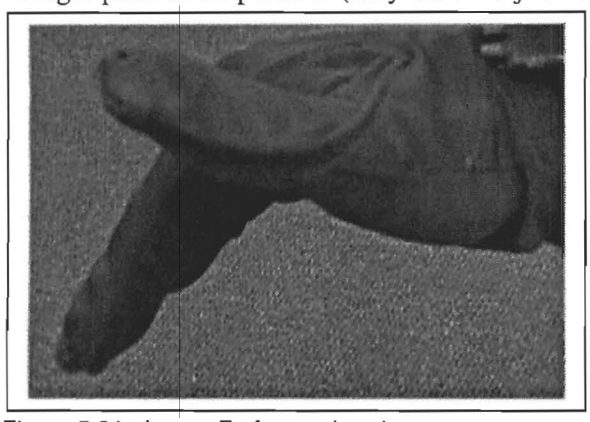


Figure 5.31 : Image F of users hand

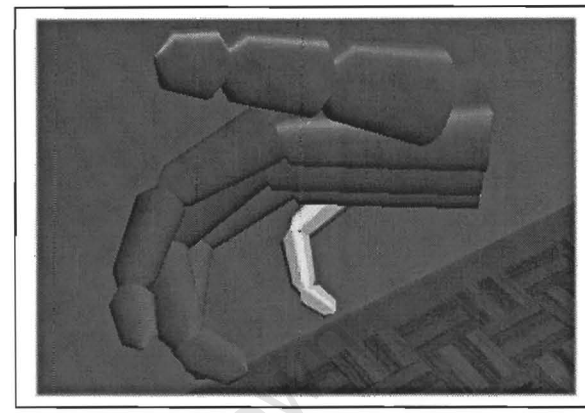


Figure 5.32 : Image of virtual glove corresponding to Image F

- First joint hook position (only the first joint is bent)

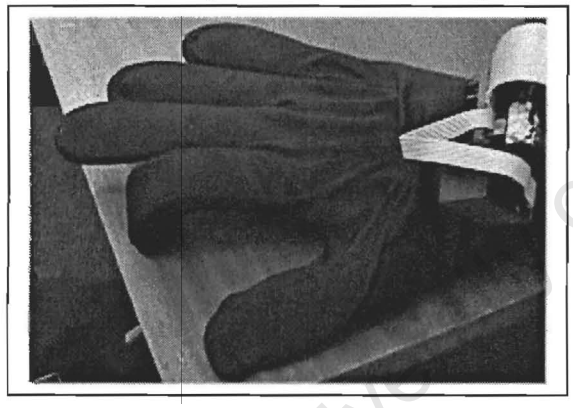


Figure 5.33 : Image G of users hand

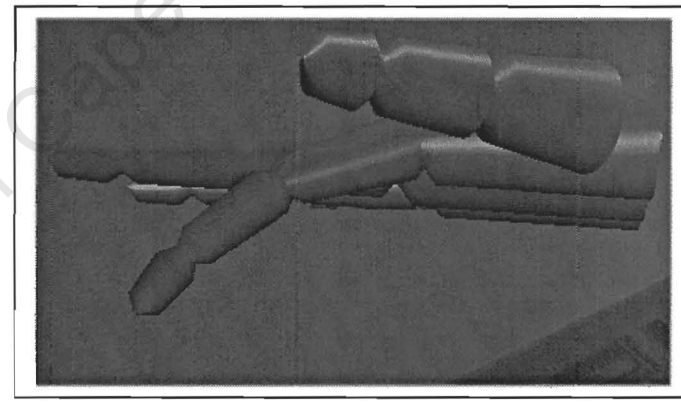


Figure 5.34 : Image of virtual glove corresponding to Image G

- Looking at nail position (third joint of all the fingers is unbent)

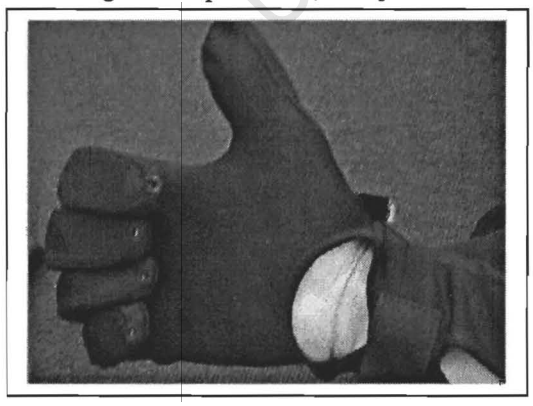


Figure 5.35 : Image H of users hand

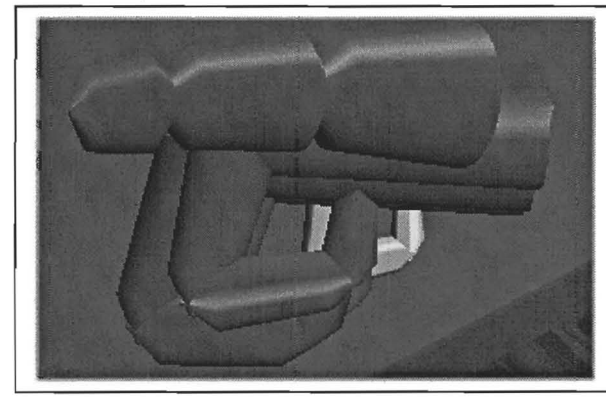


Figure 5.36 : Image of virtual glove corresponding to Image H

From these three pairs of images it is seen that using a single sensor to represent three degrees of freedom can generate inaccurate images.

Motion detection limitations by data glove

The above images showed how the flexure value read from the data glove is misinterpreted. There are however also other joints and muscles in the hand that can perform motions that are even undetectable by the data glove. The following three pairs of images show how some movements are not registered by any sensor on the data glove.

- Thumb next to hand position (sideways movement is interpreted as bending)

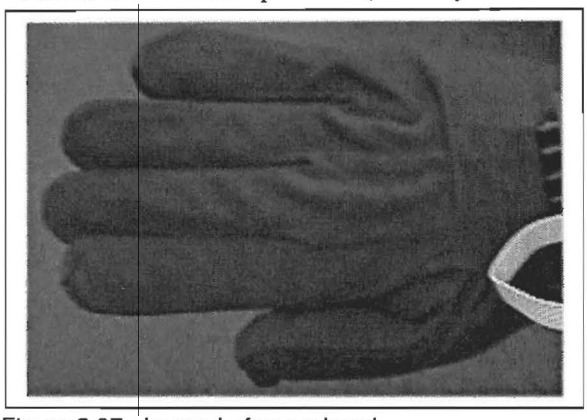


Figure 5.37 : Image I of users hand

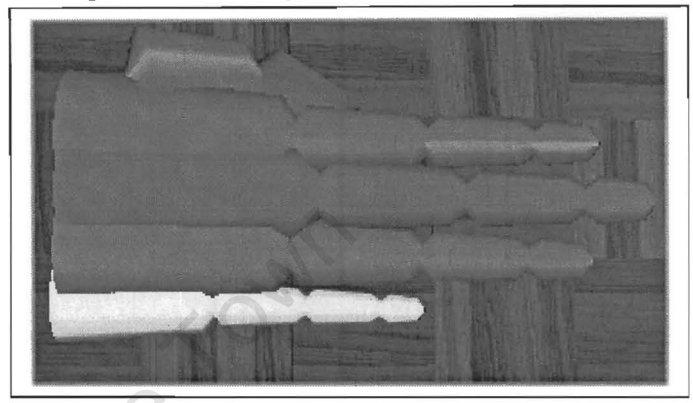


Figure 5.38 : Image of virtual glove corresponding to Image I

- Finger spread

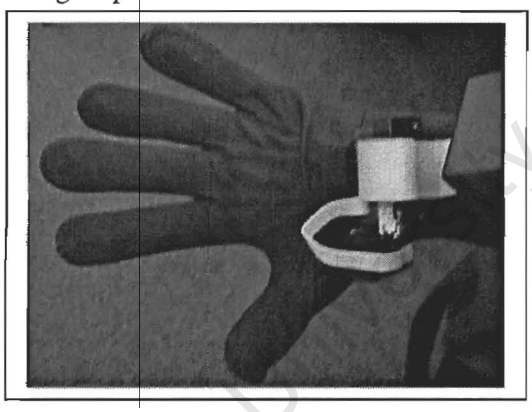


Figure 5.39 : Image J of users hand

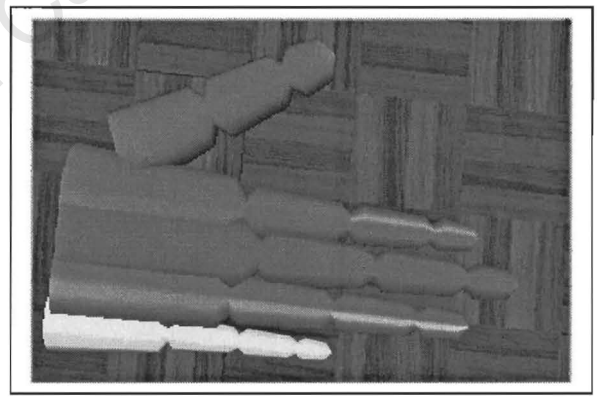


Figure 5.40 : Image of virtual glove corresponding to Image J

- Thumb touching little finger

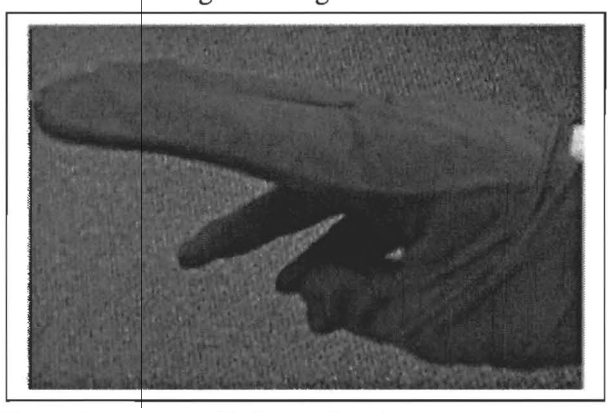


Figure 5.41 : Image K of users hand

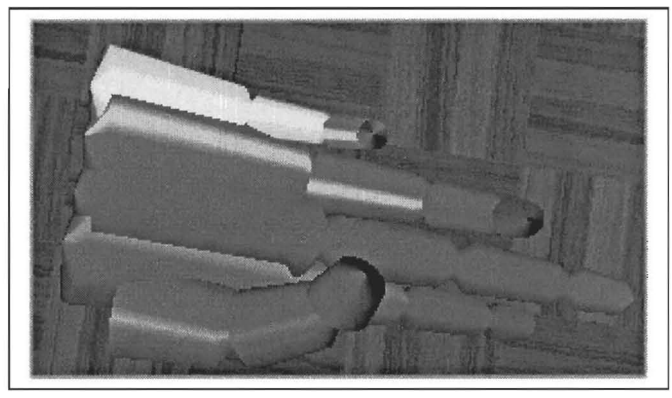


Figure 5.42 : Image of virtual glove corresponding to Image K

Effects of different hand sizes and shapes

Another factor that needs to be taken into account is that not all hands are the same. It is possible to recalibrate the system for different pairs of hands. It is even possible to store these calibration settings for each and every person. Even with the above two solutions, re-calibration to account for changes in the shape of the users' hands needs to take place.

The one-size fits all approach of the data glove, is more like a one-size fits no one in practice. The reason is that a glove that does not fit snugly, has to be re-calibrated each time the user takes its off, as it is highly dependent on the current fit. A loosely fitting glove causes up to 20% of the range having to be used for a zero offset, depending on how tightly it has been pulled on.

This section shows how the image of an open hand appears on three different hands after it had been calibrated for my hand. The hands are classified according to the length of the middle finger, which for me is 7.8cm. The images show that different hands bring up different open-hand images.

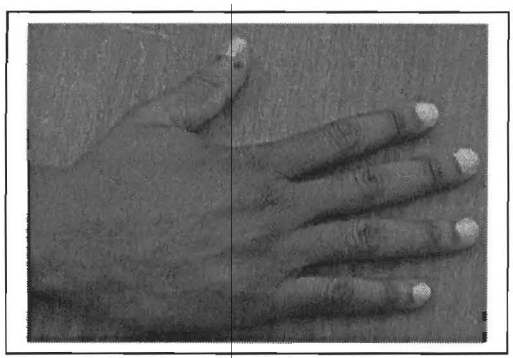


Figure 5.43 : Hand A with Index Finder length = 9.2 cm

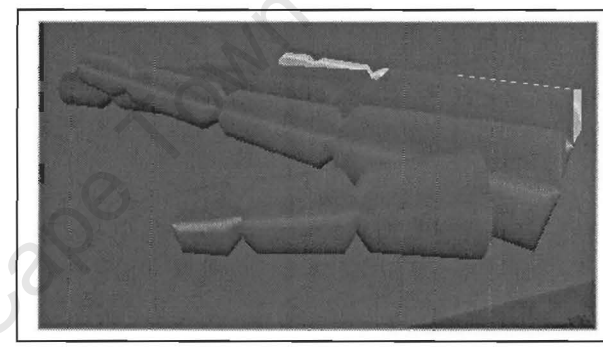


Figure 5.44 : Image of virtual hand for Hand A

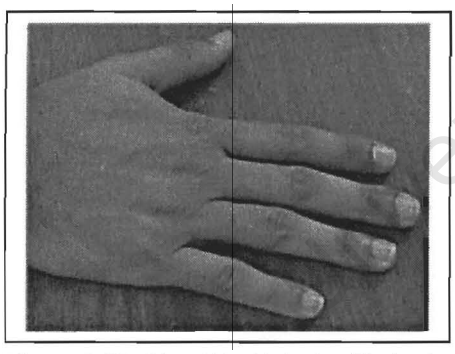


Figure 5.45 : Hand B with Index Finder length = 8.2 cm

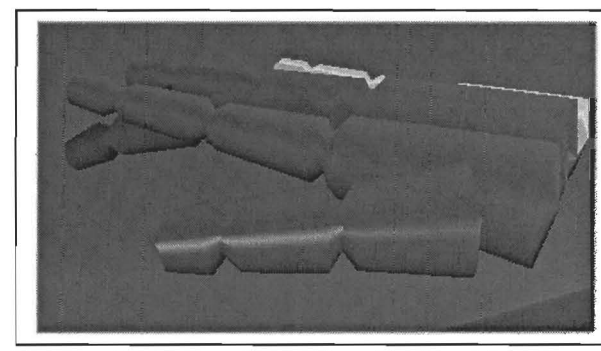


Figure 5.46 : Image of virtual hand for Hand B

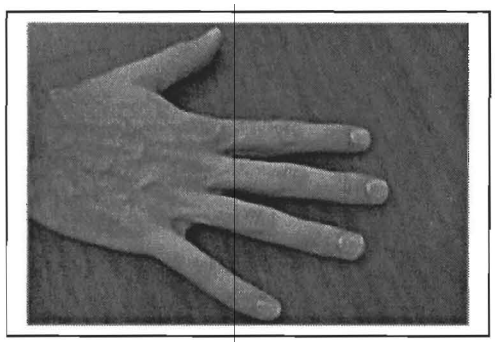


Figure 5.47 : Hand C with Index Finder length = 8.8 cm

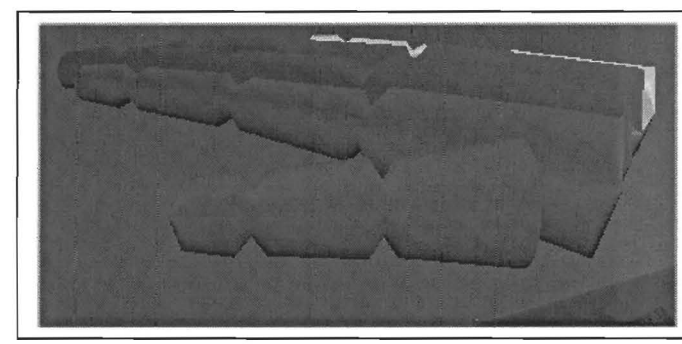


Figure 5.48 : Image of virtual hand for Hand C

Supplementing visual feedback

Up until now the discussion has focussed on how well the data glove and its corresponding virtual glove is able to fool the user visually. The capabilities of the data glove, as well as its short-comings have been illustrated, and for its price it performs well in creating a visually convincing image. Vision however, is only one of five senses. Most of them admittedly are irrelevant when it comes to simulating physical interaction with a man made environment. For example, not many control rooms currently use smell or taste to indicate to the plant supervisor how the plant is functioning.

Sound in response to physical interaction with an environment can play a small role, however it is not essential. Having audible feedback when pressing a button can play a supportive role. Take for example the popularity of click keyboards versus feather-touch buttons. Hearing that a button has been pressed provides reassuring feedback to the user. Due mainly to financial constraints, but also because of the complexity involved in generating 3D sound sources, this aspect of the virtual environment has been neglected by this project.

The other sense that needs to be satisfied is touch. When a user presses a button, they need to be able to feel a back-pressure on the finger. Working with the virtual environment it was found to be very disconcerting how ones fingers and hand can go through a virtual wall. Although building a physical wall where the virtual wall is, is a possible solution, it defeats the whole purpose of setting up a virtual control room versus a real one. Another problem is that buttons still don't move under the user's touch.

What is required is some sort of pressure feedback that can be applied to the fingers, palms, and the whole of the user's body. This is a formidable requirement considering the number of points on the user's body that the computer system would be required to apply feedback to.

A few shortcuts can however be taken. It is a natural reaction for a person to first check the solidity of a wall with his / her hands before trying to walk through it. Thus if feedback is provided to the hands and the user feels an opposing force on his / her arms, then they will not try and stick the rest of their body through the wall.

It is also possible to avoid having some sort of motion restriction. If the person wants to stick their arm through a wall then there needs to be an opposing force that is external to the hand. It has to be either mounted on the body of the person or on some secondary rigid structure. This sort of force feedback can be difficult to achieve and very cumbersome.

One way of faking it, is to have cells (pneumatic, mechanic or electric) mounted on certain parts of the glove. As the user pushes against the virtual wall, the pressure in these cells increases. There still however is nothing stopping the hand from going through the wall, so all that needs to done, is to move the virtual wall away from the hand. It is like having spring loaded walls, that if you push hard enough, they will move away.

The next step is to decide where such cells are required. Firstly one is need on each fingertip. Secondly one is needed on the palm of the hand. With a cell on the palm of the hand, having any cells on the inside of the fingers can be avoided, as most users cannot touch a flat surface with the inside of their fingers without their palm coming into contact with that surface.

If the user must be able to grip objects, then pads on the inside of the fingers must be provided. A pad on the outside of the hand is also needed, and maybe also a single pad on the outside of each finger in case the outside of the hand comes into contact with some object. These last five pads are however not essential as most people do not try to manipulate objects with the outside of their hands.

Currently there are a very small number of force and / or pressure feedback products available. Because of their still experimental nature, they are either not for sale, or available for exorbitant amounts of money. Thus no practical application to my above theorising has been tested in this thesis. The above few paragraphs form just a brief summary from the few available sources that engage in research in pressure and force feedback devices. Some of the devices that are available will be discussed in the following section.

5.1.3 Alternative Gloves and Interfaces

CyberGlove (Virtual Technologies)

CyberGlove is a low-profile, lightweight glove with flexible sensors which measure the position and movement of the fingers and wrist. The CyberGlove is available in two models and for either hand. The 18-sensor model features two bend sensors on each finger, four abduction sensors, plus sensors measuring thumb crossover, palm arch, wrist flexion and wrist abduction. The 22-sensor model adds sensors to measure the flexion of the distal joints on the four fingers.

The CyberGlove uses resistive bend-sensing technology that is linear and robust. The sensors are thin and flexible and produce almost undetectable resistance to bending. Since the sensors exhibit low sensitivity to their positioning over finger joints and to the joints' radii of curvature, each CyberGlove provides high quality measurements for a wide range of hand sizes, and ensures repeatability between uses. Calibrations typically need not be updated, even after months of use.

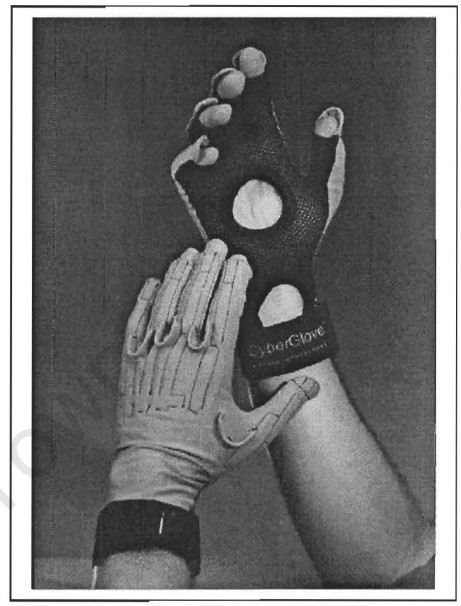


Figure 5.49 : CyberGlove

Specifications

- Sensor Linearity: 0.62% maximum nonlinearity over full range of hand motion.
- Sensor Resolution: 0.5 degrees; remains constant over the entire range of joint motion.
- Repeatability Between Glove Uses: Standard deviation of typically 1 degree.
- Interface: RS-232 with selectable baud rates up to 115.2 kbaud. Analog output also provided.
- Update Rate: Up to 112 records/sec when filtered (18 sensor records). Up to 149 records/sec when unfiltered. Preset sample period or polled I/O
- Price : $9 800 – 18 Sensor
- Price : $14 500 – 22 Sensor

CyberTouch (Virtual Technologies)

CyberTouch is a tactile feedback option for the 18-sensor CyberGlove instrumented glove. CyberTouch features small vibrotactile stimulators on each finger and the palm of the CyberGlove. Each stimulator can be individually programmed to vary the strength of touch sensation. Software developers can design their own actuation profile to achieve the desired tactile sensation, including the perception of touching a solid object in a simulated virtual world.

Price : $14 800

Figure 5.50 : CyberTouch

CyberGrasp (Virtual Technologies)

CyberGrasp is a Haptic Interface for the entire hand. The CyberGrasp haptic feedback enables CyberGlove users to actually "touch" computer-generated objects and experience force on their hand. The CyberGrasp is a force-reflecting exoskeleton that fits over a CyberGlove and adds resistive force to each finger.

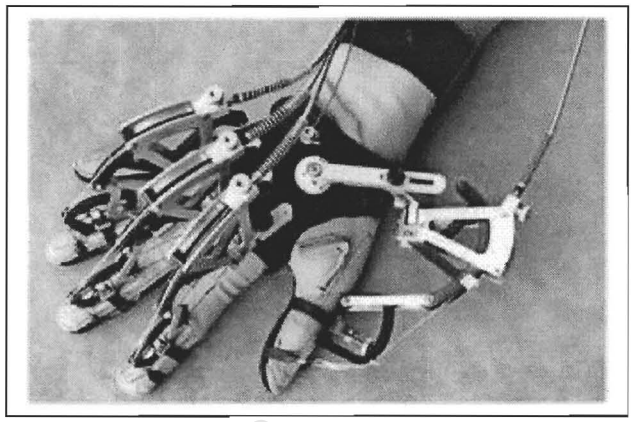


Figure 5.51 : CyberGrasp

The grasp forces are exerted via a network of tendons that are routed to the fingertips via an exoskeleton, and can be programmed to prevent the user's fingers from penetrating or crushing a virtual object. The tendon sheaths are specifically designed for low friction. The actuators are DC motors located in a small enclosure on the desktop. There are five actuators, one for each finger.

The device exerts grasp forces that are roughly perpendicular to the fingertips throughout the range of motion. The CyberGrasp system allows full range-of-motion of the hand and does not obstruct the wearer's movements. Originally developed under STTR contract to the United States Navy for use in telerobotic applications, the CyberGrasp system allows an operator to control a remotely-located robotic "hand" and "feel" the object being manipulated.

Specifications

- Maximum Continuous Force : 12 N per finger
- Force resolution : 12-bit resolution
- Weight (minus CyberGlove) : 350g
- Workspace : 1 meter radius hemisphere
- Host Interface : RS-232 and Ethernet are supported
- Price : $39 000

PHANTOM (SenseAble Technologies)

The Phantom haptic interface is a device that lets you feel virtual objects with your hands. It sits conveniently on the desktop next to the computer. The Phantom haptic interface has a stylus grip or a fingertip thimble with which users can reach into virtual worlds to touch and interact with 3D objects.

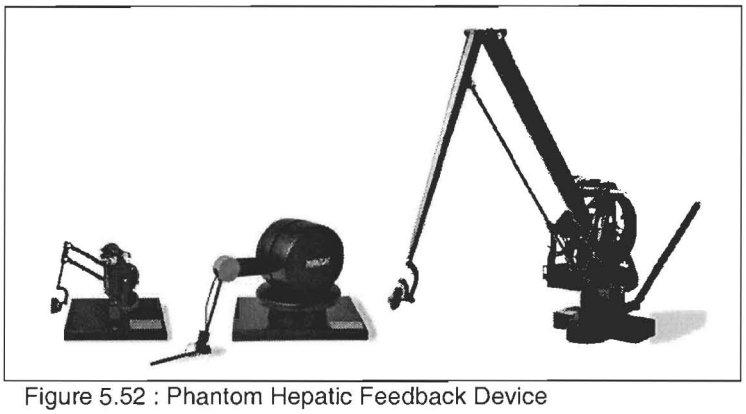


Figure 5.52 : Phantom Hepatic Feedback Device

The Phantom interface permits users to feel the forces of interaction they would encounter while touching objects with the end of a stylus or the tip of their finger. One observation in simplifying haptic interface design is that people perform well in exploring and manipulating the world through a stick or a rigid thimble. This insight is that forces generated through point contacts, especially during active exploration, contain significant spatial and temporal information that humans can easily understand.

Specifications

- Nominal position resolution : >1000 dpi , 0.02 mm
- Workspace : 42 cm x 59 cm x 82 cm
- Backdrive friction : 0.2 N
- Maximum exertable force : 22 N
- Continuous exertable force (24 hrs) : 3 N
- Stiffness : 1 N/mm
- Inertia (apparent mass at tip) : <150 g
- Footprint : 20 cm x 20 cm
- Force feedback : 3 degrees of freedom (x, y, z)
- Position sensing : 3 degrees of freedom (x, y, z)
- Interface : PCI or ISA interface card
- Price : $19 000 (Approximate 1997 price)

5.2 6 DOF Sensor Tracking System

This section covers the Polhemus InsideTrak tracking solution, and also mentions some of the other tracking solutions that are currently available. In the discussion of the InsideTrak, an attempt has been made to fully cover all the aspects of its performance and capabilities in measuring the orientation and position of the receivers.

5.2.1 Performance Testing of the InsideTrak

As already discussed previously, the InsideTrak tracking system from Polhemus is a twin receiver system with a possible range of up to 3m, but with a workable range of about 75cm. The tracker measures the three position co-ordinates X, Y and Z, and the three orientation angles Tilt, Roll and Azimuth. Over the workable range, the tracker specifies an accuracy of 1.3cm for the position co-ordinates, and 2.0 degrees for the orientation angles.

The above specifications are for the case in ideal conditions. This section covers how well the tracker performs in a typical work environment. It also covers potential sources of problems and attempts to test how serious these problems are.

Range, Accuracy and Dead-band

These values are available from the specifications, but verification of these specification values will still take place. The image alongside shows my work area, containing some of the typical equipment found in an engineering environment.

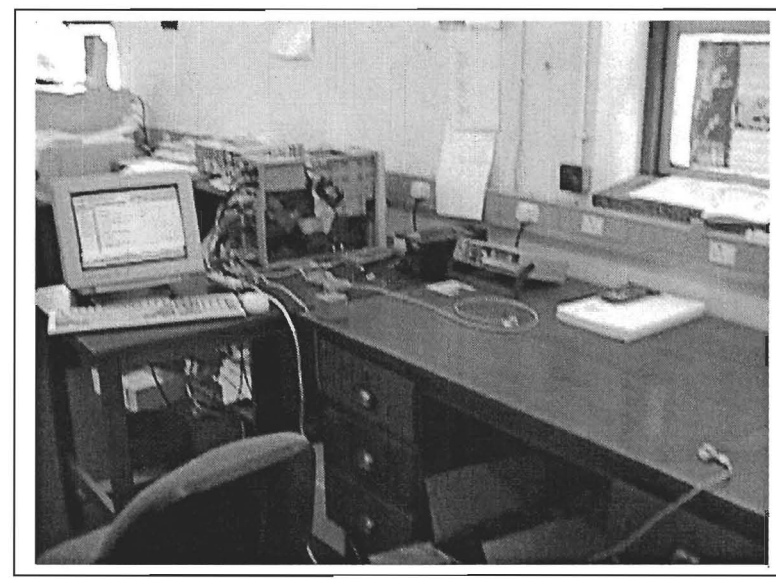


Figure 5.53 : Photo of my work area

The first test is to test the range along the Y-axis (towards the chair), the negative Z-axis (up), and the negative X-axis (towards the computer). The receiver is moved away from the transmitter until the tracker no longer registers any changes in the orientation angles. This way a maximum range over which the sensor operates can be found. One would assume that the range along the negative X-axis is going to be the least due to the two obstacles, namely the computer and the monitor.

Having done so it was found that the range in every direction is 3m. The only observation is that the values for the position and orientation start jittering with larger amplitudes. This leads to the next part of the discussion. How accurate are the readings that are acquired from the tracking system? To perform this test, the receiver is moved away from the transmitter along the positive X-axis. At different distances the static error in the X reading is recorded. This test is then repeated and the errors in the azimuth, elevation and roll recorded. Finally it concludes with the errors in the Y and Z values, while moving the receiver away along their corresponding axis.

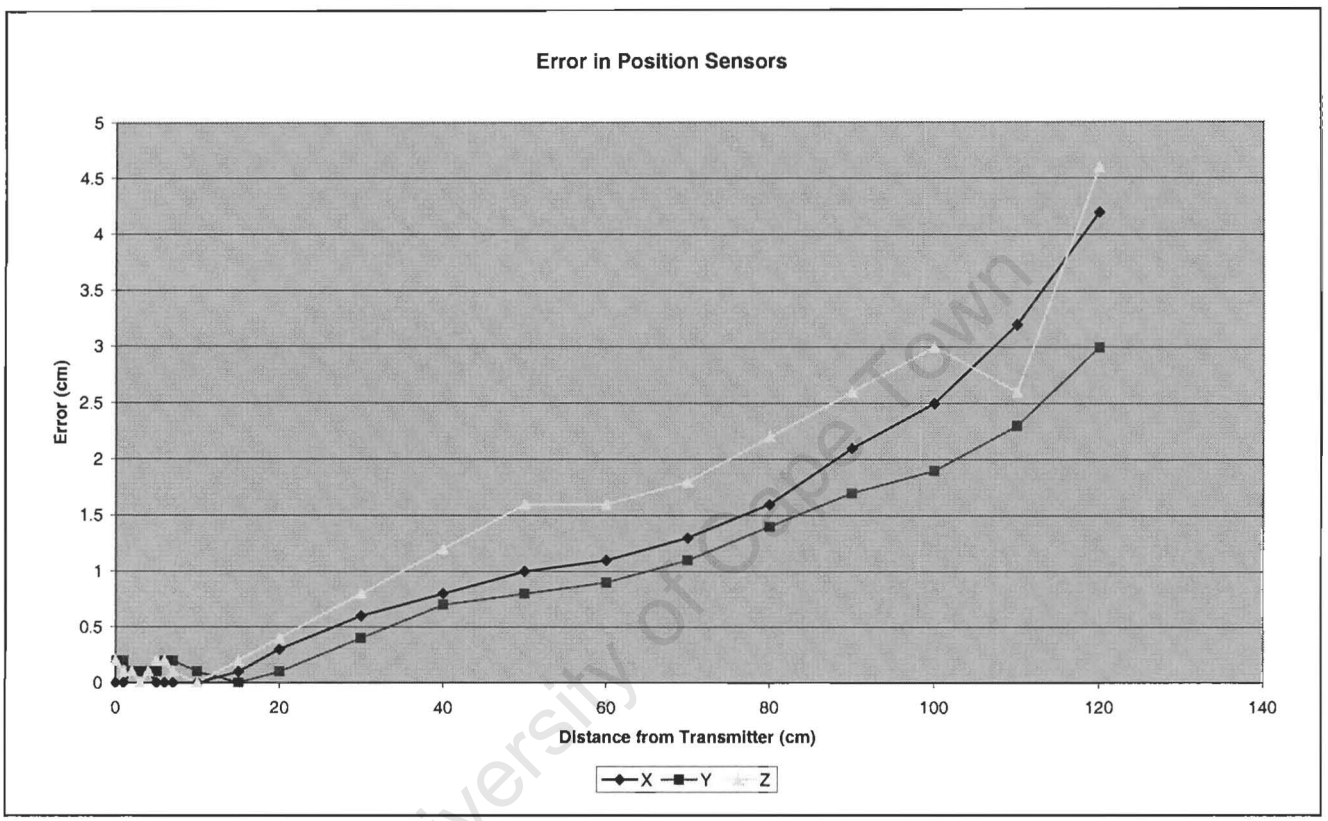


Figure 5.54 : Graph of Static Error in readings from tracker along the X, Y, Z axis.

The above graph shows that the position sensors do remain relatively close to the static error specification of being less than 1.3cm within 75 cm of the transmitter. The error within the first 15cm from the transmitter gives errors of 2mm or less. Thus within 15cm there exists precise positional measurement.

On the following page is the graph for the errors in the orientation angles. First thing to note is the offset of 4.1 for the error in the elevation reading. This error is most probably due to either a transmitter coil or a coil in the receiver not being mounted perfectly perpendicular. This would give a fixed offset. To correct for this, the one edge of the transmitter or receiver can be raised to level out the reading. Finally, for at least 65% of the working range, the static accuracy stays within the specified tolerance of two degrees.

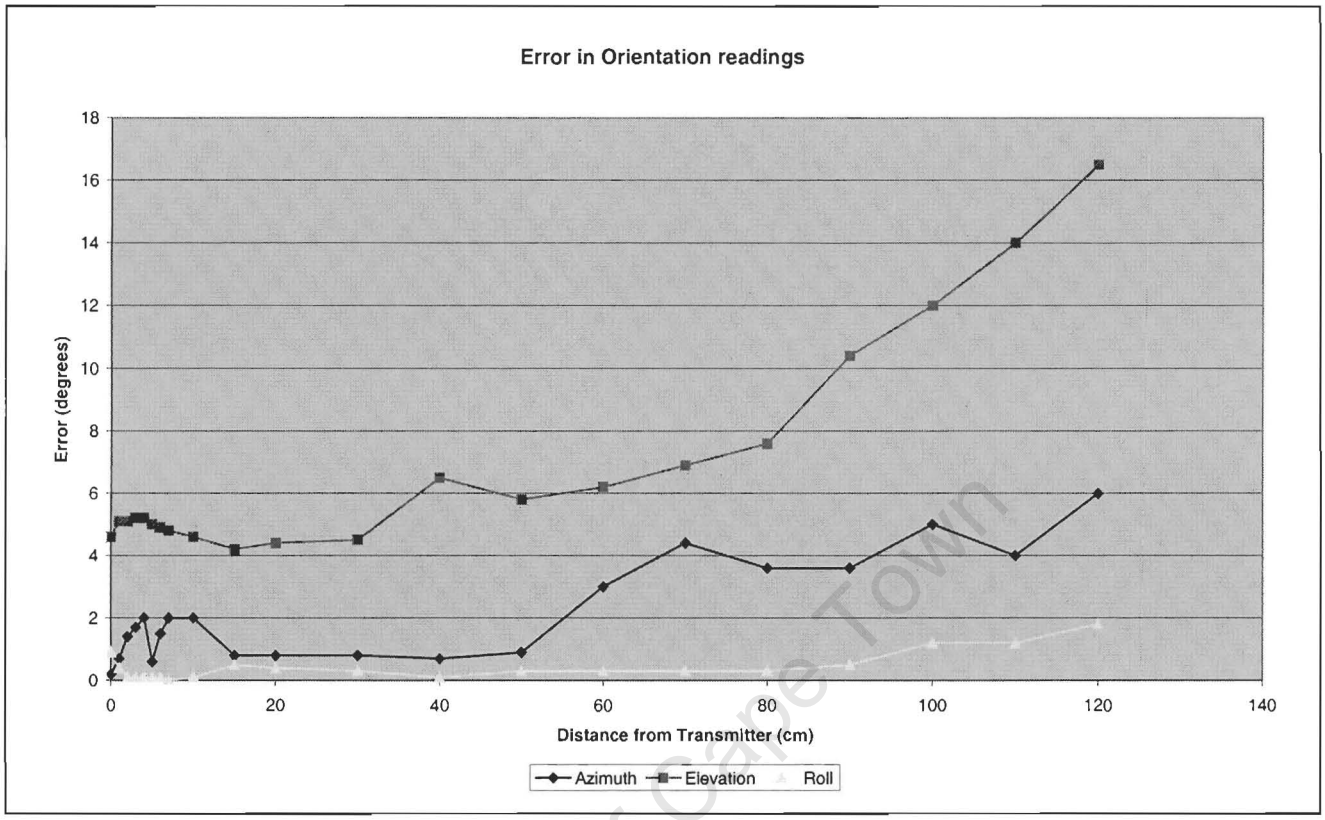


Figure 5.55 : Graph of Static Error in readings from tracker on the orientation angles

The next step is to verify if the errors shown above for the orientation angles are constant throughout the range of the orientation sensor. What was done previously, was to keep a certain orientation angle constant, and measure how much the reading changed from this set value as the receiver was moved away from the transmitter. What needs to be checked now is, if the distance between receiver and transmitter is kept constant, but the sensor is rotated through its full range, then does the error stay constant. For example, if the error is 2° at an angle of 90°, is it 2° when the angle is 180°.

The graph on the following page shows that the error does change, depending on the angle of orientation. A possible explanation for this occurrence could be derived from the fact that the signals from all three coils are required to tell each orientation angle. Now as the orientation angle changes a different (maybe less accurately made) coil starts collecting the majority of the signal.

Measuring the dead-band, it was found to be less than a millimetre for the position values and less than a degree for the orientation values. Both of these are higher resolution than any measuring equipment that was available. It can only be assume that this is true throughout the whole range of the device as the jitter in the readings becomes greater than the dead-band at distances more than a few centimetres away from the transmitter. With large jitter, the dead-band becomes difficult to measure accurately.

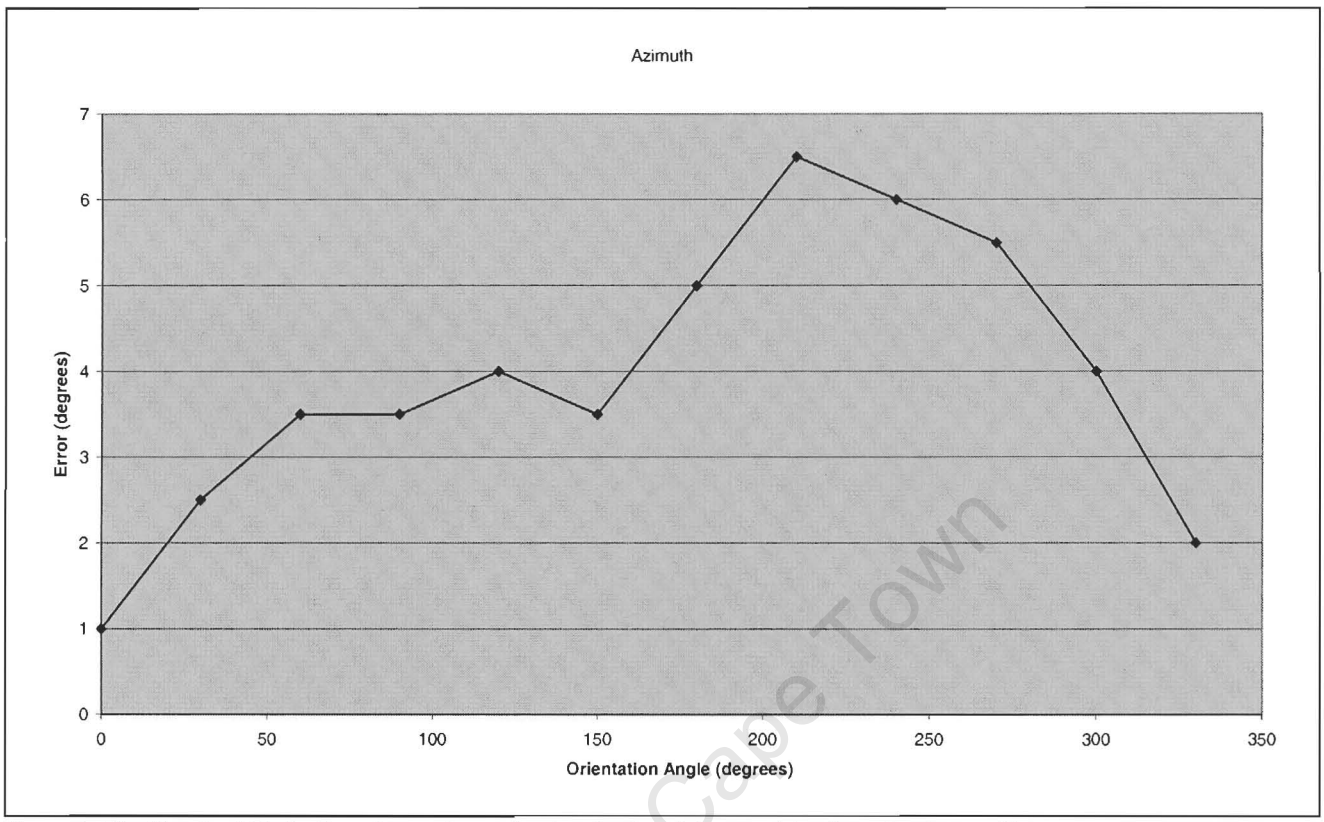


Figure 5.56 : Graph of Static Error in readings of azimuth at a distance of 1m from the transmitter

Jitter

Having already introduced it, this sub-section will now attempt to quantify it. Jitter is the noise in the reading that causes it to jump from one value to another while the receiver is stationary. On a small scale these could be due to vibrations in the table, but once these get larger than a millimetre their occurrence is due to fluctuations in the magnetic fields from the transmitter. At larger transmitter to receiver separation gaps, the magnetic signals received at the receiver are very small and it becomes more difficult to distinguish these from background electromagnetic interference. Thus background magnetic noise and the difficulty to accurately digitise a week signal cause the reading to jitter from one value to another.

What this sub-section does, is calculate the maximum value of the deviation from the average reading. By taking 50 readings, averaging them and then subtract this average from each reading a collection of jitter values is created. The maximum jitter value for the three position values, and the maximum for the three orientation values is then found. These two maximum jitter values are recorded at different distances from the transmitter. Figures 5.57 and 5.58 show the results of this test.

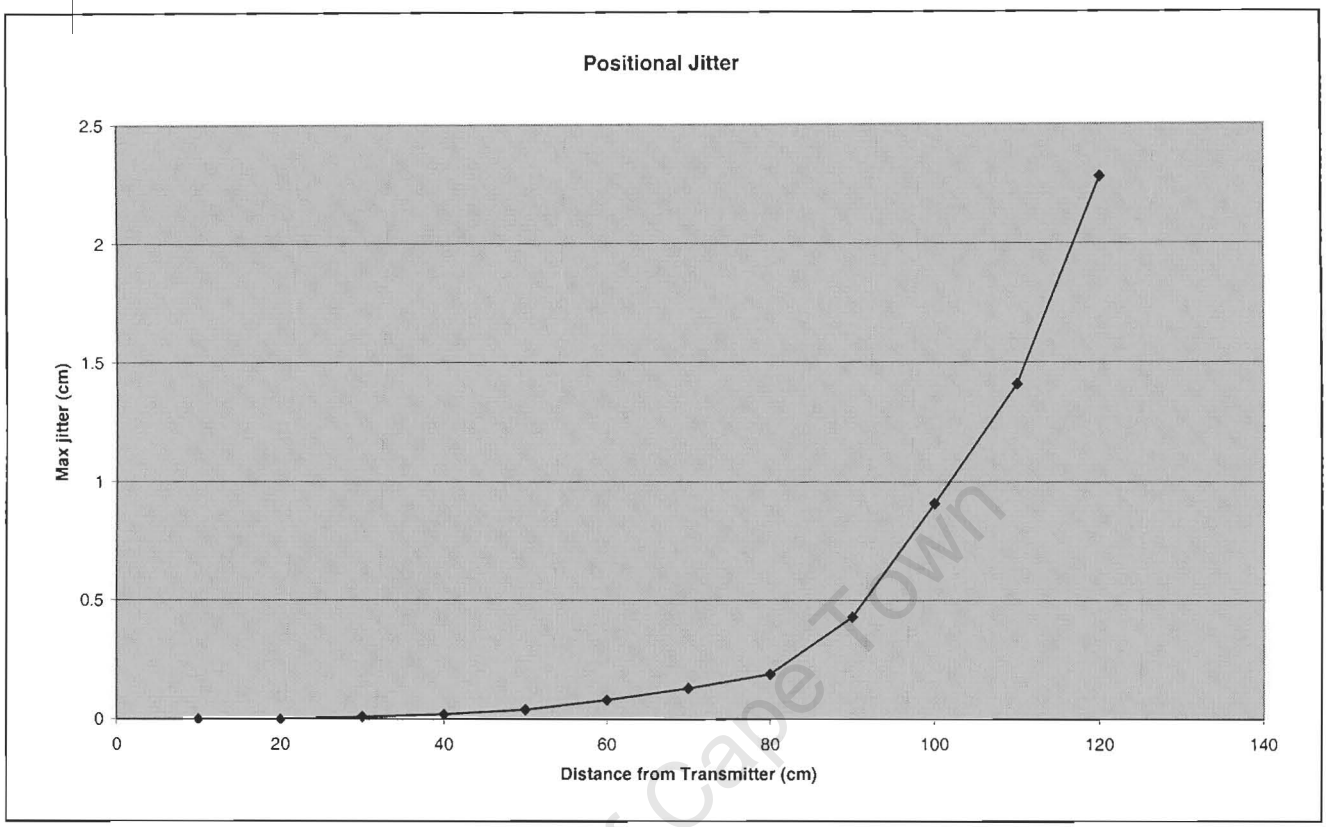


Figure 5.57 : Maximum deviation from the average position value at varying distances from the transmitter

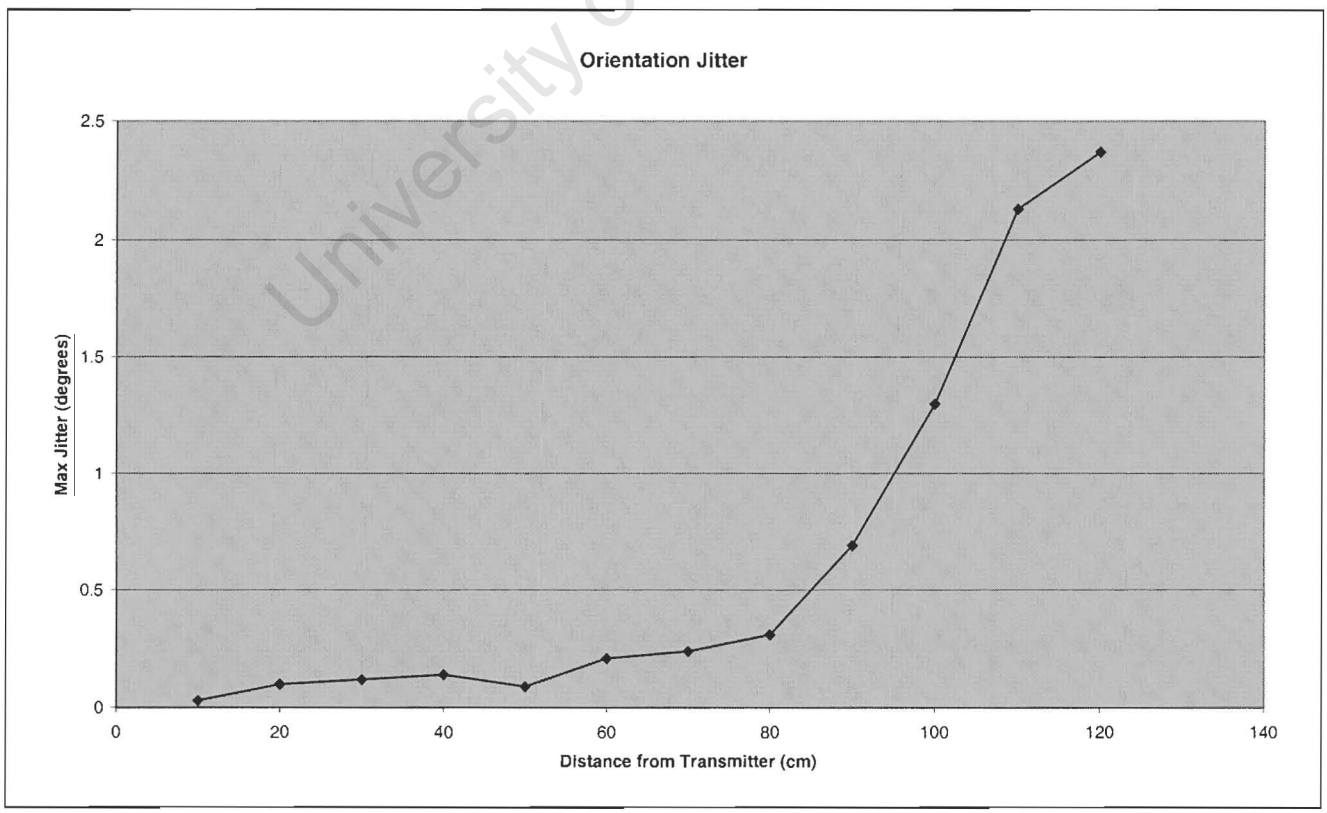


Figure 5.58 : Maximum deviation from the average orientation value at varying distances from the transmitter

The above two graphs show that the jitter increases exponentially as the receiver moves away from the transmitter. The question to ask now, is why is jitter important? Jitter is to some degree more important than the static accuracy. If a person draws a spot on the wall, closes their eyes and tries to touch this spot, then most people will not be able to get within two centimetres of this spot.

So much so, if the user sees a button in the virtual display, closes their eyes and tries to touch this spot, and upon opening their eyes finds that they have missed the button by 2cm, then they will most probable not be too surprised. Now if the user were attempting to press a button with their eyes open, as their hand moved towards the button, they will be able to see if it starts moving off course. Even without thinking about it, their arm will correct its trajectory to make contact with the button.

Thus having a static accuracy of only 2cm will not be noticed once within the virtual environment. The only problem that can occur is if there are physical objects (e.g. a table) that have directly corresponding virtual objects. Here static accuracy could play a more important role as the user might find their hand colliding with the physical table while their virtual hand is still two centimetres above the virtual table in the VR environment.

Where jitter comes into effect is that it causes the view and the virtual glove to shake. If the receiver mounted on the user's head is receiving a position signal with jitter then this will translate into the same affect as if the user had been shaking their head. This makes focussing of the eyes a bit more difficult. Secondly, if the receiver on the data glove is receiving a position signal containing jitter, then the hand will appear to shake. This will create the same effect as trying to press buttons in the real world with a shaking hand. The user could miss the button, or even double click it by mistake.

Thus the amount of jitter limits the distance by which buttons must be separated. If the hand jitters, the user must still be able to only press a single button as long as he or she aims for the button's centre. Assuming that that buttons can be placed directly next to each other, then there is a minimum size of button that has to be used at various distances from the transmitter. The graph below shows the minimum sized button at different distances from the transmitter for a fingertip of 1cm by 1cm.

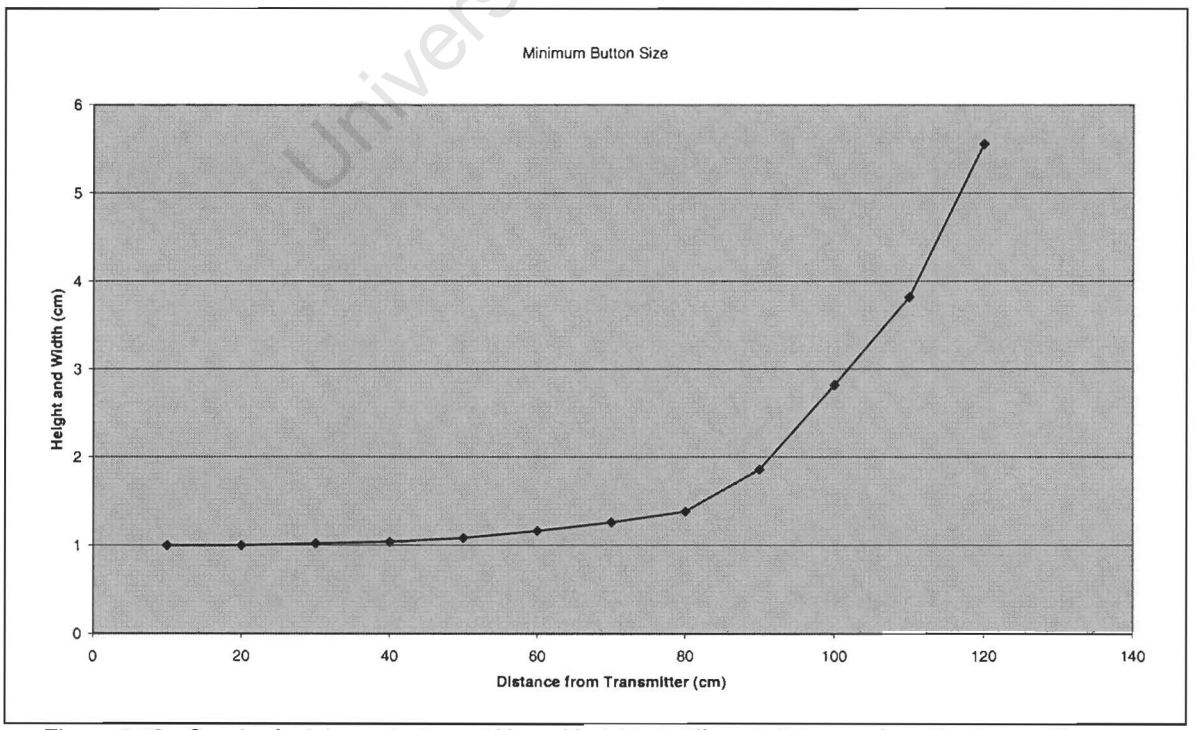


Figure 5.59 : Graph of minimum button width and height at different distances from the transmitter.

Interdependence of readings

It was shown earlier, in the discussion of the orientation sensors for the data glove, how the one sensor's reading, accuracy and range was dependent on the other sensor's value. This affect is however not observed with the InsideTrak tracking system. All the readings are independent of each other. The position readings are just as accurate irrespective of any orientation of the sensors, and each orientation reading maintains its full range and accuracy irrespective of the values of the other orientation readings. For this reason I have decided not to include any graphs of readings for this sub-section as they do not show anything interesting.

Motion Dynamics

This section discusses the dynamic response of the sensors due to changes in both position and orientation. The first step is to see how the sensor responds to being shaken. While attempting to keep the receiver perpendicular and level, the sensor is rapidly moved from side to side. Readings are meanwhile recorded at maximum sampling rate.

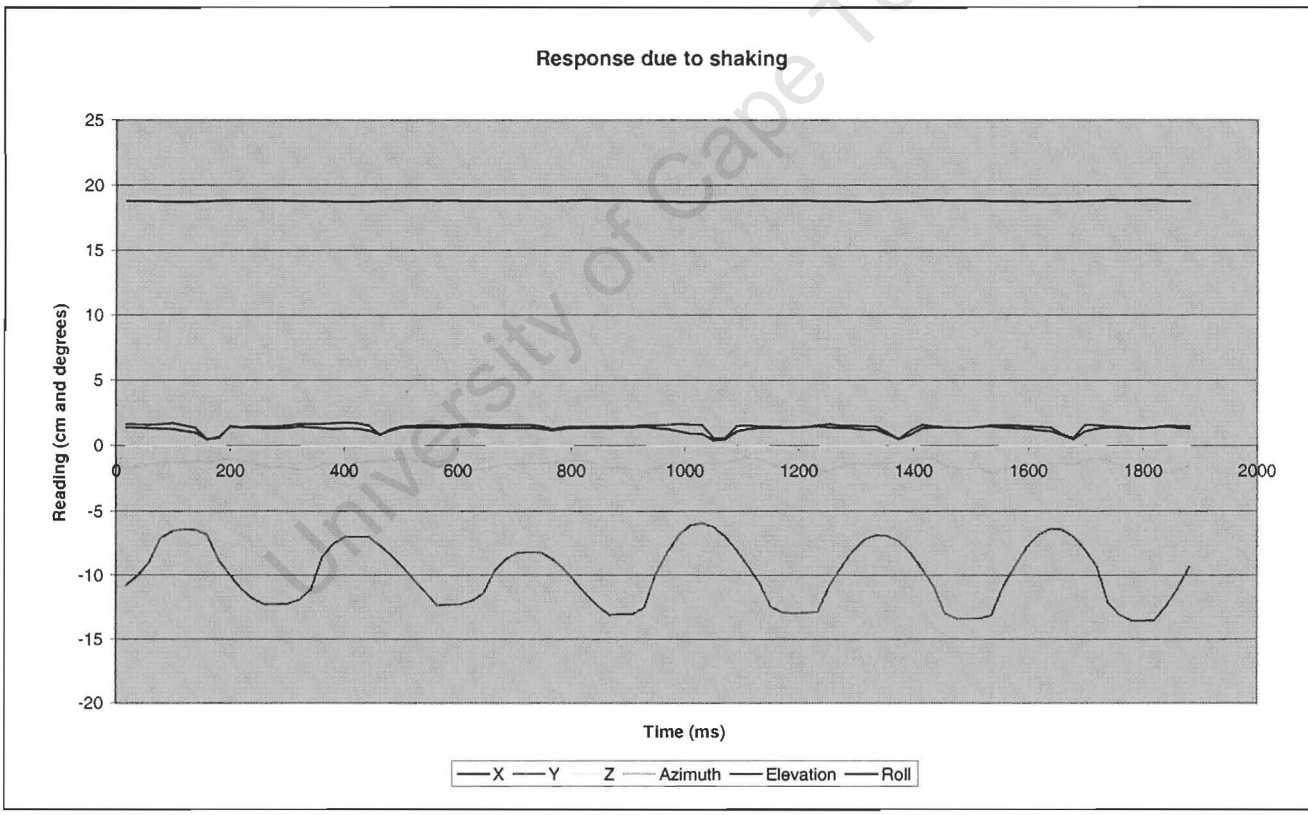


Figure 5.60 : Graph of sensor response due top motion along Y axis.

The graph above shows that the motion was along the Y-axis of the system. The other two position values, namely X (dark blue), and Z (yellow), show almost no effect to the motion. The orientation angles oscillate to only a very small degree (less than the static accuracy specification) and thus are also negligible. The oscillations in the orientation readings are most probably due to the receiver not being held sufficiently steady and level. Thus rapid motion of the receiver does not effect the readings it reports.

Next comes an investigation to see how the system responds to changes in position and orientation. Without teleportation, step changes in position are not really possible. Similarly, step changes in orientation are also difficult to achieve. Thus to get some response data to input changes for the position sensor, the sensor was dropped, and readings of its position recorded at various intervals. Ideally the position is expected to change at a constant acceleration

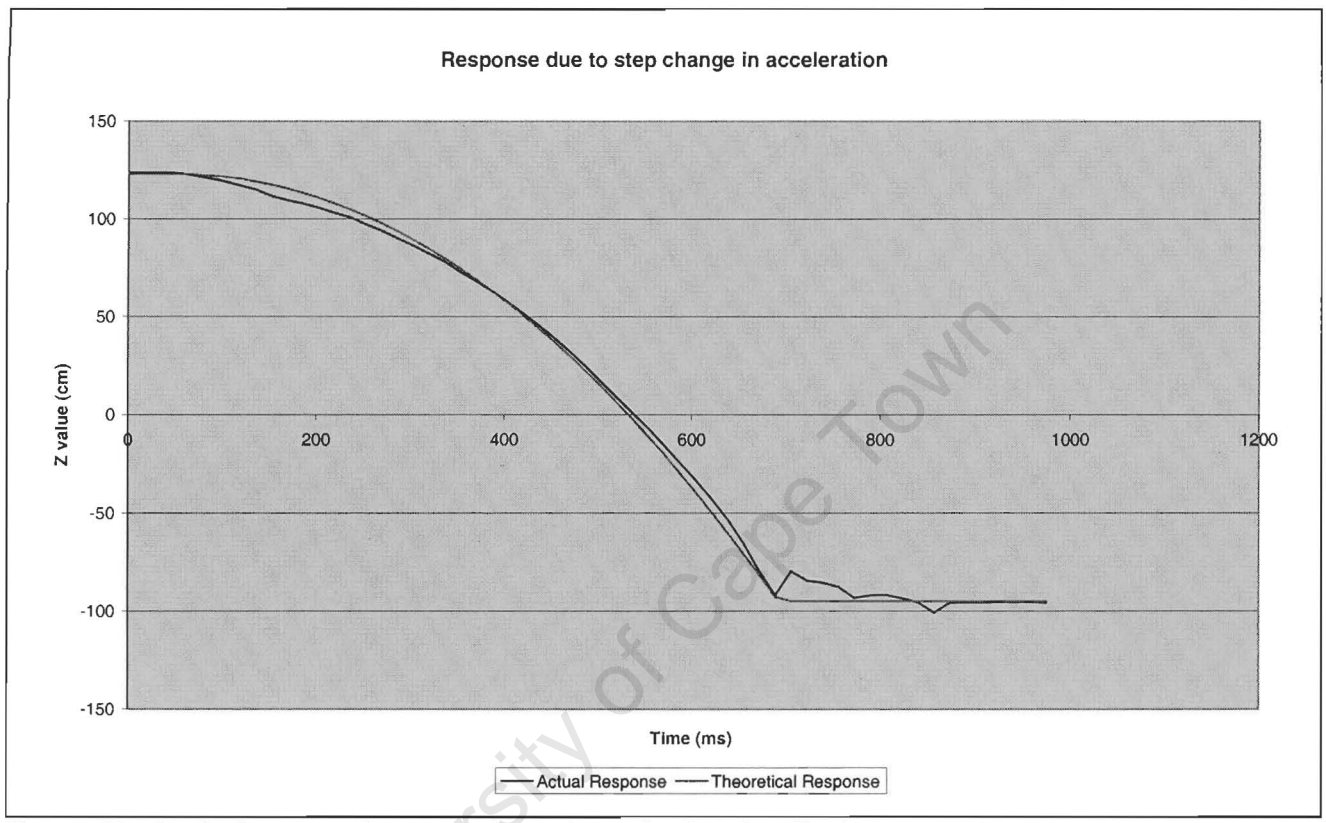


Figure 5.61 : Graph of response due to step change in acceleration along Z axis

The above graph shows the actual readings taken and a theoretically expected object undergoing free fall at 5.3 m/s^2. We see that the tracker successfully, and without any dynamics, managed to accurately record the falling receivers position. The bumps at the end of the graph are due to the receiver bouncing once it landed on the ground. The receiver was dropped onto a carpet and also covered in 'Prestik' to reduce the amount of bouncing, however some still occurred.

The reason for the reduced acceleration during freefall (namely 5.3 m/s^2) is because the receiver is attached to the computer via a cable. The receiver is lightweight and the cable retards the downward force of gravity. The 50g of putty used to prevent bouncing helped increase the weight of the receiver in comparison to the cabling.

Next is a check to see if the orientation sensors exhibit any dynamics due to changes in orientation. The receiver was positioned at an angle to the table. It was then hit with a hand that brought the sensor level to the ground. The graph below shows how the roll angle starts off at 35 degrees then when hit goes down to zero within two sample intervals without exhibiting any undesired dynamics.

From this discussion we can gather that the InsideTrak has excellent response dynamics to changes in position and orientation. No distortion or oscillation occurs in the readings due to changes in position or orientation, making these sensors ideal in situation when the user moves considerably within the virtual environment.

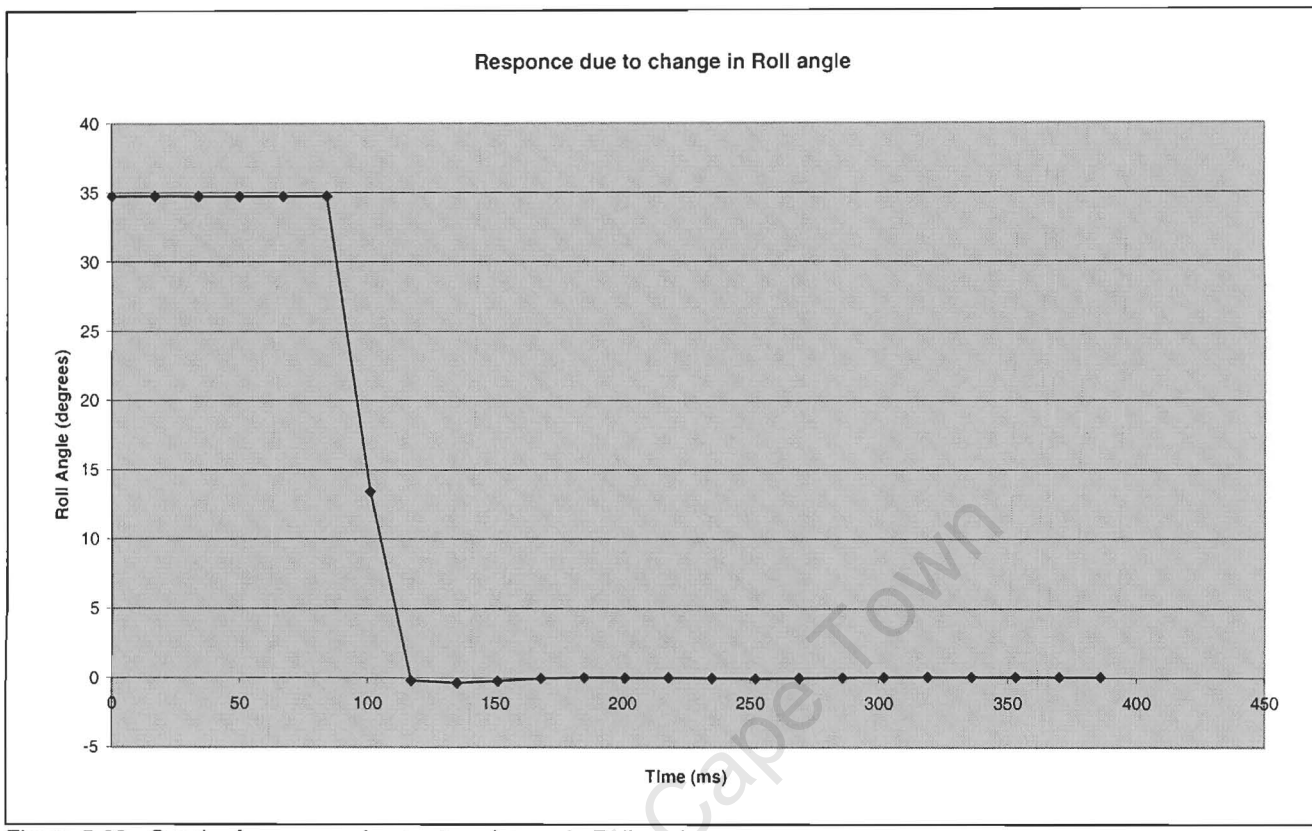


Figure 5.62 : Graph of response due to step change in Roll angle

External Interference

This sub-section covers some of the potential sources of problems that could occur in a typical industrial environment. It first shows how proximity to certain equipment can decrease the ability of the receiver to accurately detect its position. To calculate the decrease in accuracy, the amount of jitter in the signal will be measured. When the jitter increases, it indicates that either there is an interfering signal superimposed or something is blocking off the signal from the transmitter.

The first part measures how different objects adversely affect the tracking system. The receiver is placed near a computer, near a monitor, and near a radio, all of which will be within similar distances from the transmitter. Below are three photos of the receiver mounting positions.

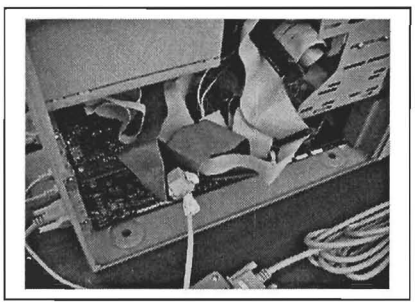


Figure 5.63 : Receiver on Computer

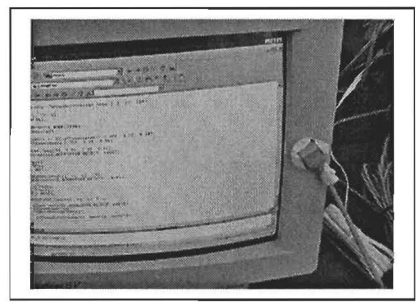


Figure 5.64 : Receiver on Monitor

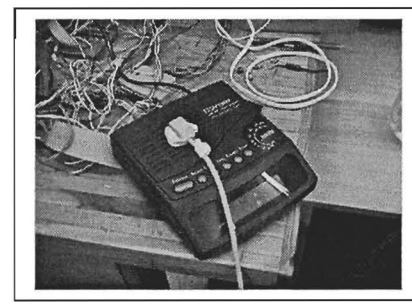


Figure 5.65 : Receiver on Radio

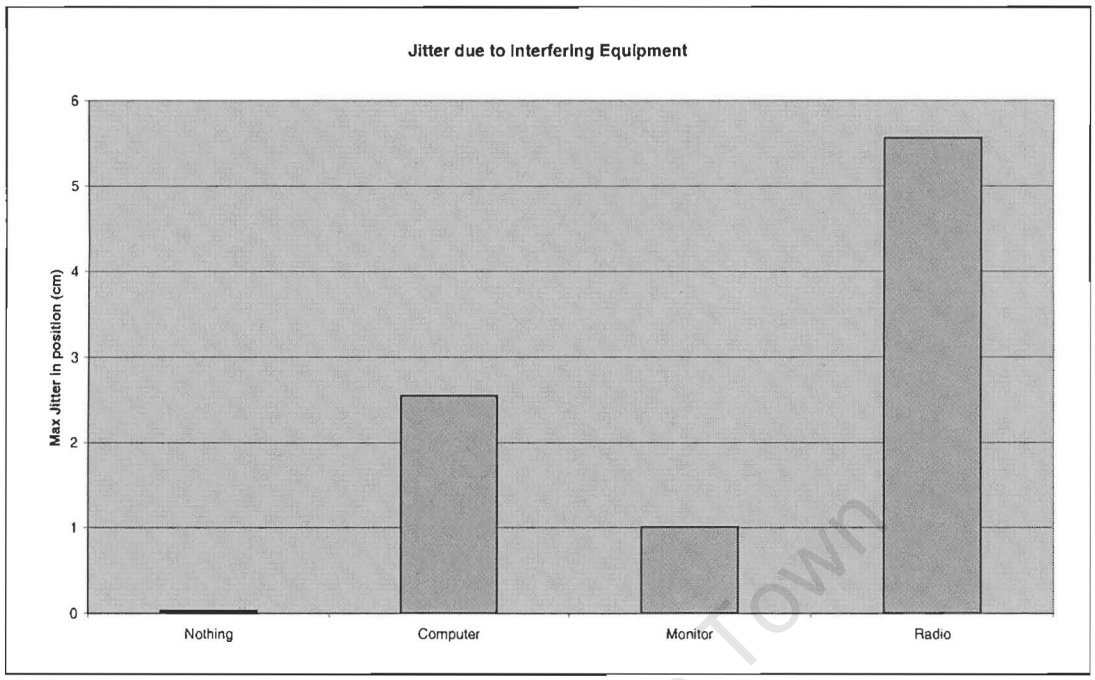


Figure 5.66 : Graph of the maximum positional jitter in the four different positions

As specified in the user documentation, any metallic object will cause interference. That is the most probable cause of the jitter due to the computer. The monitor had less jitter because the receiver was mounted on the plastic enclosure, and most of the electromagnetic radiation is shielded inside the monitor. The radio caused the greatest amount of interference because of its internal speaker. The speaker is a moving magnet with its oscillation range around the transmitter's 10kHz carrier frequency.

The next step is to see if there is any interference caused by the cabling of the receiver being repositioned. The receiver's excess wire (about 3m) was placed as a bundle on the transmitter's bundle of cable (about 2m). The receiver's cable is wound around the transmitter's cable. It is also place on top of the Transmitter Frequency Module to see if any of these affect the tracking system. The following figure shows a table of maximum jitter values with the receiver mounted 50cm away from the transmitter.

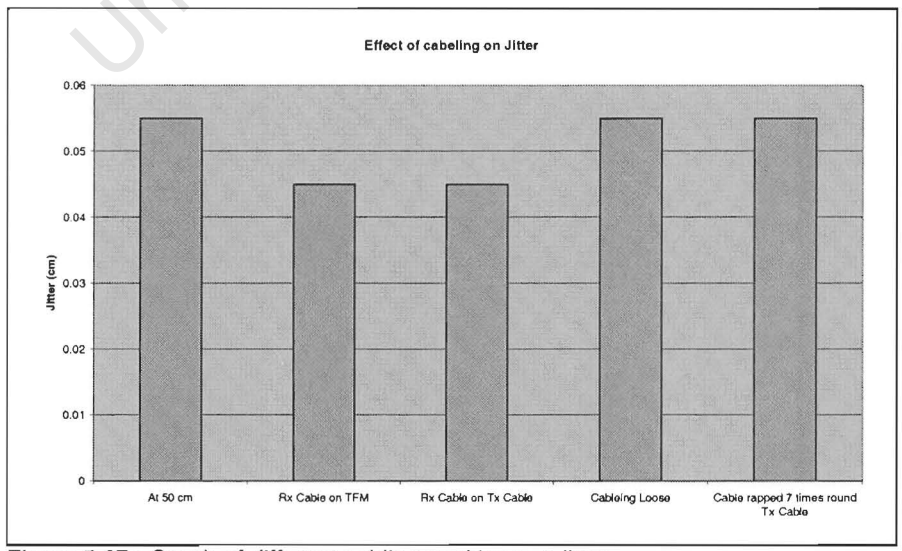


Figure 5.67 : Graph of different cabling positions on jitter

The final discussion in this section will be to see how different metallic objects affect the static accuracy of the readings. This is done by having three different metallic objects, each of which are placed at different distances from the transmitter and the static error recorded. Because the objects are of different sizes, separate graphs are presented for the errors as the objects moves towards the receiver and one for where the objects move away from the transmitter. The three objects that are used, as shown in figures 5.68 to 5.70, are a dustbin, a speaker and a metal disk.

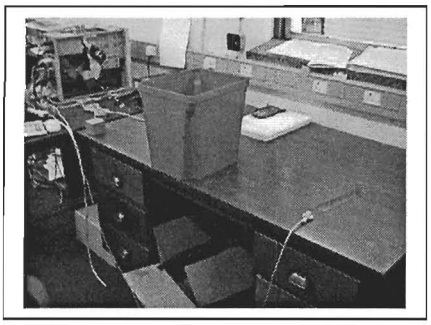


Figure 5.68 : Dustbin

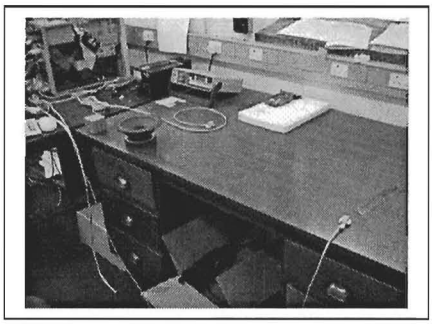


Figure 5.69 : Speaker

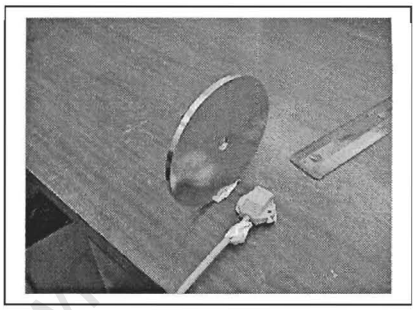


Figure 5.70 : Metal Disk

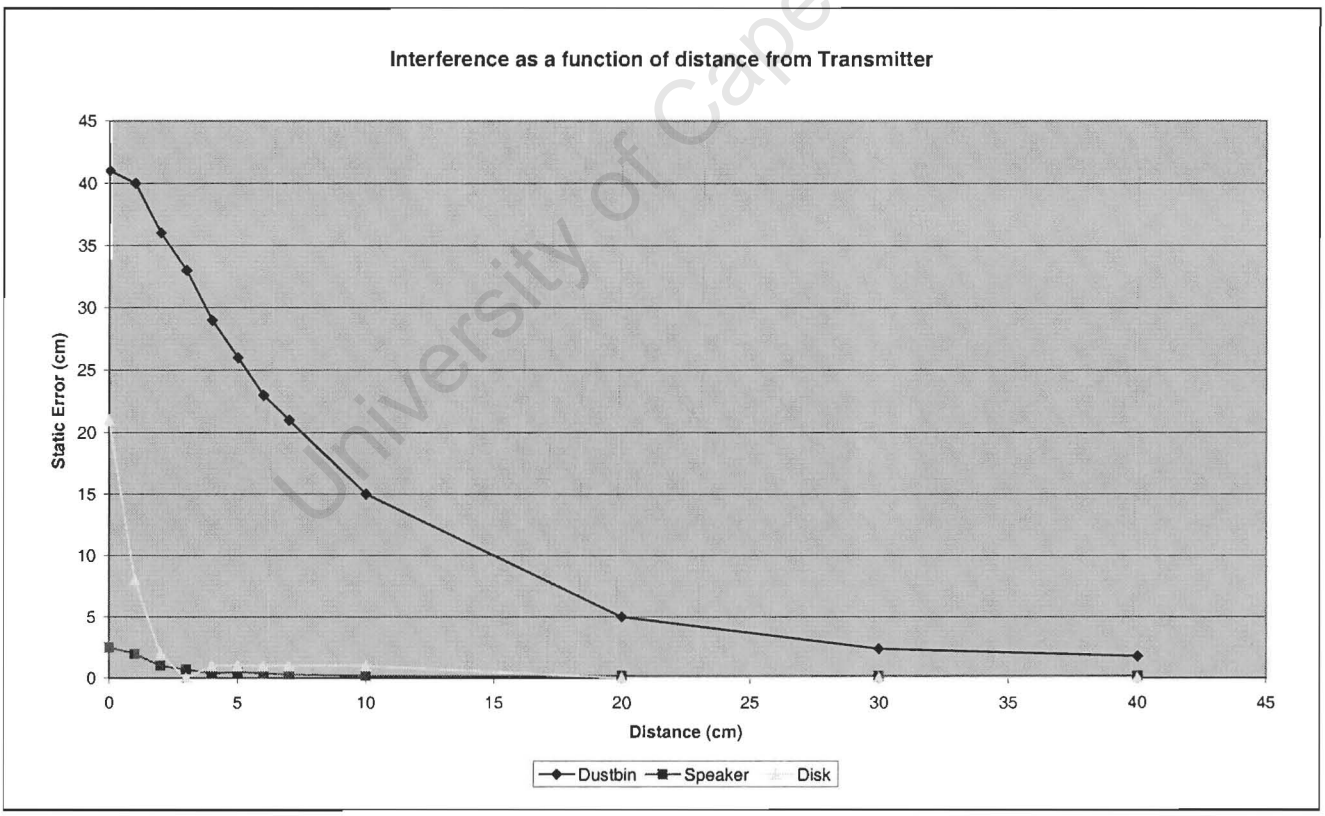


Figure 5.71 : Graph of Error in Static Accuracy for different interfering objects at different distances from the Transmitter

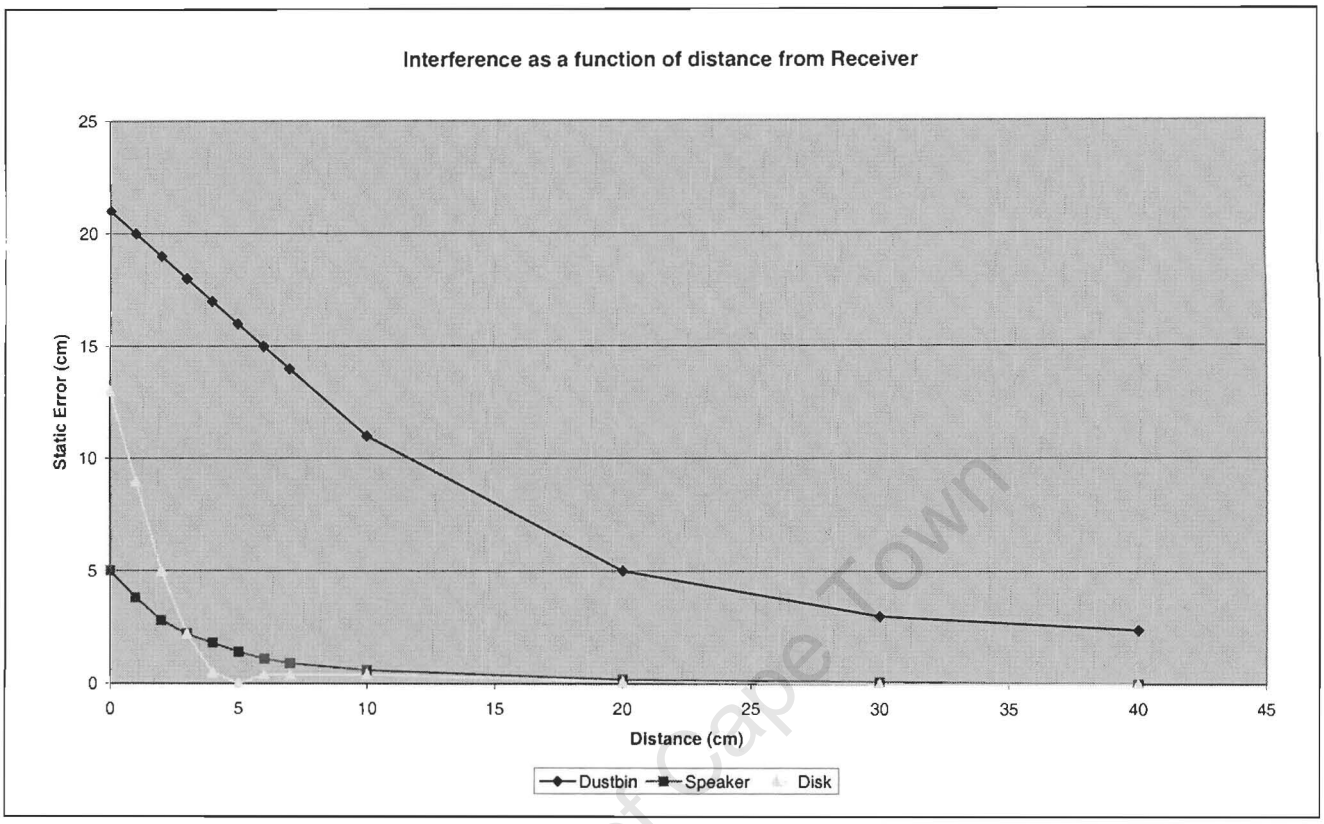


Figure 5.72 : Graph of Error in Static Accuracy for different interfering objects at different distances from the Receiver

It can be seen from both the above graphs that large metallic objects cannot be placed within ones working volume, otherwise they warp the virtual space. The dustbin is a fairly large object, so avoiding objects this size shouldn't be a problem. Smaller metallic objects, for example speakers, microphones, electronic circuitry, can within reasonable limits be used within the working volume, as long as they are kept at least 10 cm away from both the receiver and transmitter. This can be simply achieved by adjusting where the receiver and transmitter are mounted.

In overview of this section, the InsideTrak has shown to be highly resilient to interference from other electronic sources as well as to metallic objects that interfere with its magnetic pulses. Based on its resilience to environmental interference, the InsideTrak tracking system is suitable for use in an engineering and industrial environment.

Effects of the tracking system on the environment

Having discussed the degrees to which the environment is capable of effecting the tracking system, this sub-section will discuss how the tracking system effects the environment. Trying to calculate the effect of the electro-magnetic radiation on the human body is beyond the scope of this thesis. Two things I can however test for, that would be of consequence to use in an industrial situation, are its effect on display devices (monitors) and on magnetic storage media.

Starting with the effect on storage media, a 1.4MB file containing pre-calculated values was written onto a 'stiffy' disk. This disk was then placed on top of the transmitter and left there to stand while the system recorded the readings from the receiver. After 30 minutes the disk was returned to the computer and each byte compared against the original, with no errors being found.

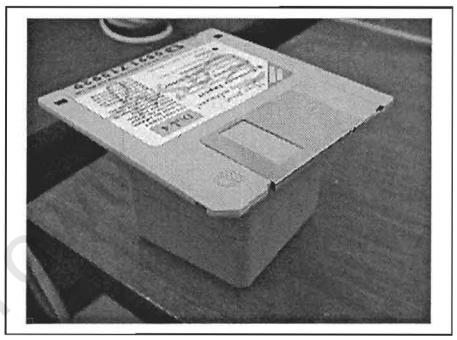


Figure 5.72 : Disk on Transmitter

The disk was then placed back on the transmitter and left there for 17 hours. After re-testing, again no errors were found on the disk. Thus the tracking system is safe to use around magnetic storage media. If no errors occurred on one of the most vulnerable of magnetic storage media after 17 hours of being right on top of the transmitter, then almost all magnetic media should be able to survive in the near vicinity of the tracking system.

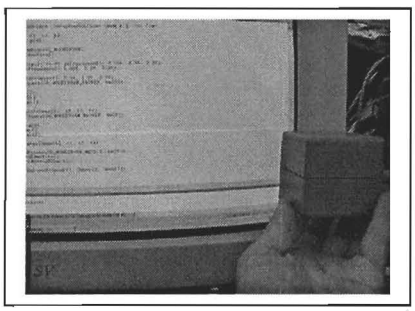


Figure 5.73 : Transmitter near monitor.

Next step was to test if the tracking system interferes with display media, namely the monitor. This was done by moving the transmitter towards the monitor and seeing if any visual disturbances became visible. About two centimetres from the monitor small ripple like disturbances would become visible moving up the monitor. Moving the transmitter right onto the monitor, made these ripple like effects appeared more prominently. These effects were however not prominent enough that they would come out on a photograph of the screen, and thus they cannot be display here.

Implications on Interactions

This sub-section summarises the potential problems and conditions for proper functioning of the tracking systems that the analysis in the previous sections has shown to exist. It will elaborate on the needs, and the complications in making the tracking system function correctly.

Starting with a discussion on whether the range and accuracy is sufficient, it moves onto the requirements on the environment that the Virtual Reality system will be operating in. Next it discusses the implications on the virtual environment that need to be obeyed for successful interaction. Finally it discuss other possible problems that can occur that have not been covered in this report.

Limitations due to range and accuracy

The accuracy is sufficient for a virtual environment as long as the receiver is within a short distance of the transmitter. If the user is able to keep the receiver within 50cm within the transmitter then the user should not experience any major problems. 50cm is however only sufficient to have a small desk-top sized virtual laboratory project. It is not suitable for a control room. Even the smallest control room is larger than two by two square meters. Ideally for a decently sized control room it would be ideal if the specified accuracy could be maintained over a range of 3m. Thus this is one area in which the InsideTrak is severely lacking.

Limitations on electronic equipment

The InsideTrak was shown to be quite tolerant of interference from the outside environment. It wouldn't be recommend for use on the factory floor, where large amounts of metallic objects or electro-magnetic radiation occur, as such an environment was unavailable for testing. However within a control room, where standard off-the-shelf computers can be used, the InsideTrak tracking system should perform satisfactorily.

Implication on the Virtual environment

The report has already discussed how the jitter in the position signal limits the minimum size of the buttons within the virtual environment. There is however another limitation that is created because of the jitter. For visual stability, and to avoid the user getting nauseous it is recommend that the view should not jitter by much more than 1%.

Using the 1% figure, for a given jitter value, there must be a minimum size of virtual object that is visible. If the jitter is 1cm, then the smallest object that can fit within the whole view must be at least 1m in width and height. The field of view is however also constant. Thus this minimum height restriction can be interpreted as a minimum distance away from the object.

For example, 1cm jitter dictates that we should see a metre's worth of height and width. The vertical field of view (FOV) is smaller than the horizontal FOV, so this will be our limiting figure. The FOV for the CyberEye head mounted display is 34 degrees. Thus the user needs to be at least 1.5 meters away from the object. This is very far considering the range of the equipment. Figure 5.74 below shows the minimum distance away from the object that the viewer has to be for the jitter to be less than 1% of the view for the values found in Figure 5.57

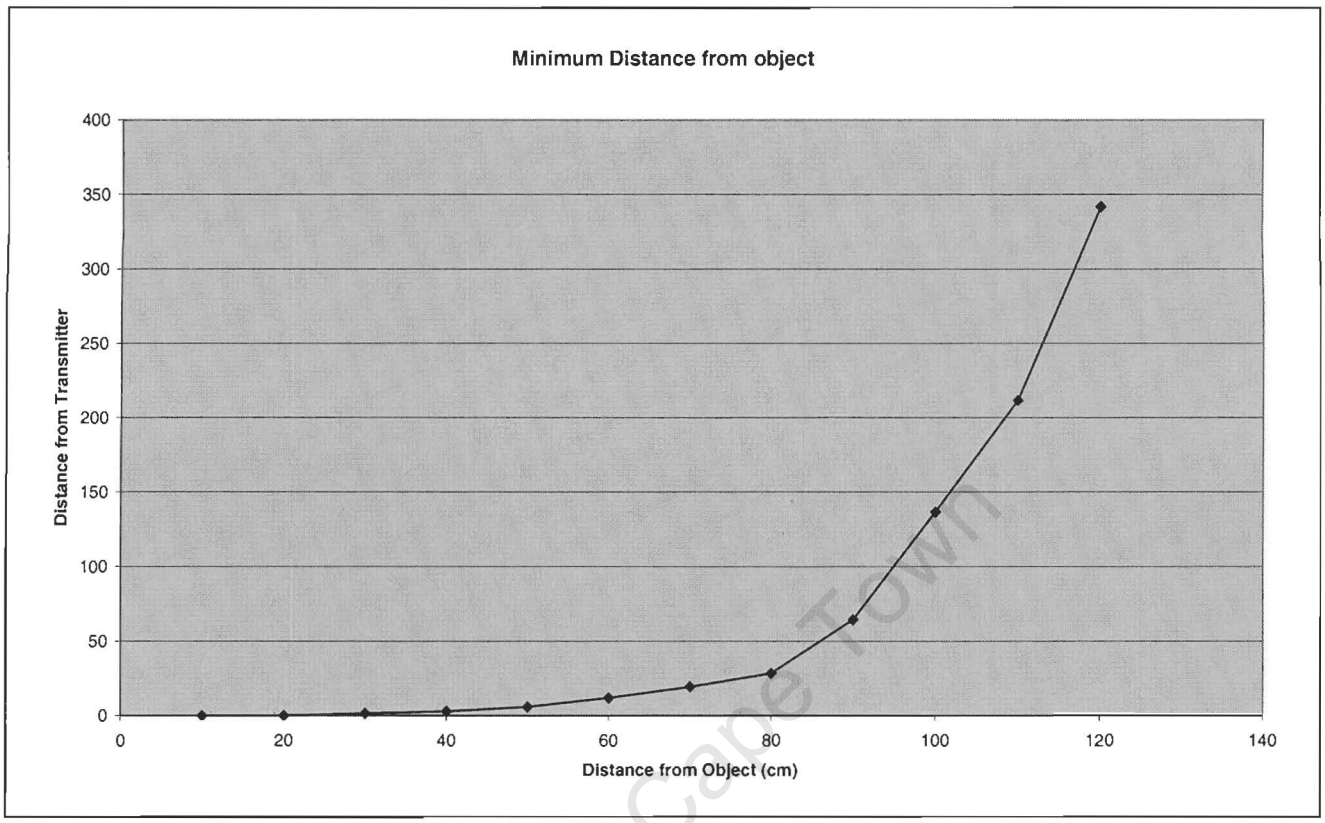


Figure 5.74 : Minimum distance from any virtual object for less than 1% jitter, shown as a function of the receiver's distance from the transmitter.

From the above graph we see that it is not really possible for the users head to move more than 70cm away from the transmitter if he / she still wants to be able to see nearby objects without them shaking too much.

The refresh rate of the tracking system, namely 30Hz for two receivers, is sufficient, as the screen update rate will be less than this. Thus no lagging view when the user rotates his / her head will be visible due to the tracking system.

Possible problem not investigated

These are some more potential sources of problems that might occur in setting up a virtual control room that have for one reason or another not been evaluated.

- Interaction between more than one tracking system in the same computer (using different carrier frequencies)
- Interaction between more than one tracking system in different computers (using the same carrier frequency)
- Increased sensitivity to the environment that might occur for higher accuracy tracking systems
- Potential long term damaging effects on the human body due to magnetic pulse radiation from the transmitter.

5.2.2 Alternative Tracking Systems

ISOTRAK II (Polhemus)

This tracking system is similar to the InsideTrak except that the static accuracy is 5 times better for the position readings and 2.5 times better for the orientation readings

Figure 5.75 : Polhemus IsoTrak

Specifications

- Latency : 20 milliseconds (without software filter).
- Update Rate : 60 updates/sec. divided by the number of receivers.
- Interface : RS-232 with selectable baud rates up to 115.2K baud.
- Static Accuracy : 0.1" RMS for the X, Y, or Z position; 0.75 degrees RMS for receiver orientation.
- Resolution : 0.0015"/" of transmitter and receiver separation; degrees orientation.
- Range : Up to 5 feet.
- Angular Coverage : The receivers are all-attitude.
- Maximum number of receivers : Two
- Price : $2875
- Extra Receiver : $475

FASTRAK (Polhemus)

The FastTrak is one of the upper range trackers with a static accuracy improvement of 20 times for position and 13 times for orientation values over the InsideTrak. With this high accuracy one is able to operate over a much larger range than the specified 75cm and still maintain an accuracy better than the InsideTrak. It also has an improved update rate (double that of the InsideTrak) and can have up to 4 receivers connected simultaneously.

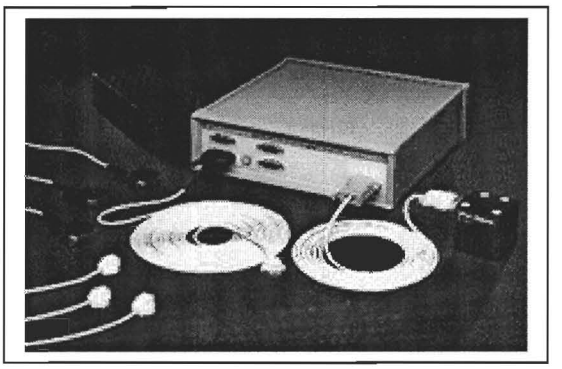


Figure 5.76 : Polhemus FastTrak

Specifications

- Position Coverage : The system will provide the specified performance when the receivers are within 30" of the transmitter. Operation over a range of up to 10 feet is possible with slightly reduced performance.
- Latency : 4 milliseconds.
- Update Rate : 120 updates/sec. divided by the number of receivers.
- Interface : RS-232 with selectable baud rates up to 115.2K baud (optional RS-422 = or IEEE-488 at up to 100K bytes/sec); ASCII or Binary format.
- Static Accuracy : 0.03" RMS for the X, Y, or Z position; 0.15 degrees RMS for receiver = orientation.
- Resolution : 0.0002"/" of transmitter and receiver separation; 0.025 degrees = orientation.
- Range : Up to 10 feet with standard transmitter.
- Multiple Systems : Up to 8 systems can be frequency multiplexed with no change in update = rate.
- CRT Interference Rejection : Provided by means of an external cable and sensor.
- Angular Coverage : The receivers are all-attitude.
- Max Number of Receivers : 4
- Price : $6050
- Extra Receiver : $475

LONG RANGER (Polhemus)

The LONG RANGER transmitter is a product that may be connected to either the FASTRAK or the INSIDETRAK system, and will have the effect of multiplying that system's range of operation by a factor of three. LONG RANGER consists of a clear acrylic globe, 18" in diameter, within which are three orthogonal coils that comprise the transmitter. LONG RANGER may be suspended overhead (LONG RANGER's weight is only 8 lbs.), or used with the 6" high tabletop support column.

Price : $3450

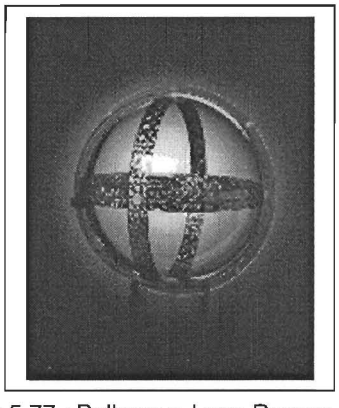


Figure 5.77 : Polhemus Long Ranger

StarTrak (Polhemus)

From fast-paced fighting to intricate dancing and tumbling, the RF wireless 3D motion capture system tracks the movements of multiple performers - in real-time and over the largest range in the industry. STAR*TRAK(R) utilises up to 32 receivers - enabling multiple performers to be tracked with a single system. When used with one supernova transmitter, STAR*TRAK covers a 25'x25' area, expanding to 50'x50' when used with two transmitters. STAR*TRAK allows users to perform directly on the studio floor and can be used in most studio spaces.

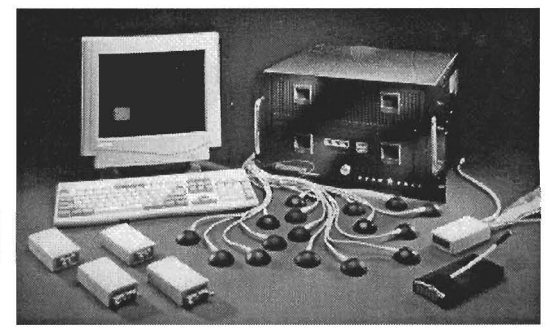


Figure 5.78 : Polhemus StarTrak

Price : $62,000 (8 Receivers)
$250,000 (32 Receivers)

pcBIRD (Ascension)

This is the competitor to Polhemus' InsideTrak and IsoTrak. It has an accuracy of 8 times better for position readings and 4 times better for orientation readings over the InsideTrak. Its range is also improved by 67%. One additional extra not on the Polhemus cards is that it can provide a rotation matrix instead of orientation values which would make the maths of rotating the view considerably easier in OpenGL. It however only supports a single receiver

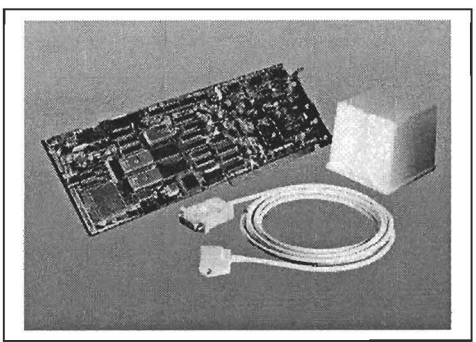


Figure 5.79 : Ascension pcBIRD

Specifications

- Degrees of Freedom: 6 (Position and Orientation)
- Translation range: 1.2m (3m optional)
- Angular range: All Attitude: ±180° Azimuth & Roll; ±90° Elevation
- Accuracy : position 0.07" RMS, orientation 0.5° RMS
- Resolution : position 0.02" RMS, orientation 0.1° RMS
- Update rate : Up to 144 measurements/second
- Outputs : X,Y,Z positional coordinates and orientation angles, rotation matrix or quaternions
- Interface : ISA-Bus
- Price : $2479

MotionStar Wireless (Ascension)

MotionStar Wireless utilises pulsed DC magnetic fields emitted by its extended range transmitter to track the position and orientation of its sensors. Sensors are mounted at key body points on the performer. Inputs from the sensors travel via cables to a miniature, battery-powered electronics unit mounted in a transmitter pack. From here, sensor data and other signals from body-mounted peripherals, such as data gloves are sent through the air to the base station. They are then transmitted to the host computer via RS-232 or an ethernet interface.

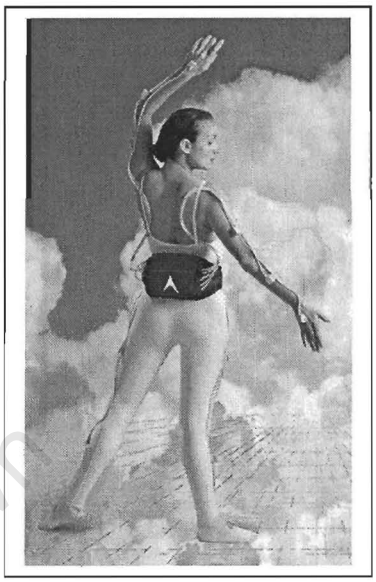


Figure 5.80 : Ascention MotionStar

Specifications

- Degrees of Freedom : 6: (position and orientation)
- Telemetry: 14 receivers per performer plus digital and analog inputs for user devices
- Translation range : ±10' in any direction
- Angular range : All-attitude: ±180° Azimuth & Roll, ±90° Elevation
- Accuracy :
  - position : 0.3 inch RMS at 5-ft range, 0.6 inch RMS at 10-ft range
  - orientation : 0.5° RMS at 5-ft range, 1.0° RMS at 10-ft range
- Resolution:
  - position : 0.03 inch at 5-ft range, 0.10 inch at 10-ft range
  - orientation : 0.1° RMS at 5-ft range, 0.2° RMS at 10-ft range
- Update rate : Up to 120 measurements/second
- Outputs : X,Y,Z position and orientation angles, rotation matrix, or quaternions
- Interface : Ethernet, RS-232C
- Price : $55,815 (6 Receivers)
- Price : $73,630 (20 Receivers)

Recommended System

The following are the tracking components that would be recommend for a pilot virtual control room. These should overcome the main problems experienced with the current tracking system, namely accuracy and range.

Polhemus FastTrak	: $6 050	
Polhemus Long Ranger	: $3 450	(for extended range operation)
Polhemus Extra Receiver	: $ 475	(for right hand)
Polhemus Extra Receiver	: $ 475	(for left hand)
Total	: $10 450	

5.3 Head Mounted Display

The head mounted display that was acquired for this thesis was the General Reality Company's CyberEye 200W. Purchased for $2 500, it was at that time one of the cheapest head mounted displays (HMD) available. As all the lower ranger displays were of similar resolutions, there wasn't much to distinguish them, apart from the type of interface that was used. Because it is the easiest to use, a HMD with a VGA input signal was chosen.

5.3.1 Capability Testing of the CyberEye

The CyberEye has a line sequential VGA input (alternate lines go to one eye) and has 181 000 pixels per eye. Because one needs three LCD pixels to form a red, green and blue triad, the effective resolution of the CyberEye is 230 x 263 pixels. Compared to currently available CRT monitors this is a very low resolution. What this section will do is to evaluate if this resolution is still usable for a virtual environment.

Resolution

The resolution of a HMD is one of its most critical specifications. However it is not just the number of pixels that are important. Filtering and lenses can often reduce the effective resolution that can be seen. What is done now, is a test to see what the highest resolution that is visible is. Start off by drawing horizontal lines, alternating black and while lines. The thickness of these lines is increased until they can be seen individually in the HMD. From this, it should be possible to calculate the effective vertical resolution.

While attempting this it was found that sometimes the lines go missing. A program that would draw a horizontal line that could be moved up and down was thus written. By moving this line up and down, it is possible to see which lines are visible in each eye of the HMD. The visible lines were thus marked as being either visible in the left eye or the right eye. By looking at Figure 5:81, the black lines on the left, are the lines visible in the left eye, and the black lines on the right are visible in the right eyes. The white lines stretching across the whole image are not visible in either eye.

These white bands indicate that horizontal sections of the image will not be visible in the HMD.

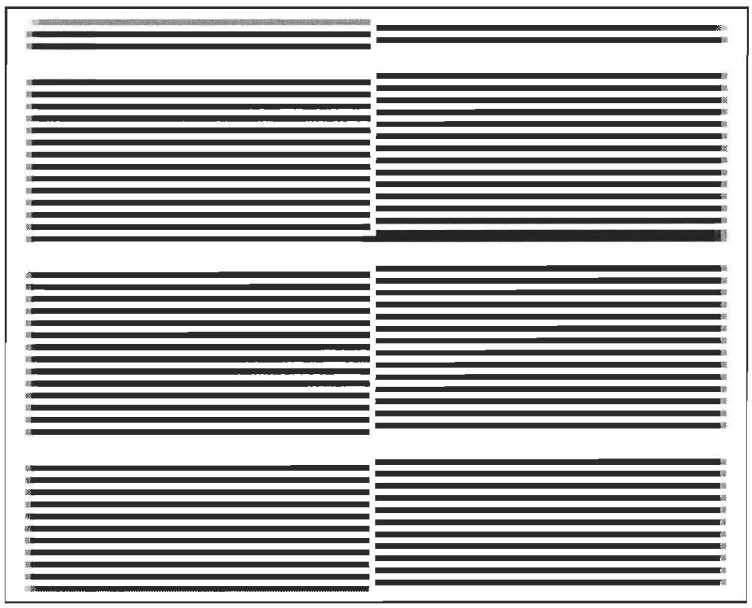


Figure 5.81 : Image illustrating Vertical Resolution

Counting the lines, it was found that out of every 32 horizontal lines (in VGA 640*480) mode there are only 14 lines that appear in the one eye of the HMD. This translates to a vertical resolution on the HMD of 210 pixels.

Next, the horizontal resolution needs to be calculated. This time, starting with a program that draws a single line that can be moved left and right by the user. By moving the line across the screen two observations were noted. Firstly, the line is always visible in both eyes unlike the horizontal lines which alternates visibility in each eye. Secondly, the is seen to move with every pixel increment. This would indicate a horizontal resolution of 640.

Attempting to verify this, a program was written that draws alternating white and black vertical lines. All that was visible, was a grey screen. Even when modified to draw two white lines alternating with two black lines, the screen was still grey. It was only when the white and black vertical lines were each three pixels wide that alternate lines became visible on the HMD. This translates to a horizontal resolution of 640 / 3 = 213 pixels.

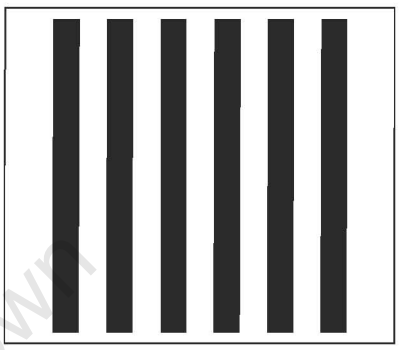


Figure 5.82 : Three pixels per line

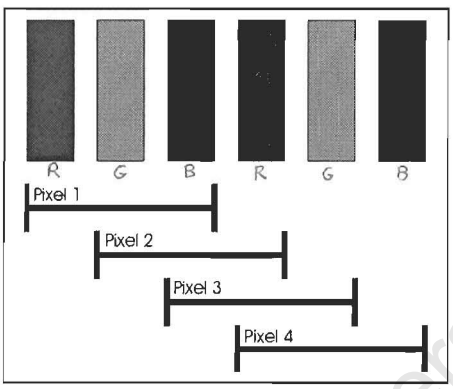


Figure 5.83 : Pixel Structure

What seemed odd was that although it took three pixels to draw a line, the progression of the line could still be seen when it was advanced by a single pixel. Taking a close look at the pixel structure of the HMD, which are fairly large in the HMD, explained this phenomenon. Figure 5.83 illustrates the pixel structure of a horizontal line. Although pixel 1 is different from pixel 2, they are not independent of each other. It is only every third pixel that is independent. This explains why the line moved pixel by pixel, but lines had to be drawn three pixels wide to give an image of vertical bands.

Image Quality

Having found the effective resolution to be 213 by 210 pixel, what does this entail for image quality? What is shown below, is using the effects discussed above, a 640 x 480 image of the virtual control room, seen in Figure 5.84, has been converted into a 213 by 210 pixel image, seen in Figure 5.85. The image in Figure 5.85 has been stretched horizontally, because the LCD panel is rectangular and displays the image wider.

Note on the bottom panel of Figure 5.85. It can be seen where horizontal bands have been removed from the once straight line. This is due to the HMD not displaying the fifteenth and sixteenth even and odd lines.

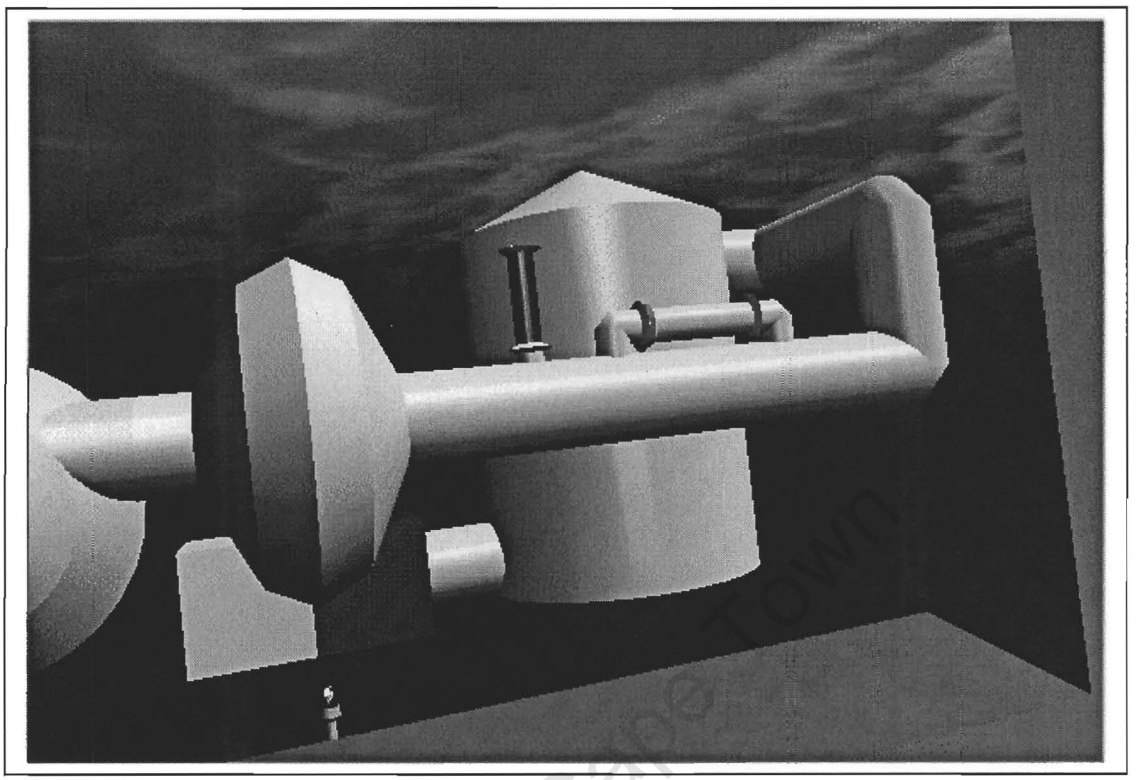


Figure 5.84 : Original 640 x 480 image

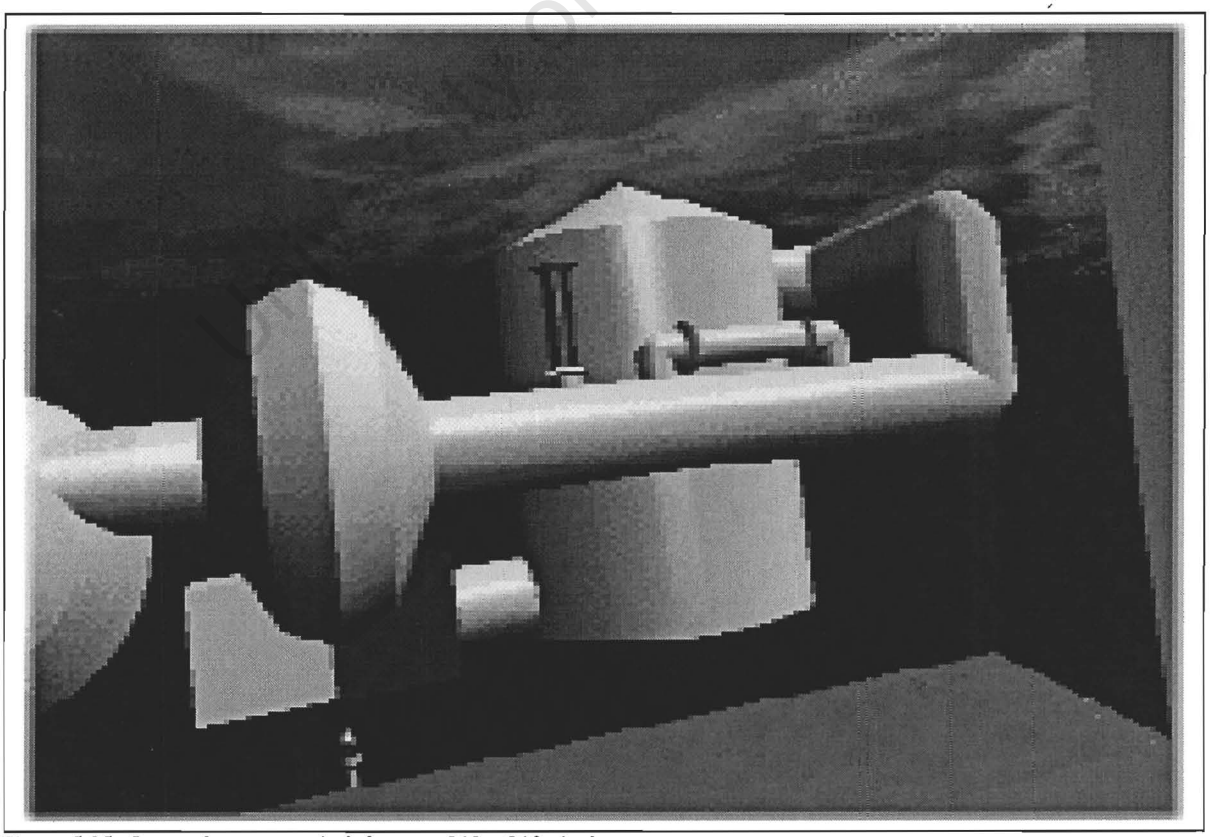


Figure 5.85 : Image that appears in left eye at 213 x 210 pixels

Demo

The next three pages show 18 frames following a sequence where a user puts one of the PLCs into local control and changes the setpoint. It can be seen how the flame has increased in size, followed by which the user looks at the graphs to see how the temperature is progressing. These next 18 frames form a selection of some of the couple hundred that made up this one minute long activity.

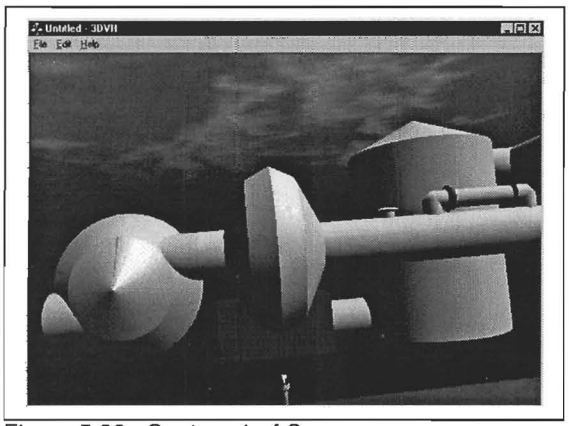


Figure 5.86 : Capture 1 of Sequence

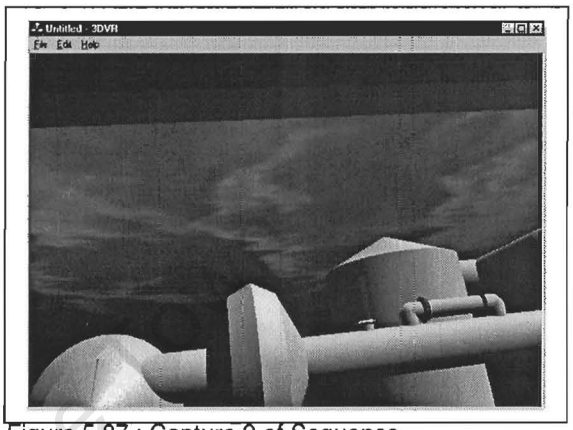


Figure 5.87 : Capture 2 of Sequence

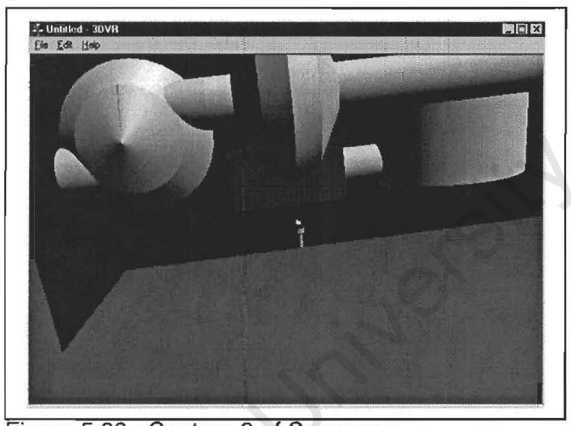


Figure 5.88 : Capture 3 of Sequence

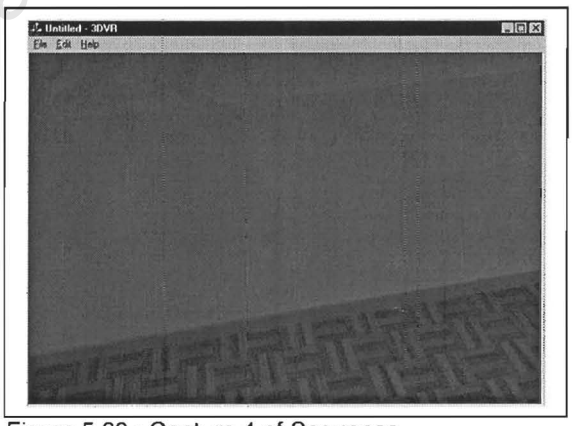


Figure 5.89 : Capture 4 of Sequence

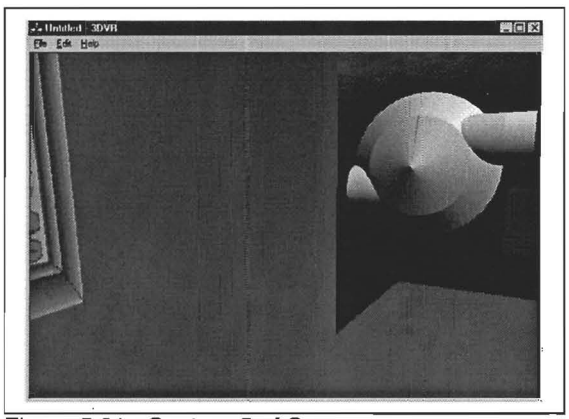


Figure 5.91 : Capture 5 of Sequence

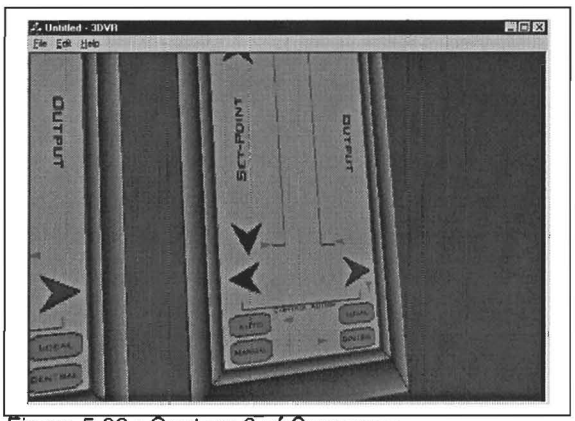


Figure 5.92 : Capture 6 of Sequence

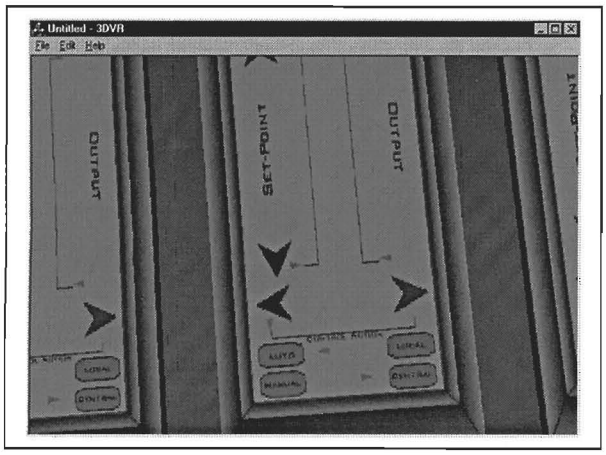


Figure 5.93 : Capture 7 of Sequence

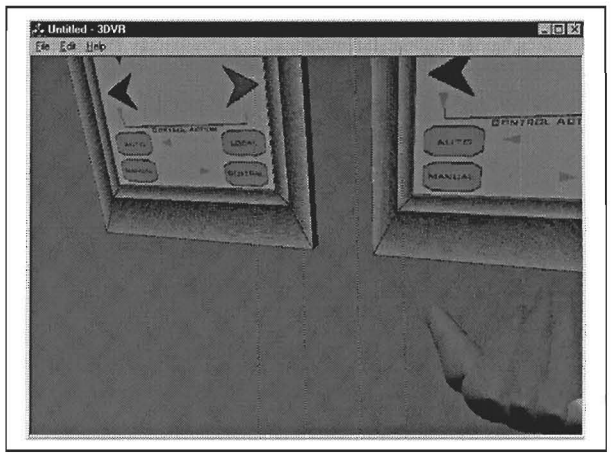


Figure 5.94 : Capture 8 of Sequence

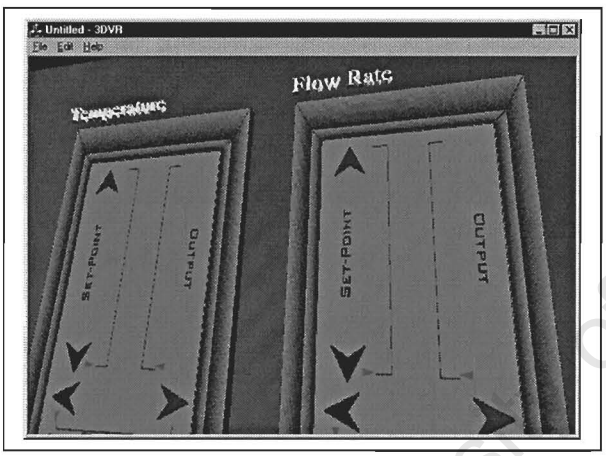


Figure 5.95 : Capture 9 of Sequence

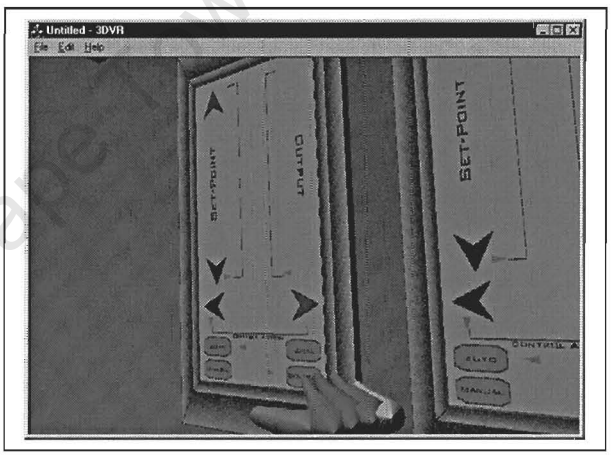


Figure 5.96 : Capture 10 of Sequence

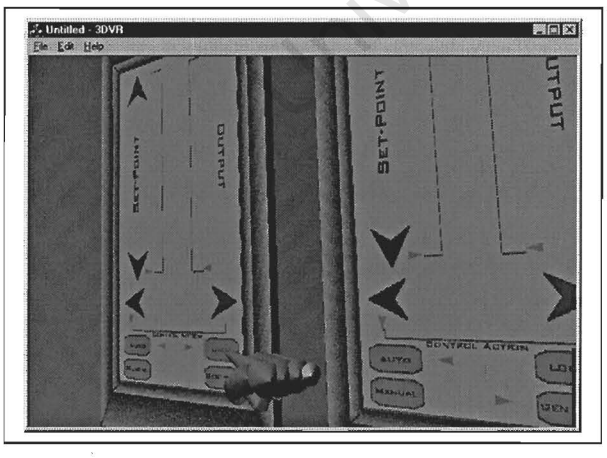


Figure 5.97 : Capture 11 of Sequence

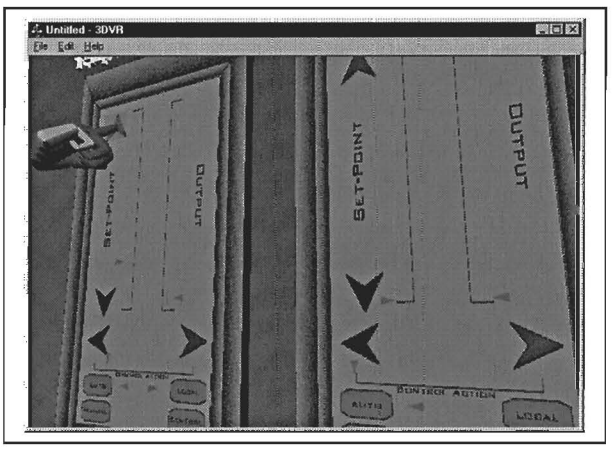


Figure 5.98 : Capture 12 of Sequence

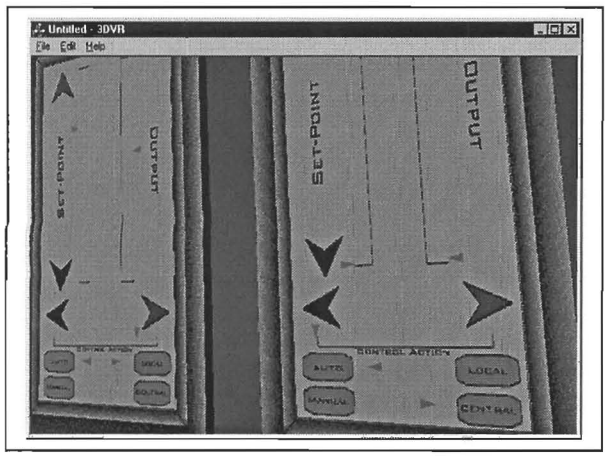


Figure 5.99 : Capture 13 of Sequence

Figure 5.100 : Capture 14 of Sequence

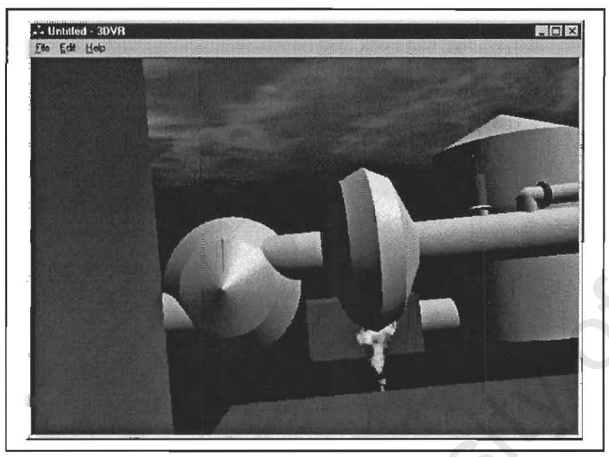


Figure 5.101 : Capture 15 of Sequence

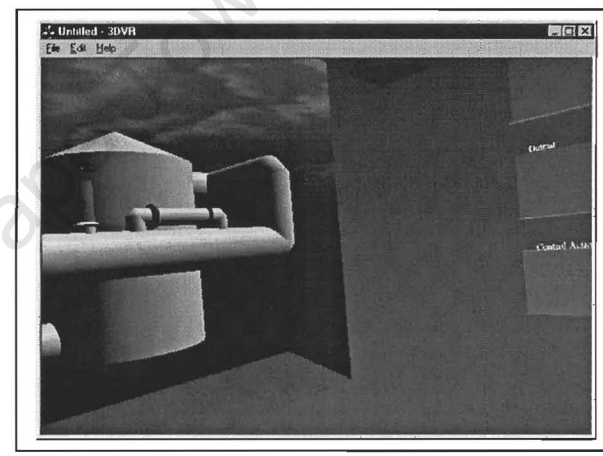


Figure 5.102 : Capture 16 of Sequence

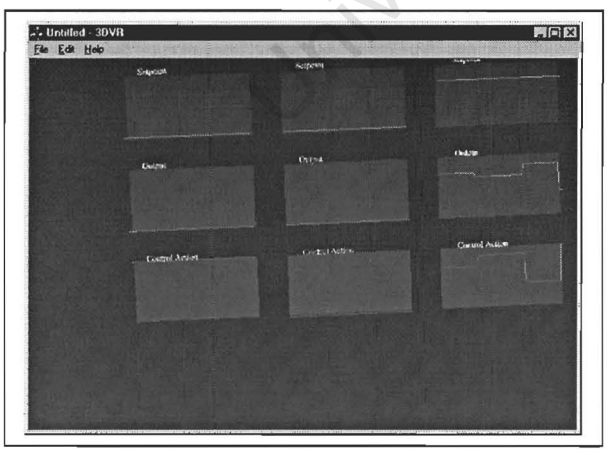


Figure 5.103 : Capture 17 of Sequence

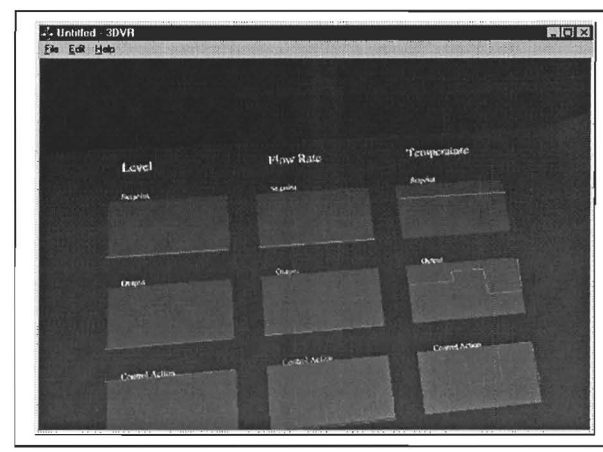


Figure 5.104 : Capture 18 of Sequence

5.3.2 Alternative Head Mounted Displays

In the last six months there has been a dramatic increase in the number of head mounted displays that have become available on the market, and even more that have been removed from the market. Where before there was only a few types of LCD panels that were used in almost all of the head mounted displays, recently there have been fairly large jumps in improvements in miniature LCD panel technology. Thus some of the HMDs below are considerably better, for a similar price, than the CyberEye that was ordered almost one year ago. They are organise in price order.

Glasstron PLM-A55 (Sony)

Stereo : No
Resolution : 800 (H) x 255 (V)
Contrast :
Field of View :
Display Type : LCD
Weight : 160g
Interface : NTSC
Price : $895

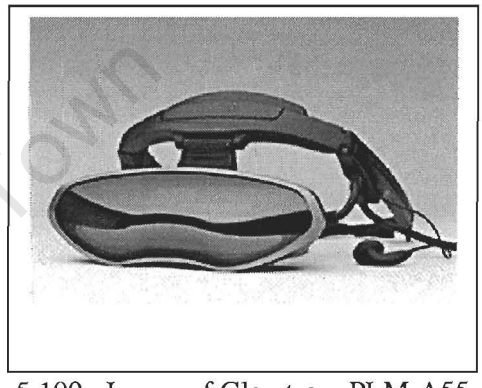


Figure 5.109 : Image of Glasstrom PLM-A55

VFX3D (Interactive Imaging)

Stereo : Yes
Resolution : 230,000 pixels
Contrast :
Field of View :
Display Type : LCD
Weight :
Interface : VGA
Price : $1 759

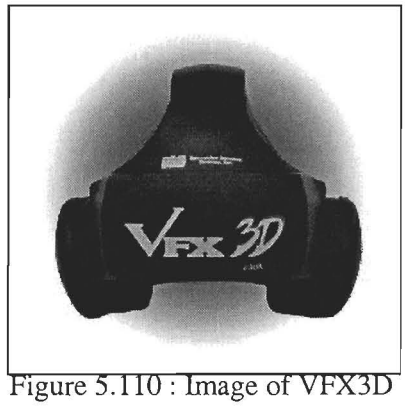


Figure 5.110 : Image of VFX3D

MRG4 (Liquid Image)

Stereo : No
Resolution : 480x234
Contrast : 30:1
Field of View : 61 Horizontal 46 Vertical
Display Type : LCD
Weight :
Interface : NTSC
Price : $2 195

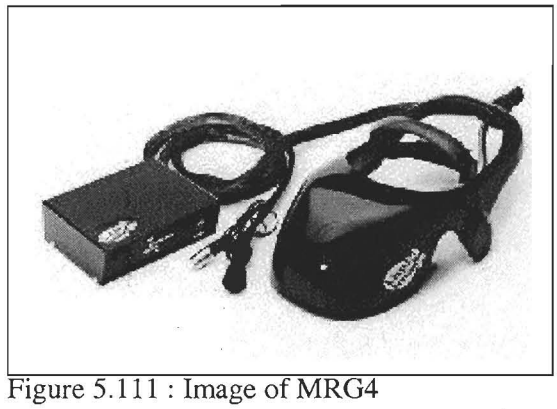


Figure 5.111 : Image of MRG4

Glasstron S700 (Sony)

Stereo : No
Resolution : 832 X 3 X 624
Contrast :
Field of View : 30 degrees H X 22.5 degrees V
Display Type : LCD
Weight : 120g
Interface : NTSC
Price : $2 995

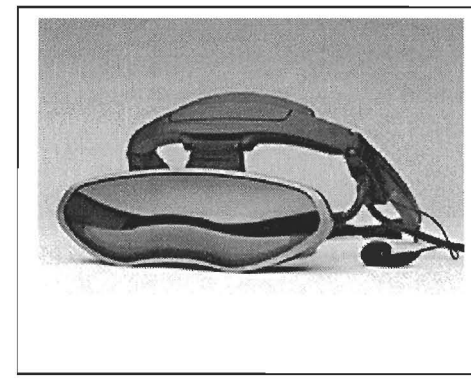


Figure 5.112 : Image of Glasstron S700

MRG2 (Liquid Image)

Stereo : No
Resolution : 720x240
Contrast : 40:1
Field of View : 84 Horizontal, 65 Vertical
Display Type : LCD
Weight :
Interface : NTSC
Price : $3 495

Figure 5.113 : Image of MRG2

Protec (I-O Display Systems)

Stereo : Yes
Resolution : 640 x 480
Contrast : 100:1
Field of View : 42 degrees diagonal
Display Type : LCD
Weight : 310g
Interface : 1 or 2 VGA
Price : $3 969

Figure 5.114 : Image of Protec

MRG3c (Liquid Image)

Stereo : No
Resolution : 768x556
Contrast : 100:1
Field of View : 84 Horizontal, 65 Vertical
Display Type : LCD
Weight :
Interface : NTSC/PAL
Price : $5 500

Figure 5.115 : Image of MRG3c

V6 (Virtual Research)

Stereo : Yes
Resolution : 640 x 480 colour elements
Contrast : 200:1
Field of View : 60° diagonal
Display Type : LCD
Weight : 820g
Interface : VGA
Price : $6 900

Figure 5.116 : Image of V6

X3 (Liquid Image)

Stereo : No
Resolution : 640 x 480 x 3
Contrast : 100:1
Field of View : 84 Horizontal, 65 Vertical
Display Type : LCD
Weight :
Interface : VGA
Price :6995

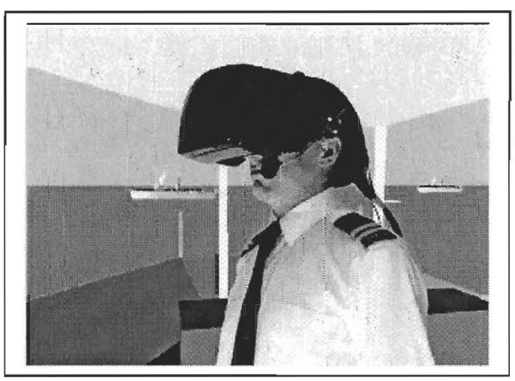


Figure 5.117 : Image of X3

ProView 30 (Kaiser Electro-Optics)

Stereo : Yes
Resolution : 640 x 3 x 480
Contrast : 25:1
Field of View : 18° (V) x 24° (H)
Display Type : LCD
Weight : 800g
Interface : One or two VGA
Price : $7 895

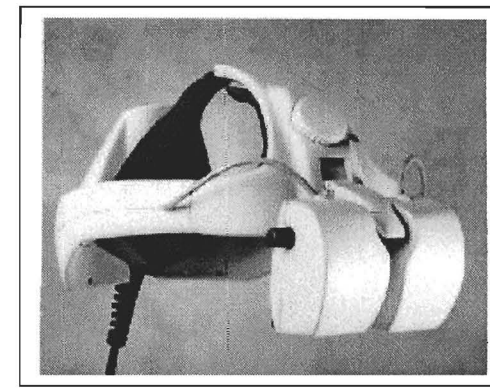


Figure 5.118 : Image of ProView 30

ProView 60 (Kaiser Electro-Optics)

Stereo : Yes
Resolution : 640 x 3 x 480
Contrast : 25:1
Field of View : 36° (V) x 48° (H)
Display Type : LCD
Weight : 750g
Interface : One or two VGA
Price : $11 595

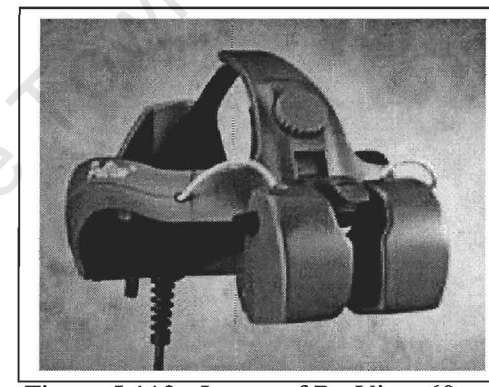


Figure 5.119 : Image of ProView 60

V8 (Virtual Research)

Stereo : Yes
Resolution : 1920 x 480 colour elements
Contrast : 200:1
Field of View : 60° diagonal
Display Type : LCD
Weight : 820g
Interface : VGA
Price : $11 900

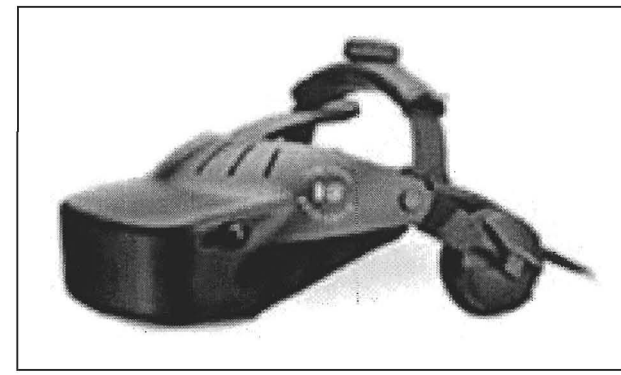


Figure 5.120 : Image of V8

ProView XL50 XGA (Kaiser Electro-Optics)

Stereo : Yes
Resolution : 1024 x 768
Contrast : 40:1
Field of View : One or two VGA
Display Type : LCD
Weight : 900g
Interface : 2 XGA (VGA)
Price : $15 000

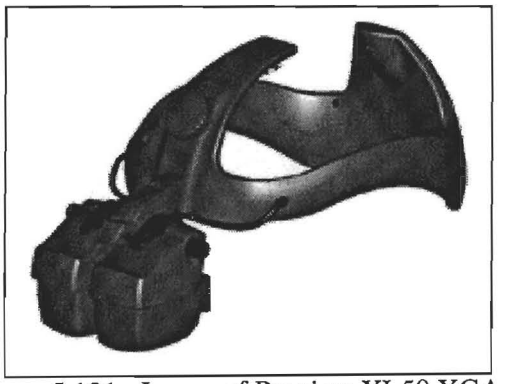


Figure 5.121 : Image of Proview XL50 XGA

ProView 80 (Kaiser Electro-Optics)

Stereo : Yes
Resolution : 640 x 3 x 480
Contrast : 20:1
Field of View : 50° (V) x 65° (H)
Display Type : LCD
Weight : 1000g
Interface : One or two VGA
Price : $34 745

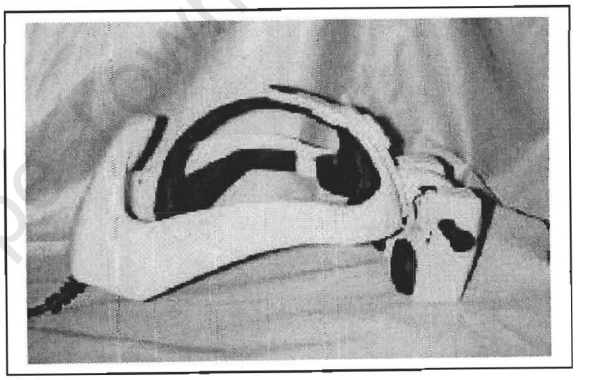


Figure 5.122 : Image of ProView 80

5. 4 Virtual World Performance

It is possible to have a near photo-realistic virtual control room, however it will be of no good if it takes 10 minutes to render a single frame. Thus scene complexity is traded off for a reasonable frame rate.

Frame rate is very much system dependent. A faster graphics card or a faster computer will improve the frame rate for a given virtual environment. However, no matter how good the computer is, most users can generate scenes complicated enough to bring any computer to a standstill. Thus there is a need to quantify how complicated a scene is. For example, what is the triangles or pixel per second rate?

These are non-uniform measurements as they are not necessarily comparable across different hardware. Even if two different sets of hardware are both rated for the same number of triangles per second. Add blending, or anti-aliasing to the scene and the difference in performance could easily be a factor of 10.

The system for this project consists of a Pentium MMX 166MHz with a Diamond Fire GL graphics card. It is faster in rendering than a Pentium II 400MHz with a standard graphics card, but could be even faster if the CPU was upgraded. The graphics card has been shown in tests to increase the speed of rendering almost linearly with an increased CPU speed. Thus the current CPU is forming the main bottleneck.

The virtual environment has been set so that it tries to redraw the scene every 20ms. Thus if the redraw takes less than 20ms, then it is not possible to pick up any changes in screen refresh rate. Each frame consists of 640 x 240 pixels (I.e. each view for each eye is counted as its own frame.) Starting with everything initialised, but with no drawing routines the measured frame rate is 50.4FPS. Adding in components, the following frame rates are recorded:

- Add Clear Screen : 50.4 fps
- Above + SwapBuffers : 50.4 fps
- Above + SetupPos : 50.4 fps
- Above + SetupLighting : 50.4 fps
- Above + DrawHand (hand not visible) : 50.4 fps
- Above with hand visible (side view) : 28.8 fps
- Above with hand visible (top view) : 28.7 fps

Note how initially there is no decrease in frame rate because all the processing takes less than 20ms. Also note how once the hand is drawn, it does not really matter how much of the hand is visible. The top view of the hand covers about 3 times the screen area of the side view as can be seen in the figures below.

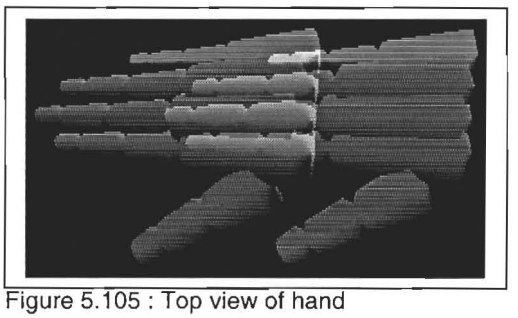


Figure 5.105 : Top view of hand

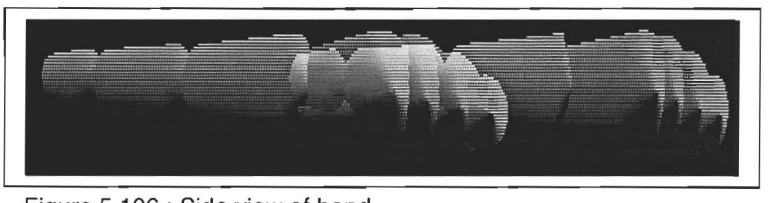


Figure 5.106 : Side view of hand

Next step disables the drawing of the fingers, and adds in a single finger at a time, so see how the frame rate decreases.

- No Fingers : 50.4 fps
- 1 Finger : 50.4 fps
- 2 Fingers : 50.4 fps
- 3 Fingers : 42.0 fps
- 4 Fingers : 33.1 fps
- 4 Fingers + Thumb : 28.8 fps

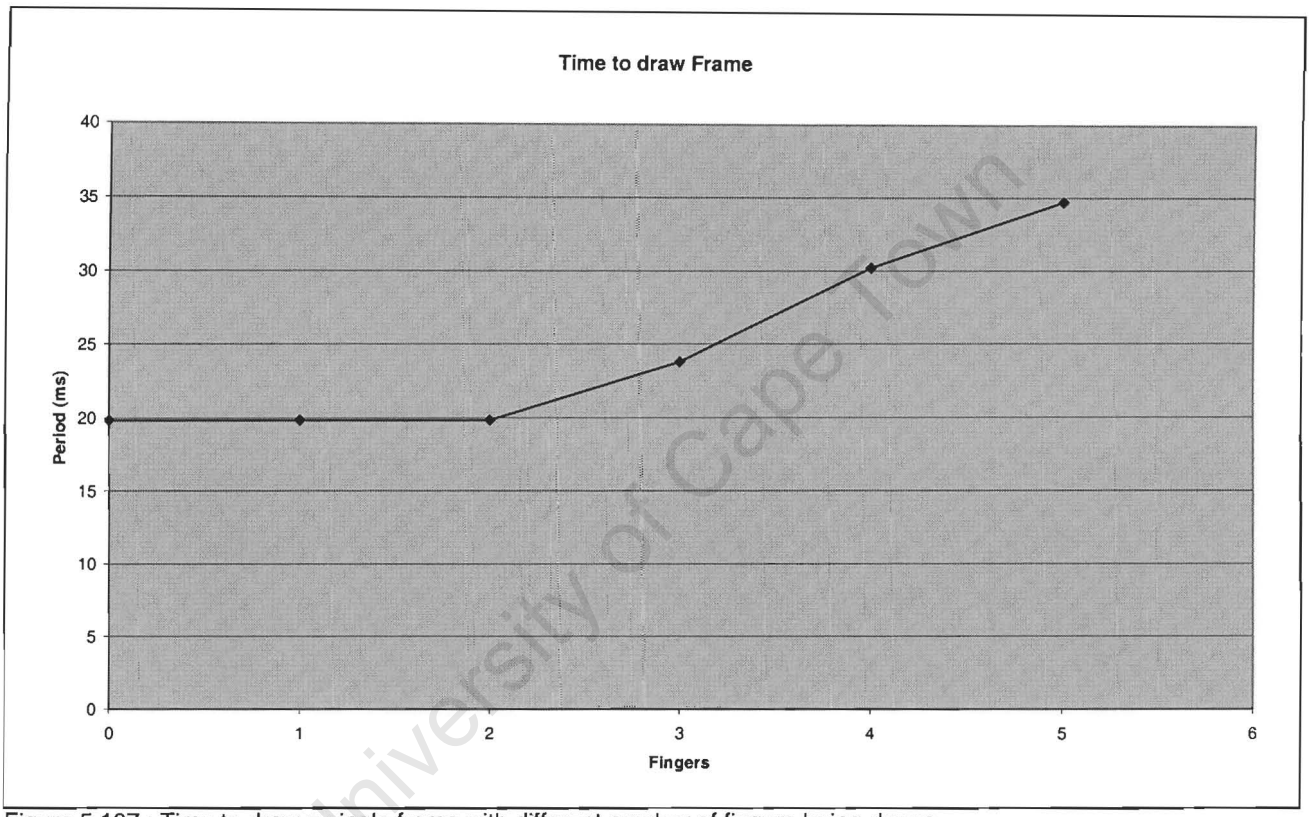


Figure 5.107 : Time to draw a single frame with different number of fingers being drawn

Inverting these to get the frame periods, and plotting them (above), it is possible to work back and find out that it takes 4.5 ms to draw the thumb, and 6.4ms to draw a finger. This would mean that the time spent doing background operations like swapping frames, clearing screen and event handling takes 34.7 – 4.5 – 6.4 * 4 = 4.6 ms. It thus takes 30.1ms to draw the full glove.

First thing to check is to see if the difference between a finger and a thumb seems accurate. Going through the source code, a finger is made up of:

- 12 SzConverts
- 15 Translates
- 3 Rotates

A thumb is made up of:

- 9 SzConvert
- 11 Translates
- 3 Rotates

Dissecting further, SzConvert to be made up of:

- 62 Vertices (60 Triangles)
- 31 Normals

Trying to see if there would be any speed difference if the normal vectors were commented in the SzConvert function it was found that the frame rate decreasing from 28.8 to 26.9. Thus having the normal vectors somehow speeds up the processing of the objects. Judging from the above, the thumb has only 75% of the statements that a finger executes. In comparison it takes only 70% of the time of drawing a finger. These two are fairly close, and it can thus be assumed that the timing estimates have some sense of accuracy.

Assume that the number of triangles form the main basis for the frame rate, then the number of triangles that there are need to be calculated. With 60 triangles per SxConvert call, and 12 such calls per finger and 9 in the thumb, there are 60 * (12 * 4 + 9) = 3060 triangles drawn in 30.1 ms. This averages to about 100 000 triangles per second.

Next step is to add in the factory. Adding in the factory, the frame rate increased from 28.8 fps (just the hand) to 32.6 fps (hand and factory). This seemed illogical, so tracing through all the statements it was found that by enabling the light sources in the DrawFactory function the rendering speed improved dramatically. Thus re-testing it was found that with light source 2 enables, the whole glove could be drawn within the minimum timing interval of 20ms. This will thus give a much higher rate of triangle drawing.

Removing all the objects that use textures, namely the sky, the floor, and the heater, the frame rate improved to 36.6 fps. Thus searching through DrawFactory, the following objects making up the factory (excluding the sky, floor, and heater):

- 5 Pipes
- 3 Knees
- 1 Pump
- 1 TempMeter
- 1 FlowMeter
- 1 SlushGate
- 1 Tank

Thus in terms of triangles:

- Pipe = 60 Triangles
- ShrinkPipe = 60 Triangles
- TempMeter = 3 Pipes, 4 ShrinkPipes = 420 triangles
- FlowMeter = 6 Pipes, 6 ShrinkPipes = 720 triangles
- Knee = 60 Triangles
- Pump = 2 Pipes, 4 ShrinkPipes = 360 triangles
- Tank = 2 Pipes, 2 ShrinkPipes = 240 triangles
- Cube = 12 triangles
- SlushGate = 3 Cubes = 36 trangles

Adding this all up within the simplified factory gives the number of triangles = 5*60 + 3*60 + 360 + 420 + 720 + 36 + 240 = 2256. Adding the 3060 triangles from the hand, there are a total of 5316 triangles for a frame rate of 36.6 fps. Inverting the frame rate and subtracting the 4.6ms, calculated previously as the set up time, the time to draw the factory and the glove becomes 22.7ms. This converts to a triangle rate of 230 000 triangles per second. This more than double improvement in speed is due just to the inclusion of a light source. This illustrates the problem experienced in performance optimisation without having a very detailed knowledge on how the CPU and graphics card handle the individual statements.

Next the three outstanding parts of the factory are added, yielding the following frame rates:

- Full Factory : 33.3 fps
- Without Floor : 34.4 fps
- Without Sky : 34.4 fps
- Without Floor and Sky : 35.2 fps
- Without Heater or Floor or Sky : 36.6 fps

Thus the following times can be calculated for the three texture objects

- Floor (193k texture) : 0.6ms
- Sky (193k texture) : 0.6ms
- Heater (16k texture + 600 triangles) : 1.1ms

As calculated previously, using the triangles per second rate, it takes 2.5ms to draw the 600 triangles needed for the heater virtual object. Yet it takes less than 1ms to draw these triangles with a small texture.

Adding in the ControlRoom, with the 3 PLCs removed, the frame rate slows down to 29.2 (while still looking at the factory floor.) This means that it takes 3.4ms to filter out all the objects that make up the room. As one cannot see them while still looking at the factory, none of these objects need to be drawn.

If the viewpoint changes to look at any point of the factory (e.g. looking at the walls, floor or ceiling) the frame rate increases to 50.4 fps. This means that to draw any part of the control room and to filter out all the objects from the factory and data glove that are not visible, takes less than 20ms.

Working back to primitives, the control room is made up of the following objects :

- Obj.Lights (22 triangles)
- Obj.Walls (22 triangles)
- Obj.Skirting (32 triangles)
- Obj.FloorCeilingPanel (102 triangles + 3 textures)

This gives a total of 178 triangles and 3 textures. Thus working on our previously calculated triangle and texture rates, the time to draw the factory ought to be 1ms (triangles) + 2ms (textures) = 3ms. This is a very small amount of time in the great scheme of the environment. The control room is however going to suffer reduced performance because it totally encloses the whole view, and every pixel needs to be rendered.

The next step is to add the PLCs. The graph below shows the increase in rendering time as I increase the number of PLC virtual objects is increased.

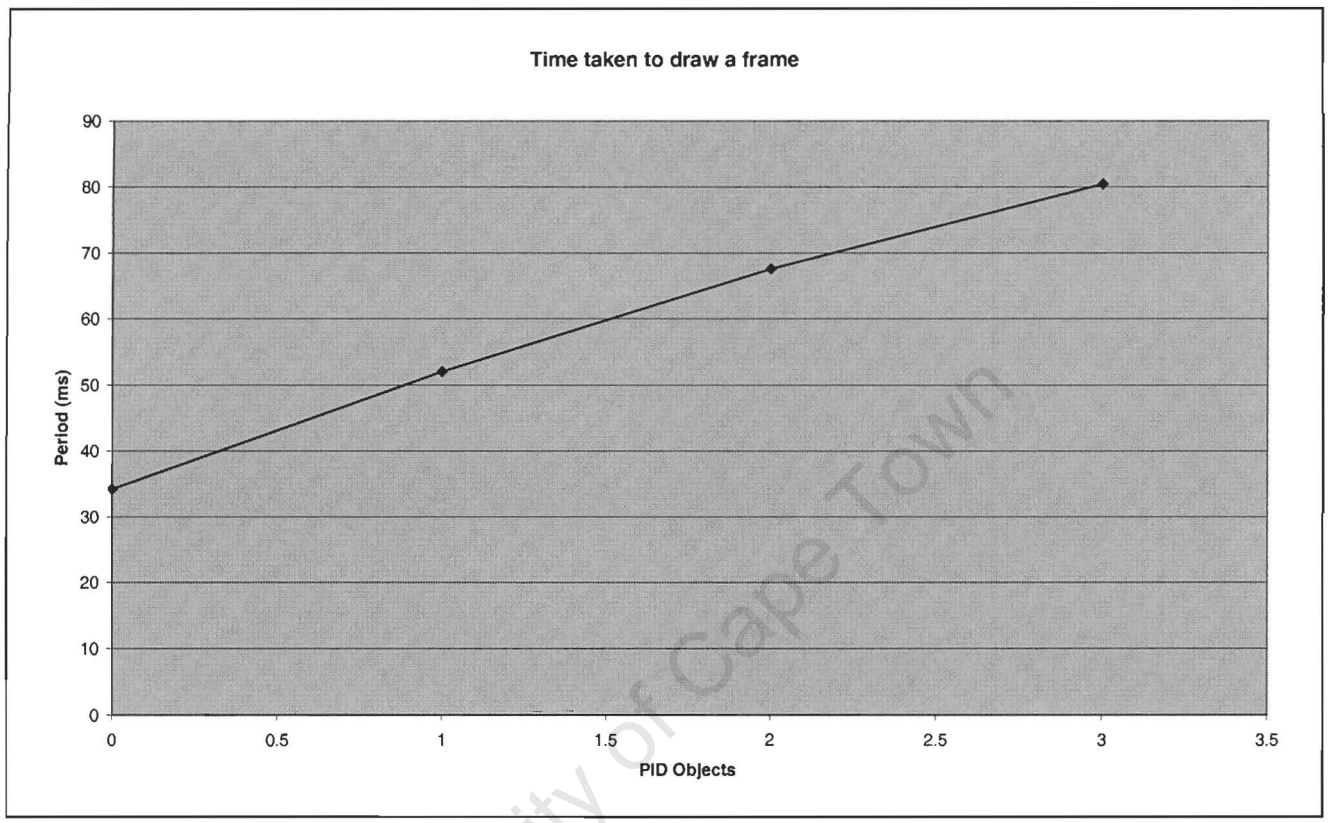


Figure 5.108 : Graph showing rendering time increasing with different numbers of PID objects visible.

The above graph shows an average cost of 15ms to draw each PLC object. Back to the basics, the PLC object is made up of:

- 15 Triangles
- 1 Texture (256k)
- 1 DrawGraph
- 1 DrawData

I take some more measurements to see how much time is spent in these various sections.

- 3 PID objects : 12.39 fps
- With No Graphs : 22.46 fps
- With No Data : 12.65 fps
- With neither Graphs or Data : 23.0 fps
- With No Title : 13.88 fps
- With No Texture : 12.45 fps

The above frame rate measurement yield the following results :

- 36.2 ms to draw the Graphs (6 Triangles + 4 Strings)
- 1.7 ms to draw the data on the Graphs (300 line segments)
- 8.7 ms to draw the Title (1 String)
- 0.4 ms to draw the Texture

Firstly it should be noted how quickly it takes to draw the textures. On normal graphics cards, textures can take as much as a few hundred milliseconds each. The Permedia 2 allows the textures to be stored on the graphics card, instead of computer memory, and thus they need not be copied across when used. This topic was discussed earlier in Chapter 3.4.2.

Next comes the check to see how long it takes to draw the graphs. Comparing the time it takes to draw the titles above the PLCs to the time spent in the DrawGraphs function, It can be calculated that the main time consumption occurs in drawing the strings. The titles for the PLCs and the Graphs are 3D renderings of true type fonts. These renderings might themselves require thousands of triangles per letter and thus cause the slow down experienced. If the view is changed to look at the graphs the frame rate drops even lower to 2.5 fps. A quicker way to display strings is as a pre-drawn texture map.

The final part of the control room is to check if the user's hand has made contact with any of the buttons. This is a very complicated procedure because of the complexity of the data glove. To check if a single 3D point is in contact with a button is done by simply checking if its X, Y and Z values fall within a certain bounding volume. This is simply and fast, however the data glove has thousands of points on its surface. Thus the way the VR environment checks for contact between the glove and the button is to clip the volume of the whole environment to the dimensions of a button, and draw the hand. If any part of the hand is drawn within this volume then it means that part of it falls within the volume occupied by the button, and thus contact is occurring.

This procedure needs to be repeated for the eight buttons for each PLC display panel. Thus the glove has to be redrawn 24 times. This equals over 70 000 triangles per frame. Thus it should not be surprising when the frame rate drops down from 12.39 fps to 5.0 fps when the contact check for a single PLC controller is included. This indicates that it takes 120ms to check for contact of the glove with one set of eight buttons. This agrees with previous calculations where drawing 24 480 triangles at 230 000 triangles per second works out to take 110ms.

Other things that influence rendering speed are shading, blending, and anti-aliasing. Because most of these are implemented in hardware, removing smooth shading, removing the Blending of transparent images with the background, and disabling the Stencil test for line sequential stereo images showed marginal if any improvement on the speed of rendering.

Recommendations

For future development the following should be tried to help improve the scene complexity that can be achieve with still usable frame rates.

- Don't use the data glove, it takes too long to draw, and too long to check if it has come into contact with the buttons. Use a pointer instead. This way only a simple check, to see if a single point has made contact with a button, is necessary. Use the second receiver mounted on a 50cm pole as a pointer.
- Don't use 3D true type fonts. Use bitmaps or textures instead to draw text strings

Chapter 6
Conclusions

Every aspect of the hardware and software has been covered, with both tested them from every possible angle. Based on all the results from the above chapters the following conclusions can be made.

The data glove is unnecessary. Without force feedback, there is no advantage in having a data glove over a pointer stick. Having a six degree of freedom point in 3D space to wave around is simpler to use than a whole data glove, and achieves a similar objective. It will only be when force feedback becomes inexpensive and simple to use that the advantage of having a hand to manipulate objects in virtual space will prove an advantage. Without force feedback, the glove is of not much use beyond single finger button presses.

The tracking system is currently accurate enough for a Control Engineering environment (maybe not for medical use), however the range is still far short of what it needs to be. A reasonable range would be three meters, and not 75 centimetres.

The head mounted display still lacks in resolution. I found no problems with the field of view, but objects were not clear enough for the labels on the buttons to be distinguishable. This causes problems when the user needs to read any form of labels, as these only become readable when the user is very close to them. Much higher resolution displays do exist, but at costs that need to come down before being accepted by the mass market.

Display technology will never be good enough. Even in a hundred years scenes will be generated that will bring computers of that day to a stand still. However, with today's hardware, it is possible to create an environment that is realistic enough that it doesn't feel uncomfortably unreal. Realism that could fool the operator for the real world is still decades away, but realism that a user is happy with, and able to work in, is currently available.

Computer hardware is still not reliable enough for a virtual environment to replace a control room in safety critical operations. Currently it should only be used for training and as a supplement (maybe for remote monitoring) to existing control rooms.

In short the technology is almost satisfactory. Although better equipment can be purchased with a larger budget, the cost effective of the virtual environment has to first justify such expenditure. The equipment has to be economically viable. The top of the range systems provide the capabilities that are needed, yet only the bottom of the range equipment is affordable.

Thus all that is required is time for the prices of the technology to drop. If the price versus performance trends of VR equipment carries forward at the same rate as that of desktop PC hardware, then within three to five years the applications expressed in the introduction of this dissertation will be possible and cost-effective.

Chapter 7
Recommendations

Based on the results from the above chapters and the conclusions, the following recommendations can be made:

- Further research into the usefulness of 3D sound is required
- Change the glove to a pointer stick for use over then next about five years.
- Try and implement a virtual control room, for monitoring purposes only, for a real installation. Choose a situation that the VR system can only aid the existing system. It must not be able to disrupt the existing system if it fails.
- Update the hardware and software on an 18-month basis, as re-development time should be less than 6 months and considerable jumps in technology take place within a 18 month period.
- Sell it to users as a free supplement to existing control systems being sold. VR has considerable marketing appeal, even if its functionality is still limited. It is also sufficiently low cost, that it should not increase the cost of the original system. From a sales point of view, virtual reality has public appeal, and makes people believe that the company providing the control system is at the forefront of technology.
- Once users of the system start spending as much time in the virtual environment as in the real environment, start up new research on the reliability of the technology, and start selling it as an alternative to existing control systems.

Chapter 8
Bibliography

3Space InsideTRAK

Programmer's Supplement to 3Space InsideTRAK User's Manual
Polhemus Incorporated
October 1996

3Space InsideTRAK User's Manual
Polhemus Incorporated
December 1993

Device Drivers

Windows NT DDK Programmers Guide (Online Book)
Microsoft Corporation
1996

Windows NT DDK Kernel Mode Drivers (Online Book)
Microsoft Corporation
1996

Visual C++ Programming

Special Edition Using Visual C++ 5
by Kate Gregory
Que Corporation
1997

OpenGL Programming

OpenGL Programming for Windows 95 and Windows NT
by Ron Fosner
Addison-Wesley Developers Press
1997

Video Accelerator Cards

Permedia 2, GLINT GMX
3D Labs
http://www.3dlabs.com/
1999

Intense 3D Wildcat
Intergraph
http://www.intergraph.com/ics/tdz2000/2000go.asp
1999

Voodoo 2
3Dfx
http://www.3dfx.com/
1999

Interaction Devices

DataGlove
Fifth Dimension Technologies
http://www.5dt.com/
1999

CyberGlove, CyberTouch, CyberGrasp
Virtual Technologies
http://www.virtex.com/~virtex/
1999

Phantom
SenseAble Technologies
http://www.sensable.com/
1999

"Phantom-Based Interaction with Virtual Objects"
Salisbury, J.K. and M.A. Srinivasan.
IEEE Computer Graphics and Applications,
September-October 1997, Vol. 17, No. 5
IEEE Computer Society.

Tracking Systems

InsideTrak, IsoTrak II, FastTrak, Long Ranger, StarTrak
Polhemus
http://www.polhemus.com/
1999

pcBIRD, MotionStar Wireless
Ascension
http://www.ascension.com/
1999

Head Mounted Displays

Ian's VR Buying Guide
http://www.cs.jhu.edu/~feldberg/vr/vrbg.html
Ian Feldberg
June 1997

HMD/VR-helmet Comparison Chart
http://www.stereo3d.com/hmd.htm
Christoph Bungert
1999

The Virtual Reality Source
http://www.thevrsource.com/
The Virtual Reality Source
1999

V6, V8
Virtual Research
http://www.virtualresearch.com
1999

MRG2, 3c, 4, X3
Liquid Image
http://www.mbnet.mb.ca/vr/hmd.html
1999

Proview 30, 60, 80, XL50 XGA
Kaiser Electr-Optics
http://www.keo.com
1999

Protec
I-0 Display Systems
http://www.iglasses.com/
1999

Glasstron PLM-A55, S700
Sony
http://www.sony.co.jp/
1999

VFX3D
Interactive Imaging
http://www.iisvr.com/
1999

Literature Review

Przemyslaw Rokita
Generating Depth-of-Field Effects in Virtual Reality Applications
IEEE Computer Graphics and Applications
March 1996
pp. 18–21

Dave Pape
A Harware-Independent Virtual Reality Development System
IEEE Computer Graphics and Applications
July 1996
pp. 44-47

Jason Leigh and Andrew E. Johnson
Supporting Transcontinental Collaborative Work in Persistent Virtual Environments
IEEE Computer Graphics and Applications
July 1996
pp. 47-50

James Cremer, Joseph Kearney and Yiannis Papelis
Driving Simulation: Challenges for VR Technology
IEEE Computer Graphics and Applications
September 1996
pp. 16-20

Lawrence Rosenblum, Jim Durbin, Upul Obeysekare, Linda Sibert, David Tate, James Templeman, Jyoti Agrawal, Daniel Fasulo, Thomas Meyer, Greg Newton, and Amit Shalev
Shipboard VR: From Damage Control to Design
IEEE Computer Graphics and Applications
November 1996
pp. 10-13

Larry F.Hodges, Benjamin A. Watson, G. Drew Kessler, Barbara O. Rothbaum, Dan Opdyke
Virtually Conquering Fear of Flying
IEEE Computer Graphics and Applications
November 1996
pp. 42-48

John W. Barrus, Richard C. Waters, and David B. Anderson
Locales: Supporting Large Multi-user Virtual Environments
IEEE Computer Graphics and Applications
November 1996
pp. 50-56

Kelly Gaither, Robert Moorhead, Scott Nations, Dan Fox
Visualizing Ocean Circulation Models Through Virtual Environments
IEEE Computer Graphics and Applications
January – February 1997
pp. 16-19

Valerie D. Lehner and Thomas A DeFanti
Distributed Virtual Reality: Supporting Remote Collaboration in Vehicle Design
IEEE Computer Graphics and Applications
March - April 1997
pp. 13 - 17

Gerold Wesche, Jurgen Wind and Martin Gobel
Visualization on the Responsive Workbench
IEEE Computer Graphics and Applications
July - August 1997
pp. 10 - 12

Larry Rosenblum, Jim Durbin, Robert Doyle, David Tate and Robert King
Situational Awareness Using the Responsive Workbench
IEEE Computer Graphics and Applications
July - August 1997
pp. 12 - 13

Bernd Frohlich, Martin Fischer, Maneesh Agrawala, Andrew Beers and Pat Hanraham
Collaborative Production Modeling and Planning
IEEE Computer Graphics and Applications
July - August 1997
pp. 13 - 15

Steve Bryson
The Virtual Windtunnel on the Virtual Workbench
IEEE Computer Graphics and Applications
July - August 1997
pp. 15

Michael J.Potel (Editor)
Virtual Reality Provides Real Therapy
IEEE Computer Graphics and Applications
July - August 1997
pp. 15-20

J. Kenneth Salisbury and Mandayam A. Srinivasan
Phantom-Based Haptic Interaction with Virtual Objects
IEEE Computer Graphics and Applications
September - October 1997
pp. 6-10

Andrew S. Forsberg, Joseph J. LaViola, Jr., Lee Markosian, and Robert C. Zeleznik
Seamless Interaction in Virtual Reality
IEEE Computer Graphics and Applications
November - December 1997
pp. 6-9

Don Allison, Brian Wills, Doug Bowman, Jean Wineman, and Larry F. Hodges
The Virtual Reality Gorilla Exhibit
IEEE Computer Graphics and Applications
November - December 1997
pp. 30-37

Douglas J. Wiley and James K. Hahn
Interpolation Synthesis of Articulated Figure Motion
IEEE Computer Graphics and Applications
November - December 1997
pp. 39-45

Jeffrey D. Cress, Lawrence J. Hettinger, James A. Cunningham, Gary E. Riccio, Michael W. Haas and Grant R. McMillan
Integrating Vestibular Displays for VE and Airborne Applications
IEEE Computer Graphics and Applications
November - December 1997
pp. 47-51

Yasushi Ikei, Kazufumi Wakamatsu, and Shuichi Fukada
Vibratory Tactile Display of Image-Based Textures
IEEE Computer Graphics and Applications
November - December 1997
pp. 53-60

Luc Emering, Ronan Boulic, and Daniel Thalmann
Interacting with Virtual Humans through Body Actions
IEEE Computer Graphics and Applications
January – February 1998
pp. 8-11

Andy Johnson, Jason Leigh, and Jim Costigan
Multiway Tele-Immersion at Supercomputing 97
IEEE Computer Graphics and Applications
July – August 1998
pp. 6-9

Doug A. Bowman, Jean Wineman, Larry F. Hodges, and Don Allison
Designing Animal Habitats within an Immersive VE
IEEE Computer Graphics and Applications
September – October 1998
pp. 9-12

Ben Delaney
On the Trail of the Shadow Woman: The Mystery of Motion Capture
IEEE Computer Graphics and Applications
September – October 1998
pp. 14-19

Marc Cavazza, Rae Earnshaw, Nadia Magnenat-Thalmann, and Daniel Thalmann
Motion Control of Virtual Humans
IEEE Computer Graphics and Applications
September – October 1998
pp. 24 – 29

Prem Kalra, Nadia Magnenat-Thalmann, Laurent Moccozet, Gael Sannier, Amaury Aubel, and Danirl Thalmann
Real-Time Animation of Realistic Virtual Humans
IEEE Computer Graphics and Applications
September – October 1998
pp. 42-54

Susumu Tachi
Real-time Remote Robotics – Towards Networked Telexistance
IEEE Computer Graphics and Applications
November – December 1998
pp. 6-9

Mark Billinghurst, Jerry Bowskill, Nick Dyer, and Jason Morthett
Spatial Information Displays on a Wearable Computer
IEEE Computer Graphics and Applications
November – December 1998
pp. 24-30

Naoya Asamura, Nozomu Yokoyama, and Hiroyuki Shinoda
Selectively Stimulating Skin Receptors for Tactile Display
IEEE Computer Graphics and Applications
November – December 1998
pp. 32-37

Martin Schulz, Thomas Reuding, Thomas Ertl
Analyzing Engineering Simulations in a Virtual Environment
IEEE Computer Graphics and Applications
November – December 1998
pp. 46-51

Jolanda Tromp, Adrian Bullock, Anthony Steed, Amela Sadagic, Mel Slater, Emmanuel Frecon
Small Group Behavior Experiments in the Coven Projects
IEEE Computer Graphics and Applications
November – December 1998
pp. 53-62

Brian Goldiez, Rodney Rogers, Pam Woodard
Real-Time Visual Simulation on PCs
IEEE Computer Graphics and Applications
January – February 1999
pp. 11-15

Mel Slater, Davis-Paul Pertaub, and Anthony Steed
Public Speaking in Virtual Reality: Facing an Audience of Avatars
IEEE Computer Graphics and Applications
March – April 1999
pp. 6-9

Appendix A
InsideTrak Commands

Commands are sent by writing the corresponding values to the input FIFO buffer. They are either single bytes, or strings that are terminated by 0x0d or 0x0a. I will now present the commands that the Polhemus InsideTRAK makes available to the user. Most of these commands are available in the development manuals that come with the InsideTRAK, but are included here so that the reader can understand the functionality provided by this particular tracking system.

Alignment Commands

The alignment commands allow the user to define an origin from which the X, Y, Z measurements are referenced, and to define a measurement plane. For example, if there is a sloped surface to measure and it is necessary to have the X, Y, Z outputs measured with respect to the reference frame defined by this sloped surface, then the alignment commands allow you to do this.

Alignment data consists of the co-ordinates, in the transmitter reference frame, of three non-collinear points in space that are used to define the "alignment reference frame." The first point is the origin of the alignment reference frame. A line from the origin through the second point defines the positive X-axis of the alignment reference frame. The plane defined by all three points defines the XY-plane of the alignment reference frame; the positive Y-direction being from the X-axis towards the third point. The positive Z-Axis is determined by the "right-hand rule" convention for co-ordinate systems.

The units of the co-ordinates are interpreted according to the value of UNITS as set by the "U" and "u" command.

A

Syntax : Astation,[Ox],[Oy],[Oz],[Xx],[Xy],[Xz],[Yx],[Yy],[Yz]<cr>

Example : "A1,O10.5,O-5.5,O3.23,X1.3,X12.72,X-10.78,Y7.7,Y1.1,Y-8.0"<cr>

Note : This command operates incrementally. If the command is entered and the user then changes his / her mind, the 'R' command must be used to reset the alignment reference frame before the command is re-entered.

Parameters :

station	: to specify the relevant transmitter / receiver pair.
Ox, Oy, Oz	: the cartesian co-ordinates of the origin of the new reference frame.
Xx, Xy, Xz	: the co-ordinates of the point defining the positive direction of the X-axis of the new reference frame.
Yx, Yy, Yz	: the co-ordinates of a third point that is in the positive Y direction from the X-axis.

If all of the optional parameters are omitted, then the command returns the current co-ordinate values to the host. The returned sting is structured as follows :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "A"	1 ASCII character
4-24	Origin Co-ordinates	3 signed numeric strings
25-45	Positive X-axis co-ordinates	3 signed numeric strings
46-66	Positive Y-axis co-ordinates	3 signed numeric strings
67-68	Carriage return, line feed	2 ASCII characters

Where a signed numeric string is defined as follows :

Byte(s)	Identification	Format
1	Sign : either "+" or "-"	1 ASCII character
2	Number "0" to "9"	1 ASCII character
3	Number "0" to "9"	1 ASCII character
4	Number "0" to "9"	1 ASCII character
5	Decimal point = "."	1 ASCII character
6	Number "0" to "9"	1 ASCII character
7	Number "0" to "9"	1 ASCII character

Example Output : "21A+010.50-005.50+003.23+001.30+012.72-010.78+007.70+001.10-008.00"<cr><lf>

R

Syntax : Rstation<cr>

Example : "R1"<cr>

Purpose : This command resets the alignment reference frame for the specified station to the station reference frame. It provides an easy way to re-align the reference frame to the factory default values.

Parameters : station : to specify the relevant transmitter / receiver pair.

Boresight

The boresight function allows one to designate any receiver orientation as the zero orientation point. For example, the receiver may be mounted on a persons head to measure where it is pointing. When the user's head is looking at a given object, he may want the system angular outputs to be zero. The user can designate this receiver orientation as the zero orientation by giving the boresight command.

This results in azimuth, elevation, and roll outputs of zero at this orientation. As the user's head moves from the boresight point, the orientation angles are still measured in the designated reference frame, with the zero points shifted to the point where the boresight occurred.

If the alignment command "A" has been previously invoked, the results of boresight are unpredictable and therefore the combination should be avoided.

B

Syntax : Bstation<cr>

Example : "B1"<cr>

Purpose : This command causes the system to redefine the specified station line of sight values as the new zero reference line of sight. This results in azimuth, elevation and roll outputs equal to the boresight reference values at the current orientaion. The system default boresight matrix is the identity.

Parameters : station : to specify the relevant transmitter / receiver pair.

G

Syntax : Gstation,[aref],[eref],[rref]<cr>

Example : "G2,175.0,-67.67,90.0"<cr>

Purpose : This command establishes the boresight reference angles for a particular station. When the system is subsequently boresighted, the line-of-sight vector will assume these values. The system default boresight reference values are : 0, 0, 0

Parameters : station : to specify the relevant transmitter / receiver pair whose reference angles are to be fixed.
: aref : the azimuth reference angle.
: eref : the elevation reference angle.
: rref : the roll reference angle.

If all the optional parameters are omitted, then the system returns the boresight reference angles for the specified station as a string of characters that are defined as follows.

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "G"	1 ASCII character
4-10	Azimuth reference angle	1 signed numeric strings
11-17	Elevation reference angle	1 signed numeric strings
18-24	Roll reference angle	1 signed numeric strings
25-26	Carriage return, line feed	2 ASCII characters

Where a signed numeric string is defined as was shown in the Alignment command:

Example Output : "21G+175.00-067.67+090.00"<cr><lf>

b

Syntax : bstation<cr>

Example : "b1"<cr>

Purpose : The system boresight rotation matrix is reset to the identity matrix for the specified station.

Parameters : station : to specify the relevant transmitter / receiver pair whose reference angles are to be fixed.

r

Syntax : rTransmitter,[A],[E],[R]<>

Example : "r1,123.452,-165.43,1.25"<cr>

Purpose : This command provides a means of electronically modifying the mounting frame of the transmitter relative to a particular receiver. This command modifies the values of the transmitter mounting frame co-ordinates when it is used with an associated receiver.

Parameters : Transmitter : Always = 1
: A : Mounting frame azimuth angle in degrees
: E : Mounting frame elevation angle in degrees
: R : Mounting frame roll angle in degrees

If the optional parameters are omitted, then the system returns the current values of the transmitter mounting frame co-ordinates relative to the associated receiver. The output string returned is specified as follows :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "r"	1 ASCII character
4-11	Azimuth mounting frame angle	1 signed numeric strings
12-19	Elevation mounting frame angle	1 signed numeric strings
20-27	Roll mounting frame angle	1 signed numeric strings
28-29	Carriage return, line feed	2 ASCII characters
30	NULL value	1 ASCII character

Where a signed numeric string is now defined as follows :

Byte(s)	Identification	Format
1	Sign : either "+" or "-"	1 ASCII character
2	Number "0" to "9"	1 ASCII character
3	Number "0" to "9"	1 ASCII character
4	Number "0" to "9"	1 ASCII character
5	Decimal point = "."	1 ASCII character
6	Number "0" to "9"	1 ASCII character
7	Number "0" to "9"	1 ASCII character
8	Number "0" to "9"	1 ASCII character

Example Output : "21r+123.452-165.430+001.250"<cr><lf><NULL>

s

Syntax : sReceiver,[A],[E],[R]<>

Example : "s1,123.456,-132.6,0.01"<cr>

Purpose : This command provides a means of modifying the boresight angles from which the receiver's boresight matrix is re-calculated. (I.e. a boresight of the specified receiver)

Parameters : Receiver : 1 or 2
: A : Boresight azimuth angle in degrees
: E : Boresight elevation angle in degrees
: R : Boresight roll angle in degrees

If all the optional parameters are missing, then the system returns the current values of the specified receiver's boresight angles. The returned string is structures as follows.

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "s"	1 ASCII character
4-11	Azimuth mounting frame angle	1 signed numeric strings
12-19	Elevation mounting frame angle	1 signed numeric strings
20-27	Roll mounting frame angle	1 signed numeric strings
28-29	Carriage return, line feed	2 ASCII characters
30	NULL value	1 ASCII character

Where a signed numeric string is now defined as was defined in command "r" :

Example Output : "21s+123.456,-132.600,+000.010"<cr><lf><NULL>

Compensation Commands

Compensation refers to the adjustments that are necessary to the system computations for dynamics of receiver movements (i.e. filtering). The following commands provide the means to adjust parameters required for these compensations.

v

Syntax : v[F],[FLOW],[FHIGH],[FACTOR]<>

Example : "v0,1,0,0"<cr> // Default – disable attitude filter.
"v0.2,0.2,0.8,0.8"<cr> // Recommended first attempt.

Purpose : This command establishes the sensitivity, boundary, and transition control parameters for the adaptive filter that operates on the attitude outputs of the tracking system. By means of this command, the user can adjust these parameters to fine-tune the overall dynamic response of any system that includes the tracker as a serial component.

The subject filter is a single-pole low-pass type with an adaptive pole location (i.e. a floating filter "constant"). The pole location is constrained within the boundary values FLOW and FHIGH, but is continuously self-adapting between these limits as a function of the sensitivity parameter F and the sensed (ambient noise plus rotational rate) input conditions. For input "rate" conditions that fall within the adaptive range, the adaptive feature varies the pole location between the FLOW and FHIGH limits so as to maximise the output resolution for static inputs while minimising the output lag for dynamic inputs.

Whenever the input conditions cause the filter to make a transition to a narrower bandwidth (i.e. increase filtering), the transition rate of the pole location is constrained to a maximum allowable rate by the parameter FACTOR.

Parameters : F : A scalar value that establishes the sensitivity of the filter to dynamic input conditions by specifying the proportion of new input data to recent average data that is to be used in updating the floating filter "constant". It has a default value is 0.0 and the value can range from 0 to 1.

FLOW : A scalar value that specifies the maximum allowable filtering to be aplied to the outputs during periods of relatively static input conditions. Default value is 1.0 and an allowable range exist of between 0 and FHIGH.

FHIGH : A scalar value that specifies the minimum allowable filtering to be aplied to the outputs during periods of highly dynamic input conditions. Default value is 0.0 and an allowable range exits of between FLOW and 1.

FACTOR : A scalar value that specifies the maximum allowable transition rate from minimum filtering (for highly dynamic input conditions) to maximum filtering (for relatively static input conditions) by proportionately limiting the incremental difference between successive filter "constant" updates whenever the input conditions effect a transition to a narrower bandwidth. The default value is 0.0 with an allowable range of between 0 and 1.

If all of the optional parameters are omitted, then the current value of each parameter is returned to the user as an output record string with the structure shown below :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Blank = <space>	1 ASCII character
3	Sub-record type = "v"	1 ASCII character
4-10	Filter sensitivity	1 spaced signed numeric strings
11-17	Floating filter low value	1 spaced signed numeric strings
18-24	Floating filter high value	1 spaced signed numeric strings
25-31	Transition rate monitor	1 spaced signed numeric strings
32-33	Carriage return, line feed	2 ASCII characters
34	NULL value	1 ASCII character

Where a spaced signed numeric string is now defined as follows :

Byte(s)	Identification	Format
1	Blank = <space>	1 ASCII character
2	Sign : either "+" or "-"	1 ASCII character
3	Number "0" to "9"	1 ASCII character
4	Decimal point = "."	1 ASCII character
5	Number "0" to "9"	1 ASCII character
6	Number "0" to "9"	1 ASCII character
7	Number "0" to "9"	1 ASCII character

Example Output : "2 v +0.200 +0.200 +0.800 +0.800"<cr><lf><NULL>

x

Syntax : x[F],[FLOW],[FHIGH],[FACTOR]<>

Example : "x0,1,0,0"<cr> // Default – disable position filter.
"x0.2,0.2,0.8,0.8"<cr> // Recommended first attempt.

Purpose : This command establishes the sensitivity, boundary, and transition control parameters for the adaptive filter that operates on the position outputs of the tracking system. By means of this command, the user can adjust these parameters to fine-tune the overall dynamic response of any system that includes the tracker as a serial component.

The subject filter is a single-pole low-pass type with an adaptive pole location (i.e. a floating filter "constant"). The pole location is constrained within the boundary values FLOW and FHIGH, but is continuously self-adapting between these limits as a function of the sensitivity parameter F and the sensed (ambient noise plus rotational rate) input conditions. For input "rate" conditions that fall within the adaptive range, the adaptive feature varies the pole location between the FLOW and FHIGH limits so as to maximise the output resolution for static inputs while minimising the output lag for dynamic inputs.

Whenever the input conditions cause the filter to make a transition to a narrower bandwidth (i.e. increase filtering), the transition rate of the pole location is constrained to a maximum allowable rate by the parameter FACTOR.

Parameters : F : A scalar value that establishes the sensitivity of the filter to dynamic input conditions by specifying the proportion of new input data to recent average data that is to be used in updating the floating filter "constant". It has a default value is 0.0 and the value can range from 0 to 1.

FLOW : A scalar value that specifies the maximum allowable filtering to be aplied to the outputs during periods of relatively static input conditions. Default value is 1.0 and an allowable range exist of between 0 and FHIGH.

FHIGH : A scalar value that specifies the minimum allowable filtering to be aplied to the outputs during periods of highly dynamic input conditions. Default value is 0.0 and an allowable range exits of between FLOW and 1.

FACTOR : A scalar value that specifies the maximum allowable transition rate from minimum filtering (for highly dynamic input conditions) to maximum filtering (for relatively static input conditions) by proportionately limiting the incremental difference between successive filter "constant" updates whenever the input conditions effect a transition to a narrower bandwidth. The default value is 0.0 with an allowable range of between 0 and 1.

If all of the optional parameters are omitted, then the current value of each parameter is returned to the user as an output record string with the structure shown below :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Blank = <space>	1 ASCII character
3	Sub-record type = "x"	1 ASCII character
4-10	Filter sensitivity	1 spaced signed numeric strings
11-17	Floating filter low value	1 spaced signed numeric strings
18-24	Floating filter high value	1 spaced signed numeric strings
25-31	Transition rate monitor	1 spaced signed numeric strings
32-33	Carriage return, line feed	2 ASCII characters
34	NULL value	1 ASCII character

Where a spaced signed numeric string is now defined as follows :

Byte(s)	Identification	Format
1	Blank = <space>	1 ASCII character
2	Sign : either "+" or "-"	1 ASCII character
3	Number "0" to "9"	1 ASCII character
4	Decimal point = "."	1 ASCII character
5	Number "0" to "9"	1 ASCII character
6	Number "0" to "9"	1 ASCII character
7	Number "0" to "9"	1 ASCII character

Example Output : "2 x +0.200 +0.200 +0.800 +0.800"<cr><lf><NULL>

Synchronisation Commands

Synchronisation refers to the command involved in controlling the system synchronisation mode. The "y" command provides a means to change system synchronisation.

y

Syntax : y[smode]<cr>

Example : "y1"<cr>

Purpose : This command allows the host to set the system synchronisation mode.

Parameter : smode 0 – Signifies that the system is internally synchronised. The next cycle starts after the previous cycle time has expired.
1 – Signifies that the system is externally synchronised to another system.
2 – Signifies that the system is synchronised via software

If the optional parameter is omitted the system returns the current value of the synchronisation mode as an output record string that is defined as follows :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Blank = <space>	1 ASCII character
3	Sub-record type = "y"	1 ASCII character
4	Sync. mode (0 = Internal, 1 = External, 2 = Software)	1 byte
5-6	Carriage return, line feed	2 ASCII characters

Example Output : "2 y1"<cr><lf>

Reset

Reset refers to the capability of changing system control values of the system to an initial start-up state.

W

Syntax : W

Example : "W"

Purpose : This command resets many system variables to their factory default values without going through the power-ON sequence. However, this is not a complete system refresh like the "^y" command. Only variables of commands that specify a particular system default are reset.

^Y

Syntax : ^Y // The ^ is the <cntrl> button.

Example : <cntrl>"Y"

Purpose : Reinitialises the entire system to the power up state. The system will run through its initial self test and power-ON sequence.

Envelope Commands

Envelope refers to the angular and positional limits in which the receiver is allowed to operate. Movement of the receiver outside these limits results in a software bit error. All co-ordinates are given in the transmitter reference frame. The units of the position co-ordinates are interpreted according to the units as set by the "U" or "u" command. The defaults are the maximum envelope allowed.

Q

Syntax : Qs,[amax],[emax],[rmax],[amin],[emin],[rmin]<cr>

Example : "Q1,70.0, 70.0, 50.0,-70.0,-50.0,10.0"<cr>

Purpose : The angular operational envelope is established with this command. This command may be used to impose software angular limits on the system outputs. If the computed system outputs for a given receiver outside of these limits, the output is flagged with a BIT error code "y". The system defaults are 180, 90, 180, -180, -90, -180.

Paramters : s : The number of the station whose angular limits are to be established or returned.
amax : The maximum azimuth value for the angular operational envelope.
emax : The maximum elevation value for the angular operational envelope.
rmax : The maximum roll value for the angular operational envelope.
amin : The minimum azimuth value for the angular operational envelope.
emin : The minimum elevation value for the angular operational envelope.
rmin : The minimum roll value for the angular operational envelope.

If all the optional parameters are omitted the system returns the current values of the parameters in a string whose structure is described below :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "Q"	1 ASCII character
4-12	Maximum Azimuth	1 spaced signed numeric strings
13-21	Maximum Elevation	1 spaced signed numeric strings
22-30	Maximum Roll	1 spaced signed numeric strings
31-39	Minimum Azimuth	1 spaced signed numeric strings
40-48	Minimum Elevation	1 spaced signed numeric strings
49-57	Minimum Roll	1 spaced signed numeric strings
58-59	Carriage return, line feed	2 ASCII characters
60	NULL value	1 ASCII character

Where a spaced signed numeric string is here defined as follows :

Byte(s)	Identification	Format
1	Sign : either "+" or "-"	1 ASCII character
2	Number "0" to "9"	1 ASCII character
3	Number "0" to "9"	1 ASCII character
4	Number "0" to "9"	1 ASCII character
5	Decimal point = "."	1 ASCII character
6	Number "0" to "9"	1 ASCII character
7	Number "0" to "9"	1 ASCII character
8	Number "0" to "9"	1 ASCII character
9	Blank = <space>	1 ASCII character

Example Output : "21Q+070.000 +070.000 +050.000 –070.000 –050.000 +010.000 "<cr><lf><NULL>

V

Syntax : Vs,[xmax],[ymax],[zmax],[xmin],[ymin],[zmin]<cr>

Example : "V1,70.0, 70.0, 50.0,-70.0,-50.0,10.0"<cr>

Purpose : The position operational envelope is established with this command. This command may be used to impose software positional limits on the system outputs. If the computed system outputs for a given receiver outside of these limits, the output is flagged with a BIT error code "s". The system defaults are 200.0, 200.0, 200.0, -200.0, -200.0, -200.0 cm.

Paramters : s : The number of the station whose angular limits are to be established or returned.
xmax : The maximum X co-ordinate for the position operational envelope.
ymax : The maximum Y co-ordinate for the position operational envelope.
zmax : The maximum Z co-ordinate for the position operational envelope.
xmin : The minimum X co-ordinate for the position operational envelope.
ymin : The minimum Y co-ordinate for the position operational envelope.
zmin : The minimum Z co-ordinate for the position operational envelope.

If all the optional parameters are omitted the system returns the current values of the parameters in a string whose structure is described below :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "V"	1 ASCII character
4-11	Maximum Azimuth	1 signed numeric strings
12-19	Maximum Elevation	1 signed numeric strings
20-27	Maximum Roll	1 signed numeric strings
28-35	Minimum Azimuth	1 signed numeric strings
36-43	Minimum Elevation	1 signed numeric strings
44-51	Minimum Roll	1 signed numeric strings
51-53	Carriage return, line feed	2 ASCII characters
54	NULL value	1 ASCII character

Where a spaced signed numeric string is here defined as follows :

Byte(s)	Identification	Format
1	Sign : either "+" or "-"	1 ASCII character
2	Number "0" to "9"	1 ASCII character
3	Number "0" to "9"	1 ASCII character
4	Number "0" to "9"	1 ASCII character
5	Decimal point = "."	1 ASCII character
6	Number "0" to "9"	1 ASCII character
7	Number "0" to "9"	1 ASCII character
8	Number "0" to "9"	1 ASCII character

Example Output : "21V+070.000+070.000+050.000–070.000–050.000+010.000"<cr><lf><NULL>

Hemisphere Command

Because of the symmetry of the magnetic fields generated by the transmitter, there are two mathematical solutions to each set of receiver data processed. Therefore only half of the total spatial sphere surrounding the transmitter is practically used at any one time without incurring an ambiguity in X, Y, Z measurements. This half sphere is referred to as the current hemisphere. The chosen hemisphere is defined by a line-of-sight vector from the transmitter through a point at the zenith of the hemisphere, and is specified by the line-of-sight direction cosines.

H

Syntax : Hstation,[p1],[p2],[p3]<cr>

Example : "H1,1.0, 0.90, -0.50"<cr>

Purpose : The operational hemisphere for a particular station can be established with this command.

Parameters : station : the number of the station whose operational hemisphere is to be adjusted.
p1 : the x-component of a vector pointing in the direction of the operational hemisphere.
p2 : the y-component of a vector pointing in the direction of the operational hemisphere.
p3 : the z-component of a vector pointing in the direction of the operational hemisphere.

Range : -1 <= p <= 1

Default : The transmitter reference frame X-axis defines the default hemisphere.

If all the optional parameters are omitted then the system will return the vector components for the hemisphere of operation for the specified station as an output record string defined as follows :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "H"	1 ASCII character
4-10	Vector x-component	1 signed numeric strings
11-17	Vector y-component	1 signed numeric strings
18-24	Vector z-component	1 signed numeric strings
25-26	Carriage return, line feed	2 ASCII characters

Where a spaced signed numeric string is here defined as follows :

Byte(s)	Identification	Format
1	Sign : either "+" or "-"	1 ASCII character
2	Number "0" to "9"	1 ASCII character
3	Number "0" to "9"	1 ASCII character
4	Decimal point = "."	1 ASCII character
5	Number "0" to "9"	1 ASCII character
6	Number "0" to "9"	1 ASCII character
7	Number "0" to "9"	1 ASCII character

Example Output : "21H+01.000+00.900-00.500"<cr><lf>

System Output Definition Commands

System output definition is accomplished by several commands. With the various commands and their multiple parameters, outputs may be tailored to each user's requirements. The following command sets affect the various output possibilities.

Output List Commands

The output list refers to the subset of data items to be included in a data record. Any combination of up to 32 data items that total less than or equal 1024 16-bit words is permissible.

O

Syntax : Ostation,[p1],[p2],...,[pn]<cr>

Example : "O1,2,1,0,4,1,0"<cr>

Purpose : This command allows the user to define the list of variables to be output to the host computer for the specified station. Any combination of up to 32 data items that total less than or equal 1024 16-bit words is permissible; however, an increased output size will have an impact on system cycle time and system latency.

Parameters : station : 1 or 2

output is 16-bit integer for the following.

0	: <NULL> 16 bit word
1	: ASCII carriage return and line feed pair.
2	: x, y, z cartesian co-ordinates of position
3	: not used (reserved for future use)
4	: az, el, roll Euler orientation angles
5	: x-axis direction cosines (line of sight)
6	: y-axis direction cosines (line of hear)
7	: z-axis direction cosines (line of plumb)
8	: x-receiver data (factory use only)
9	: y-receiver data (factory use only)
10	: z-receiver data (factory use only)
11	: orientation quaternion
12	: self calibration data (factory use only)
13	: adjusted x-receiver data (factory use only)
14	: adjusted y-receiver data (factory use only)
15	: adjusted z-receiver data (factory use only)
16 – 49	: not used (reserved for future use)

Output is 32 bit IEEE float output in accordance with the format specified by ANSI / IEEE Std 754-1985

50	: <NULL> 16 bit word (same as 0)
51	: ASCII carriage return, line feed pair (same as 1)
52	: x, y, z cartesian co-ordinates of position
53	: not used (reserved for future use)
54	: az, el, roll Euler orientation angles
55	: x-axis direction cosines (line of sight)
56	: y-axis direction cosines (line of hear)
57	: z-axis direction cosines (line of plumb)
58	: x-receiver data (factory use only)
59	: y-receiver data (factory use only)
60	: z-receiver data (factory use only)
61	: orientation quaternion
62	: self calibration data (factory use only)
63	: adjusted x-receiver data (factory use only)
64	: adjusted y-receiver data (factory use only)
65	: adjusted z-receiver data (factory use only)
66 – 69	: not used (reserved for future use)

Default : "O1,2,4,1"<cr> and "O2,2,4,1"<cr>
I.e. the three Cartesian co-ordinates, the three orientation angles, carriage return, and line feed for both stations.

The IEEE 32Bit float is defined as follows:

Bit	Byte	Format
31	Bit 7 in byte 3	Sign
30 – 23	Bit 6 in byte 3 to Bit 7 in byte 2	Exponent (e – 127, exponent ranges from –127 to 128
22 – 0	Bit 6 in byte 2 to Bit 0 in byte 0	Fraction (most significant 1 implied)

The IEEE floating-point format uses sign magnitude notation for the mantissa, and an exponent offset by 127. In a 32-bit words representing a floating-point number, the first bit is the sign bit. The next 8 bits are the exponent, offset by 127 (i.e. the actual exponent is e - 127). The last 23 bits are the absolute value of the mantissa with the most significant 1 implied. The decimal point is after the implied 1, or in other words, the mantissa is actually expressed in 24 bits. In the normal case an IEEE value is expressed as :

$(-1)^s * (2^{(e-127)}) * (01.f)$

If all of the optional parameters are omitted, the system returns the current list of selected data items as an output record string defined as follows :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "H"	1 ASCII character
4 – 5	Data item n identification	Integer
6 – 7	Data item n identification	Integer
7 – 8	Data item n identification	Integer
...	...	...
2*n+2 – 2*n+3	Data item n identification	Integer
2*n+4 – 2*n+5	Carriage return, line feed	2 ASCII characters
2*n+6	NULL value (If required)	1 ASCII character

Output Transmit Mode Commands

Transmit mode refers to whether the system automatically transmits a data record when it is ready (Continuous), or the host must request each data record by sending a "P" command or a transmit interrupt to the system (Non-continuous). The current transmit mode may be retrieved from the status record.

Default is Non-continuous.

C

Syntax : C

Example : "C"

Purpose : This command enables the continuous output mode of the system. This mode excludes "P" commands and immediate requests via interrupt. Default is non-continuous where data may be requested via "P" commands or immediate requests via interrupt.

P

Syntax : P

Example : "P"

Purpose : This command requests that a single data record be transmitted to the host computer. Continuous print mode must be disabled (c). If more than one station is active, one record for each of the active stations will be transmitted in station number order.

c

Synatx : c

Example : "c"

Purpose : Disable continuous output to the host computer. This is the default case.

Command Input / Output Units Commands

Command Input / Output Units is a reference to the distance unit assumed by the system when interpreting input data or when formatting a response to an input command request for data. The current distance unit may be retrieved from the status record. Note that these commands have no effect on the output scaling of the system data record generated to report position, orientation, etc. of any receiver. The default is inches.

U

Syntax : U

Example : "U"

Purpose : Sets the distance unit to inches. Subsequent input lengths are interpreted as inches. Subsequent output command responses are scaled to inches. The system default is inches.

u

Syntax : u

Example : "u"

Purpose : Sets the distance unit to centimeters. Subsequent input lengths are interpreted as centimeters. Subsequent output command responses are scaled to centimeters. The system default is inches.

Station Command

A station is a transmitter-receiver pair. In an InsideTRAK, the two receivers of a board, paired with the one available transmitter, are assigned station numbers one or two. Stations may be activated or deactivated. If a station is active, data records are transmitted for that station; otherwise, no data records for that station are transmitted. At least one station in a master board must always be active; any attempt to deactivate a lone station is ignored. By default both are assumed active.

l

Syntax : lstation,[state]<>

Example : "l1,0"<cr>

Purpose : Set the on / off station state.

Parameters : Station : 1 or 2
State : 0 = off
1 = on

If the optional parameter is missing, the system returns the current state for the specified station as an output record of type "l" as defined below :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "l"	1 ASCII character
4	Station 1 = 1 if active	1 ASCII character
5	Station 2 = 1 if active	1 ASCII character
6 – 7	Carriage return, line feed	2 ASCII characters
8	NULL value (If required)	1 ASCII character

Status Commands

Status refers to the capability to determine information about the system that is not available from other commands.

S

Syntax : S

Example : "S"

Purpose : Request that a system status record be transmitted to the host computer.

The returned data string is defined as follows :

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "S"	1 ASCII character
4 – 6	System Flags	3 Hexadecimal bytes
	Bit 0 : Output format (0 = ASCII, 1 = Binary)	
	Bit 1 : Units (0 = Inches, 1 = Centimeters)	
	Bit 2 : Compensation (0 = Off, 1 = On)	
	Bit 3 : Transmit Mode (0 = Non-Continuous, 1 = Continuous)	
	Bit 4 : Configuration (Always 1)	
	Bit 5 : 1 = Fastrak Family	
	Bits 6 – 7 : Always 11	
	Bits 8 – 9 : Always 11	
	Bits 10 – 11 : Product (00 = Fastrak, 01 = InsideTRAK)	
	Bits 12 – 23 : Reserved for future use	
7 – 9	Bit Error Number	3 Integers
10 – 15	Blank (Reserved for future use)	6 ASCII characters
16 – 21	Software Version ID, eg "150.00"	6 ASCII characters
22 – 53	System Identification = " InsideTRAK CPG2047-001-01 "	32 ASCII characters
54 – 55	Carriage return, line feed	2 ASCII characters
56	NULL value (If required)	1 ASCII character

T

Syntax : Tbitnbr[,0]<cr>

Example : "T72,0"<cr>

Purpose : This command allows the user to obtain additional information about a particular BIT or clear a particular bit error.

Parameters : bitnbr : The BIT number for which added information is requested
: 0 : This parameter, if used, is specifies as a 0. If present, then the bitnbr specified in the command is reset / cleared.

The system returns the current bit information for the specified bit as an output record string defined as follows:

Byte(s)	Identification	Format
1	Record Type = "2"	1 ASCII character
2	Station Number	1 ASCII character
3	Sub-record type = "T"	1 ASCII character
4 – 6	BIT Number	3 Integers
7 - ??	BIT Information (Factory Use)	?? ASCII characters
?? - ??	Carriage return, line feed	2 ASCII characters
??	NULL value (If required)	1 ASCII character

Appendix B
WinNT Device Driver

I will now provide a reasonably brief description on how this device driver works. The first point of entry is the DriverEntry function. This is the first function that gets executed by the operating system when the device driver is started up.

```
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
```

It accepts two input variables, both of which are created by the operating system. The first variable is a pointer to the Driver Object which will be filled with Device Driver specific information by the device driver. This information will then be used by the operating system in future communications with the Device Driver. The second is a path to the registry where the data for this device driver is stored. Each device driver has a space inside the Windows NT registry that it can store information and variables that remain consistent between reboots of the operating system.

The first step is to declare some variables that will be used inside this function.

```
{
    ULONG PortBase;                 // Port location, in NT's address form.
    ULONG PortCount;                // Count of contiguous I/O ports
    PHYSICAL_ADDRESS PortAddress;

    PLOCAL_DEVICE_INFO pLocalInfo;  // Device extension:
                                    // local information for each device.
    NTSTATUS Status;
    PDEVICE_OBJECT DeviceObject;

    CM_RESOURCE_LIST ResourceList;  // Resource usage list to report to sys.
    BOOLEAN ResourceConflict;       // This is set true if our I/O ports
                                    // conflict with another driver
```

The whole of the next section is there to set up the query and return buffers that will be sent to the registry. The registry will search for the two variables, namely "IoPortAddress" and "IoPortCount", and fill the variables "PortBase" and "PortCount". If the registry query fails, then it uses the default values of "BASE_PORT" and "NUMBER_PORTS" which in my case were defined to be :

```
#define BASE_PORT         0x300
#define NUMBER_PORTS      30
```

```
    {
        static  WCHAR                   SubKeyString[] = L"\\Parameters";
        UNICODE_STRING                  paramPath;
        RTL_QUERY_REGISTRY_TABLE        paramTable[3];
        ULONG                           DefaultBase = BASE_PORT;
        ULONG                           DefaultCount = NUMBER_PORTS;


        paramPath.MaximumLength = RegistryPath->Length +
sizeof(SubKeyString);
        paramPath.Buffer = ExAllocatePool(PagedPool,
paramPath.MaximumLength);

        if (paramPath.Buffer != NULL)
        {
            RtlMoveMemory( paramPath.Buffer,
              RegistryPath->Buffer, RegistryPath->Length);

            RtlMoveMemory(
                &paramPath.Buffer[RegistryPath->Length / 2], SubKeyString,
                sizeof(SubKeyString));

            paramPath.Length = paramPath.MaximumLength - 2;

            RtlZeroMemory(&paramTable[0], sizeof(paramTable));

            paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
            paramTable[0].Name = L"IoPortAddress";
            paramTable[0].EntryContext = &PortBase;
            paramTable[0].DefaultType = REG_DWORD;
            paramTable[0].DefaultData = &DefaultBase;
            paramTable[0].DefaultLength = sizeof(ULONG);

            paramTable[1].Flags = RTL_QUERY_REGISTRY_DIRECT;
            paramTable[1].Name = L"IoPortCount";
            paramTable[1].EntryContext = &PortCount;
            paramTable[1].DefaultType = REG_DWORD;
            paramTable[1].DefaultData = &DefaultCount;
            paramTable[1].DefaultLength = sizeof(ULONG);

            if (!NT_SUCCESS(RtlQueryRegistryValues(
                RTL_REGISTRY_ABSOLUTE | RTL_REGISTRY_OPTIONAL,
                paramPath.Buffer, &paramTable[0], NULL, NULL)))
            {
                PortBase = DefaultBase;
                PortCount = DefaultCount;
            }
            ExFreePool(paramPath.Buffer);
        }
    }
```

What happens now is that Windows NT requires the device driver to report any hardware resources that it intends using. If these resources are available to be used then the operating system assigns then to this device driver. This means that no other device is then able to use the resources chosen by this driver. In our case, we only need to request a segment of the port addresses starting with the PortBase value, up to PortBase + PortCount. If the requested resources are unavailable for this device driver then the DriverEntry routine aborts and returns the corresponding error code.

```
PortAddress.LowPart  = PortBase;
PortAddress.HighPart = 0;

RtlZeroMemory((PVOID)&ResourceList, sizeof(ResourceList));

ResourceList.Count = 1;
ResourceList.List[0].InterfaceType = Isa;
// ResourceList.List[0].Busnumber = 0;              Already 0
ResourceList.List[0].PartialResourceList.Count = 1;
ResourceList.List[0].PartialResourceList.PartialDescriptors[0].Type =
                                                  CmResourceTypePort;
ResourceList.List[0].PartialResourceList.PartialDescriptors[0].
    ShareDisposition = CmResourceShareDriverExclusive;
ResourceList.List[0].PartialResourceList.PartialDescriptors[0].
    Flags = CM_RESOURCE_PORT_IO;
ResourceList.List[0].PartialResourceList.PartialDescriptors[0].
    u.Port.Start = PortAddress;
ResourceList.List[0].PartialResourceList.PartialDescriptors[0].
    u.Port.Length = PortCount;

// Report our resource usage and detect conflicts
Status = IoReportResourceUsage(
               NULL,
               DriverObject,
               &ResourceList,
               sizeof(ResourceList),
               NULL,
               NULL,
               0,
               FALSE,
               &ResourceConflict);

if (ResourceConflict)
    Status = STATUS_DEVICE_CONFIGURATION_ERROR;

if (!NT_SUCCESS(Status))
{
    KdPrint( ("Resource reporting problem %8X", Status) );
    return Status;
}
```

The next step is to fill part of the DriverObject with the names of the functions to be called when certain operations on the device driver need to be performed. Because there is only one predefined name in the Device Driver, (i.e. DriverEntry) this is the only function that the operating system can find. It is thus the responsibility of the DriverEntry routine to setup a routing table of function names corresponding to calls that the operating system will be making to the device driver.

```
DriverObject->MajorFunction[IRP_MJ_CREATE]           = GpdDispatch;
DriverObject->MajorFunction[IRP_MJ_CLOSE]            = GpdDispatch;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]   = GpdDispatch;
DriverObject->DriverUnload                           = GpdUnload;
```

The remaining task is to create an instance of the device, I.e. a DeviceObject. As each device driver can control multiple devices of the same type (e.g. One CDROM device driver can access two CDROM drives), we need an individual entity that describes the device. This is the DeviceObject. GenPort calls a helper function GpdCreateDevice (discussed as the next function) to do the work of creating a DeviceObject.

```
// Create our device.
    Status = GpdCreateDevice(
                    GPD_DEVICE_NAME,
                    GPD_TYPE,
                    DriverObject,
                    &DeviceObject
                    );
```

If the device creates successfully then we can translate the base port address into one that can be understood within the Windows NT kernel. Next we fill up the DeviceExtension of the DeviceObject with information we will need to know later on. The DeviceExtension will remain coherent and will be accessible by most of the functions within the device driver that will be called by the operating system. The DeviceExtension is a place to store variables that the device driver might later need.

The DeviceExtention has a device driver defined structure and has been defined as follows by GenPort

```
typedef struct _LOCAL_DEVICE_INFO {
    ULONG               DeviceType;     // Our private Device Type
    BOOLEAN             PortWasMapped;  // If TRUE,we have to unmap on unload
    PVOID               PortBase;       // base port address
    ULONG               PortCount;      // Count of I/O addresses used.
    ULONG               PortMemoryType; // HalTranslateBusAddress MemoryType
    PDEVICE_OBJECT      DeviceObject;   // The Gpd device object.
} LOCAL_DEVICE_INFO, *PLOCAL_DEVICE_INFO;
```

```
    if ( NT_SUCCESS(Status) )
    {
        PHYSICAL_ADDRESS MappedAddress;
        ULONG MemType;

        // Convert the IO port address into a form NT likes.
        MemType = 1;                        // located in IO space
        HalTranslateBusAddress( Isa,
                                0,
                                PortAddress,
                                &MemType,
                                &MappedAddress );

        pLocalInfo = (PLOCAL_DEVICE_INFO)DeviceObject->DeviceExtension;

        if (MemType == 0)
        {
            pLocalInfo->PortWasMapped = TRUE;
            pLocalInfo->PortBase = MmMapIoSpace(MappedAddress,
               PortCount, FALSE);
        }
        else
        {
            pLocalInfo->PortWasMapped = FALSE;
            pLocalInfo->PortBase = (PVOID)MappedAddress.LowPart;
        }

        pLocalInfo->DeviceObject    = DeviceObject;
        pLocalInfo->DeviceType      = GPD_TYPE;
        pLocalInfo->PortCount       = PortCount;
        pLocalInfo->PortMemoryType  = MemType;
    }
```

If the DeviceObject failed to create then we need to return the resources that we requested from the operating system back, so that they can be used by someone else, as we failed to start up. This enables the continuous recycling of resources amongst operating system components.

```
else
    {
        RtlZeroMemory((PVOID)&ResourceList, sizeof(ResourceList));

        Status = IoReportResourceUsage(
                        NULL,
                        DriverObject,
                        &ResourceList,
                        sizeof(ResourceList),
                        NULL,
                        NULL,
                        0,
                        FALSE,
                        &ResourceConflict);
    }
    return Status;
}
```

```
NTSTATUS
GpdCreateDevice(
    IN   PWSTR              PrototypeName,
    IN   DEVICE_TYPE        DeviceType,
    IN   PDRIVER_OBJECT     DriverObject,
    OUT  PDEVICE_OBJECT     *ppDevObj
    )
```

Having already used the GpdCreateDevice inside the DriverEntry routine, now would be a good place to explain how it works. As a start, it takes three input variables and one output variable. The PrototypeName and DeviceType are filled by the DriverEntry routine with the two constants GPD_DEVICE_NAME, and GPD_TYPE which are defined at compile time to be :

```
#define GPD_DEVICE_NAME L"\\Device\\Gpd0"
#define GPD_TYPE 40000
```

The DriverObject is the object refering to the device driver. The information in the DriverObject and the other variables are used to create a DeviceObject which will store the data particular to that particular device. For example imagine a CDROM driver with two CDROM drives. The CDROM driver will have a single DriverObject and two DeviceObjects. The DriverObject will contain information on the functions accessible in the device driver, the resources requested for use by the device driver, and others. Whereas each of the DeviceObjects will contain information specific to their specific CDROM drive, like the current head position, the data currently being read, or cache of past data read.

The main purpose of this function is to create a logical entity within the kernel that will represent the physical device, and through which applications will communicate to the hardware. Having done so it also needs to create a DOS name (e.g. "com1", "lpt1", "prn") through which applications can find and connect to the device.

```
{
    NTSTATUS Status;                        // Status of utility calls
    UNICODE_STRING NtDeviceName;
    UNICODE_STRING Win32DeviceName;

    RtlInitUnicodeString(&NtDeviceName, PrototypeName);

    Status = IoCreateDevice(                                // Create it.
                    DriverObject,
                    sizeof(LOCAL_DEVICE_INFO),
                    &NtDeviceName,
                    DeviceType,
                    0,
                    FALSE,                      // Not Exclusive
                    ppDevObj
                    );

    if (!NT_SUCCESS(Status))
        return Status;              // Give up if create failed.

    // Clear local device info memory
    RtlZeroMemory((*ppDevObj)->DeviceExtension, sizeof(LOCAL_DEVICE_INFO));

    RtlInitUnicodeString(&Win32DeviceName, DOS_DEVICE_NAME);
```

```
    Status = IoCreateSymbolicLink( &Win32DeviceName, &NtDeviceName );

    if (!NT_SUCCESS(Status))     // If we we couldn't create the link then
    {                            //  abort installation.
        IoDeleteDevice(*ppDevObj);
    }

    return Status;
}
```

```
NTSTATUS
GpdDispatch(
    IN    PDEVICE_OBJECT pDO,
    IN    PIRP pIrp
    )
```

The next function under discussion is the GpdDispatch function. As you should remember from the DriverEntry routine, GenPort sets this as the function that will be called if a IRP_MJ_CREATE, IRP_MJ_CLOSE or a IRP_MJ_DEVICE_CONTROL operation is requested. Thus in this function we need to handle these three requests. You will also see how the IRP_MJ_DEVICE_CONTROL operation is further sub-divided into multiple minor codes that represent the transfer operations that we will be using.

```
{
    PLOCAL_DEVICE_INFO pLDI;
    PIO_STACK_LOCATION pIrpStack;
    NTSTATUS Status;

    //  Initialize the irp info field.
    //      This is used to return the number of bytes transfered.

    pIrp->IoStatus.Information = 0;
    pLDI = (PLOCAL_DEVICE_INFO)pDO->DeviceExtension; // Get local info struct

    pIrpStack = IoGetCurrentIrpStackLocation(pIrp);

    //  Set default return status
    Status = STATUS_NOT_IMPLEMENTED;
```

The first thing that this function does is initialise the pointers to the DeviceExtention (device specific information), to the IrpStack (Operation specific information) and set the returned data length to 0.

Now we have the switch statement that sorts out the different operations that could have caused this function to be executed.

```
    switch (pIrpStack->MajorFunction)
        {
            case IRP_MJ_CREATE:
            case IRP_MJ_CLOSE:
                // We don't need any special processing on open/close so we'll
                // just return success.
                Status = STATUS_SUCCESS;
                break;

            case IRP_MJ_DEVICE_CONTROL:             //  Dispatch on IOCTL
                switch (pIrpStack->Parameters.DeviceIoControl.IoControlCode)
                {
                case IOCTL_GPD_READ_PORT_UCHAR:
                case IOCTL_GPD_READ_PORT_USHORT:
                case IOCTL_GPD_READ_PORT_ULONG:
                    Status = GpdIoctlReadPort(
                                pLDI,
                                pIrp,
                                pIrpStack,
                                pIrpStack->Parameters.
                                  DeviceIoControl.IoControlCode
                                );
                    break;

                case IOCTL_GPD_WRITE_PORT_UCHAR:
                case IOCTL_GPD_WRITE_PORT_USHORT:
                case IOCTL_GPD_WRITE_PORT_ULONG:
                    Status = GpdIoctlWritePort(
                                pLDI,
                                pIrp,
                                pIrpStack,
                                pIrpStack->Parameters.
                                  DeviceIoControl.IoControlCode
                                );
                    break;
                }
                break;
        }
```

Firstly if it is a Create or Close major operation then we do not need to do anything because our device does not need any specific setup or shutdown procedures to be performed. Secondly, if the major operation is a Control operation, then depending on whether the minor operation is a Read or a Write, it gets forwarded to the functions GpdIoctlReadPort and GpdIoctlWritePort respectively. Finally, we report either STATUS_SUCCESS, or the status returned by the two previously mentioned functions.

```
        // We're done with I/O request.  Record the status of the I/O action.
        pIrp->IoStatus.Status = Status;

        // Don't boost priority when returning since this took little time.
        IoCompleteRequest(pIrp, IO_NO_INCREMENT );

        return Status;
    }
```

```
NTSTATUS
GpdIoctlReadPort(
    IN PLOCAL_DEVICE_INFO pLDI,
    IN PIRP pIrp,
    IN PIO_STACK_LOCATION IrpStack,
    IN ULONG IoctlCode  )
```

This next function does all (8, 16 and 32 bit) the read operations that the user could request. After creating some temporary variables and assigning the sizes of the input and output data buffers to two of them, this function checks to see if the input buffer is large enough to hold the port number, and that the output buffer is large enough to hold the data requested (8, 16 and 32 bit).

```
{
    PULONG pIOBuffer;           // Pointer to transfer buffer
                                //      (treated as an array of longs).
    ULONG InBufferSize;         // Amount of data avail. from caller.
    ULONG OutBufferSize;        // Max data that caller can accept.
    ULONG nPort;                // Port number to read
    ULONG DataBufferSize;

    // Size of buffer containing data from application
    InBufferSize  = IrpStack->Parameters.DeviceIoControl.InputBufferLength;

    // Size of buffer for data to be sent to application
    OutBufferSize = IrpStack->Parameters.DeviceIoControl.OutputBufferLength;

    // NT copies inbuf here before entry and copies this to outbuf after
    // return, for METHOD_BUFFERED IOCTL's.
    pIOBuffer     = (PULONG)pIrp->AssociatedIrp.SystemBuffer;

    // Check to ensure input buffer is big enough to hold a port number and
    // the output buffer is at least as big as the port data width.
    //
    switch (IoctlCode)
    {
    default:                    // There isn't really any default but
  /* FALL THRU */           // this will quiet the compiler.
    case IOCTL_GPD_READ_PORT_UCHAR:
        DataBufferSize = sizeof(UCHAR);
        break;
    case IOCTL_GPD_READ_PORT_USHORT:
        DataBufferSize = sizeof(USHORT);
        break;
    case IOCTL_GPD_READ_PORT_ULONG:
        DataBufferSize = sizeof(ULONG);
        break;
    }

    if ( InBufferSize != sizeof(ULONG) || OutBufferSize < DataBufferSize )
    {
        return STATUS_INVALID_PARAMETER;
    }
```

The next step is to remove the port address from the input buffer, afterwhich we have to check to make sure that the port address is not larger than the allowed number of ports. We also have to check to make sure that the port address added to the size of the data read size is also smaller than the port count (i.e. if there are only 8 ports, one cannot read a 32 bit value from the 6th port because the last 16 bits would stretch past the 8 port boundary.) The last check that needs to be made is to make sure that 16 bit reads fall on an even byte boundary, and that 32 bit reads fall on a 4 byte boundary.

```
nPort = *pIOBuffer;            // Get the I/O port number from the buffer.

if (nPort >= pLDI->PortCount ||
    (nPort + DataBufferSize) > pLDI->PortCount ||
    (((ULONG)pLDI->PortBase + nPort) & (DataBufferSize - 1)) != 0)
{
    return STATUS_ACCESS_VIOLATION;    // It was not legal.
}
```

Finally we can read the data. Sometimes the address space of a piece of hardware could be memory mapped, i.e. one just writes to memory and the device itself fetches the data from the specific memory location. If this is the case, then special read instructions need to be used, otherwise this section just reads in the value of the requested size from the IO port.

```
if (pLDI->PortMemoryType == 1)
    {
        // Address is in I/O space
        switch (IoctlCode) {
        case IOCTL_GPD_READ_PORT_UCHAR:
            *(PUCHAR)pIOBuffer = READ_PORT_UCHAR(
                                 (PUCHAR)((ULONG)pLDI->PortBase + nPort) );
            break;
        case IOCTL_GPD_READ_PORT_USHORT:
            *(PUSHORT)pIOBuffer = READ_PORT_USHORT(
                                 (PUSHORT)((ULONG)pLDI->PortBase + nPort) );
            break;
        case IOCTL_GPD_READ_PORT_ULONG:
            *(PULONG)pIOBuffer = READ_PORT_ULONG(
                                 (PULONG)((ULONG)pLDI->PortBase + nPort) );
            break;
        }
    } else {
        // Address is in Memory space
        switch (IoctlCode) {
        case IOCTL_GPD_READ_PORT_UCHAR:
            *(PUCHAR)pIOBuffer = READ_REGISTER_UCHAR(
                                 (PUCHAR)((ULONG)pLDI->PortBase + nPort) );
            break;
        case IOCTL_GPD_READ_PORT_USHORT:
            *(PUSHORT)pIOBuffer = READ_REGISTER_USHORT(
                                 (PUSHORT)((ULONG)pLDI->PortBase + nPort) );
            break;
        case IOCTL_GPD_READ_PORT_ULONG:
            *(PULONG)pIOBuffer = READ_REGISTER_ULONG(
                                 (PULONG)((ULONG)pLDI->PortBase + nPort) );
            break;
        }
    }
```

Finally, we set the returned buffer size to the size of the data request and return the status as success back to the operating system.

```
    pIrp->IoStatus.Information = DataBufferSize;
    return STATUS_SUCCESS;
}
```

```
NTSTATUS
GpdIoctlWritePort(
    IN PLOCAL_DEVICE_INFO pLDI,
    IN PIRP pIrp,
    IN PIO_STACK_LOCATION IrpStack,
    IN ULONG IoctlCode
    )
```

The GpdIoctlWritePort function, pasted below operates in a very similar manner to the GpdIoctlReadPort function, so I will skip through the explanation, leaving only source code comments.

```
{
                                // NOTE:  Use METHOD_BUFFERED ioctls.
    PULONG pIOBuffer;           // Pointer to transfer buffer
                                //     (treated as array of longs).
    ULONG InBufferSize ;        // Amount of data avail. from caller.
    ULONG OutBufferSize ;       // Max data that caller can accept.
    ULONG nPort;                // Port number to read or write.
    ULONG DataBufferSize;

    // Size of buffer containing data from application
    InBufferSize  = IrpStack->Parameters.DeviceIoControl.InputBufferLength;

    // Size of buffer for data to be sent to application
    OutBufferSize = IrpStack->Parameters.DeviceIoControl.OutputBufferLength;

    // NT copies inbuf here before entry and copies it to outbuf after return
    // for METHOD_BUFFERED IOCTL's.
    pIOBuffer     = (PULONG) pIrp->AssociatedIrp.SystemBuffer;

    // We don't return any data on a write port.
    pIrp->IoStatus.Information = 0;

    // Check to ensure input buffer is big enough to hold a port number as
    // well as the data to write.
    //
    // The relative port # is a ULONG, and the data is the type appropriate
    // to the IOCTL.
    //
```

```
  switch (IoctlCode)
  {
  default:                    // There isn't really any default but
/* FALL THRU */               // this will quiet the compiler.
  case IOCTL_GPD_WRITE_PORT_UCHAR:
      DataBufferSize = sizeof(UCHAR);
      break;
  case IOCTL_GPD_WRITE_PORT_USHORT:
      DataBufferSize = sizeof(USHORT);
      break;
  case IOCTL_GPD_WRITE_PORT_ULONG:
      DataBufferSize = sizeof(ULONG);
      break;
  }

  if ( InBufferSize < (sizeof(ULONG) + DataBufferSize) )
  {
      return STATUS_INVALID_PARAMETER;
  }

  nPort = *pIOBuffer++;

  if (nPort >= pLDI->PortCount ||
      (nPort + DataBufferSize) > pLDI->PortCount ||
      (((ULONG)pLDI->PortBase + nPort) & (DataBufferSize - 1)) != 0)
  {
      return STATUS_ACCESS_VIOLATION;   // Illegal port number
  }

  if (pLDI->PortMemoryType == 1)
  {
      // Address is in I/O space

      switch (IoctlCode)
      {
      case IOCTL_GPD_WRITE_PORT_UCHAR:
          WRITE_PORT_UCHAR(
              (PUCHAR)((ULONG)pLDI->PortBase + nPort),
              *(PUCHAR)pIOBuffer );
          break;
      case IOCTL_GPD_WRITE_PORT_USHORT:
          WRITE_PORT_USHORT(
              (PUSHORT)((ULONG)pLDI->PortBase + nPort),
              *(PUSHORT)pIOBuffer );
          break;
      case IOCTL_GPD_WRITE_PORT_ULONG:
          WRITE_PORT_ULONG(
              (PULONG)((ULONG)pLDI->PortBase + nPort),
              *(PULONG)pIOBuffer );
          break;
      }
```

```
    } else {
        // Address is in Memory space

        switch (IoctlCode)
        {
        case IOCTL_GPD_WRITE_PORT_UCHAR:
            WRITE_REGISTER_UCHAR(
                    (PUCHAR)((ULONG)pLDI->PortBase + nPort),
                    *(PUCHAR)pIOBuffer );
            break;
        case IOCTL_GPD_WRITE_PORT_USHORT:
            WRITE_REGISTER_USHORT(
                    (PUSHORT)((ULONG)pLDI->PortBase + nPort),
                    *(PUSHORT)pIOBuffer );
            break;
        case IOCTL_GPD_WRITE_PORT_ULONG:
            WRITE_REGISTER_ULONG(
                    (PULONG)((ULONG)pLDI->PortBase + nPort),
                    *(PULONG)pIOBuffer );
            break;
        }
    }

    return STATUS_SUCCESS;
}
```

```
VOID
GpdUnload(
    PDRIVER_OBJECT DriverObject
    )
```

This is the final function present in the device driver. This function is called when the user wants to unload the device driver from the kernel. To do so the device driver has to release all resources is has taken up and to remove all traces of its presence. This unload function does four main operations.

Firstly if the port addresses had been memory mapped to improve on performance, then we need to unmap these ports and free up the memory used. Secondly we need to return the resources we requested in the DriverEntry routine back to the operating system. This is done by creating a blank resource list and using it to report our resources to the operating system. The operating system will thus know that we no longer need any resources and marks them as available to other entities.

Thirdly we need to remove the DOS name, i.e. the symbolic link to the device driver as this will no longer be a valid link as the driver will soon cease to exist. Finally, the last operation that needs to be performed it to delete the logical device that we created in the CreateDevice function to represent the physical piece of hardware.

```
{
   PLOCAL_DEVICE_INFO pLDI;
   CM_RESOURCE_LIST NullResourceList;
   BOOLEAN ResourceConflict;
   UNICODE_STRING Win32DeviceName;

   // Find our global data
   pLDI = (PLOCAL_DEVICE_INFO)DriverObject->DeviceObject->DeviceExtension;

   // Unmap the ports

   if (pLDI->PortWasMapped)
   {
      MmUnmapIoSpace(pLDI->PortBase, pLDI->PortCount);
   }

   // Report we're not using any hardware.  If we don't do this
   // then we'll conflict with ourselves (!) on the next load

   RtlZeroMemory((PVOID)&NullResourceList, sizeof(NullResourceList));

   IoReportResourceUsage(
         NULL,
         DriverObject,
         &NullResourceList,
         sizeof(ULONG),
         NULL,
         NULL,
         0,
         FALSE,
         &ResourceConflict );

   // Assume all handles are closed down.
   // Delete the things we allocated - devices, symbolic links

   RtlInitUnicodeString(&Win32DeviceName, DOS_DEVICE_NAME);

   IoDeleteSymbolicLink(&Win32DeviceName);

   IoDeleteDevice(pLDI->DeviceObject);
}
```