November 1993 Technical Document PPG 93/10

Parallel implementation of a Virtual Reality System on a Transputer Architecture.

Shaun Bangay

Parallel Processing Group
Department of Computer Science
Rhodes University
P.O. Box 94
Grahamstown
6140 RSA

Telephone: (+27) (461) 22023
Telefax: (+27) (461) 25049

Internet: cspc@alpha.ru.ac.za

Parallel Implementation of a Virtual Reality System on a Transputer Architecture.

Submitted in fulfilment of the requirements for the degree of Master of Science of Rhodes University

by

Shaun Douglas Bangay

July 1993

Abstract

A Virtual Reality is a computer model of an environment, actual or imagined, presented to a user in as realistic a fashion as possible. Stereo goggles may be used to provide the user with a view of the modelled environment from within the environment, while a data-glove is used to interact with the environment. To simulate reality on a computer, the machine has to produce realistic images rapidly. Such a requirement usually necessitates expensive equipment.

This thesis presents an implementation of a virtual reality system on a transputer architecture. The system is general, and is intended to provide support for the development of various virtual environments. The three main components of the system are the output device drivers, the input device drivers, and the virtual world kernel. This last component is responsible for the simulation of the virtual world.

The rendering system is described in detail. Various methods for implementing the components of the graphics pipeline are discussed. These are then generalised to make use of the facilities provided by the transputer processor for parallel processing. A number of different decomposition techniques are implemented and compared. The emphasis in this section is on the speed at which the world can be rendered, and the interaction latency involved. In the best case, where almost linear speedup is obtained, a world containing over 250 polygons is rendered at 32 frames/second. The bandwidth of the transputer links is the major factor limiting speedup.

A description is given of an input device driver which makes use of a powerglove. Techniques for overcoming the limitations of this device, and for interacting with the virtual world, are discussed.

The virtual world kernel is designed to make extensive use of the parallel processing facilities provided by transputers. It is capable of providing support for multiple worlds concurrently, and for multiple users interacting with these worlds. Two applications are described that were successfully implemented using this system.

The design of the system is compared with other recently developed virtual reality systems. Features that are common or advantageous in each of the systems are discussed. The system described in this thesis compares favourably, particularly in its use of parallel processors.

Acknowledgements

I would like to thank my supervisors, Dave Sewry and Peter Clayton, for their advice, and support. Thanks also to the members of the Graphics Group for their interest and useful suggestions.

Many valuable discussions with the other staff and students of the Computer Science Department contributed a great deal toward this work. In particular, thanks to those who spent time browsing through numerous shops in an attempt to obtain equipment for this project.

Finally, thanks to my family and friends who supported me, even when nobody knew what virtual reality was.

Table of Contents

1. Introduction . . . . Page 1
    1.1. Introduction to Virtual Reality . . . . Page 1
    1.2. Project goal . . . . Page 2
    1.3. The Transputer . . . . Page 2
    1.4. Virtual Reality Applications . . . . Page 3
        1.4.1. Hardware used for Virtual Reality . . . . Page 3
            1.4.1.1. Polhemus tracker . . . . Page 3
            1.4.1.2. VPL . . . . Page 3
            1.4.1.3. The Powerglove . . . . Page 4
        1.4.2. General Virtual Reality systems . . . . Page 5
            1.4.2.1. The Walkthrough Project . . . . Page 5
            1.4.2.2. Reality Built For Two . . . . Page 6
            1.4.2.3. Integrated Simulation for Autonomous Underwater Vehicles . . . . Page 6
            1.4.2.4. Medical Imaging . . . . Page 7
            1.4.2.5. Visualization . . . . Page 7
        1.4.3. Transputer based Virtual Reality systems . . . . Page 8
            1.4.3.1. The INMOS multi-player flight simulator . . . . Page 8
            1.4.3.2. ProVision . . . . Page 8
    1.5. Design Criteria . . . . Page 9
        1.5.1. The Virtual Reality . . . . Page 10
        1.5.2. Parallelisation of the virtual worlds . . . . Page 10
        1.5.3. Support for external connections . . . . Page 11
        1.5.4. Attributes . . . . Page 11
        1.5.5. Object Manipulation . . . . Page 11
        1.5.6. Graphical Support . . . . Page 11
        1.5.7. Gesture Recognition . . . . Page 11
    1.6. Summary . . . . Page 12

2. Design for Virtual Reality Systems . . . . Page 13
    2.1. System requirements and design . . . . Page 13
        2.1.1. System breakdown . . . . Page 13
        2.1.2. The Virtual Reality Operating System . . . . Page 14

2.1.3. Design Specifications . . . . Page 15
2.1.4. Data structures . . . . Page 16
2.1.5. Design of modules . . . . Page 16
2.2. The Implementation environment . . . . Page 17
2.2.1. Hardware . . . . Page 17
2.2.2. Software . . . . Page 18
2.2.2.1. The development environment . . . . Page 18
2.2.2.2. Transputer characteristics . . . . Page 19
2.2.2.3. Support software . . . . Page 21
2.3. Summary . . . . Page 22

3. Output Device Drivers . . . . Page 23
3.1. Design of an output device driver . . . . Page 23
3.1.1. Requirements . . . . Page 23
3.1.2. Design . . . . Page 24
3.1.2.1. Data structures . . . . Page 25
3.2. Implementation of the graphics pipeline . . . . Page 26
3.2.1. Rendering . . . . Page 26
3.2.1.1. Line drawing . . . . Page 26
3.2.1.2. Polygon rendering . . . . Page 32
3.2.2. Hidden Surface Removal . . . . Page 37
3.2.2.1. Categories of hidden surface algorithms . . . . Page 37
3.2.2.2. Implemented Hidden Surface Removal Algorithms . . . Page 38
3.2.2.3. Comparison of the different routines . . . . Page 44
3.2.2.4. Using the hidden surface routine . . . . Page 45
3.2.2.5. Additional ways to simplify hidden surface removal . . Page 46
3.2.2.6. BSP tree generation . . . . Page 47
3.2.3. Perspective projections . . . . Page 49
3.2.3.1. Standard perspective projection . . . . Page 49
3.2.3.2. Stereo perspective projection . . . . Page 49
3.2.4. Clipping . . . . Page 50
3.2.5. Transformations . . . . Page 51
3.2.6. Performance of the pipeline . . . . Page 52
3.3. Parallelisation of the graphics pipeline . . . . Page 53
3.3.1. Parallel Decomposition techniques . . . . Page 53
3.3.2. Pipeline parallelism . . . . Page 54
3.3.3. Coarse grained parallelism . . . . Page 56

3.3.4. Fine grained parallelism . . . . Page 57
3.3.5. Object-oriented parallelism . . . . Page 61
3.3.6. Further parallel implementation features . . . . Page 63
3.3.7. Optimisations and speed improvement techniques . . . . Page 64
3.4. Other renderers . . . . Page 64
3.5. Future additions . . . . Page 64
3.6. Summary . . . . Page 65

4. Input Device Drivers . . . . Page 66
4.1. Design of Input Device Driver . . . . Page 66
4.1.1. Requirements . . . . Page 66
4.1.2. Design . . . . Page 66
4.2. Implementation of the powerglove device driver . . . . Page 67
4.2.1. Converting glove data to gestures . . . . Page 68
4.3. Implementation of the keyboard device driver . . . . Page 70
4.3.1. Converting keys to gestures . . . . Page 70
4.4. Summary . . . . Page 71

5. The Virtual World Kernel . . . . Page 72
5.1. Design of the Virtual World Kernel . . . . Page 72
5.1.1. Requirements . . . . Page 72
5.1.2. Design . . . . Page 73
5.1.2.1. Representing worlds and objects . . . . Page 73
5.1.2.2. Functionality of the kernel . . . . Page 74
5.1.2.3. Communication between worlds and objects . . . . Page 75
5.2. Implementation of the Virtual World Kernel . . . . Page 76
5.2.1. Object Processes . . . . Page 77
5.2.2. Kernel Functions . . . . Page 77
5.2.3. The Communication layer . . . . Page 78
5.2.3.1. Communication between object and world . . . . Page 78
5.2.3.2. Communication between object and object . . . . Page 80
5.2.3.3. Communication between object and device driver . . . . Page 81
5.2.4. The Servers . . . . Page 82
5.2.5. Function Implementations . . . . Page 82
5.2.5.1. Getting object position and orientation . . . . Page 83
5.2.5.2. Creating objects . . . . Page 83
5.2.5.3. Moving from one world to another . . . . Page 84

5.2.5.4. Finding neighbouring objects . . . . Page 85
5.3. Kernel Performance . . . . Page 85
5.4. Summary . . . . Page 87

6. Virtual Reality Applications . . . . Page 89
6.1. Techniques common to all applications. . . . . Page 89
6.1.1. Movement in a virtual world . . . . Page 89
6.1.1.1. Movement in a virtual world . . . . Page 89
6.1.1.2. Navigation . . . . Page 90
6.1.2. Implementation of attributes . . . . Page 91
6.1.2.1. Boundaries . . . . Page 92
6.1.2.2. Portals . . . . Page 92
6.2. Implementations of virtual worlds . . . . Page 93
6.2.1. Architecture walkthrough . . . . Page 93
6.2.2. Virtual Checkers . . . . Page 95
6.3. Summary . . . . Page 95

7. Comparisons and Conclusions . . . . Page 102
7.1. AVIARY . . . . Page 102
7.2. Cyberterm . . . . Page 104
7.3. Distributed Interactive Simulation (DIS) . . . . Page 105
7.4. DIVE . . . . Page 105
7.5. Division . . . . Page 106
7.6. Minimal Reality (MR) Toolkit . . . . Page 107
7.7. Multiverse . . . . Page 108
7.8. Rend386 . . . . Page 109
7.9. The Virtual Environment Operating Shell (Veos) . . . . Page 110
7.10. Conclusion . . . . Page 111

8. References. . . . . Page 114

List of Figures

Figure 1 VPL Eyephones . . . . Page 4
Figure 2 VPL Dataglove . . . . Page 4
Figure 3 The Powerglove . . . . Page 5
Figure 4 Transputer network for INMOS Flight Simulator . . . . Page 9
Figure 5 Top Level breakdown . . . . Page 14
Figure 6 Virtual Reality Operating System . . . . Page 15
Figure 7 Hardware used for implementation . . . . Page 17
Figure 8 Logical Operations in three-dimensional viewing . . . . Page 24
Figure 9 Conceptual model of the 3D viewing process . . . . Page 25
Figure 10 Graphics pipeline for output device driver . . . . Page 25
Figure 11 Bresenham's line-drawing algorithm . . . . Page 27
Figure 12 Segmented line-drawing algorithm . . . . Page 29
Figure 13 Modified segmented line-drawing algorithm . . . . Page 30
Figure 14 DDA algorithm . . . . Page 31
Figure 15 Triangle fill algorithm . . . . Page 34
Figure 16 Z buffer algorithm . . . . Page 39
Figure 17 Scan-line algorithm . . . . Page 40
Figure 18 Painter's algorithm . . . . Page 41
Figure 19 BSP tree generation algorithm . . . . Page 42
Figure 20 BSP rendering algorithm . . . . Page 43
Figure 21 Cyclic overlapping of polygons . . . . Page 45
Figure 22 Performance of the components of the graphics pipeline . . . . Page 52
Figure 23 Parallel decomposition taxonomy . . . . Page 54
Figure 24 Illustration of latency in pipelines . . . . Page 55
Figure 25 Processor layout for pipeline parallelism . . . . Page 56
Figure 26 Processor layout for coarse grained parallelism . . . . Page 57
Figure 27 Processor layout for fine grained parallelism . . . . Page 59
Figure 28 Processor layout for object oriented parallelism . . . . Page 62
Figure 29 State transition diagram for glove control . . . . Page 69
Figure 30 Summary of kernel design . . . . Page 77
Figure 31 Structure of communication layer . . . . Page 80
Figure 32 Complete plan of communication layer . . . . Page 81
Figure 33 Pseudocode description of object creation function . . . . Page 83
Figure 34 Pseudocode description of object transfer function . . . . Page 84

Figure 35 Views of applications (See Table XIV for details) . . . . Page 97
Figure 36 Views of applications (See Table XIV for details) . . . . Page 98
Figure 37 Views of applications (See Table XIV for details) . . . . Page 99
Figure 38 Views of applications (See Table XIV for details) . . . . Page 100

List of Tables

Table I Lines of varying length drawn per second by each routine . . . . Page 31
Table II Characteristics of Line drawing routines . . . . Page 32
Table III Performance of Triangle Fill Routine . . . . Page 35
Table IV Timing values for the different hidden surface removal routines . . . . Page 44
Table V Polygon counts when generating BSP trees . . . . Page 48
Table VI Speed of the rendering system in frames/s . . . . Page 53
Table VII Frame rates for pipeline parallelism . . . . Page 56
Table VIII Speedup values for coarse grained parallelism . . . . Page 58
Table IX Speedup values for fine grained parallelism . . . . Page 60
Table X Speedup values for object oriented parallelism . . . . Page 63
Table XI Glove configurations for various gestures . . . . Page 69
Table XII Update rate for particle processes . . . . Page 86
Table XIII Data transferred per world server in Kbytes/second . . . . Page 87
Table XIV Description of the scenes in Figure 35 - Figure 38 . . . . Page 101

1. Introduction

This chapter introduces the concept of virtual reality and describes the purpose and design goals for a virtual reality system implemented on a parallel architecture. The first section provides a motivation for the use of virtual reality. Descriptions of other virtual reality systems and the transputer processor are then presented. A description of the purpose and requirements of the system is deferred to the end of the chapter.

1.1. Introduction to Virtual Reality

A person using a computer is typically confronted with some form of user interface, be it a prompt on a monitor requiring a response with input from a keyboard, a set of menus from which options may be selected or a windowing system complete with buttons and scrollbars. While some of these are comfortable to use, they impose a restriction on the manner in which input may be provided and the way that the output may be displayed.

In contrast, the ’user interface’ to conventional (physical) reality allows access to a degree of freedom far exceeding that of a computer. Manipulation of objects can be done in three dimensions by reaching out and grasping the object. Vision is unhampered, and is independent of the physical dimensions of a piece of hardware. Even when working on a two dimensional surface, it is possible to reach up to adjust a light or turn around to pick up a pencil.

Clearly there is a difference between the current (pre-virtual reality) range of computer user interfaces and the one that most people are accustomed to using. These user interfaces are at most two dimensional. The virtual reality paradigm seeks to create three dimensional work areas with concomitant interaction techniques.

In practice, each user would don a pair of goggles and a glove to enter a computer generated world. Stereo images provided by a pair of displays in the goggles would cause computer models of objects to appear in three dimensions around the user. The glove would allow similar interaction techniques to those normally found in physical reality. In its ultimate form, all the senses of the user would be supplied with stimuli from the computer and all his responses would be monitored and used to control their situation in the virtual world.

Formally, virtual reality can be defined as "... real-time interactive graphics with three-dimensional models, when combined with a display technology that gives the user immersion in the model world, and direct manipulation" [46].

1.2. Project goal

Virtual Reality is an application well suited to implementation in a parallel environment. To simulate reality on a computer, the machine has to produce realistic images rapidly. Such a requirement necessitates either expensive equipment or a compromise on the part of the viewer. This situation has resulted in the prominent dichotomy currently evident, where two clearly distinguishable camps may be found. The first relies on sophisticated hardware rendering at great expense; the other uses more readily available personal computers and accepts slower, less realistic images.

An opening is thus available for mid range equipment, with inexpensive processors that can be connected in parallel to provide a system of some power. The transputer, described in section 1.3, satisfies the requirements for such a processor.

The intention of this thesis is to investigate the implementation of various virtual realities on a cluster of transputers. A description is given of a system that can support the development of various virtual reality applications. Examples of the types of virtual reality application that currently exist are given in section 1.4.

A detailed description of the requirements for such a system is given below in section 1.5.

1.3. The Transputer

The IMS T800 transputer is advertised as a 32 bit microprocessor with a 64 bit floating point unit and graphics support [23]. The instruction set is efficient and supports inter-processor communication as well as efficient switching between multiple processes on each processor. The floating point unit operates concurrently with the processor and is rated at 2.2 Mflops when the processor is running at 20 MHz.

The graphics support consists of a powerful block move instruction capable of transferring data as a two dimensional array. Versions of the block move exist that transfer only zero-valued, or non-zero-valued bytes.

The transputer contains 4 bidirectional communications links allowing a number of these processors to be connected and to work in parallel as a MIMD machine. Each link operates at 20 Mbits/s for 20 MHz transputers, and a single bidirectional link can transfer a theoretical maximum of 2.35 Mbytes/s, or 1.74 Mbytes/s if communication is only in one direction. Link communication is performed by DMA, thus allowing computation to proceed concurrently with communication.

1.4. Virtual Reality Applications

This section describes some virtual reality hardware and software that has been implemented with current technology. A brief overview is given of the type of equipment available, and the range of applications to which virtual reality has been applied.

1.4.1. Hardware used for Virtual Reality

This section describes some of the hardware devices used in virtual reality systems. The descriptions are intended to illustrate some of the techniques that have been used for overcoming the interfacing obstacles involved in creating a natural computer-human interface.

1.4.1.1. Polhemus tracker

The Polhemus 3Space Isotrak system is a device capable of sensing position and orientation in space [24]. It comprises sources that generate a magnetic field, and sensors that are located on the object being tracked. The sensors contain perpendicular coils which are used to determine the orientation of an object through measurement of the magnetic field.

1.4.1.2. VPL

VPL produces both hardware and software for virtual reality. The VPL eyephones (a head mounted display) and Dataglove are standard components in most high-end virtual reality systems.

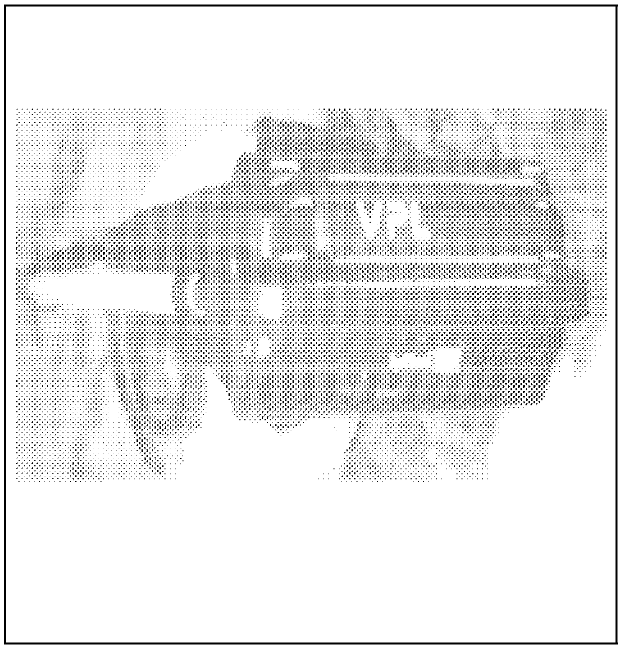


Figure 1 VPL Eyephones

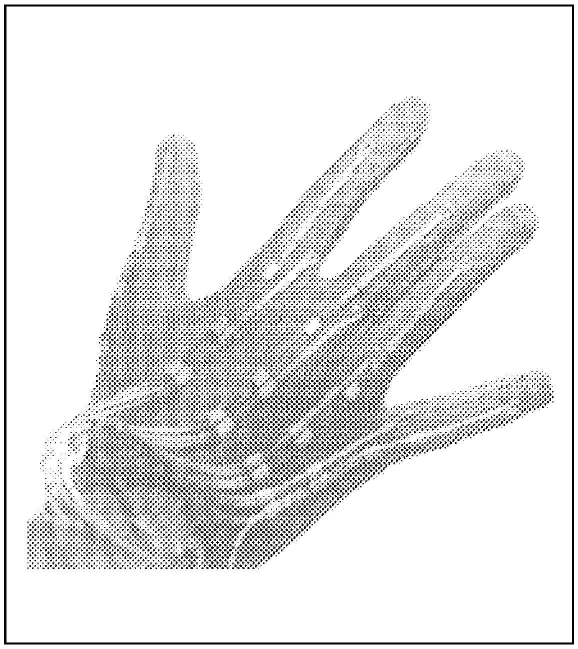


Figure 2 VPL Dataglove

The Eyephones (see Figure 1) are a colour stereo display system worn on the head. The stereo image is displayed on two liquid crystal screens and viewed through wide angle optics. Each screen has a resolution of 720 x 480 pixels. Head position and orientation may be measured using the attached Polhemus 3Space tracker. Stereo headphones may also be included. The Eyephone, together with a Polhemus tracker, was selling for $9,150 during August 1991.

The Dataglove (see Figure 2) has optical fibre sensors on the backs of the hand and fingers to measure flexing and extension of the fingers. A magnetic sensor (Polhemus 3Space Isotrak) measures the position and orientation of the hand. In August 1991, the Dataglove including Polhemus tracker cost $8,800.

1.4.1.3. The Powerglove

The Powerglove (see Figure 3) was developed to provide a cheaper alternative to the Dataglove. Strain gauges replace the fibre optics for measurement of finger position, and ultrasonics replace the Polhemus tracker for position sensing. The powerglove is capable of measuring absolute position in space, wrist rotation and finger bend. Originally developed for the computer games market, the powerglove has since been taken out of production. Surplus units are being collected by low-end virtual reality enthusiasts for about $50.

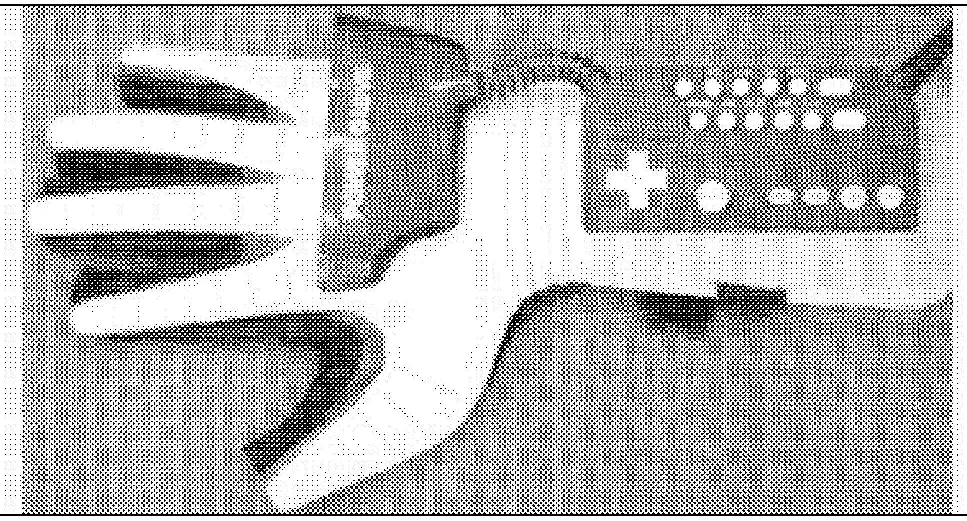


Figure 3 The Powerglove

1.4.2. General Virtual Reality systems

Descriptions of a number of existing virtual reality applications follow. The intention of this section is to illustrate some of the vast range of applications for virtual reality that are feasible with current technology. This section also describes some of the hardware and software being used to support these applications.

The rendering speed of the various systems is given where possible. These values may be used for a rough indication of the performance, but direct comparison may be unreliable as the size and complexity of the scene is often not specified.

1.4.2.1. The Walkthrough Project

An architecture walkthrough enables a viewer to explore a building by simulating a walk through a model. For virtual reality purposes, the model is in electronic form, and the simulation is performed by the computer.

The walkthrough project at the University of North Carolina at Chapel Hill [8] has been evolving for a number of years, improving both the rate at which images can be produced and their realism. The current version, Walkthrough 5.2 can achieve an update rate of 25 stereo frames per second on models of 30,000 polygons. Models are illuminated via radiosity calculations with

Gouraud-shaded[1] patches. Walls, floor, ceilings and furnishings may be textured. Sound appropriate to each location is played as the user moves around. The user can interact with the model to the extent of opening doors, carrying objects and pushing furniture.

The user wears a stereo head-mounted display, and can be tracked within a 12’x10’ area and so can move naturally around in this space. The system runs on a Pixel-Planes 5, a machine developed at Chapel Hill. The Pixel-Planes 5 is rated at 2 million Phong-shaded[2] polygons per second.

1.4.2.2. Reality Built For Two

This system is a development platform for designing and implementing virtual realities [6]. The system consists of a solid modelling package and an animation control package running on a Mac II which is used for world design. The completed world is then passed to Silicon Graphics Iris workstations for rendering. The system is able to render worlds of 1400 polygons at interactive rates of 10 Hz or higher.

The one or two users are equipped with VPL eyephones and Datagloves. A Polhemus 3Space Isotrak is used to gather data about the users’ positions and actions.

In designing the virtual reality, the user specifies lighting, object shapes, colours, mechanical linkages and motion constraints with the modelling tool [45]. Object behaviour, interaction and animation can then be controlled by the animation package, which combines the world description with the user data, and broadcasts the result to the renderers.

1.4.2.3. Integrated Simulation for Autonomous Underwater Vehicles

Autonomous Underwater Vehicles are submarines designed to operate unattended. Testing the control software for such devices is complicated by the inability to monitor them directly during operation, difficulty in recovery if an accident occurs, and the complexity of the different control

[1] Gouraud shading: A method of colouring a polygon to approximate the smooth change in shade found in curved surfaces.

[2] Phong shading: A more realistic form of shading than Gouraud shading[1] but also more computationally expensive.

modules.

An integrated simulator is being developed at the Naval Postgraduate School in Monterey [9]. Here the submarine control system is networked to a three-dimensional graphics workstation which simulates the underwater environment and allows monitoring of the response from the vehicle. The submersible is effectively being placed in a virtual reality.

The simulator software simulates details of vehicle interaction by providing the various sensor information and modelling some simulated components (e.g. control surfaces, propellers, gyrocompass). An external view of the simulated world in three dimensions with real-time response must be provided. Various components of the vehicle may be simulated as well, allowing software testing to occur even if the entire vessel is not available. Telemetry data may be easily collected during simulation.

The graphical simulations all run on Silicon Graphics Iris workstations.

1.4.2.4. Medical Imaging

Work is being done at the University of North Carolina at Chapel Hill on real-time rendering of 3D ultrasound data [5]. The application being created is a virtual environment that displays the ultrasound images in place within the patients anatomy. The overall effect is that of being able to see within the patient.

The system uses a Sun 4 with real time video digitizer to collect the ultrasound data and transfer it to the Pixel Planes 5 graphics multicomputer. Polhemus sensors track the position of the head mounted display (HMD) and ultrasound sensor. The rendered images are combined with the views from head-mounted TV cameras and displayed on the HMD.

Other suggested applications for combining computer generated images with real world objects include highlighting nearby arteries during surgery, seeing through smoke in burning buildings, and showing servicing data for complicated machinery.

1.4.2.5. Visualization

Scientific visualization is the use of a computer to produce a visual representation of some physical phenomena. It is particularly useful when physical data is too complex to be understood easily. A visual analog is often more comprehensible.

An example of the use of scientific visualization is the virtual wind tunnel being developed at NASA Ames Research Center [10]. It is designed to visualise three-dimensional fluid flows. The users can pick up and move smoke sources around using datagloves. They can move around within the flow viewing it from any angle without disturbing the system.

The system uses a Binocular Omni Orientation Monitor (BOOM) to display the images. This consists of a pair of high resolution cathode ray tubes mounted on a yoke. Position tracking is performed by measuring the angles of the joints in the yoke. Computation of the flow is performed on a Convex 3240, with rendering performed by an Iris 380 GT/VGX system. The system attempts to run at 10 frames/s or higher.

1.4.3. Transputer based Virtual Reality systems

Some systems that use transputers for interactive graphical tasks are mentioned in this section.

1.4.3.1. The INMOS multi-player flight simulator

This system was developed as an illustration of the real-time graphics capabilities of the transputer [4]. The simulation software runs on a number of transputers corresponding to the number of players. The simulation processors are linked and each is connected to a joystick and to a network of transputers used for rendering. A separate rendering network is used for each player. The rendering network consists of a pipeline to perform the required tasks of hidden surface removal, transformation, clipping and polygon shading (see Figure 4). A frame rate of 17 frames per second is obtained for the few hundred polygons being rendered.

1.4.3.2. ProVision

The ProVision system developed by a Bristol based company, Division, is a virtual reality server that can be hosted by a machine such as a PC, Sun Sparcstation or a VAX. The system consists of a parallel processing rendering engine with accompanying software [32].

The hardware consists of a box holding up to 20 processing cards. The processors may be T805 and T425 transputers used for data transfer, Intel 860’s for floating point and Toshiba HSP

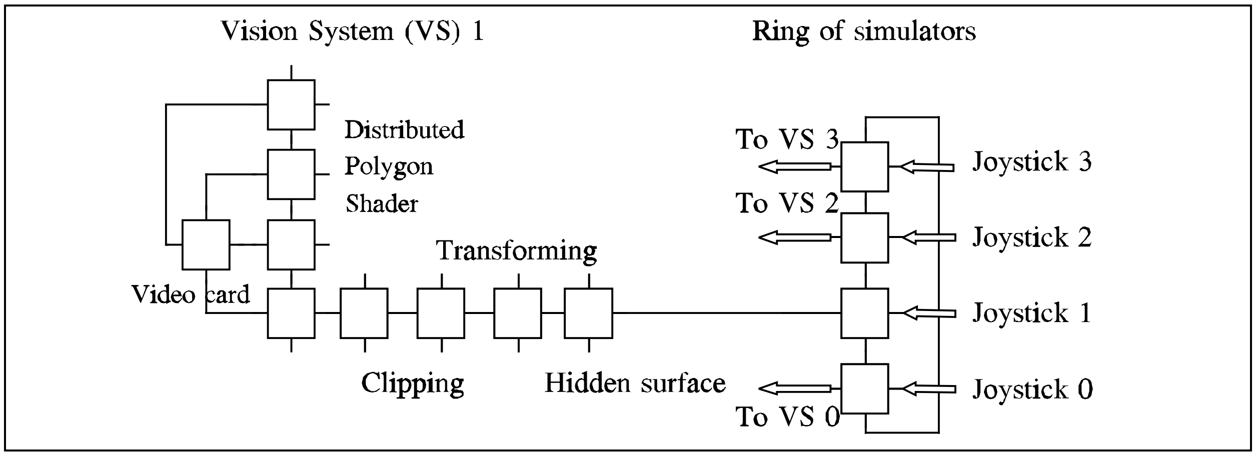


Figure 4 Transputer network for INMOS Flight Simulator

polygon processor for rendering. VPL Eyephones, Datagloves and Polhemus trackers are supported.

The software provides support for the parallel processes that are used to implement the virtual reality.

Stereoscopic ProVision systems were selling for around £30,000 in October 1991. This price excluded the goggles and gloves. The system is estimated to render between 60,000 and 80,000 polygons per second. In theory this would correspond to about 3500 polygons a frame at 20 frames per second.

1.5. Design Criteria

This section discusses the design criteria for a general virtual reality system.

Some specialised terminology is used to describe certain concepts. In this document the term ’application’ refers to a particular model of a reality. A ’world’ is the environment in which a virtual reality may be modelled. The term ’object’ is used for the individual components of a world. A ’user’ refers to a human participant in the virtual reality. For example, in a simple virtual reality application which models a solar system, the world could consist of a collection of cubes representing planets and a ’celestial’ sphere acting as a boundary for the model. Each of the cubes is an object as is the sphere. A world may have ’laws’, for example all objects attract all other objects with a force proportional to the product of their masses. The mass of an object is an ’attribute’ of that object. The intention is that these terms have similar meanings when applied to both the physical reality and to the virtual reality.

A certain parallel may be drawn to object-oriented terminology. The overlapping terms will have the meanings defined above unless it is explicitly stated otherwise.

1.5.1. The Virtual Reality

The systems mentioned above have been designed for a specific purpose. There are few systems that are suitable for use in creating general virtual realities. At the time that this project was started, none were known of that were freely available for research purposes, and that could be used with inexpensive equipment.

In a report on research directions in virtual environments [46], available systems were classified as commercial, supporting a particular vendor’s hardware, or as research software. Tools that accompany proprietary hardware are oriented more to the support of that hardware than to the development of virtual environments. The tools created for research purposes also show this tendency, although they tend to be less hardware specific.

The system described in this thesis was intended to overcome these problems. It was designed to be able to support a wide range of applications. In essence it was intended as a platform for the development of any given reality. As such the system concentrates on providing functions one would expect to use when creating a virtual reality.

1.5.2. Parallelisation of the virtual worlds

Many people and objects may interact in physical reality. The same should be possible in a virtual reality. A given virtual reality application may use more than one world, or it may be useful to run more than one application at one time. Thus the virtual reality system should provide support for multiple users and objects, and for multiple worlds running concurrently.

Modelling a reality may require considerable processing power, as will support for multiple objects and worlds. The system should take advantage of the parallel processing facilities offered by a transputer cluster in order to distribute the load. The system may eventually be spread across different architectures via a network or some other similar medium, and should be designed to provide for this as a future development.

1.5.3. Support for external connections

In addition to allowing many users to exist in each world, the system should provide support for these users to exist at remote sites. This would reduce the interfacing requirements on the host machine, and allow easy utilization of the system.

1.5.4. Attributes

In the real world, objects have certain characteristics which determine the way in which they behave. For example, the mass of the object influences the size of the gravitational force on it. In an arbitrary reality, different rules may apply. These rules are often based on certain attributes of the object. The system should support a means of assigning such arbitrary attributes to the various objects.

1.5.5. Object Manipulation

The most important part of the virtual reality simulation is the ability to interact with the objects as if they were real. The facilities provided by the system should make such interaction easy to incorporate into an application.

1.5.6. Graphical Support

In order for the ’reality’ effect to be realistic the scene must be rendered rapidly, typically 20 - 50 frames a second. A goal of this project is to use the parallel processing abilities of the transputer to produce the required images at acceptable speeds.

1.5.7. Gesture Recognition

The interaction techniques used in a virtual reality should resemble those used in physical reality. The system should be capable of sensing the position of the user and reacting accordingly. Interaction with objects should be natural. In general, the time taken to learn to interact with the virtual reality should not be much more than the time taken to gain familiarity with the interfacing hardware being used.

1.6. Summary

The intention of this chapter has been to introduce the field of virtual reality and the applications that currently exist. A proposal for a virtual reality system using a parallel architecture has been given. Future chapters will give a more detailed breakdown of the system, and discuss implementation options in the light of empirical measurements of their effectiveness.

2. Design for Virtual Reality Systems

This chapter gives the top level breakdown for the virtual reality system. The development environment for this project is also described.

2.1. System requirements and design

The implementation of virtual realities has only recently become feasible with the improvement in computer technology. As such, few general systems have been developed and their design is still a corporate secret in many cases. A few descriptions of the various systems can be found in published literature, such as those described in section 1.4. These, however, are mostly dedicated to one particular application. The sections below detail the reasoning used to produce the top level breakdown of a general virtual reality system.

Around the time that development of this system started, other researchers also recognised the need for a general virtual reality system. A number of other similar systems have been developed during the lifetime of this project. A comparison with the design of these systems will be presented toward the end of this thesis.

2.1.1. System breakdown

Many of the known virtual reality systems, for example ProVision and Reality Built for Two, make use of hardware rendering to obtain high speed graphics. Others, for example, the University of North Carolina at Chapel Hill, use parallel processing in a specialist machine for this purpose. The intention of this system is to attempt fast rendering using a non-specialised, parallel architecture. The INMOS flight simulator uses a similar approach, dedicating a number of transputers specifically for rendering, and others for the simulation itself.

The use of rendering hardware implies a separation between the management of the world, entailing interaction with the objects, and the viewing of the world, involving drawing of the objects. It therefore makes sense to separate the graphics from the world management. Using a standard protocol, the description of the world can be passed from the world manager to the rendering routine. This would also allow rendering hardware to be easily substituted for the software renderer.

The advantages of separation of the routines producing graphical output from the virtual world manager suggests a similar separation for the input routines. With the wide range of input devices currently available such as keyboards, mice and gloves, easy substitution of one for the other requires a degree of separation from the rest of the system.

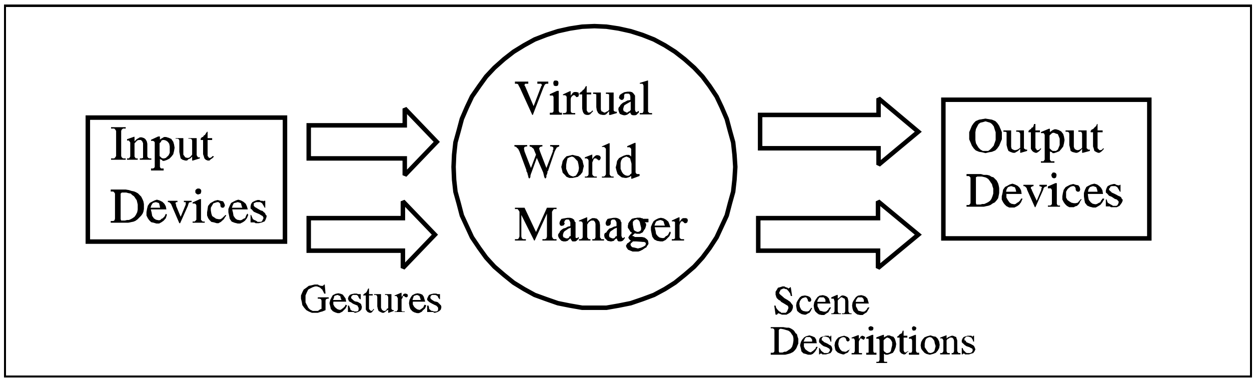


Figure 5 Top Level breakdown

The top level breakdown of the system is summarised in Figure 5. The input device drivers gather and translate the raw data from the hardware and transfer it in some standard format to the virtual world manager. For virtual reality, a suitable format would be a gesture such as MOVING LEFT or POINTING FINGER. This abstracts over the different characteristics of each device and just returns the salient information required by the virtual world manager. An output driver will take a description of the objects in the world and their positions and render them. The virtual world manager may operate at a high level working with worlds and objects. The actual three dimensional description of the objects and the physical characteristics of input devices can be safely abstracted to the device drivers.

2.1.2. The Virtual Reality Operating System

The specification for the system requires that it be able to support different applications. These applications will differ in the way objects interact with each other and with the world. Each application will however need a world containing objects and some means of controlling the objects. It would make sense to group the functionality common to all applications in a common core. A higher level application layer can then selectively use these functions to create a customised world.

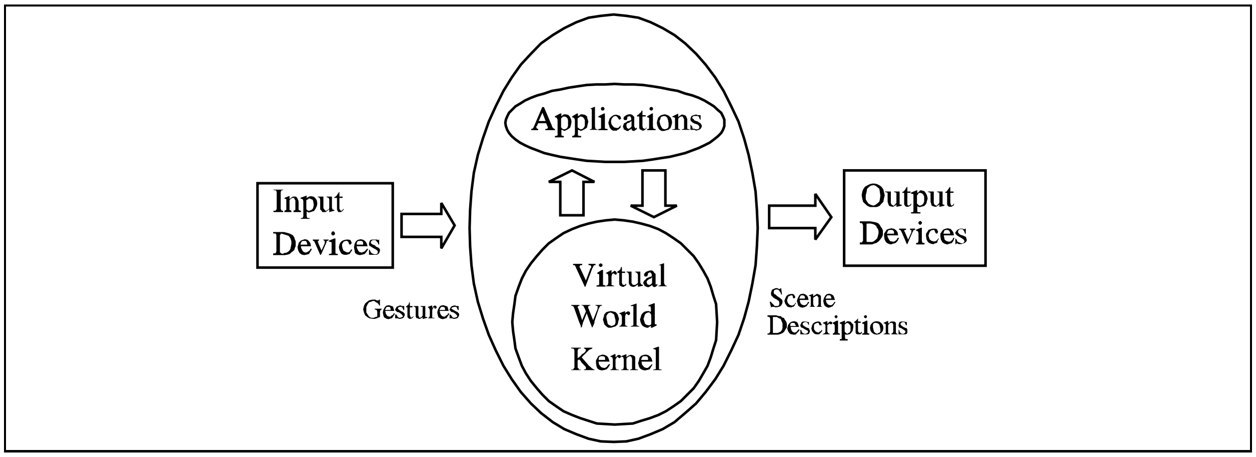


Figure 6 Virtual Reality Operating System

The system with the application layer included is shown in Figure 6. The resemblance to a conventional operating system may be seen. The virtual world kernel manages the resources of the virtual world in much the same way as, for example, the UNIX kernel manages the resources of the machine. Similarly the device drivers provide a device independent means of accessing various pieces of hardware. The application programs make use of the functions provided by the kernel.

2.1.3. Design Specifications

The specifications for the system given earlier give the following requirements for the various sections of the system:

The application layer must:

- Support the development of different virtual realities.

The virtual world kernel must:

- Support multiple worlds.
- Support multiple users in each world.
- Support connections from other machines.
- Use parallelism to reduce the load on individual processors.
- Provide the ability to associate attributes with objects.

- Allow object manipulation routines to be easily created.

The output device driver must:

- Render an image of a world.
- Use parallelism to attempt to reduce rendering time.

The input device drivers must:

- Interface with the input device being used.
- Determine which gesture is being made and pass this information to the virtual world manager.

2.1.4. Data structures

At this point it is appropriate to decide on the principal data structures for the system. The essential structures for a virtual reality system are objects and worlds.

The objects are required to have attributes associated with them. While the set of attributes is not completely defined, some attributes are required for the operating of the system. For correct display, object position and orientation must be known. In order to support general attributes, provision must be made for extra fields that can be specified at run time. The structure representing an object will then have the following fields:

- position (position in space).
- orientation (direction in which the object is facing).
- name (or type of the object).
- scale (size of the object).
- list of other attributes.

A world consists of a collection of objects.

2.1.5. Design of modules

The four areas described above can now be designed and implemented independently. This design and implementation will be detailed in the following chapters.

2.2. The Implementation environment

Certain design decisions are influenced by the need to work efficiently on the transputer architecture. The sections below describe the working environment.

2.2.1. Hardware

Two transputer clusters containing 16 processors each were available for the implementation of the system. Each cluster consisted of a motherboard equipped with cross-bar switches allowing the external links on the transputer to be connected in any configuration. The cross-bar switches were software switchable. Each transputer was mounted on a separate worker board that plugged into one of the 20 connectors on the mother board. Each worker board contained a T800 transputer as well as between 2 and 8 Mbytes of RAM.

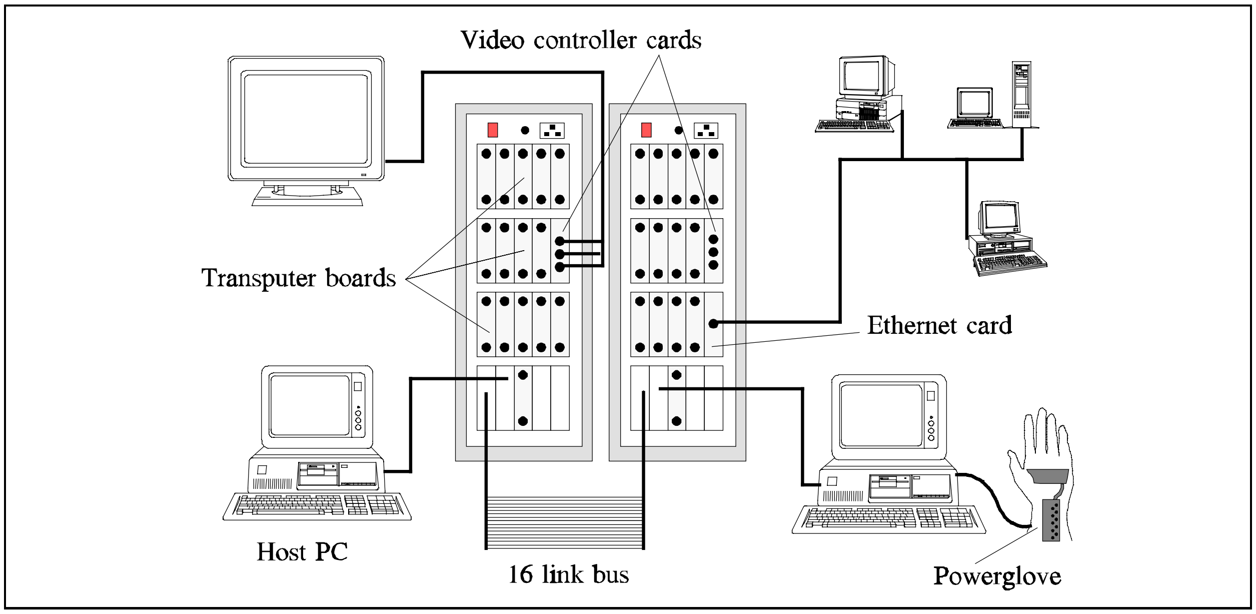


Figure 7 Hardware used for implementation

Three of the transputer boards were specialised. Two contained G300 video controllers and could be used for the display of graphics. The third was equipped for communication over ethernet.

The two clusters were linked by a bus capable of carrying 16 links. These links could also be controlled by the cross-bar switches.

Externally, each cluster was connected to a PC via a RS422 interface. Configuration information and programs were downloaded via this link. Other workstations, both PCs and Sparcstations, could communicate with the clusters via the ethernet.

A powerglove was interfaced to one of the host PCs. This device could measure finger bend, wrist rotation and the position of the hand in space. Data from the glove could be transmitted via the PC to programs running on the transputers.

The hardware used is summarised in Figure 7.

2.2.2. Software

The transputer is capable of functioning as a conventional sequential processor and, with its floating point unit, is well suited to such applications. It also contains a number of instructions suited to parallel programming. The section below details the programming environment used for the virtual reality system and some of the details whose understanding is crucial to successful transputer programming. These details are also necessary to understand some of the implementation decisions made in this project.

2.2.2.1. The development environment

Several different development options are available for software development on transputers. The programming tools available when developing this system were the Occam programming language, the Helios Operating System, and a PC based C cross compiler. Unfortunately a compiler for an object-oriented language was not accessible.

2.2.2.1.1. Occam

The Occam programming language was developed from David May's EPL (Experimental Programming Language) and Tony Hoare's CSP (Communicating Sequential Processes) [22]. Occam's development by INMOS Ltd was closely linked to the design of the transputer chip. Occam instructions thus map to transputer assembler instructions very easily.

Occam code supports programs written as a collection of processes, communicating through channels. The Occam minimalist philosophy has produced a language that is not much more expressive than the transputer assembly language. Certain programming techniques such as recursion cannot be easily implemented [4]. These considerations resulted in Occam being rejected in favour of a higher level language.

2.2.2.1.2. Helios

Helios is an operating system designed for transputer architectures [47]. The system is similar to UNIX, and includes a UNIX compatibility library with functions largely compatible with the POSIX standard. It is capable of running on a network of processors.

Use of this system indicated that the overhead of the operating system on the transputer links produced a noticeable delay in the system, making it unsuitable for fast graphics applications. The operating system also had a tendency to fail frequently making it unsuitable as a development environment.

2.2.2.1.3. Transputer Cross-compiler

The stand-alone cross compiler was found to be the most satisfactory of the three options [44]. The compiler is an ANSI C compiler with support for the extra transputer options. The extra options are in the form of library calls that are compiled to inline code, making them easy and efficient to use. Complete control of the transputer links is given making it possible to extract maximum bandwidth.

The compiler has a network loader that downloads an executable file from a PC onto each node in the cluster. It then runs a server on the PC making it possible to access the disk on the PC from the root node in the cluster using standard C file functions. The server may be modified to allow information from other peripherals to be transferred to the transputers.

2.2.2.2. Transputer characteristics

As with all devices, the transputer has certain characteristics that influence the way in which programs are constructed. This section points out the constraints that are placed on the programmer of the device.

The memory map of the transputer is a 4 Gbyte linear address space. The T800 processor has

4 Kbytes of fast on-chip RAM starting from address 8000000H. The first few addresses of this memory are used by the processor to control the external links and other processor functions. Conventional system memory normally occupies addresses directly after the on-chip memory.

Communication on the transputer is by means of message passing over channels. Passing dynamic data structures is difficult, especially where the data contains pointers which are processor dependent. The tendency is to allocate contiguous blocks of memory for those structures that will need to be passed between processes, and to avoid the use of pointers.

Since channels are memory mapped and the transputer has no built-in run-time stack checking, processes are responsible for either allocating sufficient stack space, or for performing their own overflow checks. The optimal area to use for stack space is the on-chip RAM due to the low access time. However stack overflow in this area can cause overwriting of processor control data. Thus stack control is an area for careful consideration. In particular, recursion is almost always undesirable.

Another potential problem area is channel use. There are two types of channel, the physical link implemented as a wire joining processors, and a software link for message passing between processes on the same processor. No distinction need be made by the programmer and both types of channels can be regarded as an address in memory. Synchronization occurs by placing certain values in these addresses. Communication will occur successfully only if there is one process reading from the channel and one writing to it. An easily made mistake is to have two processes reading from the same channel simultaneously. In this case the result is unpredictable, but often one process may erroneously succeed in reading.

An alternation mechanism is built into the transputer to control access to common resources. Each process is assigned a channel, and all the channels are polled to find out which processes are requesting permission to use the resource. The channels must occupy a contiguous block of memory. This makes it difficult to allow new processes to join those requiring access while a program is running. In practice, a maximum is usually defined and this is fixed during the running of the system.

2.2.2.3. Support software

While a stand-alone compiler proves useful in removing all the extra overhead of an operating system, it does have the disadvantage that many useful facilities are unavailable. To cope with this shortcoming, a number of support modules were written. These are not of direct relevance to the functionality of a virtual reality system, but are essential to the operating of the system.

One useful facility is the ability to route messages between any two processors. Normally this would require explicit routing of the data, and processes on all intermediate processors to forward the data. A communication module was built that allowed communication between arbitrary processors. The module is given a representation of the external link network, and it designs a routing table using the shortest path between nodes. It creates processes to monitor each of the external links of each transputer and to forward messages destined for transputers further on. The appearance of the function calls for this interprocessor communication resemble the original channel function calls, with the addition of a field for the number of the destination processor. No buffering is done by this module, buffering must be performed by the receiving process if link congestion is to be avoided.

The functions provided by the compiler are only capable of accessing the PC server from the root node of the transputer cluster. The server may be used to print messages on the screen, to read keystrokes, to read and write to files and to interface with any other attached peripherals, such as the powerglove. The restricted access is a problem when these facilities are needed by the other nodes, especially for debugging purposes. The link module mentioned above was used to extend the facilities.

Libraries were also constructed to prevent contention between processes on one processor for resources on that processor. The transputer can run high priority processes that are not subject to normal context switching. This was used to implement semaphores which can be used to ensure exclusive access to some device or shared memory. No restriction is placed on the number of processes that can contend for some shared resource with this method, unlike the technique mentioned in section 2.2.2.2. A set of functions to implement a shared buffer were then added.

The transputer ethernet card was only supplied with drivers for use under Helios. The low-level drivers were ported to run under the stand-alone C, and a partial TCP/IP implementation was added to this. The implementation of a standard protocol allows other operating systems to communicate easily with the transputer cluster.

2.3. Summary

This chapter has presented a top-level breakdown of the system being implemented. Further chapters will explore each of the sections mentioned in greater detail, and will describe implementation considerations for each.

3. Output Device Drivers

This chapter starts by presenting the design for the output device drivers. Implementation issues are then considered, with the emphasis on the implementation of the rendering system running on the transputer cluster. Techniques for producing fast graphics on a single processor are first considered, followed by details of extending these to multiple processors. A brief mention of the other implemented output drivers is then made.

3.1. Design of an output device driver

3.1.1. Requirements

The requirements for the output device driver were that it must:

- Render an image of a world.
- Use parallelism to attempt to reduce rendering time.

The rendering requirement is not a novel application. Many systems have been created where three-dimensional scenes have to be rendered. The use of parallelism is not particularly unique either, a number of systems have attempted a measure of parallel rendering. In general, however, these systems have not been intended for a virtual reality application.

The two principal differences where virtual reality applications are concerned are that the images must be rendered very rapidly, in real time, and that the delay between a user input and the corresponding change in the rendered output, the latency, must be small. In working virtual reality systems [30] [1], frame rates of 6 frames/second and a latency of less than 200 milliseconds are mentioned as minimum requirements for the illusion of reality.

Since speed and generality are often contradictory requirements, one must usually be sacrificed for the other. In this case the complexity of the objects to be rendered will be reduced to improve rendering time. The following are limitations that will be accepted:

- Objects will be made of polygons.
- Polygons will be convex.
- Objects will not intersect.

The first two limitations are reasonable, and are not uncommon. Curved surfaces can be approximated to any level of detail by using a sufficiently fine polygon mesh. Concave polygons can be created by using a number of convex polygons. The third limitation seems excessively strict in a virtual reality system that is concerned with objects that will be interacting. This limitation does remove the necessity for a large amount of computation, and so will be retained for the present. Future versions of the renderer may attempt to overcome it.

3.1.2. Design

Renderers are typically constructed by creating a pipeline of several standard operations (Figure 8 [18], Figure 9 [14]). These operations include clipping, transformation, projection and hidden-surface removal. The description of the scene to be displayed is fed into the pipeline which eventually produces a sequence of primitives (two dimensional lines and polygons, for example) to be drawn. The pipeline typically ends with some routines capable of producing a visual representation of these primitives.

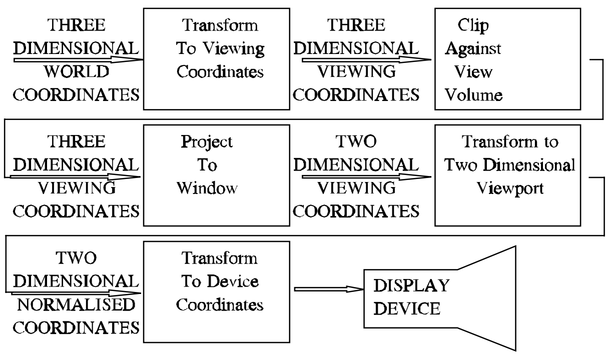

THREE DIMENSIONAL WORLD COORDINATES
Transform To Viewing Coordinates
THREE DIMENSIONAL VIEWING COORDINATES
Clip Against View Volume
THREE DIMENSIONAL VIEWING COORDINATES
Project To Window
TWO DIMENSIONAL VIEWING COORDINATES
Transform to Two Dimensional Viewport
TWO DIMENSIONAL NORMALISED COORDINATES
Transform To Device Coordinates
DISPLAY DEVICE

Figure 8 Logical Operations in three-dimensional viewing

The clipping stage slices off the pieces of the object that should be displayed. The transformation operations translate, scale and rotate objects so that they appear in the correct position relative to the viewer. Projection is necessary to display a three-dimensional representation to a two-dimensional display device. Hidden surface removal causes only the visible parts of objects that are obscured by others to be drawn.

The requirement for the renderer states that it must be able to render an image of the world. The world consists of a collection of objects as defined in section 2.1.4. The pipeline used to render

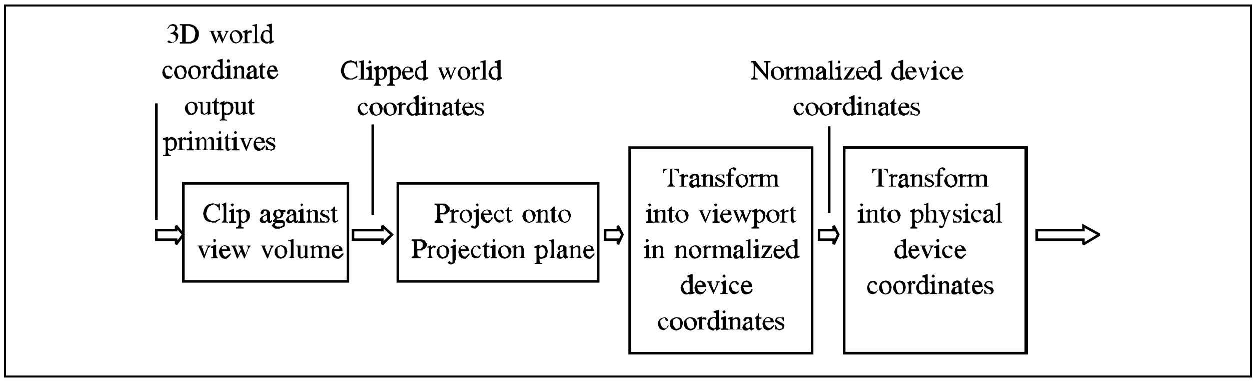

3D world coordinate output primitives
Clipped world coordinates
Normalized device coordinates
Clip against view volume
Project onto Projection plane
Transform into viewport in normalized device coordinates
Transform into physical device coordinates

Figure 9 Conceptual model of the 3D viewing process

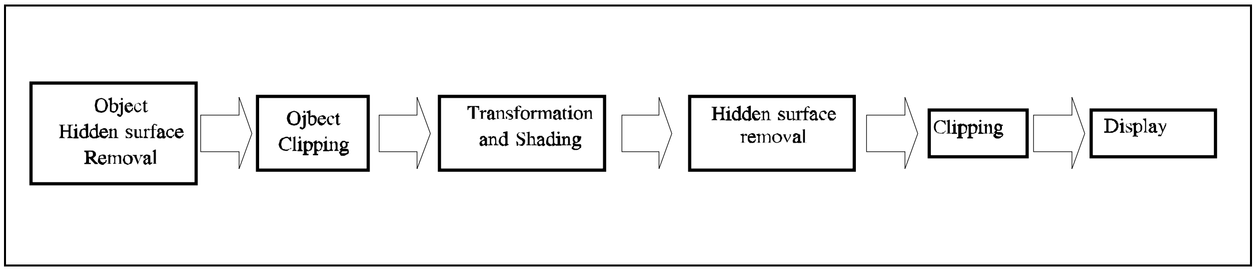


Figure 10 Graphics pipeline for output device driver

a world is shown in Figure 10.

The first stage is the hidden surface removal routine for the objects in the world. This determines which objects are obscuring others allowing each object to be drawn appropriately. The object clipping stage eliminates those objects not visible from the current viewpoint.

The rest of the pipeline works at a polygon level for each object. The polygons are moved to the appropriate position relative to the current viewpoint. At this stage they may be given appropriate colour values to simulate the effect of light sources in the world. Next, corrections are made to take into account polygons that are obscuring others. Polygons are then clipped so that only the sections that are to appear on the display device remain. Finally each polygon is drawn.

3.1.2.1. Data structures

The data structure used to represent each polyhedral object is a standard one. It consists of a list of the vertices in the object, and a list of the polygons in the object. Each polygon is described as a list of vertices, each vertex specified by its index in the vertex list. This representation is similar to the Object File Format (OFF) used for interchange and archiving of three dimensional objects [33].

In many three dimensional computer graphics systems, rotation matrices are used to control the orientation of objects. Matrix multiplication is used to combine different rotations. An alternative to the rotation matrix is the quaternion [31]. A quaternion may be represented by a four component vector, encoding a vector and an angle of rotation about that vector.

Quaternions are equivalent to rotation matrices when representing the orientation of a three dimensional object, but can be composed more efficiently since rotation matrices have redundant entries. The single rotation angle about a vector has been found to be more intuitive to use than the Euler angles used in a rotation matrix.

Quaternions were thus selected to represent orientation. They are easily converted to rotation matrices when transforming points.

3.2. Implementation of the graphics pipeline

A number of techniques exist for implementing the various components of the graphics pipeline. This section compares some of the more appropriate.

3.2.1. Rendering

This section deals with the conversion of the polygon representation from an abstract collection of coordinates to an area of colour on the screen.

3.2.1.1. Line drawing

One of the simpler methods of rendering three dimensional objects is by generating the outlines, producing a ’wireframe’ image. Efficient line drawing is also required when drawing solid areas. There are a number of accepted methods for generating lines, the most common is Bresenham’s line algorithm. The following sections present this algorithm and some alternatives that were found to be superior.

3.2.1.1.1. Bresenham's algorithm:

This routine uses a decision function to select which one of two adjacent points is to be plotted next [29]. The decision function uses only shift, addition and subtraction operations on integer values. It can thus be implemented efficiently. Pseudo-code for this routine implemented for lines with positive slope less than one, is shown in Figure 11.

```
For a line: $(x, y)$ to $(x + \Delta x, y + \Delta y)$ where
$0 \le \Delta y \le \Delta x$,
$0 \le \Delta y$, $0 \le \Delta x$

    $f = 2 * (\Delta y - \Delta x)$
    $h = 2 * \Delta y$
    $e = 2 * \Delta y - \Delta x$
    for i := 0 to $\Delta x$ do
        plot (x, y)
        if e > 0 then
            y := y + 1
            e := e + f
        else
            e := e + h
            x := x + 1
        end if
    end for
```

Figure 11 Bresenham's line-drawing algorithm

The Bresenham algorithm has several problems. It works pixel by pixel, even when the 'jaggy' nature of the representation of the line on a raster display permits the use of a single operation to fill a number of adjacent pixels. Secondly it requires a comparison at each pixel.

3.2.1.1.2. Segmented line drawing algorithm:

A great problem when drawing lines is the fact that the slope is usually some non-integral value. Thus either floating point arithmetic must be used to represent it, or some integral approximation is required. The latter case is the one often chosen since floating point operations are normally slow.

Display devices are normally discrete, displaying points as quantised pixels. Lines drawn on such displays appear jagged, constructed of a number of smaller horizontal or vertical segments. If it were possible to calculate the size of these segments, a single block fill instruction could colour many pixels at once.

The length of a single horizontal 'jag' can be found by calculating the change in the horizontal (x) coordinate corresponding to a unit change in the vertical (y) coordinate. This change in x is typically a non-integer value, namely the quotient dx / dy. Since the display device is discrete, only the integer part is of interest. The fractional part may not be discarded since this will result in rounding errors. The fractional error term must be accumulated and used to make corrections when necessary.

The error term may be stored in integer form by keeping it as the numerator of a fraction with dy as denominator. Thus the change in x is:

    dx DIV dy   (DIV = integer division)

and error term is:

    dx MOD dy.

The value of the change in x can be used to fill in several pixels simultaneously, speeding up drawing if a block move instruction is used. The error term can be kept as a measure of the fraction of the line that was lost due to rounding, and incorporated into the calculation of the next segment of the line.

A procedure implementing this technique is shown in Figure 12. The error term is initialised to give an extra half pixel initially to provide a balanced appearance to the lines. The FillHorizontalSegment (x, y, l) function is assumed to draw a horizontal line of length l from the point (x, y).

When implementations on the transputers are compared, the speed of the Bresenham algorithm does not vary much from that of the segmented routine. This is due to the MOD and DIV instructions that are slower (39 clock cycles) than addition (1 clock cycle). Another factor is the block move that takes a relatively long time to start, although it transfers data rapidly once underway. The block move operation is thus only suitable for cases with long segments, namely lines with a small gradient.

```
For a line: (x, y) to (x +  Δx, y + Δy)  where
0 ≤ Δy ≤ Δx,
0 ≤ Δy, 0 ≤ Δx

    error := Δx / 2
    i := 0
    end := x + Δx

    while i < Δy do
        length := (error DIV Δy) + 1
        error := (error MOD Δy) - Δy
        error := error + Δx
        FillHorizontalSegment (x, y + i, length)
        x := x + length;
        i := i + 1;
    end while
    FillHorizontalSegment (x, y + Δy, end - x)
```

Figure 12 Segmented line-drawing algorithm

A more efficient version of the segmented line-drawing routine uses subtraction to simulate the MOD and DIV operations. It also only invokes the block move once the gradient is below a certain value. This routine may be found in Figure 13. This routine uses the block move if the slope of the line is less than 1/10. This value was found to maximise the effect of the tradeoff.

3.2.1.1.3. DDA algorithm:

The DDA (Digital Differential Analyzer) algorithm plots pixels by making unit increments along one axis, while incrementing the other axis by the slope [18]. The problem with this routine is that it makes use of floating point values to represent the slope.

It is possible to use the faster integer arithmetic on these floating point values as shown in [4]. The floating point number is stored in two integers, the integral part of the number is stored as an integer and the fractional part is multiplied by 2^{32} and stored as an integer.

The algorithm is shown in Figure 14. In this case since the slope < 1, only the fractional part of the slope is necessary. The y coordinate is represented by the variable y holding the integral part of this coordinate and yfraction holding the fractional part in the manner described above. The

```
For a line: (x, y) to (x + Δx, y + Δy)  where
0 ≤ Δy ≤ Δx,
0 ≤ Δy, 0 ≤ Δx

    error := Δx / 2
    i := 0
    end := x + Δx

    if Δy * 10 > Δx then
        for i := 0 to Δy do
            while error > 0 do
                error := error - dy
                Plot (x, y + i)
                x := x + 1
            end while
            error := error + dx
        end for
    else
        for i := 0 to Δy do
            length := 0;
            while error > 0 do
                length := length + 1
                error := error - dy;
            end while
            error := error + dx;
            FillHorizontalSegment (x, y + i, length)
            x := x + length
        end for
    end if
    FillHorizontalSegment (x, y + Δy, end - x)
```

Figure 13 Modified segmented line-drawing algorithm

fractional parts of y and slope are added and placed in the fractional part of y. The carry from this addition is then added to y.

3.2.1.1.4. Comparison of line drawing techniques

A comparison of the various line drawing routines is shown in Table I. The implementation of Bresenham's algorithm, the two segmented routines and the DDA routine all use approximately

```
For a line: (x, y) to (x + Δx, y + Δy)  where
0 ≤ Δy ≤ Δx,
0 ≤ Δy, 0 ≤ Δx

    slopefraction := dy / dx * $2^{32}$
    yfraction := 0

    for x := x to x + Δx do
        Plot (x, y)
        yfraction := yfraction + slopefraction
            (using 32 bit integer addition)
        y := y + carry from above addition
    end for
```
Figure 14 DDA algorithm

Table I Lines of varying length drawn per second by each routine

Line Length /[pixels]	Bresenham	Segmented	Optimised Segmented	DDA	Optimised DDA
11	15939	15789	17396	19297	21464
114	1748	1816	1964	2436	2866
229	879	916	987	1236	1460
344	587	612	659	828	980
458	441	460	495	623	737
573	353	368	396	499	591

the same level of optimization. Speed differences in these cases can be attributed to the nature of the algorithm. The optimised DDA routine consists of hand optimised transputer assembler code, and gives an idea of the capabilities of the graphics system itself.

Many graphics systems are rated on the number of vectors drawn per second, but not many give the length of the vectors tested. The table shows the speed of line drawing together with the average length of line drawn. The lines tested had slopes varying between 0 and 1. The overhead involved in taking the measurements has been subtracted from the timing values.

Table II Characteristics of Line drawing routines

	Bresenham	Segmented	Optimised Segmented	DDA	Optimised DDA
Clock Cycles per pixel	98	94	87	69	58
Clock Cycles for setup	145	205	144	253	271

The correspondence between the time taken to draw the line and the line length matches a linear function very closely. The coefficients for the lines tested are shown in Table II, given in 20 MHz transputer clock cycles. It can be seen that there is a trade off between the time taken to setup the values required, and the time taken to plot each pixel. For example the shift operations required when starting the DDA take longer than the simple arithmetic involved in Bresenham's algorithm, even though less time is required to plot each pixel. Comparing the two extremes, the optimised segmented routine and the optimised DDA shows that the former is superior for lines of less than 4 pixels length, while the latter is faster in the other cases.

3.2.1.2. Polygon rendering

Various algorithms exist for drawing polygons. They generally fall into two classes [18]. This first class draws the outline of the polygon and starts colouring from a point on the inside, checking that it stays inside the boundary while doing so. The other class calculates the boundaries of the polygon and colours the interior region at the same time.

The first class of fill algorithm is not suitable for fast polygon rendering due to the complexity of pixel plotting. For each pixel plotted, checks must be made on the adjacent pixels to determine whether they are on the boundary of the area being filled.

The second class is more suitable since it can be used to fill blocks of pixels at a time, rather than by working on an individual pixel basis. Since the polygons to be rendered are all convex, the fill can be performed by traversing the polygon from top to bottom (in screen coordinates), calculating the left and right boundaries simultaneously and filling in the horizontal line of pixels between them. The boundary points can be calculated using one of the line drawing algorithms described above.

The following discussion is limited to the consideration of a triangle fill routine. A general convex polygon can easily be decomposed into triangles, and any routines described here can be extended to polygons with more than three sides without much trouble.

3.2.1.2.1. Triangle Fill Routine

A general outline for the triangle fill routine is shown in Figure 15. This algorithm is just a formal version of the idea described earlier. The DDA line drawing algorithm can be used to calculate the points on the boundary. A block move can be used to draw the horizontal line segments.

3.2.1.2.2. Optimisations to the Triangle Fill Routine

There are a number of ways to make the fill routine faster. While a description of coding techniques is not often considered relevant to the description of the algorithms, in this case an exception is made. The optimisations to improve rendering speeds are often mentioned, but are rarely described in detail in computer graphics literature. Some techniques described here are specifically for the transputer, but should also have a more general application.

Use of macros instead of procedure calls is a useful technique. Apart from avoiding the time taken in the call itself, the overhead of passing parameters is removed. With the aid of a suitable macro pre-processor, the appearance of the code is not greatly altered, and the program retains a structured appearance.

Relating specifically to the transputer, the horizontal fill can be optimised by use of the block move. The block move on the transputer is very powerful, being able to copy two-dimensional blocks of data. The instruction requires six variables before starting which is a high overhead when just copying linear arrays. A simpler copy can be used, which is faster but limits some of the extensions to the fill routine described in section 3.2.1.2.4.

Referring to Figure 15, it may be seen that the expression x2 - x1 occurs a number of times, but

```
For a triangle with vertices: (xa, ya), (xb, yb) and (xc, yc)

    Sort and relabel the vertices so that ya ≤ yb ≤ yc

    if xb < xc then

        for y := ya to yb do
            x1 := point such that (x1, y) is on the line segment joining (xa, ya) and
            (xb, yb)
            x2 := point such that (x2, y) is on the line segment joining (xa, ya) and
            (xc, yc)
            FillHorizontalSegment (x1, y, x2 - x1)
        end for

        for y := yb to yc do
            x1 := point such that (x1, y) is on the line segment joining (xb, yb) and
            (xc, yc)
            x2 := point such that (x2, y) is on the line segment joining (xa, ya) and
            (xc, yc)
            FillHorizontalSegment (x1, y, x2 - x1)
        end for

    else
        for y := ya to yb do
            x1 := point such that (x1, y) is on the line segment joining (xa, ya) and
            (xc, yc)
            x2 := point such that (x2, y) is on the line segment joining (xa, ya) and
            (xb, ybc)
            FillHorizontalSegment (x1, y, x2 - x1)
        end for

        for y := yb to yc do
            x1 := point such that (x1, y) is on the line segment joining (xa, ya) and
            (xc, yc)
            x2 := point such that (x2, y) is on the line segment joining (xb, yb) and
            (xc, yc)
            FillHorizontalSegment (x1, y, x2 - x1)
        end for

    end if
```

Figure 15 Triangle fill algorithm

x2 never occurs alone. The difference x2 - x1 is the difference between two linear expressions and is thus itself linear. This means that the line drawing algorithm could equally well be used to calculate the difference term instead of the second boundary. This removes the need to perform the subtraction repeatedly.

The use of an efficient line drawing routine to calculate the boundaries has been mentioned. For cases where large numbers of very small polygons are going to be drawn, increased speed can be obtained by using a routine with a smaller setup time.

The display device is memory mapped into an area of memory. The address corresponding to a pixel at (x, y) can be found with the expression "StartOfDisplayMemory + y * DisplayWidth + x". Rather than repeat this calculation for every new pixel, the offset from the previous point is added to the previously calculated address. Only addition is used which is generally faster than the multiplication.

3.2.1.2.3. Performance of Fill Routine

Performance figures obtained for the triangle fill routine are shown in Table III. Measurements are averaged over a number of triangles of different shapes. The boundary figure is the sum of the vertical component of the boundaries.

Table III Performance of Triangle Fill Routine

Boundary [Pixels]	Area [Pixels]	Speed [Triangles / second]
16.65	16.5	7617
174	1825	1158
348	7303	491
523	16433	278
709	30193	176
884	46876	124

The time taken to fill a triangle is dependent both on the perimeter of the triangle, and the area.

The former affects the amount of boundary calculation and the latter determines the number of pixels to be plotted. Since the line drawing algorithm moves one step at a time in the vertical direction, the time can be expected to have a linear dependence on the vertical component of the perimeter. The area determines the number of pixels plotted and can also be expected to contribute to the time in a linear fashion.

$$Time = c + b \bullet Area + a \bullet VPerimeter \tag{1}$$

Fitting the data to (1) gives the following values for the coefficients:

$a = 3.41 \times 10^{-6}$ second/pixel
$b = 1.06 \times 10^{-7}$ second/pixel
$c = 7.62 \times 10^{-5}$ second

The constant **c** gives a measure of the time to set up the variables required by the routine, corresponding to 1524 cycles of a 20 MHz clock. This sets an upper limit for the number of triangles that may be drawn in one second, about 13100. The coefficient **b** gives an indication of the time taken to plot one pixel, just over 2 clock cycles. The block copy operation is specified as taking 2 clock cycles per byte with a small extra initialisation overhead. The coefficient **a** gives the time taken to calculate one boundary coordinate, about 68 clock cycles. This is slightly higher than the value given in Table II, since the version for the fill routine is generalised to lines of any slope, and could not be optimised to the same level.

3.2.1.2.4. Extensions to the Fill Routine

When rendering red/green stereo images on a single monitor, a horizontal interlacing system can easily be implemented. Every even numbered pixel is allocated to the left eye and every odd numbered pixel is allocated to the right eye. This is implemented very simply by using a different step size in the block move instruction that draws horizontal line segments. For stereo viewing a step size of two is used, allowing every second pixel to be skipped.

Polygon rendering is dependent on the hidden surface removal technique used, as will be seen later. Some hidden surface removal routines require depth comparisons to be made for every pixel, requiring rendering to be handled on a point by point basis. In others, comparisons only need to be made once per polygon allowing block moves to be used.

3.2.2. Hidden Surface Removal

The hidden surface removal problem has no optimum solution in computer graphics as yet, although a number of commonly used algorithms do exist. Every application must choose a technique suitable to the problem. Some routines work reliably but slowly, others work faster but fail in special cases.

A taxonomy of a number of different hidden surface removal algorithms was developed in [38]. A brief description of the various categories follows in the next section.

In the case of a virtual reality where speed is important, four algorithms were considered worthy of a trial implementation. The four are discussed below together with their relative merits and demerits. The term 'depth' will be used to refer to component of the displacement of a polygon from the viewer in the direction that the viewer is looking.

3.2.2.1. Categories of hidden surface algorithms

The categorization of different hidden surface removal techniques can be based on various criteria.

Finding out which surfaces are hidden involves sorting the points in the object in order to determine those that are in front. Since the objects occur in three dimensional space, each algorithm can be classified according to the order in which each coordinate axis is searched. Generally the X and Y axes are taken as horizontal and vertical respectively and the Z axis is used to represent depth. Examples of the orders used in various routines are ZYX and YXZ.

The algorithms also can be classified according to whether they operate in image space or object space. Object space routines perform to arbitrary precision, usually that of the data type representing the object. Image space routines work at a resolution determined by the display device. A ray tracing procedure is an example of an object space routine. Some image space routines are described in the next section.

Finally algorithms can be categorised according to the nature of the algorithm. The remainder of this section describes some of the different methods available at the time of [38].

The list-priority category of algorithm involves assigning a priority to each face. The relative priorities of two faces are based on the way in which the two faces obscure each other. Much of the priority information is independent of the angle from which the object is viewed, and can be calculated beforehand. Separation of the scene into disjoint clusters simplifies the priority calculation. Dynamic list-priority algorithms assign priorities to the faces based on depth information. Corrections may be made for overlapping faces.

The depth-priority algorithms consist of two categories, area-sampling and point-sampling. The area-sampling algorithm attempts to divide the screen into areas that can be given the same colour. Areas containing two or more visible faces are divided until this condition is satisfied. The point-sampling algorithms decompose the polygonal faces into a set of horizontal scan-lines. As the screen is traversed from top to bottom, only the relevant scan-lines for each level are considered. Depth calculations need only be made where a scan-line starts or stops. These routines make use of coherence, the tendency for adjacent pixels to have the same value. The area-sampling works on area coherence, the point-sampling on the coherence properties of the segments.

3.2.2.2. Implemented Hidden Surface Removal Algorithms

Four hidden surface removal routines were chosen for implementation. The Z buffer and Scan-line routines were chosen because of their popularity in computer graphics literature. The Painter’s algorithm and the BSP tree technique were chosen due to their apparent suitability for high speed rendering.

3.2.2.2.1. Z buffer hidden surface removal

The Z buffer algorithm is an image space hidden surface removal technique. A pseudocode description of the algorithm is given in Figure 16.

Each pixel is associated with the depth value of the last point plotted there. The pixel may only be overwritten if the new point being plotted is closer to the viewpoint than the previous one. Plotting the pixel updates both the pixel colour and its associated depth. The depth information is normally stored in a two dimensional array, the Z buffer, whose size corresponds to the limits of the display device.

```
zbuffer [x, y] := -∞,  ∀(x, y) on the display

for each polygon in the scene
    for each point in the polygon
        let (x, y) be the position of the point on the display
        let z be the depth of the point (x,y)
        if z > zbuffer[x, y] then
            Plot (x, y)
            zbuffer[x,y] := z
        end if
    end for
end for
```

Figure 16 Z buffer algorithm

With the Z buffer method, the polygon rendering and hidden surface removal are closely linked. Rasterisation of each polygon occurs first and the Z buffer is then consulted for every pixel to be drawn.

The disadvantages with this method are the large amount of memory required to store the matrix, and the large amount of computation required per pixel. A depth computation and a comparison need be performed for every pixel in every polygon.

A further disadvantage is the inability to use a single instruction to fill a horizontal line segment, as each pixel requires individual consideration. Thus the full speed of the fill routine cannot be obtained.

Some improvement in speed can be obtained by using one of the line drawing algorithms for the depth calculations. Across a scan-line, depth can be linearly interpolated between the values at the two boundaries. The depth along the boundaries may be similarly calculated from that of the vertices. A further optimisation can be made by just calculating a single depth value for an entire horizontal line segment. This last optimisation will cause the routine to fail for polygons that penetrate others, however.

Clearing the Z buffer can also be a time consuming task. Instead a floating offset can be used for cases where the total depth of the scene is some finite value much less than the range of the type to represent depth. This offset holds the maximum value put in the buffer in the previous frame and is added to the depth value computed for each pixel. This ensures that the pixels in the current scene are always in front of, and thus unaffected by, those from the previous frame. When the limit is reached, the buffer can be cleared and the offset reset.

3.2.2.2.2. Scan-line hidden surface removal

This routine is a YXZ image space, point-sampling, hidden surface removal algorithm. A pseudocode definition is given in Figure 17.

```
Sort all polygons on their least y coordinate.

for y := ScreenTop to ScreenBottom do

    Sort all horizontal segments of polygons on scan-line y on their least x
    coordinate.

    for x := ScreenLeft to ScreenRight do

        Sort all pixels at horizontal position x on the relevant scan-lines on their
        depth.

        Plot the pixel that is closest to the viewer.

    end for

end for
```

Figure 17 Scan-line algorithm

The scan-line routine traverses the display from top to bottom. It uses an active edge table to keep track of the polygons present on the current scan-line. By sorting the polygons initially, the new polygons that are coming into view on each scan-line can be found easily. The sorting is done by a bucket sort allowing simple lookup of the polygons starting on a particular scan-line.

A sorted list of the horizontal segments visible on a particular scan-line is maintained. While traversing the scan-line from left to right, a list of those horizontal segments that overlap the current point is kept. Depth comparisons can then be limited to those polygons that overlap. If the assumption is made that no polygons intersect each other, then depth computations are only required when entering a new polygon or leaving an obscuring one.

The polygon rendering is again embedded in the hidden surface removal routine. The active edge table stores the end positions of the horizontal line segments and computes the pixel positions. Plotting the point can occur once the depth comparison has been completed.

The disadvantage with this technique is the relatively (to the other hidden surface removal techniques) complex data structure that it uses. Maintaining these structures entails a certain overhead. This method also performs a large number of comparisons per polygon, both in determining depth and in maintaining the edge list in ascending order. The routine is more complicated than the Z buffer routine due to the need to handle all the polygons concurrently. This results in the need to perform an operation, similar to context switching, between the polygons, saving the state of the rendering process for one and starting with another. This contributes to an extra overhead.

3.2.2.2.3. Painter's algorithm

The painter's algorithm is a ZYX image space dynamic list priority algorithm. A brief formal description is given in Figure 18.

```
Sort all polygonal faces on their depth value
Draw polygons from back to front
```

Figure 18 Painter's algorithm

This routine involves sorting the polygons according to their depth and then rendering from back to front. Any polygon that is in behind another will be overwritten, appearing as if it were obscured by that polygon.

The hidden surface routine and the rendering can be completely separated. The hidden surface routine first performs the sort, working only with polygon data. This is followed by the polygon renderer that can operate as described in a previous section.

This technique has the disadvantage of sometimes failing to operate correctly when two polygons intersect or overlap in both depth and area. The sorting stage is also time consuming, although a significant speed improvement is noted when using the previous sorted list as a starting point for the next sort. In one test, this technique reduced the sorting time by a factor of 8.

The advantage of this method is the speed improvement resulting from the reduced amount of computation, since the routine works at a polygon level as opposed to the pixel level of the previous routines.

3.2.2.2.4. Binary Space Partitioning (BSP) trees

The BSP tree algorithm is a development of list-priority algorithms. Actual priority values as such are not calculated, instead the manner in which one polygon obscures another is encoded in a binary tree structure. The tree is generated by a pre-processing stage. When rendering the object, all that is required is a traversal of the tree.

```
procedure maketree (pl : list of polygons) : tree

FrontList := ∅
BackList := ∅

Let k be the index of a polygon in pl

for i := 1 to length of pl do
    if i <> k then
        Split polygon i on polygon k to get FrontParts and BackParts
        Add FrontParts to FrontList
        Add BackParts to BackList
    end if
end for

return a binary tree with polygon k as root,
    maketree (FrontParts) as left subtree and
    maketree (BackParts) as right subtree
```

Figure 19 BSP tree generation algorithm

An algorithm for the tree generation phase in given in Figure 19 [15]. Given a set of polygons, $\{p_1, p_2, ... p_n\}$, choose one p_k. The plane in which p_k lies partitions space into two half spaces.

A front side for p_k can be found by calculating the normal by some consistent method. The other polygons in the set then either fall into the front half space for p_k, the back half space or both. By splitting those that lie in both half spaces, two sets of polygons can be obtained, one that lies totally in front of p_k, and one that lies totally behind. From any viewpoint in front of p_k, the polygons in front will obscure it, and p_k will obscure the polygons behind it. For a viewpoint behind p_k the reverse will apply. The binary space partitioning tree can be constructed with p_k as the root, and the trees produced from the front and back sets as subtrees.

```
procedure drawtree (eye : viewpoint, tree : BSP tree)

if tree is not empty then
    if eye is in front of root polygon of tree then
        drawtree (eye, backbranch (tree))
        DisplayPolygon (root polygon of tree)
        drawtree (eye, frontbranch (tree))
    else
        drawtree (eye, frontbranch (tree))
        DisplayPolygon (root polygon of tree)
        drawtree (eye, backbranch (tree))

    end if
end if
```

Figure 20 BSP rendering algorithm

The routine for generating an image using a BSP tree is shown in Figure 20. The BSP tree method is similar to the Painter's algorithm in that it works by drawing the polygons in a particular order, and painting over those that are obscured. As with the Painter's algorithm, the hidden surface removal and rendering stages are separate.

The disadvantage of this method is the sometimes large numbers of sub-polygons produced during the clipping process. A further problem is that the tree structure cannot easily be modified and is thus not suitable for situations where objects may move relative to one another.

The advantages, however, are that the tree traversal can be performed rapidly and that the algorithm successfully renders scenes where the Painter's algorithm would fail.

Implementing these algorithms on transputers requires some special considerations. Both algorithms are recursive, which poses a problem for the transputer architecture where no stack overflow protection exists. The recursion may be removed by explicitly simulating a stack, in which case overflow protection may be included.

3.2.2.3. Comparison of the different routines

Timing values allowing comparison of the four different hidden surface removal routines are shown in Table IV. These times were taken for one particular test scene. Differences in the relative values occur for different scenes, these can be explained by the tendency of each routine to perform better in different situations. For example, the scan-line algorithm would perform better for an image with few overlapping polygons, whereas the Z Buffer routine would be unaffected by overlap.

Table IV Timing values for the different hidden surface removal routines

	Z Buffer	Scan-line	Painter’s	BSP Tree
Time / [s]	1691	7836	225	229

The interdependency of the hidden surface removal algorithm and the polygon fill in some of the cases makes it difficult to isolate the effects of each different hidden surface removal approach. An effect that does remain consistent across all but the most exceptional scenarios, is the relative ordering of the routines.

Generally the Painter’s algorithm produces the fastest times. It consists of little more than a loop to draw each polygon. It does have a disadvantage in that it occasionally produces an incorrect image. These errors can occur when polygons intersect or overlap. The Painter’s algorithm is unable to correctly render cyclic overlapping, as is shown in Figure 21.

The BSP tree routine comes a close second, also doing little more than just filling each polygon. The added overhead of performing the tree traversal, maintaining a stack for backtracking and calculating the orientation of each polygon makes this technique slightly slower than the Painter’s algorithm. The orientation calculation can form part of the back face removal, however (see section 3.2.2.5). This routine does have a disadvantage with regard to complex objects. To

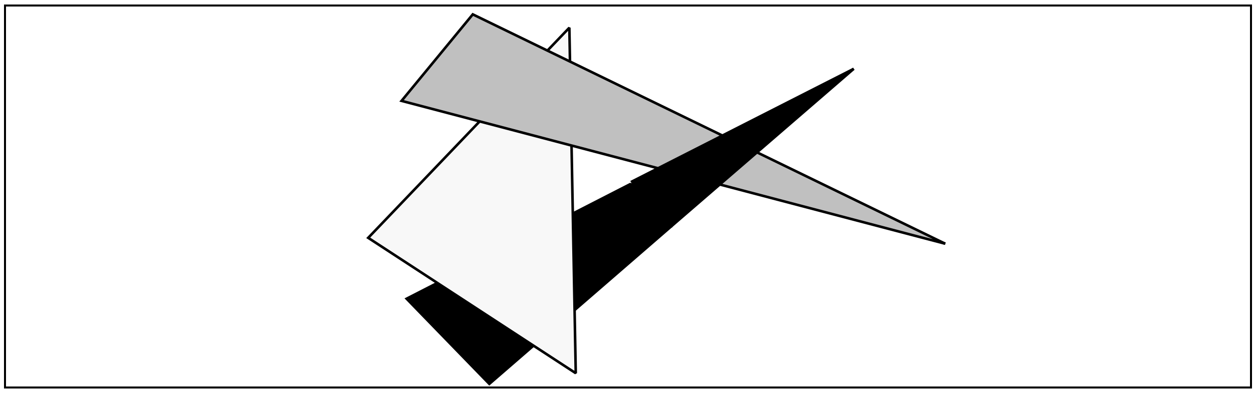


Figure 21 Cyclic overlapping of polygons

generate the tree, many polygons have to be split, further increasing the complexity of the object.

The Z buffer routine is the next most successful, although still far slower than the two techniques just mentioned. The need for a point by point depth calculation and comparison is the major factor in slowing this routine.

The scan-line routine is the slowest of the four. While the boundary calculations were performed using floating point operations for convenience in the implementation of this algorithm, this could slow down the routine by a maximum factor of 2. Although the data structures used in this routine improve the efficiency of the routine, the cost of maintaining them is high. Dynamic memory management has its cost, as does the sorting steps required when merging segment lists. Coherence does not provide much assistance in scenes with small polygons, and the need for many depth calculations per polygon is also expensive.

3.2.2.4. Using the hidden surface routine

Given the above results, the two hidden surface removal techniques that are best suited for a virtual reality application are the Painter’s algorithm and the BSP Tree method. The Painter’s algorithm tends to fail on occasion, destroying the illusion of reality. For this reason, the BSP Tree method was chosen as the hidden surface removal technique to be used.

BSP Trees have the disadvantage of not being able to adapt easily to changes in the scene being rendered. This problem can be partially overcome by adding an intermediate level between scene and polygon. The polygons can be clustered according to the object to which they belong. The shapes of the objects are not going to change, only the positions of the objects relative to each other. The Painter’s algorithm can be used at the object level to ensure that objects obscure each other correctly. The BSP tree routine can be used within each object to ensure that the polygons obscure each other correctly. There are still cases in which the Painter’s algorithm will produce the incorrect output, but these are decreased by disallowing object intersection.

The idea of sorting at an object level first is not unique. Hidden surface removal using a variation on the scan-line method to sort the objects reports a significant decrease in the number of comparisons required [12]. This method orders the bounding boxes of the objects by y values. The objects associated with a particular scan-line are kept in an active object list, sorted by x value. This allows the objects that may be visible on a particular portion of the scan-line to be easily found. The data structure describing the object keeps track of which polygons adjoin which others, allowing adjoining coherence to be used. When the edge of one polygon is reached, the next polygon will be one of the neighbours.

The other hidden surface removal routines could also be extended to work at an object level. The BSP routines, being the most successful at a polygon level, could be extended into a spatial partitioning system at an object level. However, since objects are not stationary, the partitioning would have to be revised periodically. For this method to be viable, an efficient partitioning algorithm would have to be devised.

The complete system thus has two levels of hidden surface removal, one at object level and the other at polygon level for each object.

3.2.2.5. Additional ways to simplify hidden surface removal

While the hidden surface removal techniques are sufficient for most applications, there are ways to decrease the amount of work needing to be done if the scene can be constrained in some way. A classical example is the flight simulator where the landscape is always behind everything else, and so may always be drawn first and painted over.

A similar situation occurs in other scenarios when all objects are constrained to be inside some object. In these cases, the outer object can be specified as a boundary object and always drawn first. In addition, if there are several boundary objects that do not overlap, or that always overlap in a particular sequence, these can always be drawn first in some fixed order.

An easy way to remove up to 50% of the polygons requiring consideration during hidden surface removal, is to remove those polygons at the back of the object [19]. For opaque solid objects, these back faces are not visible. By prearranging that the surface normal to each polygon in the object points outward, those polygons that are not facing the viewer can easily be distinguished. It does have the disadvantage of making the object invisible when the viewpoint is inside the object. For this reason, it is used selectively in this system.

3.2.2.6. BSP tree generation

The algorithm for generating a BSP tree is given in section 3.2.2.2.4. Mathematically it is quite correct, but problems arise when attempting to implement it.

The limited range and accuracy of the number representation on a computer makes it difficult to split polygons accurately with an arbitrary plane. Polygons containing 4 or more points may no longer be perfectly flat. This requires either complicated mathematics to find the intersection, or triangulation of the polygon. Calculation of the intersection points of plane and polygon is difficult, particularly when these points are close to vertices. There, a small rounding error can place a point that is inside a polygon on the outside and vice versa. The polygons resulting from the split may have slightly different orientations than the original, which may cause them to intersect other polygons which would previously have been untouched.

One approach to solving this problem is to take this error into account [28]. Instead of testing if a number is greater or equal to zero, it is tested to see whether it is greater than $-\varepsilon$, or in the range $[-\varepsilon, \varepsilon]$ for some suitable ε. This does make the algorithm more complex as previously impossible situations must now be detected and corrected. Our approach was to make use of an infinite precision rational type for all calculations. This sacrifices speed against reliability, and simplifies the logic.

A similar limitation arises from the data files that are used to store the objects. Firstly, only finite precision is used in the storage, so accuracy is lost, and polygons that were once parallel may now be at some other angle to each other. In some cases when many sided polygons were triangulated, the planes containing the triangles are no longer parallel. This may result in more of the triangles being split by others during BSP tree generation. This problem is worsened by some object generation packages which do not properly produce closed objects. Adjacent polygons not only do not share vertices, the vertices which should be common actually have slightly different positions. This results in thin fragments being sliced off polygons during tree generation.

A more serious problem with respect to the generation of BSP trees is the large numbers of polygons that may be split, increasing the complexity of the object. Since the relationship: polygon A splits polygon B is not always reflexive, rearranging the order in which the polygons are placed in the tree can affect the number of splits which occur. In an early paper [15], the root polygons were selected so that the number of splits of those below was reduced. A conflict reduction strategy was also employed which selected a root polygon so that possible splits are now placed on opposite sites of the root. Later work used heuristics to reduce splits and also to produce a balanced tree [40].

Table V Polygon counts when generating BSP trees

Object	Original	Naive	Conflict reduc.	Split reduc.	Conflict = Split	Conflict ≪ Split
Dodecahedron	36	52	53	36	50	36
Epcot	96	249	199	128	130	127
Glass	138	378	268	299	253	298
Hang-glider	117	652	283	251	284	243
Lamp	145	363	321	299	308	294
Pawn	304	1207	766	608	689	609

Our work used heuristics dependent on both split reduction and conflict reduction to produce minimal splitting. Used independently, both gauges reduced the number of splits over a naive ordering, as shown in Table V. The split reduction was generally the most effective. Combining these two, so that they both contributed equal weights produced an intermediate saving. The optimum results were obtained using split reduction with the conflict reduction used to resolve the cases when split reduction heuristic gave equal values for a number of polygons.

3.2.3. Perspective projections

3.2.3.1. Standard perspective projection

The transformation stage of the graphics pipeline (See section 3.2.5) arranges the objects and transforms the coordinate system so that the viewer is looking down the Z axis, with the X-Y plane at Z = 0 corresponding to the display screen. The viewer is situated a distance d behind this plane.

$$\begin{aligned} x_p &= \frac{xd}{z+d} \\ y_p &= \frac{yd}{z+d} \\ z_p &= 0 \end{aligned} \qquad \textbf{(2)}$$

The perspective transformation that projects the points on the objects onto the screen is well documented [18][19] and is shown in (2).

3.2.3.2. Stereo perspective projection

Producing a stereo image involves creating two separate images, one for the left eye, the other for the right eye, and combining the two. The two images may be displayed on two separate displays, for example in a head mounted display, or on one display using some method to separate the two images. The technique currently in use involves colouring the left image green and interleaving it with the right image which is coloured red. Red/green filter glasses are used to separate them.

$$\begin{aligned} &\textit{Left eye}: \\ x_p &= \frac{xd-ze}{d+z} \\ y_p &= \frac{yd}{d+z} \\ &\textit{Right eye}: \\ x_p &= \frac{xd+ze}{d+z} \\ y_p &= \frac{yd}{d+z} \end{aligned} \qquad \textbf{(3)}$$

The difference between the two images is created in the projection stage. The stereo perspective projection is an extension to the standard projection described above [20]. The difference is that instead of the viewpoint lying on the Z axis, the left eye viewpoint is taken as a distance e to the left of the Z axis and the right eye viewpoint as distance e to the right of the axis. The projection equations are shown in (3).

3.2.4. Clipping

Regions of the polygons that will not appear on the display device must be removed or clipped before an attempt is made to draw them. Plotting non-existent points, particularly with memory mapped devices, can have disastrous effects.

Clipping can be performed at a pixel level, checking that each point corresponds to a visible pixel on the screen before plotting it. As may be evident from discussion in previous sections, this would add a sizeable overhead to the fill routines. A more satisfying alternative is clip off the pieces of each polygon that do not fall into the visible area of the display.

While a number of general polygon clipping techniques exist [39], [25], [41], the special case of a convex polygon being clipped against a rectangular display window is not well documented. While this special case can be clipped by those routines, no advantage is made of the extra constraints.

The clipping algorithm used in our implementation is a development of the technique described by Sutherland and Hodgman in [39]. The algorithm traverses the polygon, considering each successive edge. Each edge is clipped against the boundary and the points inside and on the boundary are used to construct a new clipped polygon. The edge clipping routine makes use of the line clipping routine presented in [7], which combines a number of well known line clipping algorithms.

The order in which clipping and perspective projection is performed is important. Dividing through by the depth value results in a loss of information needed for clipping. This affects points in front of the display screen in particular. A possible solution to this problem is to clip before projecting. This requires clipping against the pyramidal shape that is the pre-projection view volume. An alternative is to clip away the points in front of the display first, project, and then clip against a rectangular view volume.

Clipping is done against pairs of boundary planes at a time, avoiding the complication of catering for turning points, as in [25]. Z clipping is performed first. This is then followed by perspective transformation, and clipping against two Y and two X boundaries.

The position occupied by the clipping stage in the graphics pipeline varies according to the hidden surface removal method used. For Scan-line and Z buffer it occurs before hidden surface removal and polygon fill. For BSP tree and Painter's algorithm, it occurs between hidden surface removal and polygon fill, as the unclipped polygon data is required by the hidden surface removal routines.

Clipping is also done at an object level. This corresponds to a simple rejection if the object is entirely non visible. Partially visible objects are passed on to the polygon level routines where more detailed clipping is done.

3.2.5. Transformations

The transformations performed involve positioning each object in the world, shading each object, and transforming the coordinate system so that the viewer is placed at a convenient point.

The initial transformations transform the object from the object space to the world space. Each object is scaled, translated to the position it occupies in the world, and rotated to face the correct direction.

The next step is to calculate the shading for each polygon. The direction of the normal vector for each polygon is compared with the direction to a light source, giving an indication of the brightness of the polygon. This value determines the colour with which the entire polygon is painted.

The whole world is then transformed to viewpoint coordinates as described in section 3.2.3.1.

The transformations are all performed using 3x3 matrices. The added generality of homogeneous coordinates was not considered necessary. The calculations are performed using floating point values. This was found to be about 5% slower than using fixed point, but the greater range of the values possible is a compensatory advantage.

3.2.6. Performance of the pipeline

The proportion of processor time that each of the components of the pipeline occupies is shown in Table V. These figures are given for a certain set of viewpoints into a simple world and are not always constant. The complexity of the environment can influence the relative times of each component. Where not specified, measurements were taken when using the BSP Tree method for hidden surface removal.

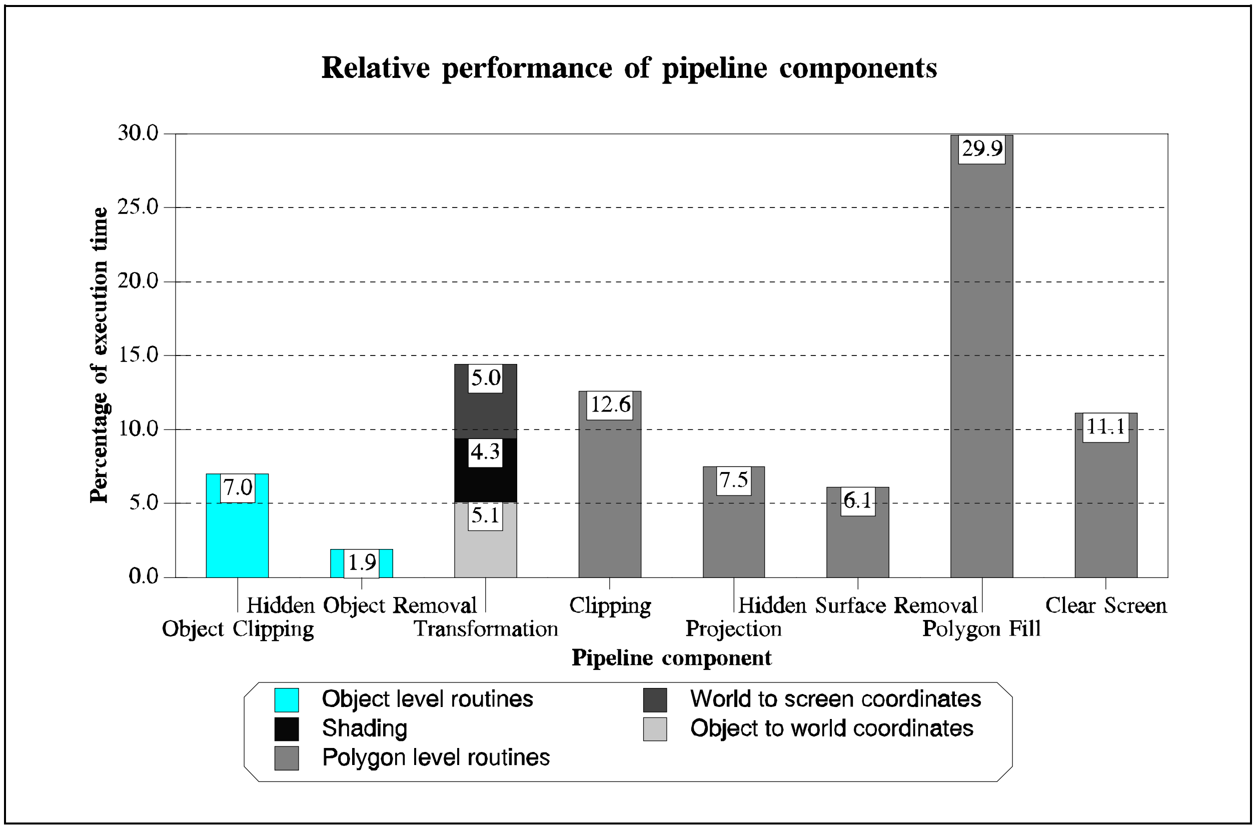


Figure 22 Performance of the components of the graphics pipeline

A point worth noting from these figures, is that the polygon fill time dominates that of the other components. Thus finding a faster fill method probably would have the greatest effect. Many other graphics systems make use of specialised hardware for this function.

The amount of the time spent clearing the screen between frames is also unfortunately significant.

Speeds for the rendering system are given in Table VI. The simple test world contains 30 objects amounting to about 250 polygons (each with 4 or more sides). The complex world consists of

Table VI Speed of the rendering system in frames/s

Frames rates:	Simple world	Complex world
512 x 512 resolution	4.17	0.87
320 x 200 resolution	5.67	0.88

a single object with over 1650 triangular polygons. The resolution of the screen determines the size of the window into the world, no attempt is made to scale the scene to the window size. There is not a lot more to be drawn for the single object at different resolutions, so rendering speed is not significantly affected.

3.3. Parallelisation of the graphics pipeline

All of the above implementation was carried out on a single processor. Further improvements in speed can possibly be obtained by making use of more processors. This section describes the results of adding parallelism to the graphics pipeline.

A number of different parallel decomposition strategies for polygon rendering are described in a paper by Whitman [43]. The next section describes the taxonomy. Four decomposition strategies were implemented, and are described in the following sections.

3.3.1. Parallel Decomposition techniques

The upper levels of the taxonomy produced by Whitman is shown in Figure 23, which classify parallel decomposition techniques into four main categories.

Level I.A. partitions the scene according to the polygons. The polygons are grouped according to some scheme, possibly the object to which they belong, or according to their position relative to some other polygon.

Level I.B. decomposes the scene amongst 3D spatial regions. This option is considered more appropriate to ray tracing applications than polygon rendering.

The image space pixel based routines, II.A., involve dividing the image amongst processors. In the ’worst’ case this involves placing the computation for each pixel on a separate processor. The more common approach is to place the computation for areas of the screen, such as horizontal

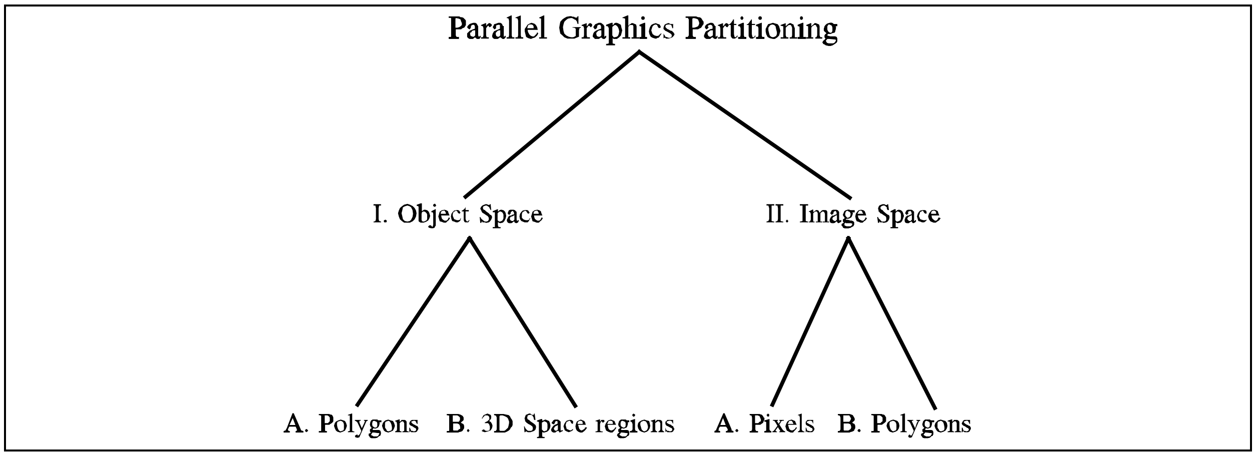


Figure 23 Parallel decomposition taxonomy

or vertical strips, on separate processors.

The polygon methods for image space, II.B., divides the scan-lines of the polygon amongst different processors.

Whitman considers the object space algorithms to be less efficient than the image space algorithms. The reason given was that object space routines considered could only parallelise the hidden-surface removal and scan-conversion was ignored.

3.3.2. Pipeline parallelism

The most obvious step when considering the diagram of the graphics pipeline is to place each component of the pipeline on a separate processor. The connections between components can be constructed using transputer links. There are two factors worthy of consideration when setting up a pipeline configuration for virtual reality on a transputer architecture.

Firstly, greater bandwidth is obtained from the transputer links when larger messages are transmitted. There is less overhead from the protocol used, and there is less delay waiting to each of the communicating processors to synchronise. The few bytes representing a point are not worth transmitting alone, it is better to wait until a number of points are ready to be transmitted and then send them all together. For convenience, all the data corresponding to an entire object is sent in one packet.

The second consideration is the latency problem, the delay between input and the corresponding output. Lots of data is held in long pipelines, even with a limited amount of data at each node. All of this must be processed before new data entering the node can have any effect on the image being displayed. Since there is a certain overhead associated with the communication between pipeline components, the delay between data entering the pipeline and it reaching the output increases with pipeline length, even though the rate at which images are produced is increased. This may be illustrated by example: Consider a pipeline in which each node shares equally in the workload, and the time taken for one processor to process a piece of data is 10 seconds (see Figure 24). The data is arriving as fast as possible, but takes 1 second to transmit across a link. For 5 nodes, each works on a datum for 2 seconds, and results are output every 2 seconds. The time between a piece of data being input and the corresponding output is at least 16 seconds. With 10 nodes, output occurs once every second but at least 21 seconds pass between input and corresponding output.

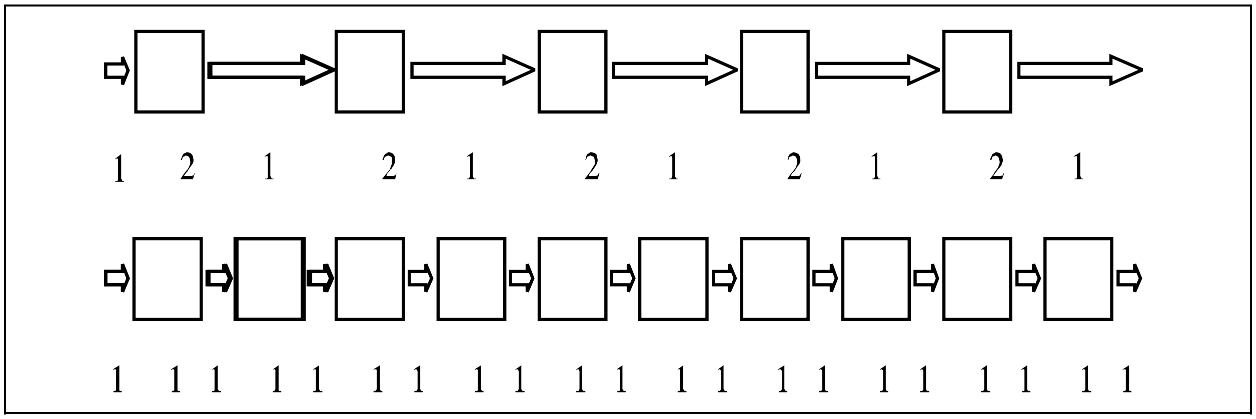
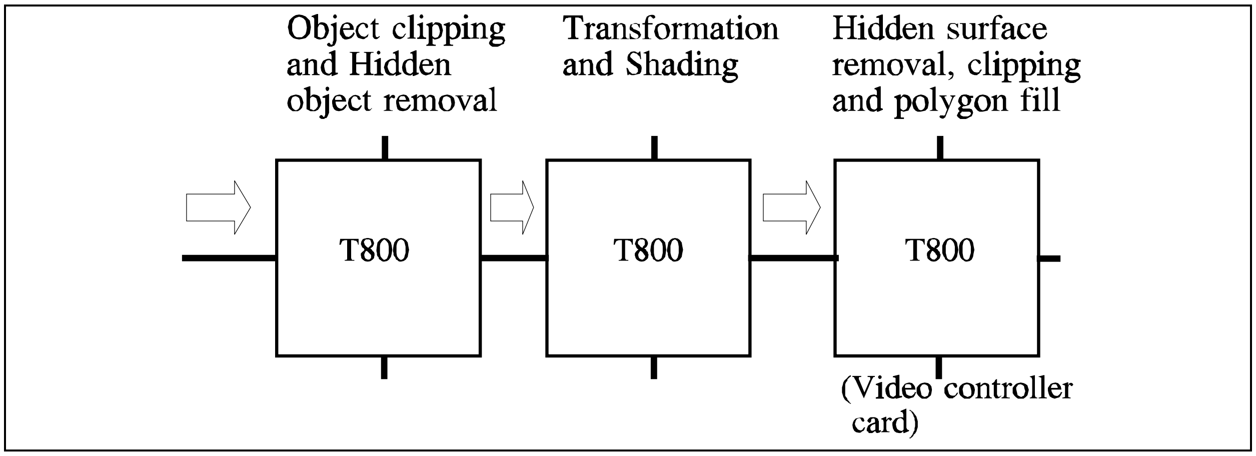


Figure 24 Illustration of latency in pipelines

In general for N processors, for work taking T seconds on one processor, capable of arbitrarily parallelism, and with communication time C between processors, the data output rate is T/N. The latency is at least (N+1)C + T seconds. The latency increases with number of processors for non zero communication time. For zero communication time it remains constant, independent of the data output rate. For this reason pipeline parallelism is better suited to systems that require computationally intensive rendering with limited user interaction.

A limited pipeline with three nodes was implemented. The processor layout is shown in Figure 25. Speedup figures are given in Table VII. The difference in speedups at different screen resolutions is due to change in the load balance. The renderer has to do more work at higher


Object clipping and Hidden object removal
Transformation and Shading
Hidden surface removal, clipping and polygon fill
T800
T800
T800
(Video controller card)
Figure 25 Processor layout for pipeline parallelism

resolutions.

Table VII Frame rates for pipeline parallelism

Frames rates:	Simple world	Complex world
512 x 512 resolution	6.99	1.09
320 x 200 resolution	10.00	1.11

3.3.3. Coarse grained parallelism

The coarse grained parallel decomposition strategy renders successive frames on separate processors. The transputer network is arranged to allow a number of independent pipelines to be created, as shown in Figure 26. Each pipeline renders one frame in memory which is then transferred directly to the graphics node. This technique promises the greatest speedup since it requires no load balancing. Each processor can start work on another frame as soon as the current one is finished. The load is also approximately equal from one frame to the next, so they are likely to stay in step. There is thus no waiting for other processors to catch up.

The delay between input and output is still the time taken for the pipeline to render the frame. No decrease in this time occurs even when the number of parallel pipelines is increased. The latency is thus constant and independent of the number of processors used.

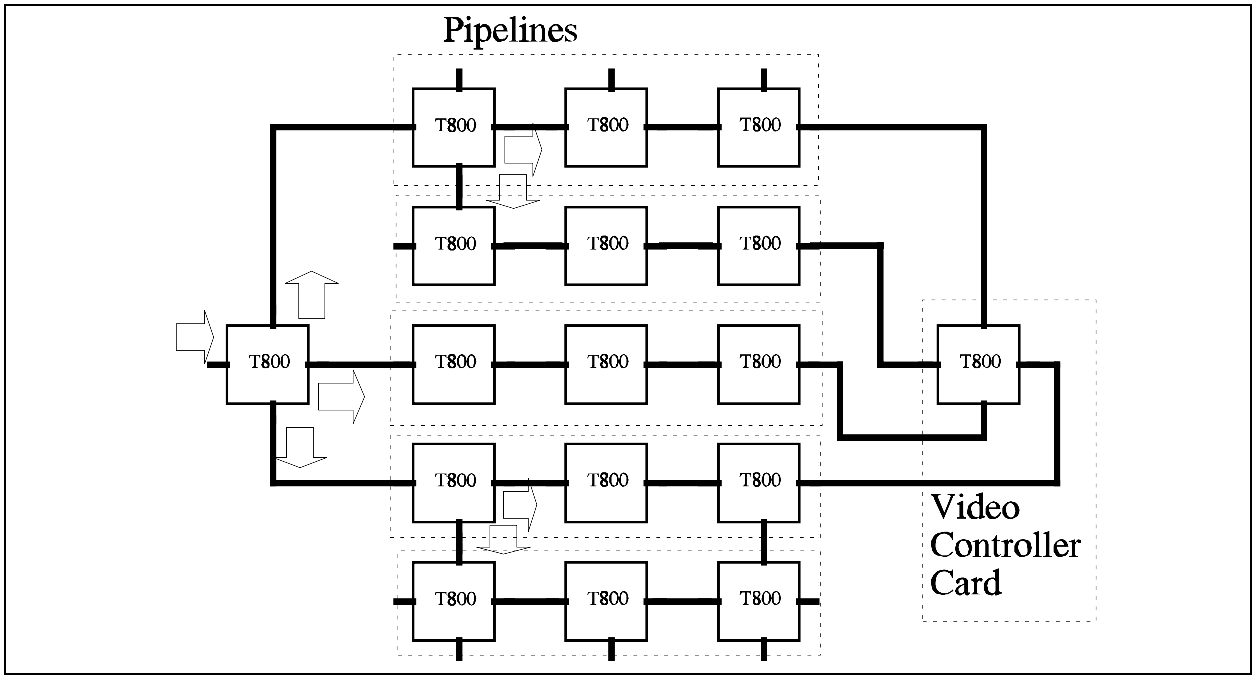


Figure 26 Processor layout for coarse grained parallelism

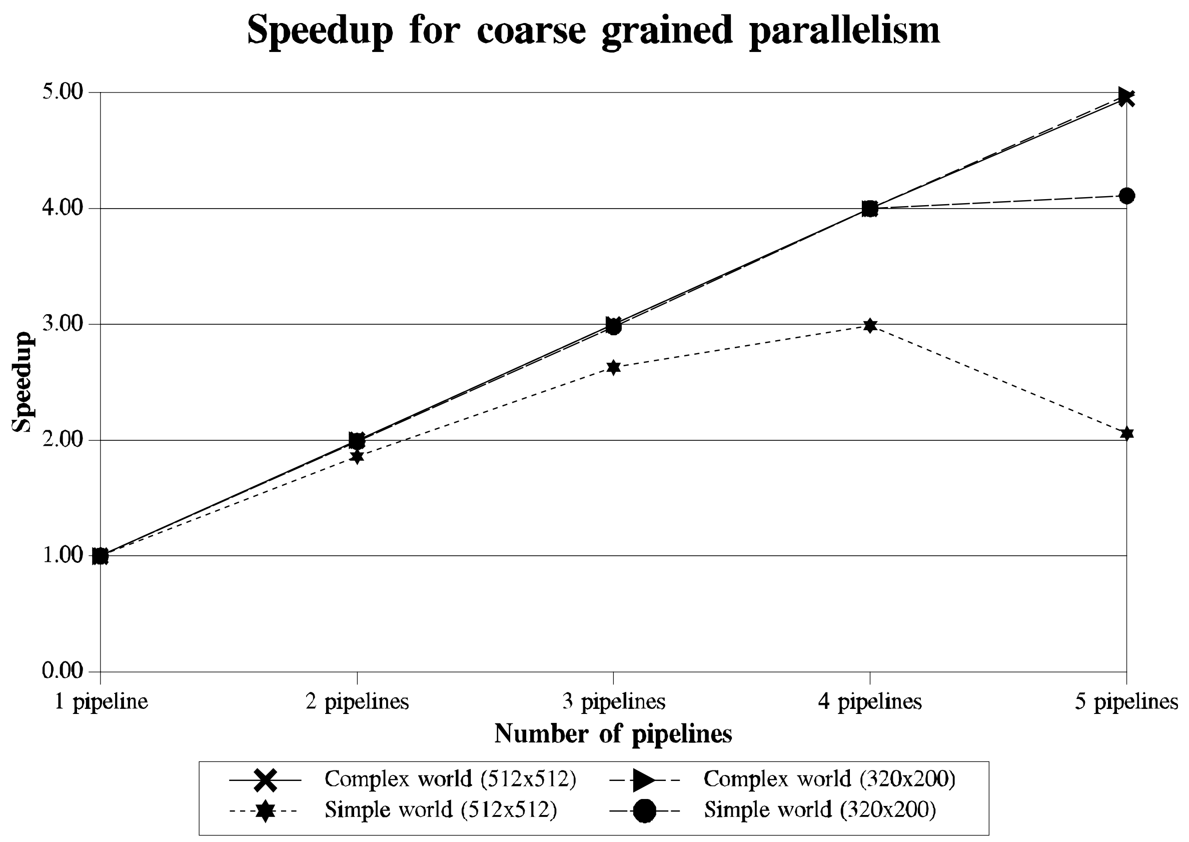
Values for the speedup obtained from an implementation of this technique are shown in Table VIII. As may be seen, linear speedup is obtained for most of these examples. The only factor preventing linear speedup in all cases is the bandwidth restriction on the links to the graphics node. With each link capable of 1.7 Mbytes /second and four links, this allows a maximum of 26.6 frames/second at 512 x 512 resolution or 108 frames/second at 320 x 200 resolution. The example with the simple world at 512 x 512 is using about half of the total bandwidth available.

There is a general tendency for a drop in the speedup when more than four processors are used, especially in high traffic situations. This tends to occur with all the parallelism strategies tried. It can be ascribed to the necessary doubling of the traffic on one of the links to the graphics node.

3.3.4. Fine grained parallelism

The pipeline approach uses a number of transputers connected in series. It has the disadvantage of high latency. A more suitable parallel decomposition for interactive graphics applications would have the processors connected in parallel, and all working on the same frame. In this way

Table VIII Speedup values for coarse grained parallelism



	Frames/s for 1 pipeline	Speedup for 2 pipelines	Speedup for 3 pipelines	Speedup for 4 pipelines	Speedup for 5 pipelines
Complex world (512x512)	1.09	2.00	3.00	4.00	4.95
Complex world (320x200)	1.12	2.00	3.00	4.00	4.98
Simple world (512x512)	3.98	1.86	2.63	2.99	2.06
Simple world (320x200)	8.06	1.99	2.98	4.00	4.11

the delay between input and corresponding output would be shortened.

The profile of the graphics routines running on a single processor (Table V) reveals that the bulk of the effort is spent on the stage involving hidden surface removal and polygon rendering. Thus the bulk of the effort in parallelisation has gone into decreasing the time spent on this stage. The other stages of the graphics pipeline are left as a short pipeline on separate processors.

The fine grained parallelisation technique is an image space pixel based parallel decomposition strategy. The display area is partitioned into horizontal strips, and the hidden surface removal and rendering for each strip occurs on a separate processor. The strips are rendered in memory and transferred to the graphics node which fits each block into the correct area of screen memory.

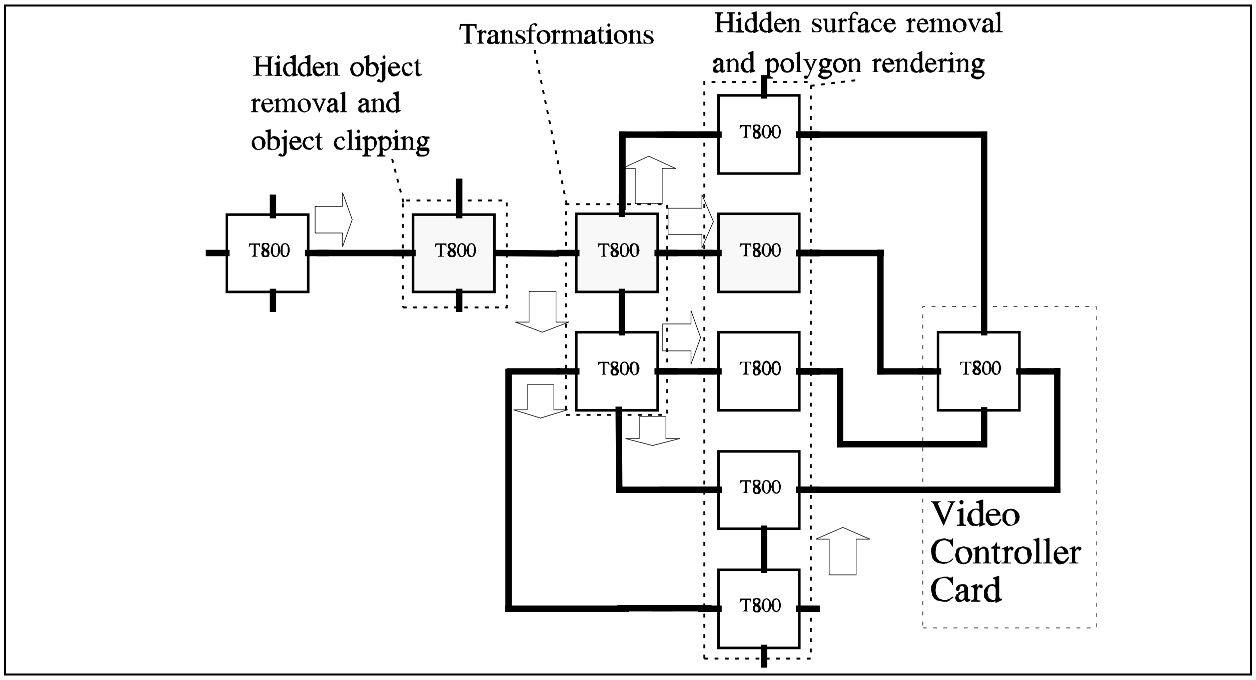

Transformations
Hidden surface removal and polygon rendering
Hidden object removal and object clipping
T800
Video Controller Card

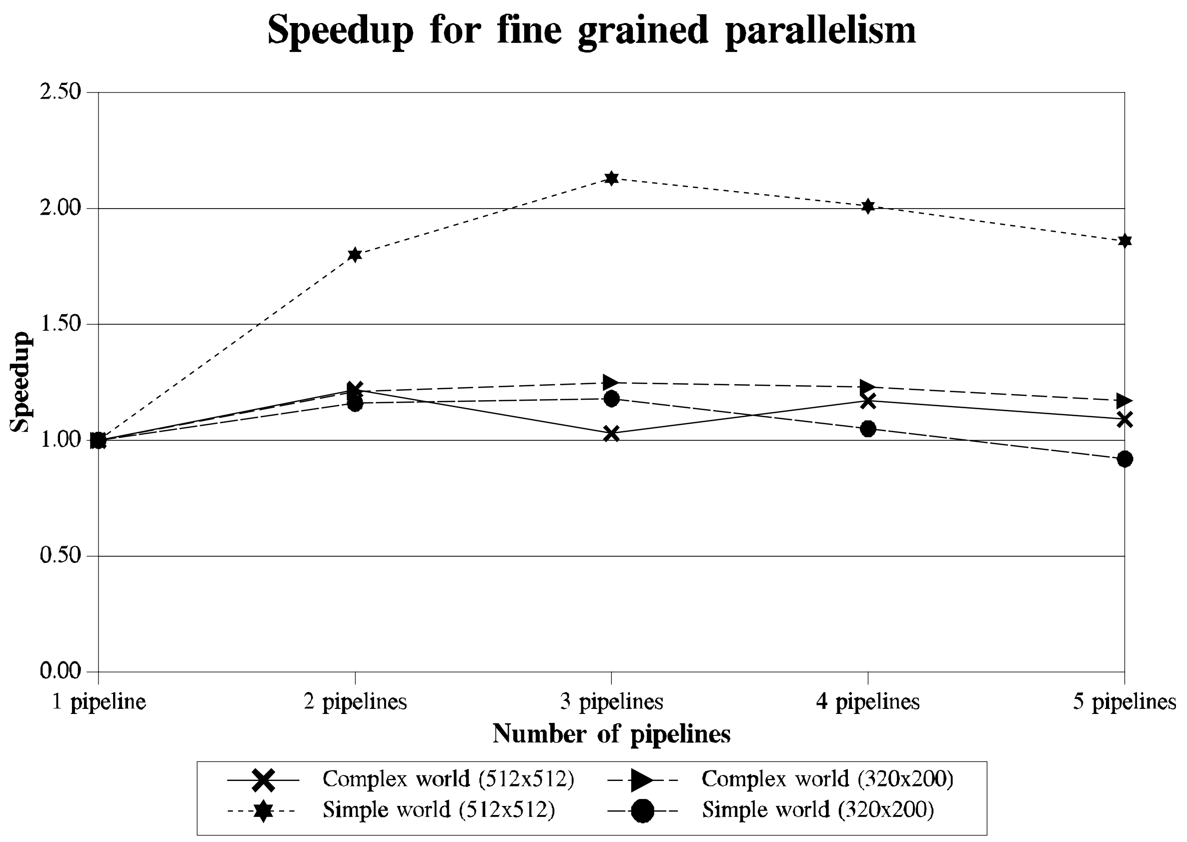
Figure 27 Processor layout for fine grained parallelism

The amount of data that is transmitted to the graphics node is the same as with the coarse grained parallelism. The messages are shorter but more frequent. The processor layout used to implement this technique is shown in Figure 27. The shaded processors represent one possible graphics pipeline for rendering a single strip. The same transformations need to be done for all the pipelines, so it is sufficient to use a single processor for the transformation stage and distribute the results to each rendering node.

Consider a simplified version of the graphics pipeline, consisting of K stages each taking equal time T to complete their task. By using N processors on one of the stages and assuming linear speedup will result in that stage taking time T/N. The total time taken to traverse the pipeline is (K - 1)T + T/N. The latency decreases as the number of processors is increased. The restrictions given above can be relaxed, and as long as speedup is obtained, latency will decrease.

The speedup values obtained from implementations of this strategy are shown in Table IX. The times measured are for the entire graphics pipeline, and not just the hidden surface removal and polygon rendering. This latter part of the pipeline does most of the work however, and controls the overall speed. This technique does not produce large speedups, nor does the speedup increase very much with the number of processors used. Two factors were found to contribute to these

Table IX Speedup values for fine grained parallelism



	Frames/s for 1 pipeline	Speedup for 2 pipelines	Speedup for 3 pipelines	Speedup for 4 pipelines	Speedup for 5 pipelines
Complex world (512x512)	1.06	1.22	1.03	1.17	1.09
Complex world (320x200)	1.08	1.21	1.25	1.23	1.17
Simple world (512x512)	3.64	1.80	2.13	2.01	1.86
Simple world (320x200)	8.62	1.16	1.18	1.05	0.92

effects.

When rendering the horizontal strips, the polygons that fall partially outside these strips must be clipped. As the number of strips becomes increases with each strip becoming smaller the number of polygons needing to be clipped increases. In one test with the screen divided into 4 strips the time taken for clipping increased from about 25 % of the time taken for rendering to about 500 % of the rendering time. The rendering time did decrease by a factor of 4 due to the parallelism, but the clipping overhead is still significant. The clipping was a significant factor with the single object in the complex world with its many polygons. The clipping limited the speedup the most for that case.

The second factor involves load balancing. The picture is not always evenly distributed over the screen. Sometimes all the objects cluster in one region of the screen. In this case one processor is required to render most of the objects while others render almost nothing. This situation is almost equivalent to a single pipeline. The effect of this can be seen in the speedup values for the complex world with 512 x 512 resolution. The single object occupied the middle half of the screen. Thus for three pipelines, the processors rendering the top and bottom slices had little work to do and the processor rendering the middle slice did most of the rendering. This explains the drop in speedup over the two pipeline version where the load was more evenly balanced. The object filled the screen in the 320 x 200 resolution and clipping limited the speedup in this case.

Alternative techniques for distributing sections of the display area over N processors include:

- Placing successive scan-lines on different processors.
- Dividing the screen into N^2 horizontal slices and placing successive slices on different processors.

The second alternative would solve the load balancing but at the expense of worsening the clipping.

The frame rates obtained with this method were not high enough for bandwidth limitations to be a constraint.

3.3.5. Object-oriented parallelism

This technique is an object space hidden surface routine. Rendering for different objects occurs on different processors. For the case of N processors, the objects are sorted into N groups based on their depth. This is easily done as part of the depth sort involved in the hidden object removal routine. Each group is then rendered on one of the processors. The images produced are sent to the graphics node where the images are combined. The combination technique is similar to the Painter’s algorithm where images corresponding to nearer objects are painted over those belonging to more distant ones. The transputer has a specialised block move instruction that allows this to be done relatively quickly and easily. Both the transformation and hidden surface/polygon rendering stages can be done in parallel with this method.

The transputer network used to implement this strategy is shown in Figure 28. The shaded nodes show the graphics pipeline for one group of objects. This parallelisation strategy has one

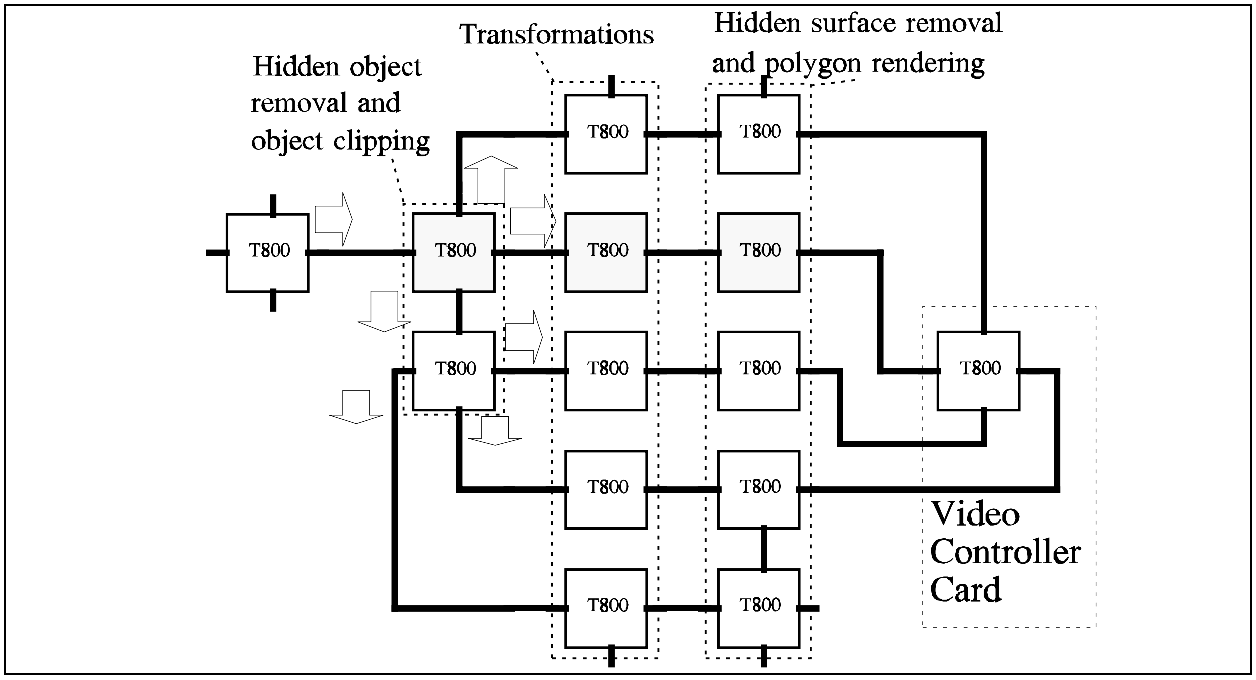

Transformations
Hidden surface removal and polygon rendering
Hidden object removal and object clipping
T800
Video Controller Card

Figure 28 Processor layout for object oriented parallelism

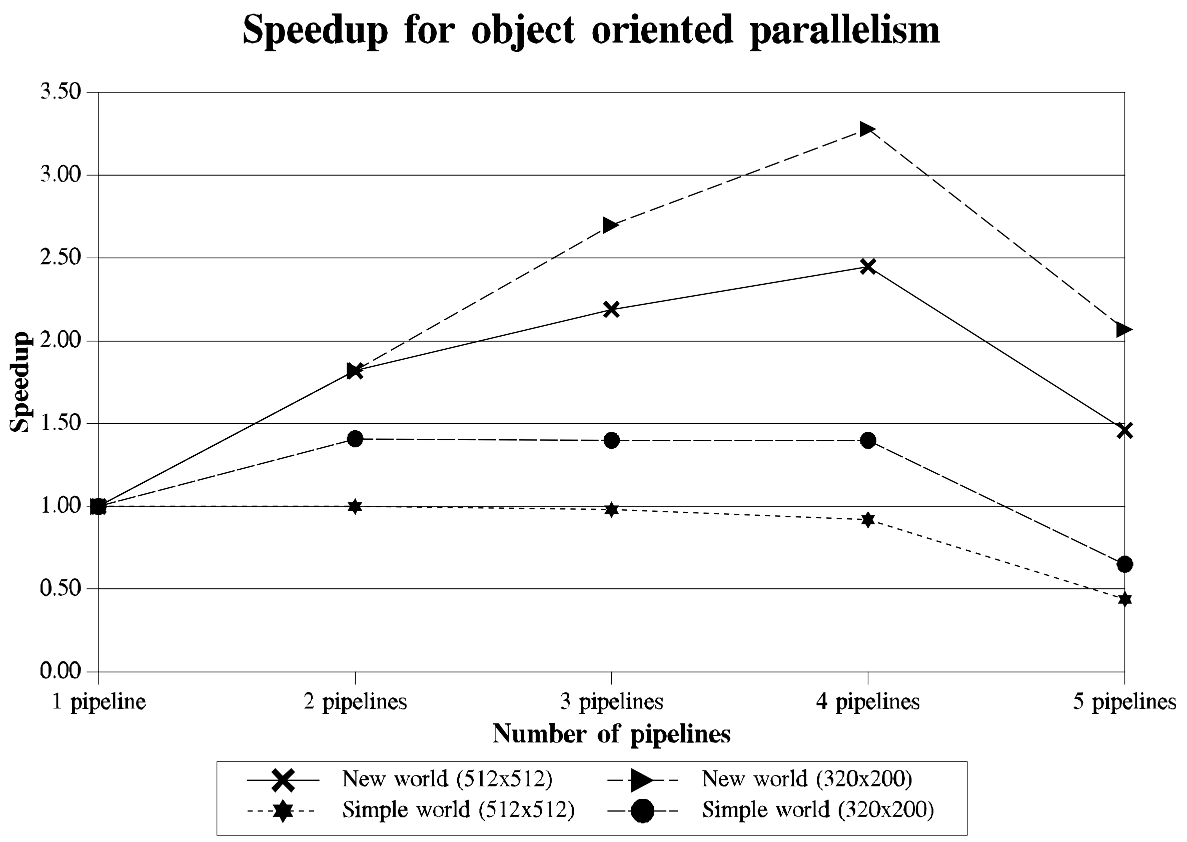
immediate disadvantage. If the scene consists of only one object, then there is little point in using more than one processor. Consequently a world containing 16 objects, each with 276 triangles was used instead of the complex world used for previous tests.

Values obtained for the speedup for an implementation of this strategy are shown in Table X. The bandwidth limitation is particularly severe in this case. In the case where the number of processors used is less than five, an entire image needs to be sent down each of the transputer’s links. This places an upper limit on the frame rate of 6.6 frames per second for 512 x 512 resolution or 27 frames per second for 320 x 200 resolution. The speedup values tend to level off as this barrier is reached. Adding a fifth processor just worsens the situation.

A further problem with this technique is load balancing. Some objects are more complex than others, and just distributing equal numbers of objects may result in some processors having to do more work. A finer, more balanced approach would be to distribute on a polygon basis. This could be implemented relatively easily by taking advantage of the BSP tree structure. However such an approach is not worth exploring until higher bandwidth systems are available.

The parallelism would also suffer when the number of visible objects is less than the number of

Table X Speedup values for object oriented parallelism



	Frames/s for 1 pipeline	Speedup for 2 pipelines	Speedup for 3 pipelines	Speedup for 4 pipelines	Speedup for 5 pipelines
New world (512x512)	0.50	1.82	2.19	2.45	1.46
New world (320x200)	0.50	1.82	2.70	3.28	2.07
Simple world (512x512)	3.65	1.00	0.98	0.92	0.44
Simple world (320x200)	8.85	1.41	1.40	1.40	0.65

processors. Polygon distribution will be valuable for this situation as well.

Considering the simplified version of the graphics pipeline used in the previous latency calculation, gives the total time taken to traverse the pipeline as (K - 2)T + 2T/N, since two stages of the pipeline are parallelised. The latency is less than that for the fine grained decomposition and also decreases as the number of processors is increased.

3.3.6. Further parallel implementation features

The polygon lists for all objects in the current world are distributed to every node when the object is first used. The polygon list specifies only the interconnection of the vertices and is unchanged throughout the graphics pipeline. The vertex list, containing the three dimensional points in the objects, is transmitted from one node to the next through the graphics pipeline. At each stage it is modified and passed on to the next node.

3.3.7. Optimisations and speed improvement techniques

Thorough analysis of the run-time profiling information showed that processor time was being wasted in waiting for other processors to catch up. Processors were blocking, waiting for others to communicate. This problem was overcome by adding extra processes to monitor the communication links and to buffer several messages. In this way wasted time was reduced. A notable speedup in the frame rate was achieved.

This solution had a corresponding disadvantage. Adding extra processes on the communication links has the effect of lengthening the pipeline. This increases the latency. The size of the buffers on each link also affects the latency adversely.

In order to get the maximum benefit from this technique the size of the buffers should be related to the complexity of the scene. Small buffers are better suited for scenes with a few complex objects, while larger ones smooth over the delays for large numbers of simple objects.

3.4. Other renderers

A number of other renderers were created to test the multi-user capabilities of the system. These were simpler programs which produced ’wire-frame’ images.

The design of these renderers was similar to the transputer version described above. The graphics pipeline consisted of fewer elements since hidden surface removal is unnecessary for wire-frame images.

The world description was obtained from the transputer cluster via the ethernet.

3.5. Future additions

In order to make the transputer renderer comparable with the hardware renderers, further extensions would need to be made. In particular, most hardware renderers provide support for Gouraud and Phong shading.

Another requirement for the renderer would be to enable easy use of text in the three dimensional world. The ability to read documents or use menus may be a useful feature. At present, any text would have to be converted to polygon format, making it both slow to use and difficult to read. It may be better to merely overlay text on top of the image.

Development of the renderer has concentrated on increasing speed of rendering while minimizing latency. Other techniques for improving speed are possible. For example, a less detailed model of an object can be used when the object is far away from the viewer. Another alternative is to produce less detailed images when moving, and refine the image when the viewpoint is no longer changing as is done in [8]. This technique can also be used to maintain a constant rendering speed by decreasing the image quality when the frame rate starts dropping [21].

3.6. Summary

This chapter has described the design and implementation of a fast renderer for polyhedral objects. Each stage of the graphics pipeline was considered and alternative techniques were compared in terms of speed. An efficient polygon fill routine was described. The BSP Tree hidden surface removal method was found to be fast and gave accurate output.

A number of different parallelisation strategies were tested in an attempt to increase the rendering speed and decrease the interaction latency. A frame per processor approach gave almost linear speedup, but left the latency unaffected. Strategies which distributed the calculations for a single frame over a number of processors resulted in latency decreasing as the number of processors increased. These techniques did, however, have lesser speedups. The factor that most limited the speedup was the limited bandwidth of the transputer links. Faster communication is essential if real-time images are to be produced.

4. Input Device Drivers

This chapter describes the input device drivers that were implemented, and describes observations that were made regarding the use of the various devices.

4.1. Design of Input Device Driver

4.1.1. Requirements

The input device drivers must:

- Interface with the input device being used.
- Determine which gesture is being made and pass this information to the virtual world manager.

Input may be taken from a number of physical devices. Traditional input devices are keyboards and mice. Virtual reality applications generally use a glove containing sensors to measure finger bend and hand position.

The goggles used to display the worlds often contain sensors to measure head position, allowing the view to be changed according to the direction in which the user is looking. Such sensors would also be classified as input devices.

The device driver is required to control the physical hardware and convert the data into a standard form.

4.1.2. Design

The device drivers should ideally convert the data from the various input devices into gestures that are independent of the nature of the hardware. For example, the ’g’ key pressed on the keyboard or a clenched fist detected by a glove could correspond to a ’GRAB’ gesture. Some of the devices provide information that can be used directly; the three dimensional coordinates of the hand may be useful in directly positioning the hand object in the virtual world. Providing visual feedback to the user also requires the production of an image of the hand with fingers bent appropriately.

For this reason it was decided to allow the application writer direct access to the raw data returned from each device. Higher level routines provided as part of the virtual world kernel would perform the conversion of this data to gestures.

A description of these conversion routines will be included as part of this chapter.

4.2. Implementation of the powerglove device driver

The Nintendo powerglove, described in section 1.4.1.3, is used by many virtual reality researchers as an inexpensive input device. It contains four strain gauges for measuring finger bend. The strain gauges, made from plastic coated with a resistive ink, are found in the thumb and first three fingers. A pair of ultrasonic transmitters mounted on the back of the glove are detected by three receivers mounted in an L shaped configuration. These allow measurement of the position of the hand in space. Position is measured to about 3 mm in the X and Y directions, and to about 14 mm in the Z (depth) direction. The glove is operable within a pyramidal volume limited to about 4 m from the receivers, with resolution decreasing with distance. The two transmitters also make it possible to detect wrist rotation. A keypad containing a number of keys is mounted on the back of the glove.

The glove was intended only for use with Nintendo system, being equipped with a proprietary Nintendo connector. A recent article gave instructions for interfacing it to a standard parallel port [13]. Usage of the glove was limited to emulating a joystick due to the encryption of the data stream from the glove. This was recently decrypted and drivers were released to allow access to all the glove functions.

This glove is not particularly reliable, the ultrasonics echo off walls and other furniture and introduce spurious values into the position data. The fingers need to be flexed occasionally to recalibrate the A/D convertors attached to the strain gauges. The glove driver includes routines to remove most of this noise in the data. The glove needs to be manually centred occasionally to define the origin of the coordinate system that it uses. The position of the powerglove is therefore relative to a random position in space, which may not always be constant.

With these drivers for the powerglove already available, implementation of the device driver was simplified. Since the glove worked through a parallel port, and none were available for the transputer cluster, the glove was connected to one of the host PCs. The device driver, running on one of the transputers, polls the PC regularly to get the latest glove data. The data is in the form of 8 bits for each of the x, y and z coordinates, a value from 0 to 11 giving wrist rotation, 2 bits per finger for finger bend and a byte giving the key pressed on the keypad.

4.2.1. Converting glove data to gestures

It is difficult to measure the bend of the finger, the transitional positions between fully open and fully closed are close together. The only states that can be measured with any degree of certainty are the two extremes.

The gestures which can be interpreted are limited by a number of constraints. The fingers can only be bent in a limited number of ways. Certain configurations are difficult to achieve or maintain for a long period of time. The sensors on the powerglove are unable to measure other values such as the angle between thumb and forefinger. Thus other possible gestures are eliminated.

For new users of the system, the interface should be easily to learn and intuitive to use. The gesture used should fit the action. The number of gestures used for principal tasks should be limited, and should be useful for other similar tasks.

A number of simple finger and wrist configurations were selected as basic gestures. These are listed in Table XI. Use of these to control motion is discussed in section 6.1.1.1. The GRIP gesture was chosen as reverse since this closest to the ’inverse’ of the POINT gesture that could be comfortably achieved.

Further glove control is slightly more complicated. Not only it is necessary to avoid reacting to the occasional glitch in the data from the glove, but glove actions need to be interpreted over a period of time. Parsing a motion such as waving requires records of glove position in the past. The first problem can be minimised by requesting confirmation for critical operations. The latter problem can be addressed by using state transition diagrams, triggered by the different gestures. Such a state transition diagrams are shown in Figure 29.

Gestures such as POINT and GRIP are immediately implemented as movement. The use of history information is illustrated when turning. When the TURN gesture is made in the FREE

Table XI Glove configurations for various gestures

Gesture	Thumb	1st finger	2nd finger	3rd finger	Wrist	Intended use
FIST	>1	>1	>1	>1	X	Grasping objects
FLAT	≤1	≤1	≤1	≤1	X	No operation
YES	≤1	>1	>1	>1	between 50° and 160°	Confirmation of operation
NO	≤1	>1	>1	>1	between 230° and 340°	Cancellation of operation
POINT	>1	≤1	>1	>1	X	Move
TURN	≤1	≤1	>1	>1	X	Rotate
GRIP	>1	>1	≤1	≤1	X	Reverse

(Fingers: 0 - open, 3 - closed. Wrist: 0° has right hand thumb horizontal and pointed left, 90° has thumb pointed upward.)

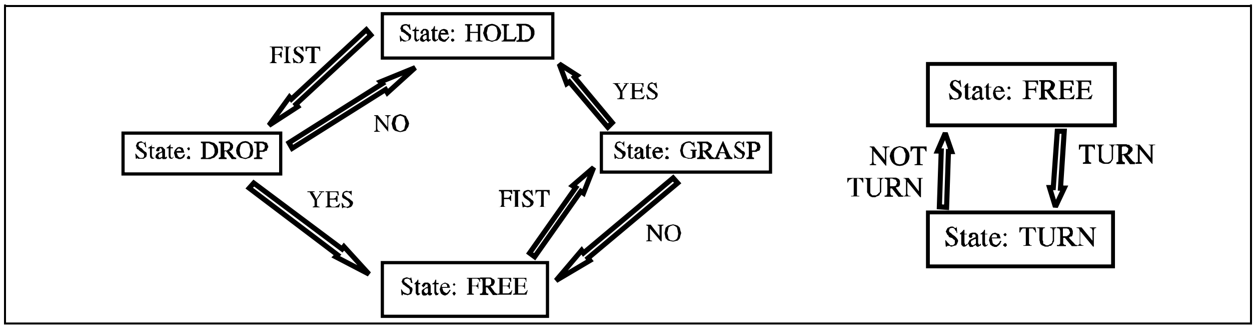


Figure 29 State transition diagram for glove control

state, the current glove position is memorised. When this gesture is made in the TURN state, the offset of the glove from this previously measured position is calculated and the user is turned in the corresponding direction. Any other gesture returns the system to the FREE state.

Confirmation is required when picking up and releasing objects. A FIST gesture is made and if an object is present then the state will change to GRASP. A ’thumbs up’ YES gesture will then pick the object up, a ’thumbs down’ NO gesture will cancel the operation. The same routine occurs when releasing objects. Possibly a more intuitive alternative is to hold the object when the fist is closed and release it when the hand is open. This was rejected because holding fingers bent for long periods is physically tiring when wearing a powerglove. This alternative also would make it impossible to control movement with the glove while holding objects.

The keys on the keypad are converted to a sequence of numbered gestures. The purpose of these is undefined, and it is left to the application writer to define the functions corresponding to these gestures.

4.3. Implementation of the keyboard device driver

A device driver using a keyboard was created for a number of reasons. Firstly the number of powergloves is limited, and keyboard input was useful for testing the system with multiple users. Secondly keyboards are useful for precise control, and are useful for doing finer manipulations that may not be possible with a powerglove. Also keyboards have more degrees of freedom, and are more appropriate to some functions, such as entering text.

The keyboard driver is implemented in a similar manner to the powerglove driver. A process runs on one of the transputers, polling the keyboard whenever input is requested. This polling may be over the link to the host PC or over the ethernet to a remote machine.

The device driver returns all the keys pressed between the current and previous calls to the driver. This allows the process reading the keyboard to either discard extra key strokes, or better, to act on all the keypresses before updating the display. The latter policy allows movement at constant speed even when the frame rate varies.

4.3.1. Converting keys to gestures

When using a keyboard several of the limitations of a glove device are removed. Lots of keys are available and these can be clearly marked with their function. This makes it possible to use a different set of gestures that are more comprehensive. A limitation is imposed in that there is no position sensing. Also all input is digital and limited to binary or unary, no analogue input is possible. The keyboard is better suited for navigation in the virtual world than for object interaction.

The limited range of values that a single key may return means that a number of key strokes are required to perform the same function as a single movement using a position tracking device. Keyboards are thus not ideal as input devices for virtual reality applications.

The gestures chosen for the keyboard perform movement or rotation in various directions, for example **MoveUp** and **RotateLeft**. The number keys are converted into numbered gestures, as for the glove keypad.

4.4. Summary

Discussion in this chapter has been limited to the available devices. Comparisons of other hand input devices and their relative advantages can be found in [37].

Having visual feedback on the state of the glove has made use of the glove much easier. Initially, when developing the system, only numerical values describing the configuration of the glove were available. Producing the correct gesture was difficult since the state being measured by the glove was not immediately obvious. Once a visual representation of a hand was created limitations in the glove could quickly be compensated for by changing the bend of various fingers.

The powerglove with its three dimensional movement ability is well suited to interaction with objects, while its limited number of gestures limit its usefulness for movement in a virtual world. The keyboard is limited to navigational functions. This suggests that a combined keyboard (or joystick) and glove system is well suited for inexpensive virtual reality applications. More advanced systems use trackers built into the head mounted displays to perform the navigational tasks.

5. The Virtual World Kernel

5.1. Design of the Virtual World Kernel

The virtual world kernel is the module that will support the virtual reality applications. It must assist in modelling realities. The design section describes the choices made when creating such a modelling system on a parallel architecture.

While a number of specific virtual reality applications did exist when design of the virtual world kernel was started, documentation of general virtual reality systems was scarce. However similar systems were being constructed at the same time as this one. Versions of these systems, or reports are about them started becoming available as this project reached completion. A comparison of the designs of these virtual reality systems is given in a later chapter.

5.1.1. Requirements

The requirements for the virtual world kernel state that it must:

- Support multiple worlds.
- Support multiple users in each world.
- Support connections from other machines.
- Use parallelism to reduce the load on individual processors
- Provide the ability to associate attributes with objects.
- Allow object manipulation routines to be easily created.

These requirements fall into two classes. There are those that deal with the layout of the kernel and its support for multiple processors, worlds and users. The second set of requirements concerns the functionality of the kernel, and the services it will provide to support the development of virtual reality applications.

In order to give an idea of the relative scale of operations, the system is expected to run on 10 to 20 processors and support about 15 worlds and 150 objects at any one time. These values are not constraints but may be useful parameters when visualising the following discussion.

5.1.2. Design

The design of the kernel is shown as three stages. The first describes the manner in which worlds and objects are represented, and how they will be distributed. This is followed by a description of the functionality of the kernel. Finally the design of the communication system supporting interaction between worlds, objects and users is given. This last stage shows the manner in which the functionality of the kernel is implemented on the distributed system.

5.1.2.1. Representing worlds and objects

A world is required to store details about its objects. It must therefore contain a data structure of the type described in section 2.1.4. Beyond that it could be either active or passive. An active world would control all its objects, implementing all the physical laws for that reality. A passive world would simply serve as a central data store, each object would be responsible for constraining itself to the laws of the reality.

The design that is chosen is required to make efficient use of multiple processors. While the active world is the closest to the manner in which physical reality works, it suffers from the disadvantage that most of the complexity is confined to the world module. Modification to the laws of the world would involve working with a large complex program. Most of the processing is done by the one process. This could be parallelised by placing computations for each object on separate processors, but then this reduces to the passive world case.

A third approach that does not use a world data structure, but distributes the data over each object was rejected because of the high communication overhead. One of the most common operations in a virtual reality system is likely to be the display of the world. Having to gather object data whenever the world needs to be redrawn is likely to place too high a load on the system.

The passive world approach is the one implemented. Each world consists of the world data structure and a simple server to supply data and update the world in response to requests from objects. The objects are active, consisting of an object data structure, together with a process. The object data structures are components of the world data structure, and so the object process must query the world server to access the object data. The data for each object consists of its attributes as described in section 2.1.4. The object process must supervise the actions of the object to ensure that the laws of the virtual world are enforced.

This approach also allows the creator of the virtual reality application a greater degree of freedom. Objects are permitted to ignore some or all of the laws governing a particular world. This does increase the possibility of accidentally leaving an inconsistency in the world, however. This possibility may be lessened by providing a default routine to control the object according to the attributes of the object and of the world, i.e. to make the object obey the laws of the world.

Once the idea of an autonomous object is accepted, then the distinction between a user, where a human controls the motion of an object, and an object, which controls its own motion, becomes blurred. The only difference is that users take motion control information from an input device and display their views on an output device. Such system calls can easily be included in the control routine for an object. There is no longer a need to distinguish between objects and users.

The parallelisation strategy involves distributing all the objects and worlds across all the nodes in the system. Since the transputer cluster contains a relatively small number of quite powerful processors, this level of decomposition is adequate. The number of objects will generally be greater than the number of processors available, so several objects will coexist on one processor.

5.1.2.2. Functionality of the kernel

The design of virtual worlds is a new field and the requirements are not completely established. Several classes of functions are proposed as a basic set for the manipulation of virtual worlds.

Each object in the world must be able to control its own attributes. This is a necessity since it has been decided that each object must control its own response to the laws of the world. Thus a set of functions to get and set the values of these attributes is required.

In the physical world, the attributes of other objects can usually be measured. One can generally measure position and size of other objects. This should be the case for virtual worlds as well. This requires a set of functions to get the values of the attributes of other objects.

Controlling other objects introduces a measure of complexity. In the physical world, there is usually no difficulty involved in picking up a ball and dropping it. Considerable opposition may be experienced if another person is the subject of this action. Some mechanism is required to distinguish between the two cases. This should be dynamic; there may be times when picking up another user is appropriate. The notion of ownership is proposed as a means for avoiding contention. Objects may either be ownerless, or owned by a specific object. Ownerless objects may only be controlled by their corresponding process. Owned objects may only be controlled by the process corresponding to their owner. To control another object, ownership must first be acquired. Ownership may then be relinquished at a later stage. This technique also provides the advantage of eliminating contention for objects. Only one object can control another at any time.

Ownership can be made hereditary. If object A gains ownership of object B who happens to own object C, then A can be considered as the owner of C. This could lead to problems with contention between objects A and B to control C. The approach taken to date is to allow only single generation control, A would control B, B would then have to change C accordingly. An exception is made in the case of transfer between worlds. When an object moves from one world to another, all directly and indirectly owned objects must be transferred as well.

Changing the attributes of other objects requires a set of functions that implement the ownership concept.

Since the intention of the system is to support implementation of virtual reality applications, functions are needed to support this. In particular functions to support interactions amongst objects, and between object and worlds are necessary. Implementing virtual worlds is not yet an exact art and the functions to best support application development are not well known. While some of the needed functions may be guessed at, some others may be found useful by the application programmer and added at a later stage.

The user objects will need to be interactive, getting data from some devices and outputting data to others. Device drivers have already been mentioned in an earlier chapter as a technique to provide a standard interface to the physical hardware. A set of functions to communicate with the devices is necessary.

5.1.2.3. Communication between worlds and objects

Communication between object and world is necessary for a number of purposes; to locate other objects in the world, to change position and other attributes or to display all the objects in the world. For each world there can be many objects. Objects need not remain in one world. They can, especially if they correspond to users, move from one world to another.

A limited interaction between objects is also necessary, particularly with regard to the implementation of ownership. Owned objects can be forcibly transferred from one world to another. The object process should be notified of this to allow it to keep track of where it can find the object data.

Rather than expose the application writer to the problems of the communication, an intermediate communication layer is used. This will keep track of which object is in which world, and direct communications accordingly. When the object changes worlds, only the routing table need be updated; the object process is not affected.

Communication with the devices also can make use of a communication layer. Devices will be identified by a single device number, all other data will be kept at a lower level. This will allow the devices to be used without the object process needing to know the address or nature of the device.

Communication with other machines can be included in the design without much trouble. Support for objects running on other machines can be added by placing an object process on the local machine that relays data to and from the remote machine. To support devices on remote machines, a device driver can be created locally that also functions as a data transfer process.

The simplified view of the overall structure of the virtual world kernel is given in Figure 30.

5.2. Implementation of the Virtual World Kernel

The functions required for supporting virtual reality applications are not overly complex and can be implemented without much trouble. The area of interest with respect to the implementation of the virtual world kernel is the parallel processing and support for multiple worlds and objects. This section deals with the implementation of these aspects, particularly where it is affected by the message passing architecture of the transputers.

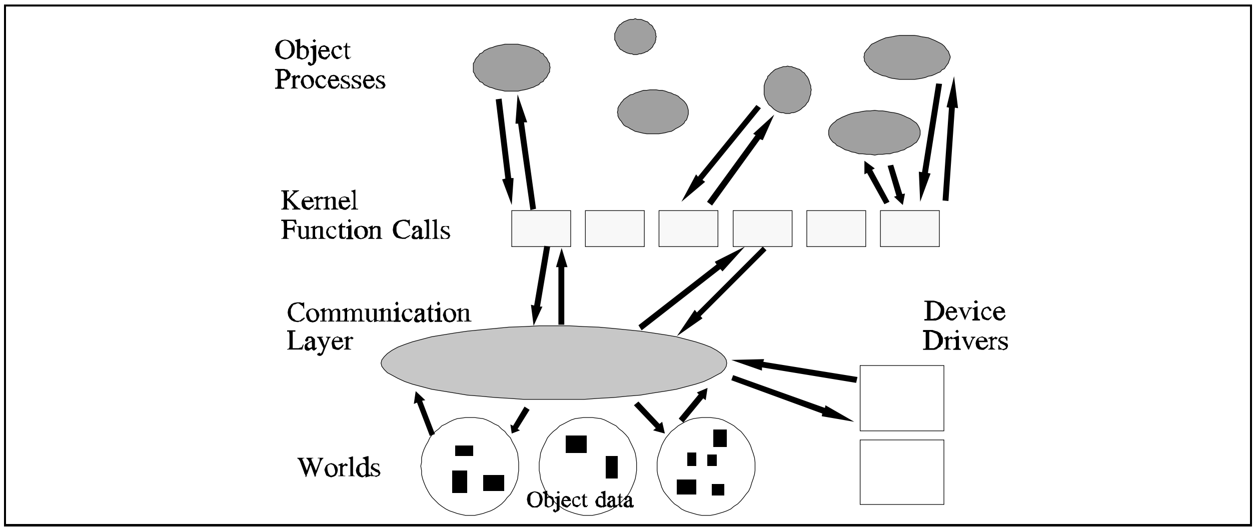


Figure 30 Summary of kernel design

5.2.1. Object Processes

Creating separate processes for the objects is not difficult when using transputers. The processor provides support for concurrent processes, and performs context switching quickly and automatically.

5.2.2. Kernel Functions

The description of the various kernel functions was given in the design section previously. This section gives a few examples of the functions that fall into each section. The list is not complete but may assist in understanding the functionality intended for the system.

This first class of functions is intended to get and set the values of the attributes belonging to the current object. These attributes include position, orientation and size as well as other more general characteristics. Functions in this category include **GetPosAndOri**, **SetPosAndOri**, **GetScale** and **SetScale**. The general attributes are stored in a nonspecialised data structure and may be accessed by functions such as **GetAttributes** and **SetAttributes**.

The second class of functions is for manipulating other objects. Getting the attributes of other objects can be done by calls such as **GetOtherPosAndOri**, **GetOtherScale** and **GetOtherAttributes**. Setting the attributes of other objects first requires ownership of that object. This may be acquired through the **ClaimOwnership** function and forfeited with

ReleaseOwnership. Once the object is owned, the attributes may be set with **SetOtherPosAndOri**, **SetOtherScale** and **SetOtherAttributes**.

A third class of function supports interaction between object and object, and between object and world. Among the functions found in this class are **FindClosestObject** and **FindClosestFreeObject**, to find the nearest neighbour. The latter function finds the closest neighbour that does not yet have an owner. Functions such as **MoveToWorld** and **MoveOtherToWorld** support the transfer of objects from one world to another.

The functions to control device drivers are similar to those used to control device drivers in other operating systems. They include an **OpenDevice** to gain control of the device and a **CloseDevice** to release control. There are also functions to read and write to devices, in this case **ShowObjects** to display a world and **ReadGesture** to get data from a device.

Functions to create and remove objects and devices are also useful, although the current implementation also allows a configuration file to create these automatically when the system starts.

Details of the implementation of some of these functions follows later, after a description of the implementation of the other parts of the system has been given.

5.2.3. The Communication layer

The communication layer is responsible for routing messages between the object processes and world servers. The messages are not generated directly by the object process, but indirectly by one of the kernel functions called by such a process. The communication system then performs the routing of the communication between different nodes.

The presence of multiple processors should not affect the kernel functions or object processes. This layer of the kernel is only necessary to make the rest of the system independent of the parallel implementation details. It is not relevant to the construction of virtual reality systems in general.

5.2.3.1. Communication between object and world

Each node needs to maintain a routing table giving details of which objects belong to each world.

Messages sent from the object processes are addressed to the node containing the correct world by means of this table.

Some form of scheduling control is needed to allow all object processes an equal opportunity to communicate with their world. Allowing all processes to communicate simultaneously will could lead to situations where some processes rarely get access to external links. The ability to implement a fair scheduling policy is a requirement.

Since the transputer architecture requires that only one process use each channel, and every object can conceivably communicate with any world, the number of channels required is the product of the number of worlds and number of objects. While this is a large number of channels it is not entirely inconceivable. Each of these channels, however, requires a buffer. If messages are not buffered, the routing processes will block, replies cannot be sent, and the system will deadlock.

The scheme described above could be implemented but it would be expensive in terms of memory. Scheduling problems can also arise in gaining access to the external communication links of the transputers. An alternative scheme was implemented that uses fewer links, and that enforces a scheduling policy. Each node contains a communications manager that multiplexes all messages from that node onto the external links of the transputer. Messages are queued so that each process gets a chance to communicate. Each message from an object process is forwarded to the appropriate world server, and the reply returned to that process before the next message can be serviced. This method could exhibit reduced performance, by restricting the object processes so that only one per processor can communicate at a time. This problem can be easily overcome by adding more communications managers per node, allowing a number of objects to communicate with their worlds simultaneously, but still limiting the number of channels required. The communication with the world server is designed to be as short as possible to minimise any delay.

The extra process layer for communication also would be useful when expanding the system to run over a network of heterogeneous machines. Here the routing will be more complicated, and protocol conversion may need to be included along with the routing.

Each world requires a channel for each processor. Each object requires a channel to communicate with its local manager. The total number of channels required is the sum of the number of objects, N, and the product of the number of world, W, and processors, P, i.e.

Channels = N + W x P.

This is less than required with the previous scheme, namely N x W, since the number of processors should be less than the number of objects.

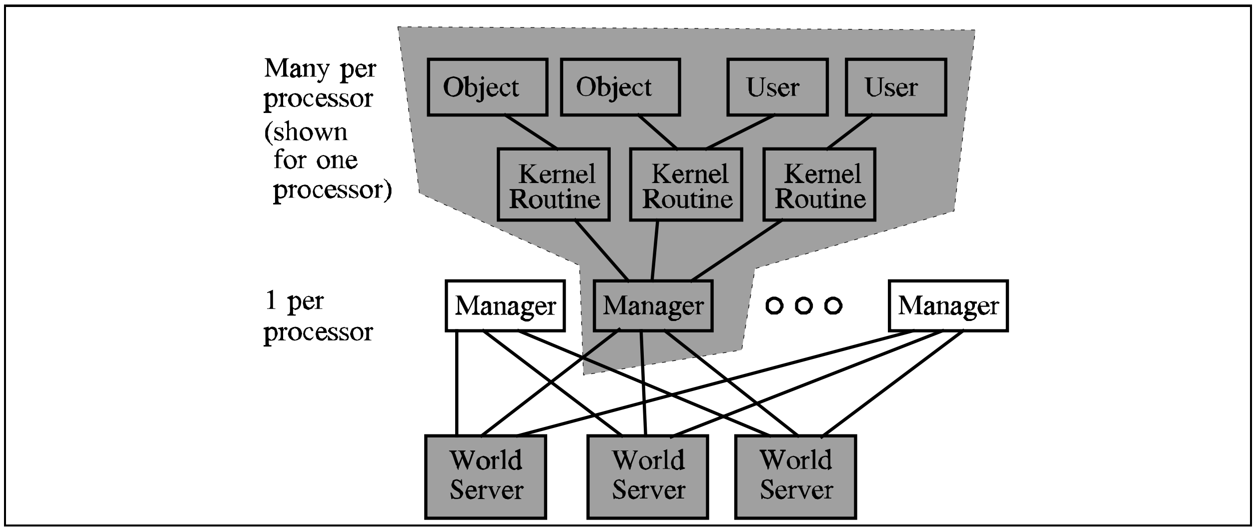


Figure 31 Structure of communication layer

The structure of the communication layer at this point is shown in Figure 31.

5.2.3.2. Communication between object and object

Users can transfer other objects from one world to another. Objects can create other objects. This requires that the communication managers be able to communicate, to allow the necessary updating of routing information.

Extra channels can be added to the communication manager process for this type of communication. Deadlock can arise with this sort of communication when two managers try to communicate, each waiting for the other to reply before replying itself. A solution to this problem is to separate the communications manager process into two separate processes. A communications manager will still handle communication with world servers. An object coordinator will be added to control communication with other managers. The routing table may be shared between the two processes since they are present on the same node. The communications manager can send messages and receive replies either to world servers, or to object coordinators. As long as no object coordinator then tries to send a message to, and receive a reply from a communications manager deadlock is prevented.

If greater bandwidth is required, then more communications manager processes can be added.

5.2.3.3. Communication between object and device driver

Communication between objects and devices is simplified by the restriction that at most one object can operate a device at any time. A single channel can be allocated for each device. A simple routing table can be used to direct messages to the appropriate device, indexed by the device number.

Some form of arbitration scheme is required to prevent different objects from opening the same device. This can be implemented by creating a single device server process, which fits into the same category as the world servers. Object processes issue requests for devices that are forwarded through the communication managers. The device server handles one request at a time, and issues the devices to the requesting process. Once a device is issued, it cannot be given to another process until it is released by the one to whom it was issued.

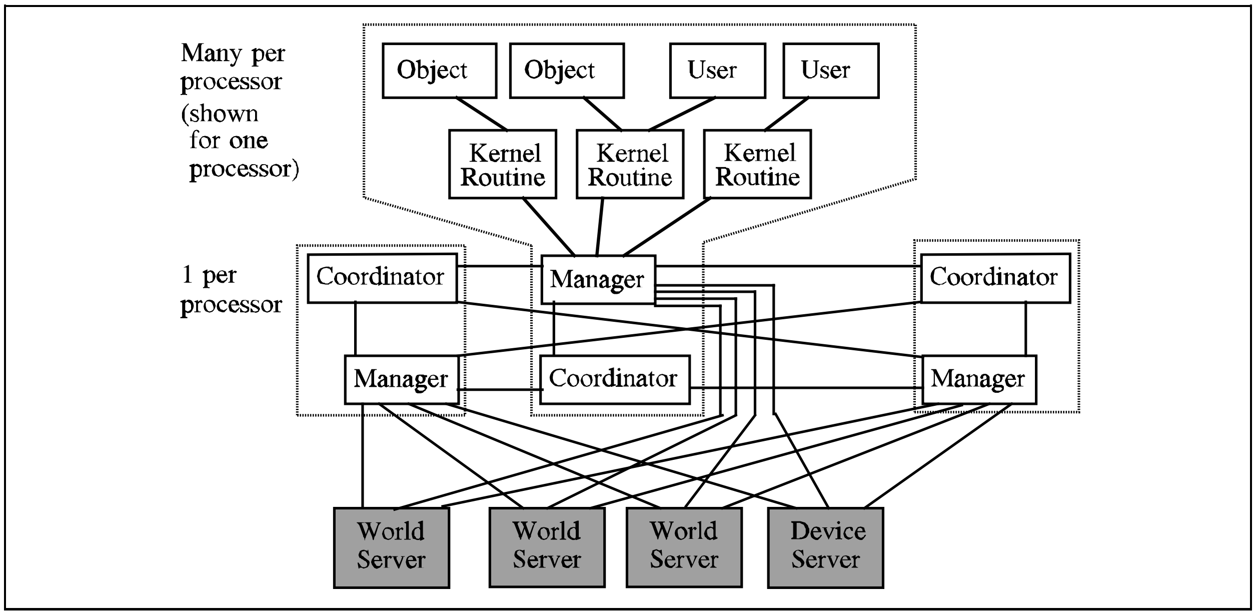


Figure 32 Complete plan of communication layer

The complete communication layer is shown in Figure 32.

5.2.4. The Servers

There are three types of server process used in the system. Each process consists of a loop that repeatedly picks up an incoming request, services it, and returns the result before continuing with the next message. In this way, contention is avoided on the shared resource. Buffers are placed on the incoming links and prevent blocking in the processes that perform the data routing, leading to eventual deadlock. Each message requires an acknowledgement, which allows the buffer size to be limited to one.

The servers present in the system consist of the object coordinators that allow access to objects on remote nodes, the world servers that control access to the object data for each world, and the single device server that prevents contention for the various devices.

5.2.5. Function Implementations

This section describes the implementations of some of the functions described in section 5.2.2. The algorithms in this section are intended to illustrate the manner in which the communication with the worlds is handled. The application programmer uses these functions, and is not required to understand any of the details presented in this section.

The implementation of the functions is intended to be general and not impose restrictions on the manner in which they can be used. It is hoped that they could even be useful for applications not yet envisioned. They are intended to assist in keeping the virtual reality consistent, not to impose restrictions on the way in which a system can be modelled.

An object may be uniquely identified by three different values. These are:

(1) the world that it occupies and the object number in that world,

(2) the node on which its process runs and an object number local that node, and

(3) a unique global identification number allocated when the object is created.

Each object process can be identified by a value of type (2). This needs to be translated into a value of type (1) to access the object data. To access another object, it is most convenient to use values of type (3). The routing table for the communication layer relates all three allowing easy conversion from one form to another. The routing table is kept on all nodes of the kernel, and updated when required.

5.2.5.1. Getting object position and orientation

Most of the functions are involved with transferring data to and from the world servers. They typically consist of a single call to the server. An example of such a function is **GetPosAndOri** that retrieves the position and orientation of an object. This function, together with the corresponding function to set position and orientation, is useful for moving objects around the world.

5.2.5.2. Creating objects

One of the more complicated functions is the one to create new objects. It contains examples of calls to the world server and object coordinators. A pseudocode description is given in Figure 33. The device server is used as a source of object identification numbers. Since only one such server exists, uniqueness of such numbers can be guaranteed.

```
function CreateObject (Attributes, ProcessProcedure, World)

    call device server to get a new global identification number

    for i := 1 to number of processors in kernel do
        call object coordinator on node i to get number of objects on that node.
    end for

    let n be the node with the least objects

    call world server to add Attributes to World

    call object coordinator for each node and update the routing table entry for the
    new object

    call object coordinator on node n to run the ProcessProcedure on that node
```

Figure 33 Pseudocode description of object creation function

This routine also implements the load balancing technique used by the system. This assumes that all object processes are likely to require the same amount of processor time. The objects are distributed so that equal numbers of objects are found on each processor.

5.2.5.3. Moving from one world to another

This is also one of the more complicated functions. It is unlike most of the other functions in that it imposes a predefined policy on how objects may move from one world to another. Most of the other functions are general; policy is defined by the application creator. The rule for moving from one world to another requires that all owned objects must be transferred as well. Each world is intended to be self contained, and ownership across worlds is undesirable. The pseudocode for this function is given in Figure 34.

```
function MoveObject (DestinationWorld)

    call world server for source world to get object data

    call world server for DestinationWorld to add object to that world

    loop
        call world server for source world to get identification number of an
        owned object

        if no owned object exists then
            exitloop
        end if

        MoveOwnedObject (DestinationWorld)
    end loop

    call world server for source world to remove the object from that world

    call object coordinator on each node to update routing tables
```

Figure 34 Pseudocode description of object transfer function

The code for the **MoveOwnedObject** is similar to the code for **MoveObject**. It also contains a recursive call to **MoveOwnedObject** to transfer any objects belonging to the already owned objects.

5.2.5.4. Finding neighbouring objects

Some functions involve a reasonable amount of computation. An example of this is the function to find the distance to the closest object. The approach with functions of this sort is to have the computations included as part of the function. This way the calculations are performed by the object process. The alternative is to have the server perform the calculation. This would both limit the advantage of the parallelism, and tie up the server slowing communication with other object processes.

The **FindClosestObject** function first calls the world server to get the positions of all the objects before calculating the distance to each object and returning the minimum.

5.3. Kernel Performance

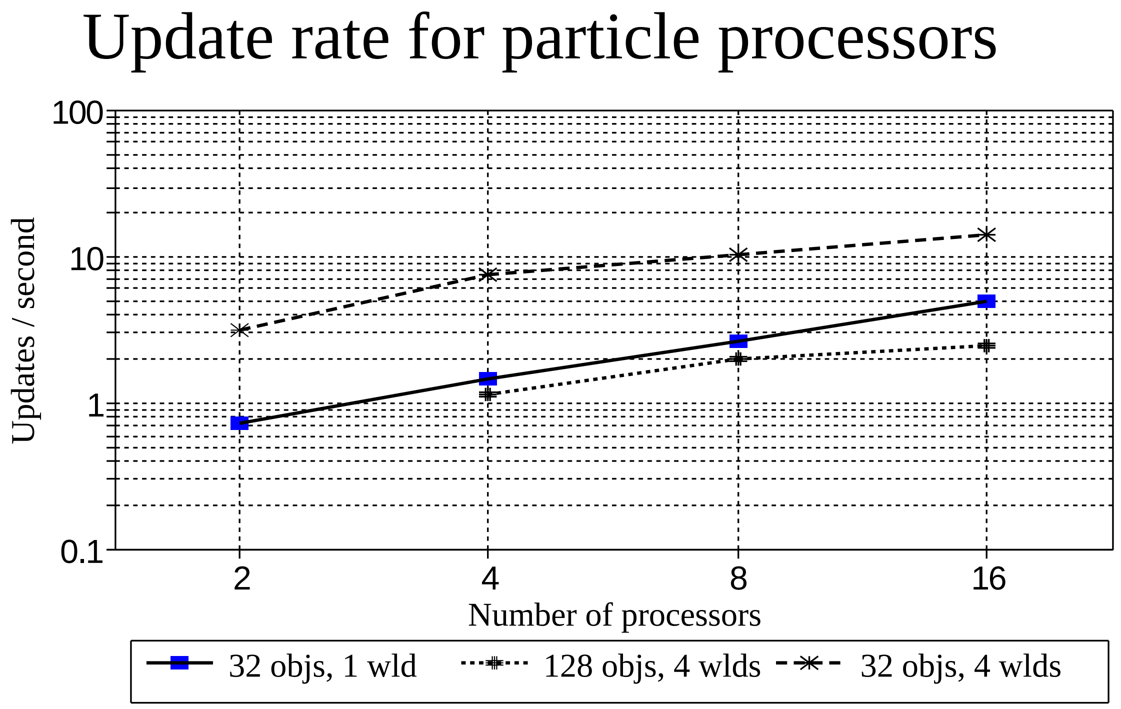
The effect of the parallel implementation of the virtual world kernel was measured for a simple simulation. This simulation modelled gas particles in a sealed container. Each particle could collide with the container walls and with all other particles.

The main loop of the object process controlling each particle is relatively simple. It first looks for the closest object to itself. If that object is closer than a critical distance, then it changes direction to indicate a collision. The particle process then checks the world attributes to locate any boundaries. If collision with a boundary is imminent, then a direction change is made. Finally, the object calculates its new position and stores this in the world database.

Three versions of the simulation were tested. The first used 32 particles and each particle process was given extra work to do so as to just prevent saturation of the communication links in the 16 processor case. The second used 4 worlds, each identical and each containing 32 particles. Extra work was given to each particle process but much less than for the first test case. This simulation was too complex to run on only two nodes. The last simulation again had 4 worlds, but with only 8 particles in each. No extra work was added. Timing figures may be seen in Table XII.

The processors were configured as a hypercube of order $\log_2 N$ where N is the number of processors used. The communications time limits the speedup particularly in the last two test

Table XII Update rate for particle processes



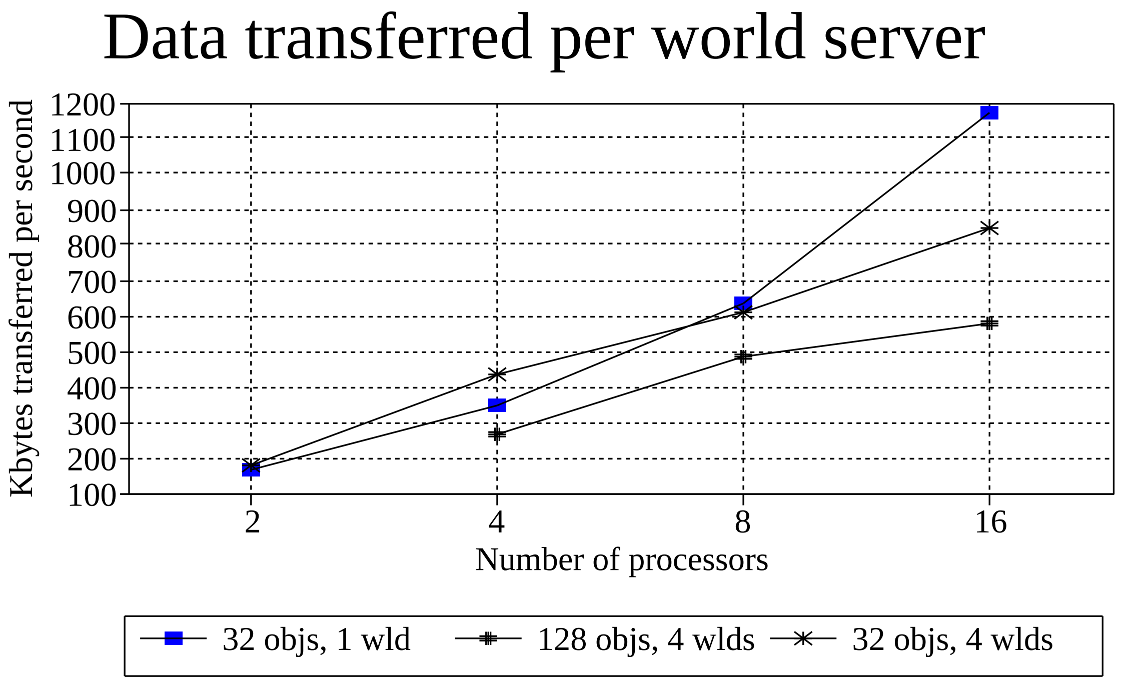
32 objects, 1 world	0.72	1.45	2.65	4.89
128 objects, 4 worlds		1.12	2.04	2.41
32 objects, 4 worlds	3.15	7.53	10.21	14.15

cases where much of the time used by each object process was spent waiting to messages to arrive. The collision detection is a particularly expensive operation with regard to communication, since it requires transfer of about 7 Kbytes per call. An estimate of the amount of data transferred per world server is shown in Table XIII, given that each particle process transfers about 7.5 Kbytes per iteration.

For the first test case, increase in speed is due to the increase in the number of processors available to do the work. For the other simulations, bandwidth seems to be the dominant feature in determining the speedup. In these cases the speedup is closer to $\log_2 N$, which is also the number of links used per transputer in constructing the N processor hypercube.

The greater than linear speedup when going from 2 to 4 processors in the final simulation can be attributed to the presence of multiple world servers on a single node in the 2 processor case.

Table XIII Data transferred per world server in Kbytes/second



32 objects, 1 world	173	349	635	1172
128 objects, 4 worlds		268	490	579
32 objects, 4 worlds	182	436	612	848

Each server is a high priority process and would normally act immediately to an incoming message if alone on the processor. When sharing, it may have to wait for the other server to finish processing before it is able to run. Thus delays are removed when increasing the number of processors in this case.

5.4. Summary

The system described above allows the manipulation of a variety of virtual worlds. All objects are controlled by their own processes. Data for all objects corresponding to a particular world are stored in a single data structure from where they may be accessed. The system attempts to make maximum use of parallelism, by distributing each world and the objects in them onto a network of processors.

A communication system exists to route data between object process and world database. The manner in which the communication system is implemented provides a limit on the speed of communication, but allows a fair scheduling scheme to be implemented. It also would assist in extending the system to include other different processors. No problems have been experienced with this method of implementation, most object processes require delays.

A number of functions exist for use in creating virtual world applications. These functions allow the object processes to read and update the world database entries, both for their own object and other objects that are owned. Functions are also provided to perform other calculations needed by virtual reality applications.

Measurements of the performance of the kernel show that the use of parallelism provides improved performance.

6. Virtual Reality Applications

This section gives some details of the techniques used when creating virtual worlds. Descriptions of the implementation of two virtual reality applications are included.

6.1. Techniques common to all applications.

This section gives the description of functions that were found useful in implementing virtual worlds, but that are a level above the functionality required by the kernel. These routines combine several of the kernel functions into commonly used functions. They are less general than the kernel routines and may be useful only in certain types of virtual realities. In the analogy where the virtual world kernel is compared to a UNIX kernel, these routines would be the equivalent of the C standard libraries.

6.1.1. Movement in a virtual world

Various functions are present to control movement of a user, based on considerations mentioned in the following sections.

6.1.1.1. Movement in a virtual world

Movement is generally more controllable and predictable when performed relative to the frame occupied by the mover. The concepts of up, down, left, right, in and out are familiar, as are turn left, turn right, turn up turn down, turn clockwise, and turn anticlockwise.

To implement this, one must find the displacements and rotations in the untransformed (world) coordinates that will cause unit displacement on- and rotation around- the axes of the transformed (screen) coordinate system. The view is produced by shifting the scene so that the view point becomes the origin, and then rotating it about the origin according to the direction of view.

Let R be the rotation matrix in the viewing transformation.

To produce movement along one of the axes (say the x axis) in the axes of the fully transformed system.

• Invert the matrix to get R^{-1} (inverting quaternions is not complex).

• Project along the axis ([1 0 0] R^{-1}).

• The projection corresponds to the unit displacement in the fully transformed system.

A motivation for this is shown by (4) where the variable OriginalPoint is the transformed point before the displacement occurred.

$$\begin{aligned}
&\textit{For translation } [S_x, S_y, S_z],\\
&\textit{Rotation } \boldsymbol{R} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad \textit{with } \boldsymbol{R}^{-1} = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}\\
&\textit{Transforming } [x, y, z] \textit{ after a displacement}\\
&\textit{of } (\Delta x, \Delta y, \Delta z) \textit{ as derived previously gives:}\\
&\boldsymbol{R} \circ \textit{Displace}[S_x + \Delta x, S_y + \Delta y, S_z + \Delta z]\ ([x, y, z]) \quad \textbf{(4)}\\
&\quad = \boldsymbol{R} \circ [x + S_x + \Delta x, y + S_y + \Delta y, z + S_z + \Delta z]\\
&\quad = \textit{OriginalPoint} + \boldsymbol{R} \circ \textit{Displace}\ [\Delta x, \Delta y, \Delta z]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textit{Distribution})\\
&\quad = \textit{OriginalPoint} + \boldsymbol{R} \circ [1, 0, 0]\ .\ \boldsymbol{R}^{-1}\\
&\quad = \textit{OriginalPoint} + \begin{bmatrix} a & b & c \\ d & e & f \\ h & i & j \end{bmatrix} \circ [A, D, G]\\
&\quad = \textit{OriginalPoint} + [1, 0, 0]
\end{aligned}$$

To produce rotation around one of the axes (say the x axis) in the axes of the fully transformed system:

• Let A correspond to the desired rotation about the axis in the screen coordinate system.

• Create a new rotation matrix R’ = A.R which represents the rotation to be applied in the world coordinate system.

The motivation for this may be seen in (5), where it may be seen that the required rotation occurs on the point in the screen coordinate system.

6.1.1.2. Navigation

The range of the movement tracking device is usually not sufficient to allow the user to reach every part of the virtual world. The volume with which the user can interact must be capable of

Given a point $[x, y, z]$ *to be displaced by* $[S_x, S_y, S_z]$
and transformed by rotation matrix $\boldsymbol{R}$

The point after additional rotation $\boldsymbol{A}$ **(5)**

$$
\begin{aligned}
&= \boldsymbol{R}' \circ \mathit{Displace}[S_x, S_y, S_z]([x, y, z]) \\
&= \boldsymbol{A} \circ \boldsymbol{R} \circ \mathit{Displace}[S_x, S_y, S_z]([x, ,y, z]) \\
&= \boldsymbol{A}\ (\mathit{OriginalPoint})
\end{aligned}
$$

moving to different regions of the virtual world. The technique employed in the walkthrough project at the University of North Carolina at Chapel Hill used buttons to fly forward or backward in the direction of gaze [8]. This is copied, using two gestures with the powerglove to control forward and backward motion.

In order to make this sort of movement efficient, it must meet two requirements: Rapid movement is required when going long distances to avoid unnecessary delays. However, fine control is still needed when interacting with nearby objects.

One solution to this problem involves selecting the target object and moving toward it with speed proportional to the distance from the object [27]. A system of logarithmic movement without selection of a destination object was tested, where speed is increased as the movement gesture is repeated. This does make movement easier, although an upper limit was placed on the velocity to prevent movement becoming unmanageable.

The latency in the system could be reduced at the input device stage by predicting the next input. Reacting before a movement is made will reduce the delay between the time at which the motion actually occurs and the time at which the result is displayed. Position can be predicted from velocity and acceleration. Such a system is already in use in the powerglove driver to detect errors when the glove is too far from where it is predicted to be. Prediction for the purpose of latency reduction has not yet been attempted, however, and remains a future goal.

6.1.2. Implementation of attributes

The kernel provides support for the storing, getting and setting of attributes but ignores the semantics of these attributes. Evaluation of these attributes is left to the application writer. Code to use a number of attributes has been assembled into a library routine.

Attributes have been assigned to all the objects in the world, and to the world itself. For example, when simulating gravity, each object will have an attribute corresponding to its air resistance, and the world will have an attribute giving the acceleration due to gravity.

Attributes are an entirely voluntary feature. It is up to the application writer to ensure that each object in the world evaluates its attributes. This is a satisfactory solution when prototyping virtual reality operating systems, but may become a problem when trying to maintain consistency with multi-user systems. A future version of the system may compel objects to satisfy this requirement.

Some of the attributes used in the applications described in the following sections, are mentioned below.

6.1.2.1. Boundaries

A boundary is a world attribute used to keep an object within a specified area. This is useful if the user is supposed to be inside some object to prevent walking through walls.

The boundary system currently used restricts objects to a cubical volume of space. If more than one boundary is used, the user is constrained to the union of all the volumes. This allows more complicated restriction volumes to be constructed.

6.1.2.2. Portals

With multiple worlds, some form of transport mechanism is necessary. The concept of a portal has been frequently mentioned in virtual reality discussions. By entering some area of space, for example a doorway, the user is magically transported into a new world.

The portals currently implemented consist of a rectangular area of space and an associated destination world. When the motion of an object intersects this rectangle automatic transfer between worlds occurs. Portals are one way, to return the destination world must also be given the appropriate portal attribute.

Portals are invisible, usually objects are placed on them to indicate their location.

6.2. Implementations of virtual worlds

The object control routines are written in C, and linked in with the rest of the system. All the process creation and message passing functions provided by the compiler and the system are available when creating these routines. The only restriction placed on the object routines is that the node on which the object process will run cannot be predicted, it is determined by the load balancing in the kernel.

The standard method of building a world is through creation of a configuration file. This gives details of all the objects in the world, including their object processes, positions, orientations and other attributes. These files are read in when the system starts and the internal representations of the worlds are created. Extra objects may be created at run-time if necessary.

A description of two virtual reality applications that run on this system is given below.

6.2.1. Architecture walkthrough

The architecture walkthrough application was developed to evaluate the effects of some proposed changes to a section of the Computer Science Department at Rhodes University. A corridor was to be removed and replaced with an open-plan layout. The apparent spaciousness of the area was of more interest than representing the details of each room.

The first stage in the development of this application was the construction of two objects, the building as it was originally and the building incorporating the proposed changes. The user was permitted to navigate through each of the buildings, flipping between them with the press of a button. In this way, a comparison of the views from any point could easily be made.

The results of this initial implementation were encouraging. The building consisted only of walls, without furniture. The colour scheme was one chosen to aid in distinguishing the different walls, rather than the conventional white used in reality. In spite of these flaws, everybody familiar with the department soon became comfortable with the illusion once a few familiar landmarks had been pointed out. Once oriented, they could keep track of their position and direction relative to the real building. Even the ability to walk through walls did not discourage anybody. Those unfamiliar with the department tended not to know where they were, but did feel the illusion of moving through a building. The monitor used for this demonstration was a large 19" monitor.

There was a trade off between using a higher resolution and slower frame rate versus a smaller image and faster update. It was found that the larger picture with the greater field of view was the more realistic. The size of the monitor permitted viewers to sit in front of it and have the virtual representation of the ceiling appear above them, with the floor below. The smaller image where everything was in front was less acceptable. The diminished frame rate did not substantially lessen the effect of reality.

The second step of the walkthrough was an attempt to improve the speed and add furniture to the building. The problem with adding furniture was that placing objects inside others caused a failure in the hidden surface removal routines. The hidden surface removal routines work on an object by object basis, only sorting the polygons for a single object at a time. While the hidden surface removal routines could have easily been extended to sort all the polygons in the scene, this would have increased the time taken by the routine. In particular, a new BSP tree would need to be generated every time an object was moved, a process taking several seconds even for simple scenes.

A solution to this problem can be found by considering the building as a boundary to the world (See section 3.2.2.5). The furniture and users are constrained to exist inside building. This solves the hidden surface removal problem for the outer boundaries of the building, but objects in other rooms are still painted over the internal partitions inside the building.

A better solution to this problem, and a method of increasing the frame rate, involves making use of separate worlds for each room. In this way, the internal partitions become boundaries for each world. Only the objects inside a particular room exist in that world with the result that hidden surface removal works satisfactorily. Adjoining rooms may be added as part of the background, allowing a view through the doorway. The worlds are linked by providing a portal attribute to the world, so that moving through a doorway results in automatic transfer to the next world. Thus movement is unchanged. The only difference is the sudden appearance of furniture in a particular room when one enters, and the inability to see inside other rooms without moving into them. These drawbacks are considered a reasonable tradeoff against the obvious advantages.

A number of sample images produced with this application can be seen in Figure 36, Figure 37 and Figure 38.

6.2.2. Virtual Checkers

The virtual checkers application was an attempt to create an application requiring interaction with the world. The board game of checkers (or draughts) requires pieces to be moved around on a board. The corresponding scenario in virtual space has the board defined as part of the background. The pieces and the players are constrained to move about in the area above the board. The pieces are coloured red and blue according to the player to which they belong. A red and a blue cone are present, one on each end of the board. The cone corresponding to the player about to make a move, rotates slowly.

A simulated gravity is used to keep all the pieces at the same level, even when they are moved and placed at different levels. The controlling routine for each piece moves the object if it is not at the correct height. The object is accelerated smoothly to the correct level. For variety, the blue objects simulate an inelastic collision with the board and stop abruptly, the red objects collide elastically and bounce.

The ownership concept is used to prevent contention for an object. The object is initially ownerless and so can only be controlled by the corresponding object process. This process monitors the position of the object and attempts to restore it to the correct level if it is moved. A user may claim ownership of the object, which allows only that user process to move the object. When the user relinquishes ownership, the object process regains control and can move the object to the correct level.

Implementing the rules of a game of checkers is a problem that can be solved without requiring any use of the virtual reality operating system. This aspect of the simulation is thus considered irrelevant to this discussion.

A number of sample images produced with this application can be seen in Figure 35 and Figure 36.

6.3. Summary

This section has outlined some of the higher level routines used in controlling virtual worlds. Details of two virtual realities that were implemented using this system were given. The creation of the realities shows how the various facilities provided by the virtual world kernel may be applied.

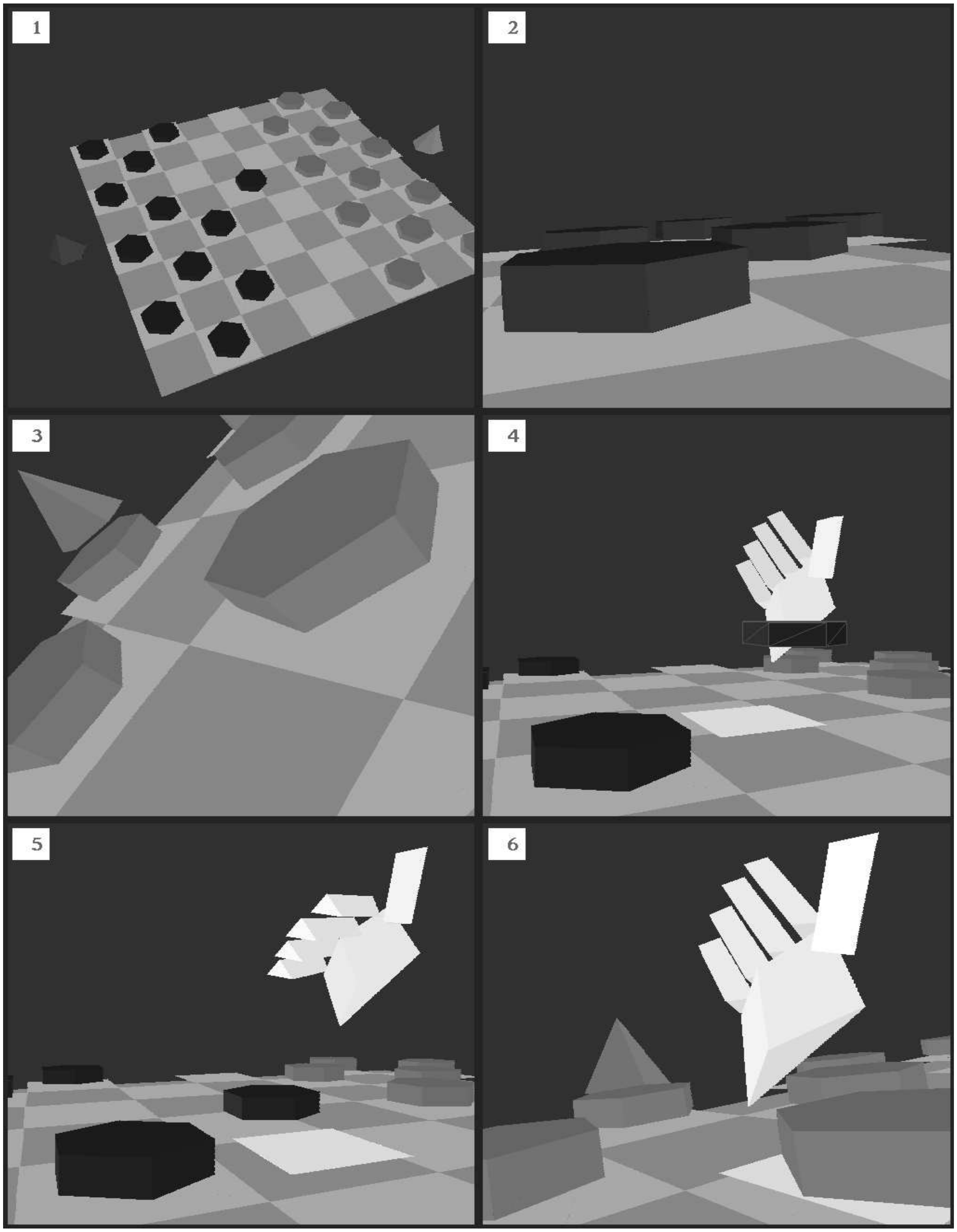


Figure 35 Views of applications (See Table XIV for details)

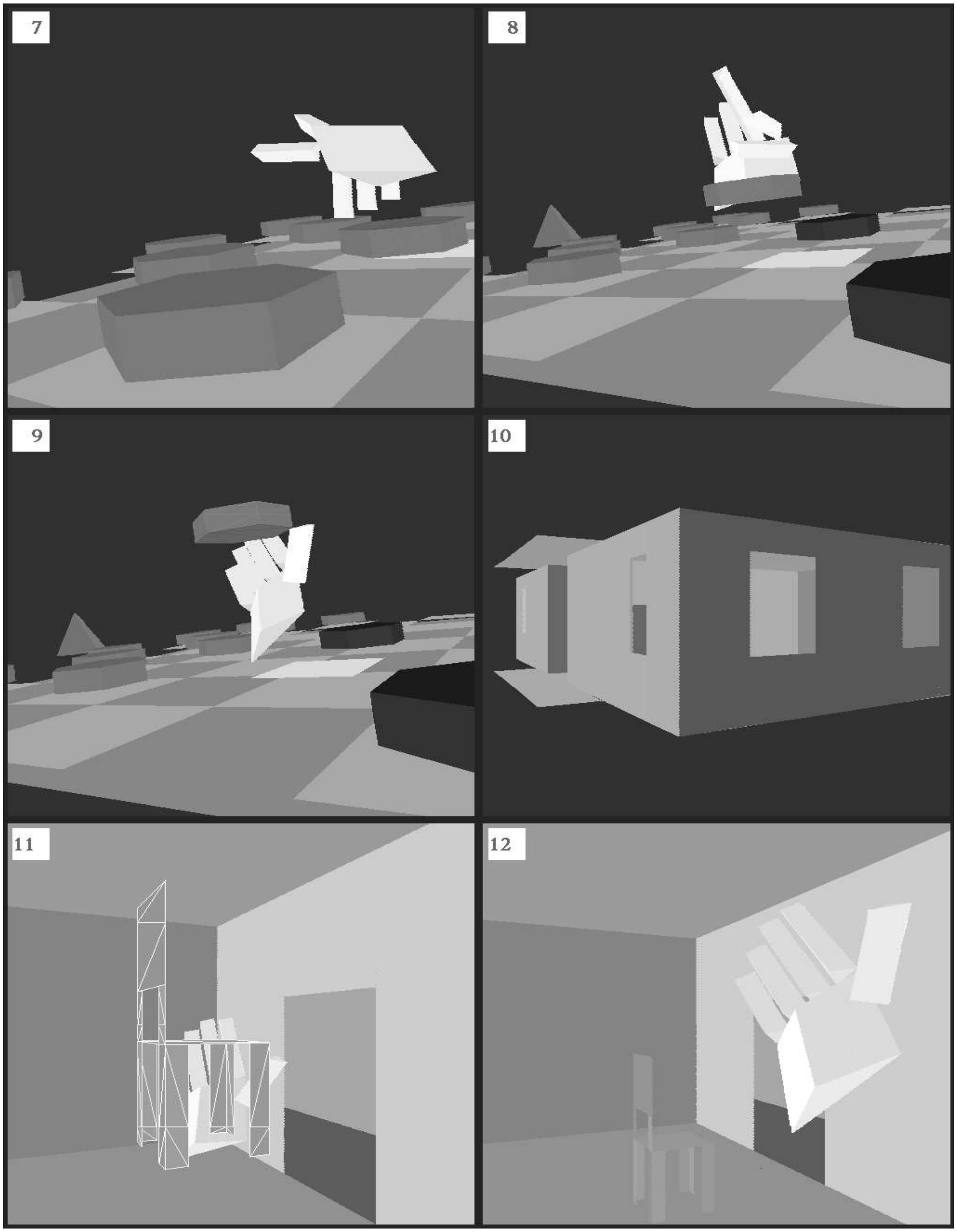


Figure 36 Views of applications (See Table XIV for details)

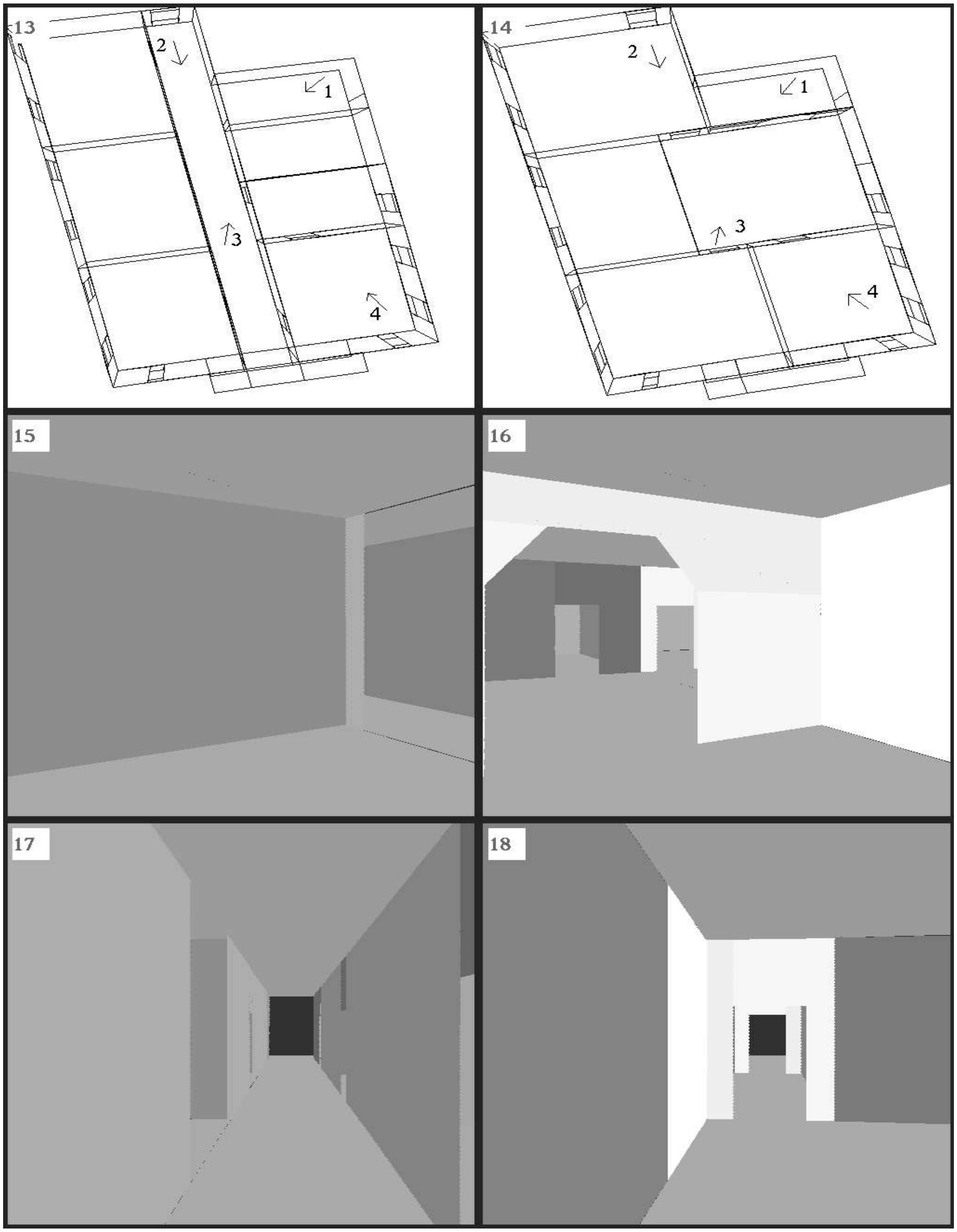


Figure 37 Views of applications (See Table XIV for details)

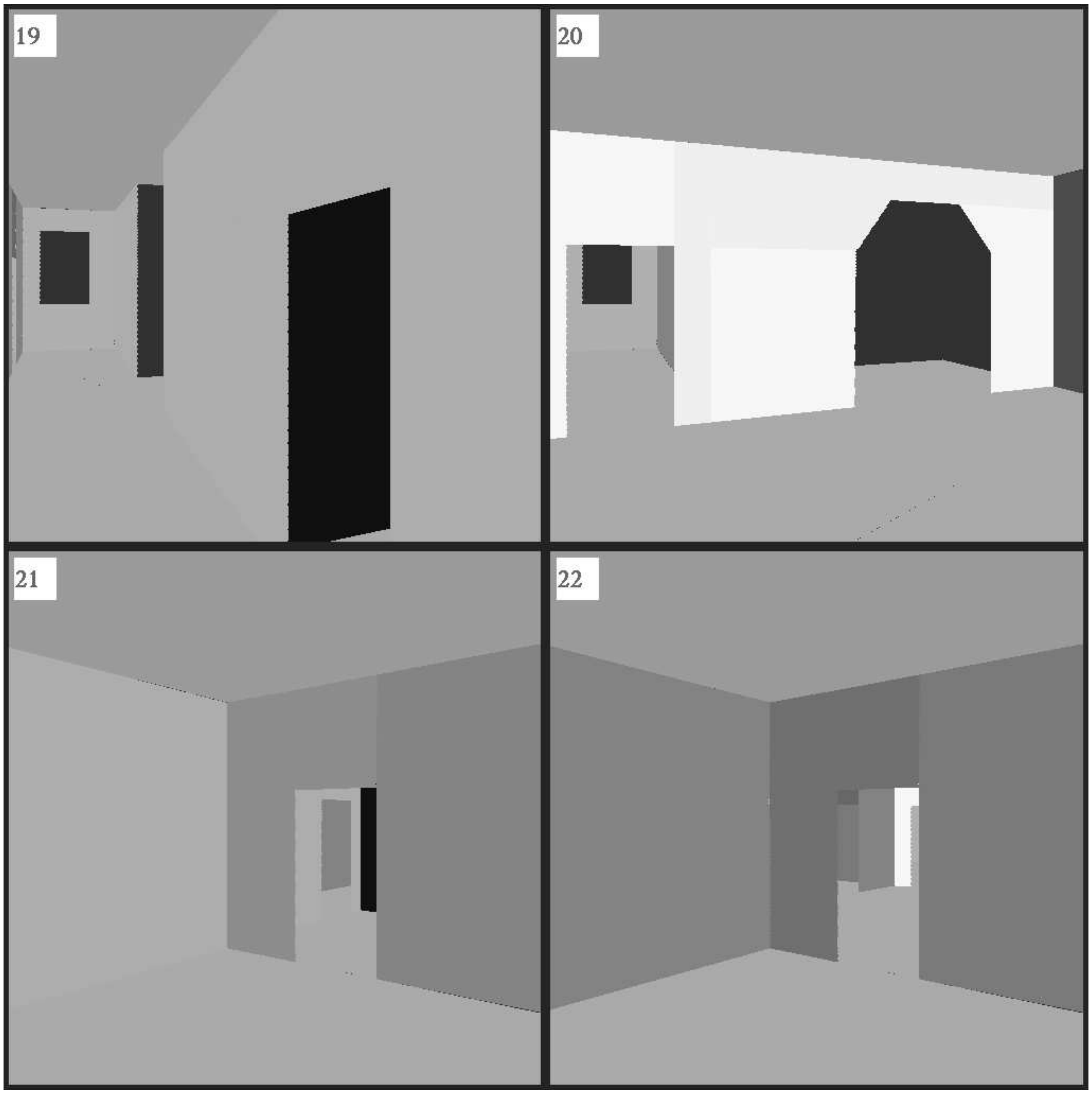


Figure 38 Views of applications (See Table XIV for details)

Table XIV Description of the scenes in Figure 35 - Figure 38

Picture Index	Description
1	View of the complete checkers board
2	View of blue pieces at board level
3	View of red pieces, with red cone in the background
4	Carrying a blue piece, gesture is FLAT
5	Placing a blue piece, gesture is YES
6	View of red pieces, gesture is FLAT
7	Moving amongst the red pieces, gesture is TURN
8	Moving a red piece, gesture is GRIP
9	Dropping the red piece, gesture is FLAT
10	Outside view of walkthrough model
11	Moving virtual furniture in one room, gesture is FLAT
12	Room after the chair has been placed, gesture is FLAT
13	Plan of original building, with viewpoints marked
14	Plan of building with proposed alterations, with viewpoints marked
15	View of entrance of original model, from viewpoint 1
16	View of entrance of modified model, from viewpoint 1
17	View of passage of original model, from viewpoint 2
18	View of the modified model from viewpoint 2
19	View of interior of original model, from viewpoint 3
20	View of the modified model from viewpoint 3
21	View from an office in original model, from viewpoint 4
22	View from the office in modified model from viewpoint 4

7. Comparisons and Conclusions

During the lifetime of this project (January 1992 to July 1993), the field of virtual reality developed from a subject known to a few select researchers to a household discussion topic. Research into the subject over that period has increased the available software from a few systems running on expensive machines to systems that can give reasonable results on an inexpensive PC.

Many of the newer systems are intended to allow the development of diverse applications and are not restricted to one application. These systems are mostly university research projects, not commercial products, and, as a result source code and/or design documentation for these systems is more widely available.

The next few sections present some of the details of these systems, concentrating in particular on the extent to which they compare to the system described in the previous chapters. Practical experience with these systems is limited since many require specialised hardware, both for computation and for interaction. For this reason, the discussion will concentrate mainly on design issues rather than performance issues. To simplify the comparison, the acronym VROS (Virtual Reality Operating System) will refer to the system whose development is the subject of this document.

Conclusions that have been reached as a result of the research into the development of this system are presented at the end of this chapter.

7.1. AVIARY

The Advanced Interfaces Group at the University of Manchester is working on the development of a general framework for advanced interfaces, which they are calling AVIARY [42][35]. This system is intended to support a broad range of Virtual Reality environment.

The model of reality used in developing the AVIARY system is similar to that used in the VROS. Many of the same terms are used, but subtle differences exist in the meanings ascribed to these words. ’Worlds’ are collections of attributes (eg. mass) and laws (eg. gravitation) rather than the data structure containing collections of objects and a few attributes as for the VROS. Applications are distinct processes that manipulate the objects in the world. Many of the applications can exist in a single world, controlling the various objects. An application in the VROS is a more abstract concept consisting of one or more worlds, a collection of objects, and the manner in which they interact.

The problem of supporting the range of features necessary to implement any reality is present in AVIARY as well. The solution implemented is to provide a basic world that may be customised to the purpose required. This results in a conflict between the need to provide assistance to the application writer while still allowing sufficient generality. The solution for the VROS is to provide library routines to handle the most common cases with the hope that few additions will be needed for more esoteric functions. The approach taken for AVIARY is far more rigid. The set of all possible worlds is structured as a hierarchy. The top of the hierarchy contains all possible worlds. Further down these laws are more refined. For example, some worlds may have gravity, while others do not. This information may also be used to restrict the types of objects that may be moved from one world to another. Consistency may be maintained by making sure that the object is capable of obeying the laws of the new world. A system of portals is used to link different worlds.

As with the VROS, objects are also permitted to be bound to processes which control their behaviour. An extra form of control is present, however, from users or applications. This differs from that found in the VROS, where the only interaction between objects is through the ownership concept. A user under AVIARY combines characteristics of objects, in that it also has a visible manifestation, and of applications, in that it is also subject to control beyond that of the physical laws of the world.

AVIARY is segmented into processes that can run in parallel. A communication system similar to that used in the VROS is present to allow communication between processes. The processes in the system consist of:

- Input processes (coinciding with input device drivers).
- Output processes (coinciding with output device drivers).
- A Virtual Environment Manager (where the VROS has multiple world servers).
- Environment Database that provides spatial management such as collision detection.

- Object Servers (corresponding to the object processes).
- Applications to control users or manipulate the virtual environment.

The communication and parallel strategies differ between AVIARY and the VROS. In the VROS objects communicate only with the servers, and support for communication between objects is minimal. The world servers maintain a central data base for each world, and computational workload is limited to the object processes. With AVIARY, the various processes communicate extensively. Each object keeps the data relevant to it, and updates are transmitted when changes occur. Much of the computation is contained in the applications, and in the Environment Database which may limit the degree to which parallelism can be used.

The AVIARY system currently runs on transputers and SUN workstations. The communications system is the module most affected by different architectures. Versions are implemented for transputer networks and SUNs connected by ethernet. Graphics are produced by a hardware renderer.

A strength of the AVIARY design lies in the ability to implement physical laws without excessive involvement on the part of application writer. The object oriented nature of the system with the use of inheritance to control attributes for different worlds is well suited to the design of a support environment for implementing virtual worlds.

7.2. Cyberterm

This system is intended to implement a single virtual world, a cyberspace that allows multiple users to share a common virtual area [34]. The single world is distributed over a number of workstations with each machine acting as a server for a portion of the world. Movement from one ’sector’ of the world to another requires connecting to a different server. A similar system could be implemented in the VROS using multiple worlds to represent the various sectors. The difference, however, lies in the fact that the different processors in Cyberterm are separated by greater geographical distances.

The position of objects is kept by the server database. When an object enters a sector it makes a local copy of this data. Velocity information is used to update the position of other objects, and updates are periodically issued when another object changes direction. This is appropriate where communication is over long distances and over limited bandwidth connections. With the VROS, it is acceptable to poll the server each time due to the fast transputer links.

The servers must issue permission for various actions, such as movement. Private areas of space can be created where rules decided on by the owner are enforced. This is the opposite policy to that taken with the VROS where such rules must be voluntarily obeyed by the objects. The bounding box attribute under the VROS is one way of defining a boundary, but if the object process does not implement the attribute, no further action will be taken. This relaxed attitude is reasonable for a prototype system, but may need to be more rigorously enforced in a commercial system.

The system is currently being implemented on PCs and SUNs. Graphics are produced by public domain rendering libraries such as VOGLE and REND386.

7.3. Distributed Interactive Simulation (DIS)

DIS and its predecessor SIMNET are standards for distributed interactive simulations [26]. They are specifically intended for battlefield simulations. The simulations may involve thousands of objects and take place over a wide area network.

Communication occurs over a relatively low bandwidth medium, such as ethernet. Each host machine controls its own vehicle and keeps track of others by dead reckoning. Each host keeps track of its dead reckoned position and, when this differs significantly from its actual position, it transmits an update to all other hosts.

This approach is quite different to the VROS where all object data for a world is maintained by a single server process.

7.4. DIVE

DIVE (Distributed Interactive Virtual Environment) is a loosely coupled heterogeneous distributed virtual reality system based on UNIX and running over local and wide-area networks using Internet protocols [2][3]. It provides shared memory over a network and controls the sending of signals to processes.

The DIVE system consists of a set of processes each capable of manipulating the world and its objects. The processes consist of visualizer processes that allow users to interact with the world and application processes that operate on objects or introduce applications in the virtual world.

The world consists of a set of objects and various parameters. It is a data structure, as in the VROS. Processes are capable of moving from one world to another by intersecting gateway objects. The implementation of a shared world differs from the server approach used by the VROS. Under DIVE the world is maintained as a replicated database. Each process has its own copy of the structure. Functions are provided to allow updating of entries in each copy for all the processes in the world. If all processes leave a world, the database is discarded.

An event handling system is present in DIVE allowing processes to register for certain types of event. The process can be notified when objects are created, removed, changed, or when interaction between a user and an object occurs. A timer event allows certain tasks, such as object movement, to be called periodically. Objects may be given primitive behaviour by specifying a state machine which performs certain actions on various events. A limited number of actions are possible, including moving, sending signals, and changing appearance.

A number of high level tools are available for creating applications in DIVE. These functions support the selecting and grasping of objects. A vehicles module exists which uses the users actions to control the virtual environment. This is similar to the gesture interpretation under the VROS.

7.5. Division

The ProVision system produced by a Bristol based company, Division, is a virtual reality server that connects to a number of host machines [32] (see also 1.4.3.2). The system is based on T425 and T805 transputers. Various support software is available, including the Distributed Virtual Environment System (DVS).

This system provides real time control and distributed event handling. All activities and environment handling under DVS are performed by processes called actors. Sharing of data between the actors is controlled by DVS.

Parcels of data can be shared between various actors. Each actor makes a local copy of the data.

In order for one actor to update the data, it must send an update request to a special actor, the director, which will then propagate the update to other actors holding that data. Updating can be done in exclusive mode which ensures that all actor processes have consistent copies at one time. The alternative is general mode which is faster, but actors separated by low bandwidth connections may experience delay in receiving the update. This is the opposite approach to that used in the VROS, where only one copy of the data is kept by a world server.

The actors control everything from 3-D input devices to geometry databases. This approach is more general than that used in the VROS, where specialised processes with customised communication interfaces are used for each particular task. The approach taken by DVS may make creating applications more complex, with greater understanding of the system required.

In order to cope with real time constraints, each actor can maintain its own local time. When communicating, the director will compare the different times of each actor and adjust them so that they are in step. This is useful in synchronizing different hardware devices that are operating at different speeds.

Rendering is done in hardware, using Toshiba HSP polygon processors. A renderer process called Paz converts a high level scene description, similar to the world data structure used in the VROS, to the polygon equivalent. Calls to Paz can be made to alter the position, motion and illumination of the objects.

7.6. Minimal Reality (MR) Toolkit

The MR toolkit is a library of functions for supporting the development of Virtual Reality interfaces [17]. It provides support for a number of peripheral devices used for Virtual Reality. It also provides facilities for distributing the Virtual Reality over multiple workstations.

The system provides the basic services. Support for creating virtual reality applications, as found in the VROS, will be provided by high level tools still being developed.

The toolkit consists of three levels of functions. The first level consists of device support functions. These are implemented as a client-server pair, with the server continuously polling the device so the client can have access to the most recent value without delay. The server also performs the low-level processing of the data such as filtering. This approach is the same as is used in the powerglove device driver in the VROS.

The second level converts the data from the devices into a convenient form for the application programmer. This corresponds to the gesture recognition stage in the VROS.

The third level of functions provides services for the application programmer. These include the maintenance of distributed data structures. This level would correspond to the virtual world kernel in the VROS.

The processes in an MR application can have three roles. One must be a master to control the application and start the other processes. There can be a number of slave processes that are used to produce graphical output. There may also be a number of computational processes that receive input from the master and return results to it. Data sharing is done by keeping local copies of the data with each process. The data structures must be periodically synchronised to ensure all processes have the correct values. The application programmer is responsible for specifying when this update occurs. This contrasts with the approach taken in the VROS. Here data is not shared, and the mechanics of updating the single copy of data structure is hidden from the application programmer.

Communication is possible between separate MR applications. The master processes of each application can send device and application-specific data to other master processes. Slave processes must communicate via the master.

7.7. Multiverse

Multiverse is a multi-user X-Windows based Virtual Reality system [16]. The system is based on a client-server model and consists of servers that model the virtual world, and clients that are used for user interfaces. Each client and each server is a separate process, and each may run on a different machine.

The clients consist of a single program that performs roughly equivalent functions to the input and output device drivers under the VROS. The clients are generic, and independent of the world being modelled by the server. They consist of a loop which renders the world, and sends any input from the user back to the server.

A server process is the equivalent of a world server and its corresponding objects under the VROS. The main functions of managing a virtual world are taken care of transparently; the application writer is required to supply only a few functions. These are mostly trivial, the one of interest being the **animateWorld** function that defines the nature of the world. It is called from the main server loop and is usually used to move the objects in the world.

The objects may have special code to control their movement. Objects interact with each other and with the world using an event handling mechanism. These events include MOVE_EVENT that should cause the object to move, COLLISION_NOTIFY_EVENT for when objects have collided and TERM_NOTIFY_EVENT for when an object ceases to exist. The objects are not separate processes as with the VROS, but have to be called as part of the server process. The object control routines are generally invoked when an event occurs which affects them. This sort of inter-object communication must at present be created by the application programmer when using the VROS.

The breakdown is similar to that of the VROS. The principal difference is the degree to which parallel processing is done. Simulation of the world in Multiverse uses a single thread of execution, as opposed to the multiple processes under VROS. However, the machines that would support Multiverse typically contain a single processor, and so creating more processes would be redundant.

7.8. Rend386

Rend386 is a polygon rendering library for 386 and 486 based systems with VGA displays [36]. The system is designed for speed rather than photo-realism. Support for a number of input devices including the powerglove is provided. This library performs similar functions to the input and output device drivers under the VROS.

The renderer is provided with a list of objects and a viewpoint from which to draw them. Transformation calculations are performed using 32 bit integer arithmetic, with accuracy is maintained by multiplying the floating point value of each number by 2^{16} and storing this as the 32 bit integer. Thus the least significant 16 bits store the fractional part of the value. A similar trick is used in line drawing in the VROS (see section 3.2.1.1.3). The coprocessor in the transputers makes the use of floating point values for transformations viable for the VROS.

The objects in Rend386 can have several representations corresponding to different levels of detail. Figures that consist of a hierarchy of objects can also be defined. Objects are stored relative to the parent object in the hierarchy. For example, in a human figure the arms and legs may be made children of the torso object. This makes it easy to animate objects.

Hidden surface removal is done using the Painter’s algorithm. This has recently been supplemented by adding binary trees of splitting planes. This is similar to the BSP Tree hidden surface removal method used by the transputer renderer for VROS, although it is not carried out to the same degree. Only a limited number of splitting planes are used, and objects intersecting the planes are not split, but placed on one side or the other.

A few refinements are made to the polygon rendering. Polygons can be flat shaded as done in the VROS, or painted in a range of shades to simulate a metallic effect. Transparent polygons can be created where alternating rows of dots are used instead of solid colour.

7.9. The Virtual Environment Operating Shell (Veos)

Veos is an environment for creating distributed applications for Unix [11]. It is designed for prototyping distributed Virtual Reality applications.

The processes required to implement a virtual environment are known as entities and can be distributed across a number of Unix workstations. A data type known as the ’grouple’ is used as the standard data structure. The grouple is an extension to the ’tuple’ used in the Linda programming paradigm. Grouples consist of nested tuples. Lisp is used as the programming interface to Veos.

Each Veos entity consists of a distinct Unix process that controls interpretation of the task written in Lisp. Each entity has associated grouplespaces for which pattern matching facilities are provided. Asynchronous message passing of grouples between entities is supported.

The use of interpreted Lisp makes the system flexible and easy to use. It also allows evaluation of program stubs passed as messages. This however will often limit the performance of the system.

The Veos system provides support for general distributed applications. Creating a Virtual Reality application still requires a great deal of work on the part of the programmer. The pattern matching facilities for the grouplespaces can assist in the modelling of virtual worlds.

Even though the grouplespaces may suggest use of shared memory, process communication still involves message passing.

7.10. Conclusion

A virtual reality system has been implemented that provides a framework for the development of various virtual environments. The transputer-based system does not require the use of dedicated hardware, yet still achieves a reasonable performance.

The input/output routines are separated from the virtual world simulation functions. This allows new hardware to be quickly and easily integrated with the system. Hardware details, and the visual appearance of objects, are isolated from the modelling of the realities.

The rendering system is capable of displaying polyhedral representations of virtual worlds. A number of alternative techniques for the rapid generation of the images on a single processor were tested and compared. A hidden surface removal technique using BSP trees was found to have the best performance in terms of speed and reliability. The use of parallelism was investigated, and a number of different decomposition techniques were tested. These differed in the speed at which images were produced and the interaction latency involved. In the best case, where almost linear speedup was obtained, a world containing over 250 polygons was rendered at 32 frames/second. The maximum frame rate was obtained with the parallel decomposition technique which renders separate frames on different processors. The technique generally used, however, renders different strips of the image on different processors. This technique yielded the best results in terms of the tradeoff between speed and latency. The factor that most limited the speedup was the bandwidth of the transputer links. Faster communication is essential if real-time images are to be produced.

Intuitive interaction with the virtual world is provided by interpreting gestures made by the user while wearing a powerglove. The limitations of the input device were overcome by providing visual feedback, and by requiring confirmation of certain gestures.

The virtual world kernel provides a development platform for virtual reality applications. It provides intrinsic support for multiple worlds existing concurrently, and for multiple users interacting with these worlds. The kernel makes effective use of multiple processors. The system can also make use of devices on separate machines communicating via ethernet. The design of the kernel with its distinct communication layer could allow the system to be extended to operate on other machines.

A means is provided of attaching arbitrary attributes to objects. This permits a wide range of applications to be created. Numerous functions exist to support interaction with a virtual world. Two test applications were successfully implemented using the facilities provided by the kernel.

The system was successfully demonstrated to a number of people with varying degrees of computer literacy. There was an intuitive understanding of the virtual world; concepts such as up and down were translated automatically to the computer model. Interaction with the virtual world by using the virtual hand controlled by the powerglove also required little explanation. The control gestures were easily learnt after they had been demonstrated once or twice. The greatest challenge experienced was that of getting the glove to produce the correct gesture, a difficulty which was noted in only a few of the test subjects. This could, however, be ascribed to the glove not fitting properly in some cases.

A powerful virtual reality system has been developed which runs on a general purpose parallel computer, and which does not need expensive or dedicated hardware. The system is general purpose and is suitable for the creation of diverse virtual reality applications. New applications can be added with ease; with this system, virtual worlds have been created in a matter of minutes. Models of various physical phenomena have taken only a few hours to implement.

Successful use has been made of the parallel processing facilities offered by the transputer. The bottleneck in both the graphics system and the virtual world kernel was the limited bandwidth of the inter-processor communication channels. Greater bandwidth will be more important than faster processors in future systems.

A comparison with other recently developed general virtual reality systems showed a number of common facilities, although these have been implemented in different ways. The system that was described in this thesis lacks facilities for inter-object communication at present, this must be done by the application programmer. However, it makes greater use of parallelism than any of the others, and achieves a more even distribution of the computational load.

8. References.

[1] Airey, J.M., **Rohlf**, J.H., and **Brooks**, F.P. Jr., "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments", *ACM SIGGRAPH Symposium on Interactive 3D Graphics*, **24**(2), March 1990, 41-50.

[2] Andersson, M., **Carlsson**, C., **Hagsand**, O., and **Ståhl**, O., "DIVE - The Distributed Interactive Virtual Environment Tutorials and Installation Guide", Technical Report, Swedish Institute of Computer Science.

[3] Andersson, M., **Carlsson**, C., **Hagsand**, O. and **Ståhl**, O., "DIVE - The Distributed Interactive Virtual Environment Technical Reference Manual", Technical Report, Swedish Institute of Computer Science.

[4] Atkin, P., and **Packer**, J., "High Performance Graphics with the IMS T800", INMOS Technical Node 37, INMOS Limited.

[5] Bajura, M., **Fuchs**, H., and **Ohbuchi**, R., "Merging Virtual Objects with the Real World: Seeing Ultrasound Imagery with the Patient", *ACM SIGGRAPH '92*, **26**(2), July 1992, 203-210.

[6] Blanchard, C., **Burgess**, S., **Harvill**, Y., **Lanier**, J., **Lasko**, A., **Oberman**, M., and **Teitel**, M., "Reality Built For Two: A Virtual Reality Tool", *ACM SIGGRAPH Symposium on Interactive 3D Graphics*, **24**(2), March 1990, 35-36.

[7] Blinn, J.F., "A Trip Down the Graphics Pipeline: Line Clipping", *IEEE Computer Graphics & Applications*, **11**(1), January 1991.

[8] Brooks, F.P. Jr., **Airey**, J., **Alspaugh**, J., **Bell**, A., **Brown**, R., **Hill**, C., **Nimscheck**, U., **Rheingans**, P., **Rohlf**, J., **Smith**, D., **Turner**, D., **Varshney**, A., **Wang**, Y., **Weber**, H., and **Yuan**, X., "Final Technical Report : Walkthrough Project", University of North Carolina at Chapel Hill Technical Report TR92-026, June 1992.

[9] Brutzman, D.P., **Kanayama**, Y., and **Zyda**, M.J., "Integrated simulation for rapid development of Autonomous Underwater Vehicles", *IEEE Oceanic Engineering Society Autonomous Underwater Vehicle (AUV) 92 Conference*, June 4-5 1992.

[10] Bryson, S., "Virtual Reality Takes on Real Physics Applications", *Computers in Physics*, **6**(4), July/August 1992, 346-352.

[11] Coco, G.P., "The VEOS project : VEOS 2.0 Tool Builders Manual", available for anonymous ftp from milton.u.washington.edu as public/veos/veos.tar.Z.

[12] Cottingham, M.S., "Interpolative hidden surface removal method for polyhedra", *The Australian Computer Journal*, **21**(2), May 1989.

[13] Eglowstein, H., "Reach out and touch your data", *Byte*, **15**(7), July 1990.

[14] Foley, J.D., and **Van Dam**, A., "Fundamentals of Interactive Computer Graphics", Addison-Wesley, 1984.

[15] Fuchs, H., **Kedem**, Z.M., and **Naylor**, B.F., "On Visible Surface Generation by a A Priori Tree Structure", *Computer Graphics*, **14**(3), July 1980.

[16] Grant, R., Multiverse description and sources, available by anonymous ftp from ftp.u.washington.edu as public/virtual-worlds/multiverse-1.0.2.tar.Z.

[17] Green, M., "Minimal Reality Toolkit Version 1.2 : Programmer's Manual", Technical Report, University of Alberta, Edmonton, Alberta.

[18] Hearn, D. and **Baker**, M.P., "Computer Graphics", Prentice-Hall (New Jersey), 1986.

[19] Hill, F.S. Jr., "Computer Graphics", Macmillan Publishing Company (Singapore), 1990.

[20] Hodges, L.F., "Tutorial: Time-Multiplexed Stereoscopic Computer Graphics", *IEEE Computer Graphics & Applications*, **12**(3), March 1992.

[21] Holloway, R.L., "Viper: A Quasi-Real-Time Virtual-Worlds Application", University of North Carolina at Chapel Hill Technical Report TR92-004, December 1991.

[22] INMOS Ltd, "Occam 2 Reference Manual", Prentice Hall International Ltd (Hertfordshire), 1988.

[23] INMOS Ltd, "IMS T800 engineering data", The Transputer Databook, Second Edition, Redwood Burn Ltd (Towbridge), 1989.

[24] Kuipers, J., "Apparatus for generating a nutating electromagnetic field", United States

Patent 4,017,858.

[25] Liang, Y.-D., and **Barsky**, B.A., "An analysis and algorithm for polygon clipping", *Communications of the ACM*, **26**(11), 1983, 868-877.

[26] Locke, J., "An Introduction to the Internet Networking Environment and SIMNET/DIS", available by anonymous ftp to sunee.uwaterloo.ca as pub/vr/documents/DISIntro.ps.

[27] Makinlay, J.D., **Card**, S.K., and **Robertson**, G.G., "Rapid Controlled Movement Through a Virtual 3D Workspace", *Computer Graphics*, **24**(4), August 1990.

[28] Naylor, B., **Amanatides**, J. and **Thibault**, W., "Merging BSP Trees Yields Polyhedral Set Operations", *Computer Graphics*, **24**(4), August 1990.

[29] Newman, W.M., and **Sproull**, R.F., "Principles of Interactive Computer Graphics", Magraw-Hill (Tokyo), 1979.

[30] Pausch, R., "Virtual Reality on Five Dollars a Day", *Proceedings of the ACM SIGCHI Human Factors in Computer Science Conference*, New Orleans, April 1991.

[31] Pletincks, D., "The Use of Quaternions for Animation, Modelling and Rendering", New Trends in Computer Graphics: Proceedings of CG International '88, Springer-Verlag, 1988.

[32] Pountain, D., "ProVision: The Packaging of Virtual Reality", *Byte*, **16**(10), October 1991.

[33] Rost, R.J., "OFF - A 3D Object File Format", Digital Equipment Corporation Technical Report, October 1989.

[34] Snoswell, M., "Overview of Cyberterm, a Cyberspace Protocol Implementation", available by anonymous ftp to sunsite.unc.edu as pub/academic/computer-science/virtual-reality/papers/Snoswell.Cyberterm.

[35] Snowdon, D.N., **West**, A.J., **Howard**, T.L.J., "Towards the next generation of Human-Computer Interface", *Informatique '93: Interface to Real and Virtual Worlds,* March 24 -26 1993, 399-408.

[36] Stampe, D. and **Roehl**, B., "REND386 - A 3-D Polygon Rendering Package for the 386 and 486 : LIBRARY Documentation Version 4.01 - September 1992", available for anonymous ftp from sunee.uwaterloo.ca as pub/rend386/devel4.zip.

[37] Sturman, D.J., "Whole-hand Input", PhD thesis, Massachusetts Institute of Technology.

[38] Sutherland, I.E., **Sproull**, R.F., and **Schumaker**, R.A., "A Characterization of Ten Hidden-Surface Algorithms", *Computing Surveys*, **6**(1), March 1974.

[39] Sutherland, I.E., and **Hodgman**, G.W., "Reentrant polygon clipping", *Communications of the ACM*, **17**(1), 1974, 32-42.

[40] Thibault, W.C., "Set Operations on Polyhedra Using Binary Space Partitioning Trees", *Computer Graphics*, **21**(4), July 1987.

[41] Vatti, B.R., "A generic solution to polygon clipping", *Communications of the ACM*, **35**(7), 1992, 56-63.

[42] West, A.J., **Howard**, T.L.J., **Hubbold**, R.J., **Murta**, A.D., **Snowdon**, D.N., **Butler**, D.A., "AVIARY - A Generic Virtual Reality Interface for Real Applications", An invited paper for "Virtual Reality Systems" May 1992 sponsored by the British Computer Society.

[43] Whitman, S., "Parallel Graphics Rendering Algorithms", *Proceedings of the 3rd Eurographics Workshop on Rendering*, May 1992.

[44] "Logical Systems C for the Transputer: Version 89.1 User Manual", Logical Systems.

[45] "RB2 Model 2 Virtual Reality System", VPL Promotional Document, VPL Research Inc, 1991.

[46] "Research Directions in Virtual Environments", Report of an NSF Invitational Workshop, *Computer Graphics*, **26**(3), August 1992.

[47] "The Helios Operating System", Perihelion Software Ltd, Prentice Hall International Ltd (Hertfordshire), 1989.