1996

The Design and Implementation of a Toolkit for the Creation of Virtual Environments

Jesse Kinross-Smith
Edith Cowan University

Edith Cowan University

Copyright Warning

The Design and Implementation of a Toolkit for the Creation of Virtual Environments

J. Kinross-Smith
B. Sc. (Computer Science) Hons
1996

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

THE DESIGN AND IMPLEMENTATION OF A TOOLKIT FOR THE CREATION OF VIRTUAL ENVIRONMENTS

By

Jesse Kinross-Smith

A Thesis submitted in partial fulfilment of the requirements for the award of

Bachelor of Science (Computer Science) Honours

at the Faculty of Information, Science and Technology,
Edith Cowan University.

Date of Submission: 31st March, 1996

Field of Research Code/Socio-Economic Objective/Research Type

059903 Virtual Reality and Related Simulation/100101 Application tools and system utilities/Applied Science

TABLE OF CONTENTS

ABSTRACT **4**

DECLARATION **5**

ACKNOWLEDGMENTS **6**

1 INTRODUCTION **7**

2 PROBLEM **10**

2.1 THE BACKGROUND OF THE PROJECT 10

2.2 THE SIGNIFICANCE OF THE PROJECT 10

2.3 THE PURPOSE OF THE PROJECT 11

2.4 PROJECT OBJECTIVES 11

2.5 DEFINITION OF TERMS 13

3 LITERATURE REVIEW **16**

3.1 GENERAL LITERATURE 16

3.2 LITERATURE ON PREVIOUS FINDINGS 18

3.3 SPECIFIC STUDIES SIMILAR TO THE CURRENT PROJECT 20

3.4 LITERATURE ON METHODOLOGY 21

4 ANALYSIS AND DESIGN **23**

4.1 DESIGN OVERVIEW 23

4.2 SYSTEM MODEL 24

4.3 SYSTEM INTEGRATION PLAN 26

4.4 USER INTERFACE DESIGN 27

4.5 SUMMARY 34

5 IMPLEMENTATION **35**

5.1 IMPLEMENTATION OVERVIEW 35

5.2 DEVELOPMENT STANDARDS AND PROCEDURES 37

5.3 SYSTEM STRUCTURE 39

5.4 EVALUATION AND TESTING 49

5.5 SUMMARY 49

6 FINDINGS **50**

6.1 DISCUSSION 50

6.2 APPLICATION AND USE OF THE PROJECT 54

6.3 FURTHER RESEARCH 55

6.4 CONCLUSION 56

REFERENCES **58**

APPENDIX: IMPLEMENTATION SOURCE CODE **60**

LIST OF FIGURES

FIGURE	DESCRIPTION	PAGE
2.1	RELATIONSHIP HIERARCHY FOR REND386	13
3.1	GENERATIONS OF USER INTERACTION	20
3.2	PROTOTYPING MODEL	21
4.1	CURRENT SYSTEM MODEL FOR VR-386	24
4.2	SYSTEM INTEGRATION MODEL	26
4.3	POWERGLOVE GESTURE TABLE	29
4.4	ACTIONS AND EQUIVALENTS	30
4.5	TRANSIT POINTS IN THE VIRTUAL CONSTRUCTION SITE	31
4.6	PALETTES - EXTENSIONS TO THE USER'S WORKSPACE	32

ABSTRACT

Virtual Reality is a field that is steadily increasing in popularity and interest. New developments in both hardware and software have empowered developers with new devices allowing faster and better quality interaction with virtual environments. However, the emphasis of research in virtual environments has been more concerned with development of new display and input devices, as opposed to the investigation of different methods of interaction that a three-dimensional environment offers.

This project designs and implements a three-dimensional, interactive, virtual environment development system upon an existing three-dimensional rendering engine. The aim of the project is to allow users to generate virtual environments with ease through a simple and intuitive user interface.

Support for a gesture-based input device has been provided, as well as for more conventional two-dimensional input devices such as the mouse and joystick. By catering for a variety of input devices, various different forms of input have been examined in terms of their strengths and weaknesses.

It is through the use of techniques developed throughout this project that designers of virtual environments may go about their work with greater efficiency and simplicity, allowing users to concentrate on the development of the environment, rather than being limited by the tools they possess.

DECLARATION

I declare that this thesis does not incorporate without acknowledgment any material previously submitted for a degree in any institution of higher education, and that, to the best of my knowledge and belief, it does not contain any material previously published or written by any other person except where due acknowledgment is made.

Signature:

Date: 31st March, 1996

ACKNOWLEDGMENTS

I would like to thank the following people for making this work possible.

Dr Thomas O'Neill, whose invaluable assistance enabled me to finish this thesis, and who gave me his time and patience to help and guide me towards completing the project.

Dr James Cooper, who encouraged me in designing and developing the software portion of the project.

My parents, Russell and Shirley, and my sister Grace, for supporting me and giving me the space I needed, and helping me throughout the whole course.

My friends, who offered advice, support, and that nagging 'Shouldn't you be doing some work on your thesis?' when I needed it.

Sharon, who supported me and kept me going.

Special thanks to Darren, Scott, Chris, and Andrew for their comments, criticisms, opinions and ideas.

1 INTRODUCTION

Today, researchers in the field of Computer Science investigate many interesting topics that are expected to enrich human endeavour. To this end, the provision of a user-friendly human-computer interface enhances the acceptance and viability of the researcher's efforts. A topic that attracts considerable attention is that of virtual reality, which permits a user to interact with a computer simulated environment.

For the reader unfamiliar with this area of Computer Science, some initial insight may be gained through the following dictionary definitions:

> **Virtual**: *"being in essence or effect but not in fact"*
>
> (Webster's Dictionary, 1981)

> **Environment**: *"act of surrounding; surroundings; external conditions in which a person or organism lives"*
>
> (Penguin English Dictionary, 1974)

> **Metaphor**: *"a figure of speech in which a word or phrase literally denoting one kind of object or idea is used in place of another to suggest a likeness or analogy between them (as in the ship plows the sea): an implied comparison"*
>
> (Webster's Dictionary, 1981)

The aim of this project is to produce an intuitive, user-friendly, and effective development system for the creation of virtual environments in three-dimensional space. The realisation of this aim is strongly dependant on the notion of metaphors, whose concepts are employed in the design of the accompanying user-interface. This is not the first usage of metaphors in user-interface design; indeed, they are intrinsic to the Macintosh Operating System (Mountford, 1994) which has been commercially available for more than a decade. This system provides a very intuitive environment for learning and development; hence its widespread acceptance and promotion in educational establishments is understandable. Some of the intuitive metaphors employed by the Macintosh Operating System to simulate real objects are its virtual desktop, files, folders and trashcan. Furthermore,

within the desktop simulation, the provision of layered windows suggests the concept of sheets of paper that might be stacked upon the desktop (Mountford, 1994).

Humanity's habitat occupies three-dimensional space, therefore a virtual environment should also exist in three-dimensions if it is to provide a complete human-oriented service. To achieve a high level of service, special care needs to be exercised when designing appropriate interaction methods for the user-interface. With this foremost in mind, the project has been directed towards the experimentation with new interaction methods for three-dimensional environments.

In association with the conventional forms of interaction via the keyboard and mouse, the user of a gesture-based input device for interaction with the virtual environment is explored. Specifically, the exploration is limited to a device called the PowerGlove, which was developed for interactive use with the computer. The PowerGlove is a more economic alternative to the Dataglove, which is prevalent in virtual reality applications and research. Consisting of a number of resistive-ink flex sensors attached to a lycra glove, the PowerGlove provides feedback on the relative dispositions of the finger joints. Furthermore, in conjunction with information from acoustic trackers mounted on the back of the lycra glove, the PowerGlove provides the means of accurately locating its position in three-dimensional space (Sturman & Zeltzer, 1994).

The PowerGlove offers a three-dimensional form of interaction superior to the more conventional two-dimensional forms (e.g. keyboard and mouse) and it allows direct mapping of points to a three-dimensional environment. Thus, the combination of hand gestures and the ability to pinpoint location allow a more intuitive interaction with such environments.

Though physical feedback (i.e. the ability to feel objects) from the environment is not possible through the PowerGlove, feedback may be provided visually in order to permit the user to employ natural gestures, such as grabbing an object by "closing one's hand over it". This type of gesture is routine for a user and it demonstrates the intuitiveness of the interaction with the virtual environment.

The realisation of the aforementioned aim, i.e. a three-dimensional virtual environment development system, is named the Virtual Construction Site (VCS) and it is essentially a system that may be used to create, in a bottom-up fashion, such environments taking advantage of a collection of primitive three-dimensional building blocks.

The discussion to this point has been in general terms; consequently, Chapter Two provides greater detail on the background of the project, outlines its significance, states its purpose and objectives, and defines the fundamental terminology.

Chapter Three contains a literature review of research into virtual environments, user-interface design and gesture-based interfaces. The review covers general literature in these areas, describes previous findings, discusses specific studies similar to the current project, and includes commentary on the methodology employed.

Chapter Four treats the analysis and design of the project. Specifically, there is a design overview (incorporating the goals, objectives, etc), a detailed system model, a system integration plan and an investigation into the design of the user interface. This chapter ends with a summary of its achievements in this phase.

Chapter Five describes the implementation in intimate detail. Beginning with an overview of the requirements for hardware, software, language, development tools, etc., then continuing with the development standards and procedures, a discussion on the structure of the system, and the evaluation and testing performed to ensure quality. It concludes with a concise résumé of attainment in these regards.

Chapter Six discusses the findings, suggests potential application areas, and outlines the implications for future research in this field. Finally, it draws conclusions about the overall success and achievements of the project.

The document incorporates the implementation source code as an appendix.

2 PROBLEM

2.1 The Background of the Project

Virtual Reality (VR) may be defined as "an advanced human-computer interface that simulates a realistic environment and allows participants to interact with it" (Latta & Oberg, 1994, p. 23). A virtual environment is the end result of this process, being an environment which simulates the real world as close as is needed for the particular problem at hand.

As Richard Quinnell stated (Quinnell, 1993, p. 48) in his article on virtual reality design software: "It is software, not specialised hardware, that lies at the real core of VR". To develop an effective virtual reality system, emphasis needs to be placed upon the software for designing the environments to ensure that it is not an area of contention in the development of virtual environments.

One of the main reasons behind the decision to support a gesture-based device is to provide a more intuitive and functional interface for the design of virtual environments. A gesture-based input device, such as the PowerGlove, allows a task to be performed in three-dimensional space more intuitively than it would be under a two-dimensional oriented input device. The enhanced intuition provided by a gesture-based input device is due largely to a "direct mapping between gesture and the manipulated object" (Kurtenback & Hulteen, 1994, p. 311).

Care must be taken in designing a gesture-based interface, however, as "[The mixing of metaphors] presents inconsistent information to the user, and can ultimately prove disastrous. An example from VR is using a glove to both steer in 3d and to select options from a 2d menu". (McGuinness & Meech, 1992, p. 3).

2.2 The Significance of the Project

The successful design and development of any product relies on the use of effective and efficient tools. In the field of virtual reality, there are a very

limited number of tools for building virtual environments that do not require expensive hardware and software. Users that cannot afford these costs have to turn to freeware packages like REND386 or one of its derivatives (AVRIL or VR-386) in order to do their work. The interfaces to these packages, being text-based, tend to discourage anything other than casual use, limiting any benefits that might be achieved through the construction of virtual environments to that of the larger budgeted projects which may afford the more expensive packages.

2.3 The Purpose of the Project

By designing a three-dimensional interactive virtual environment development system on top of a package like VR-386, users may generate virtual environments with ease and simplicity, allowing them to concentrate on the problem at hand, and thereby increasing their effectiveness in producing virtual environments. There are a few similar tools available (such as Sense8's WorldToolKit and Autodesk's Cyberspace Development Kit); however, these tools are very expensive which precludes anything other than commercial use. By producing a system that allows the design and construction of virtual environments on a low-end scale, it is possible for a larger group of users to develop products and perform research in the field of virtual reality.

2.4 Project Objectives

In order to create a virtual environment development system, the project has been broken up into a number of distinct objectives to be achieved. These are outlined below:

Software

- **Virtual Construction Site (VCS)**

  The Virtual Construction Site is the name given to the project, and consists of a three-dimensional interactive environment for the creation of virtual environments that offers the following:

- The means by which the user may construct a virtual environment using a consistent and intuitive interface.
- Support for a gesture-based interface, as well as a conventional interface for traditional forms of input such as the mouse, keyboard, and joystick.
- The ability to select simple three-dimensional objects from a defined set, allowing the user to incorporate these into the virtual environment being created.
- Facilities for storing commonly used objects, allowing them to define a set of common objects that they may reuse to create their environment.

Hardware

- **PowerGlove Conversion**

  In order to utilise a PowerGlove on a IBM PC compatible platform, modification of the PowerGlove is necessary. The modification was performed using the information given in Englowstein's BYTE article (Englowstein, 1990), and involved rewiring the PowerGlove cable so that it uses a 25-pin parallel port connector instead of the standard Nintendo socket. A source of power became an issue, as the PC parallel port offers no voltage lines, however, through utilisation of the 15-pin game port found on many personal computers, this problem was solved.

Theoretical

- **Study of three-dimensional interaction techniques**

  In developing an interface for a three-dimensional environment, one has been able to explore different metaphors for navigation, object manipulation and interaction within the environment. Experimentation and feedback are key principles in designing successful user interfaces, as stated by Myers, "the only reliable way to generate quality interfaces is to test prototypes with users and modify the design based on their comments" (Myers, 1989).

2.5 *Definition of Terms*

Below are the definitions for a number of important terms which are used in this proposal.

AVRIL

AVRIL is a virtual reality programming library which has its origins from REND386. AVRIL is similar in feel for the end user as REND386, however the code has been extensively modularised and re-coded, so as to make it easier for the programmer. The AVRIL library is still being improved and updated, and is currently in version 2.0, with a version 2.5 looking to offer additional enhancements. One of the drawbacks of AVRIL, however, is that it does not contain the support for the PowerGlove that existed in REND386.

Figure 2.1 below illustrates the relationship between the project and the packages REND386, AVRIL and VR-386.

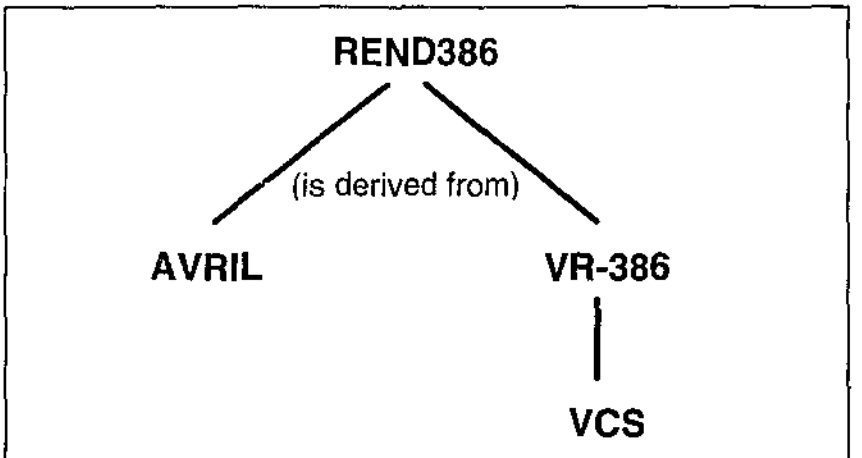

Figure 2.1: Relationship Hierarchy for REND386

Garage-VR

Garage-VR is the term given to virtual reality hardware and software that is generally developed by individuals with very limited budgets; in effect, 'working with the parts from one's garage'.

PowerGlove

The PowerGlove gives feedback on the current positioning of the hand in 3-dimensional space, as well current finger positioning and hand rotation. Aimed at being a cut-down consumer version of the DataGlove, the PowerGlove is very rugged and hardy in design and, hence, is an ideal 'Garage-VR' device. Due to lack of consumer interest at the time of production, however, Mattel stopped producing the PowerGlove several years ago, and so obtaining one is sometimes a difficult venture.

REND386

REND386 is a virtual reality rendering engine and library which was developed upon the 80386 architecture for general use. The source code to REND386 is freely distributed; thus, it is quite popular amongst Garage-VR enthusiasts. The code provides support for a number of Garage-VR input and output devices, including the PowerGlove.

Virtual Environment (VE)

A world that is simulated entirely within the memory of a computer. A virtual environment might consist of a three-dimensional model of a house, or a visualisation of a set of complex data, or any number of things. It is through virtual reality that users are able to create and explore these virtual environments. (Aukstakalnis et al., 1992, p. 12)

Virtual Reality (VR)

There are a number of definitions for virtual reality, and a number of different interpretations of what constitutes a virtual reality system. Probably the most general description of virtual reality is that given by Jaron Lanier (who coined the term): "A computer-generated, interactive, three-dimensional environment in which a person is immersed". (Aukstakalnis et al., 1992, p. 12)

VR-386

VR-386 is another derivative of REND386, and consists of a clearly defined programming interface to a virtual reality programming library. Like AVRIL, it has been extended to provide greater functionality

and ease of use for the programmer. VR-386 also provides support for the PowerGlove, however, it is not currently being maintained by the author. It was this package that was eventually chosen as the basis for implementation of the Virtual Construction Site.

3 LITERATURE REVIEW

3.1 General Literature

In order to understand the field of virtual reality, and what research and development is currently being undertaken in this field, one must first have a good understanding of the basic concepts involved.

One of the first books to describe the field of virtual reality was written by a journalist who travelled to a number of important research centres around the world for virtual reality research. Rheingold's book, 'Virtual Reality' (1991) covers the many areas of research and development in this field.

Written in a biographical manner, the book gives the reader a good understanding of where various areas of virtual reality are in terms of research and development; however, the book does not explain the reasons behind their particular approaches to research, nor does it describe in detail the underlying theory.

Watt (1989), Foley (1991), and Vince (1992) all provide excellent in-depth information on 3-dimensional graphics and modelling, areas in which the project has investigated to some depth.

Aukstakalnis (1992) contains various useful definitions of terms used in the field of virtual reality, and also describes the many diverse areas of research that are being done in this field, as well as illustrating the theoretical concepts behind the research.

Roehl (1993) and Gradeki (1994) are primarily aimed at the 'Garage-VR' enthusiast. Both books also contain large collections of information about companies doing research in the field of virtual reality, and locations where further information may be obtained via online services or through the Internet. Two of the authors of 'Virtual Reality Creations' were also the authors of REND386, and so it is not surprising that most of the discussion is about the REND386 software package. The book does, however, give a good introduction to Virtual Reality, as well as details on how to modify a variety of equipment, including a PowerGlove, for use with a PC.

Gradeki (1994) explores REND386 from a programming perspective, and contains many good examples and source code for adding special features to REND386 applications, including adding allowing shared virtual worlds through the serial port or modem lines. Gradeki also examines many design and performance issues of note, including that of realism versus speed in a virtual environment.

Though a realistic world may be the ultimate goal, Gradeki asserts that one should not necessarily sacrifice speed in order to achieve this, as a faster frame rate has the advantage of creating a smoother sense of immersion within the environment. Therefore a well designed virtual environment should have a careful balance of the two, combining smooth movement and response times, with enough detail for the user to clearly recognise everything in the environment.

Benedikt (1994) and Laurel (1994) both contain large collections of papers aimed at discussing the theoretical design concepts behind user interfaces and virtual reality. In particular Bricken's paper (1994) which describes the differences between the paradigms used for screen-based interface design and that used for creating virtual worlds.

There are also two excellent electronic discussion groups on the Internet which are invaluable for research in the field of virtual reality. The USENET news group 'sci.virtual-worlds' is the medium by which many of the researchers and developers in the field of virtual reality discuss aspects of research, design and implementation of projects. A special electronic 'mailing list' for the discussion of utilising and modifying the PowerGlove for use with various hardware is also available on the Internet.

3.2 *Literature on Previous Findings*

The first article to outline how the PowerGlove could be modified for the PC was published in BYTE magazine (Englowstein, 1990), since then, the PowerGlove has been one of the most commonly used devices in Garage-VR research. Englowstein's article also described how the PowerGlove could be used as a simple mouse or joystick, however, this was only utilising the low-resolution mode of the glove and no three-dimensional positioning information was given using this mode. In order to access the high-resolution mode of the PowerGlove, and thereby gain the full use of the PowerGlove's capabilities, a special initialisation protocol needed to be used.

A framework for designing an interface for the creation of three-dimensional objects is outlined in a paper by Sittas (1991) who describes a new user-interface method for creating and manipulating three-dimensional objects. The framework is based on a set of three-dimensional grids and finite construction planes which can be used to define sub-spaces within the environment. Each sub-space can then contain more complex detail about the objects within their volume, thus alleviating the difficulties of sketching and defining complex solids and assemblies in three-dimensional space via a two-dimensional screen quickly.

In McGuinness and Meech's article (1992) about the human factors in virtual world design, the reader is confronted with the problem of mixing metaphors whilst designing the interface to a virtual environment. The article maintains that virtual reality systems must possess consistent information structuring and representations which can be readily perceived and understood by the user, and that for these systems it is even more important to have a cohesive design theory.

Quinnell's article (1993) breaks a virtual reality system into a number of integral parts which cater for various aspects of the environment created. Quinnell defines that virtual reality software must be comprised of three main sections:

- an object database, containing the descriptions of the objects, as well as attributes such as colour, motion, and orientation;

- a device manager, provide a link to the input and output devices, monitoring the various positioning and tracking devices, and updating the display; and,
- a simulation manager, the heart of the system, coordinates the other sections' activities and tracks object attributes, collisions. and user viewpoints.

Latta and Oberg (1994) define a number of elements to a virtual reality development system in their paper "A Conceptual Virtual Reality Model". By breaking it into a number of conceptual parts, a better understanding of the human and technical elements that make up virtual reality systems may be achieved. Furthermore, they describe how human perceptual and muscle systems can be used to provide sensation and action within a virtual environment. The cognitive and psychological centres of the human brain interpret the sensations given by the environment and respond with appropriate stimuli, thereby creating the sense of immersion within the environment.

The concept of metaphors and how they should be used is further outlined in an article called 'Tools and Techniques for Creative Design' (Mountford, 1994). Mountford's describes metaphors as powerful verbal and semantic tools for conveying both superficial and deep similarities between familiar and novel situations. Mountford also presents a number of steps which can be used to help designers create new interface concepts.

Apart from a number of books which illustrate how to connect the PowerGlove to the PC (Roehl, et al., 1993; Gradeki, 1994), and provide software for communicating to it, the glove has also been compared to more expensive devices such as the DataGlove (Sturman & Zeltzer, 1994). Sturman and Zeltzer review the underlying theory behind glove-based input, and compare a number of the glove-based input devices available.

One article of particular interest when considering using the PowerGlove as an input device is "Gestures in Human-Computer Communication" (Kurtenbach & Hulteen, 1994) which discuss a number of ways in which a gesture-based device (such as the PowerGlove) may be used for interaction, navigation, or immersion, into a three-dimensional environment.

When designing the virtual environment interface itself, there are a number of elements that need to be considered: namely, the type of interaction methods used, the forms of feedback given, and the level of complexity incorporated into the interface. In order for a system to be intuitive and easy to learn and use, careful attention needs to be paid to the design of the interface, which should consist of a fine balance between simplicity and functionality.

Fisher (1994), in his paper on "Virtual Interface Environments" describes the experiences of the Aerospace Human Factors Division at the NASA Ames Research Center. Fisher describes the evolution of the Ames Virtual Environment Workstation and the problems they encountered when designing the interface to the virtual reality system.

In an article entitled "Through the Looking Glass", Walker (1994) outlines the history of user interaction methods. According to Walker, user interactions may be categorised into five generations, as listed in Figure 3.1.

Generation	Means of Operation	Barrier
First	Plugboards, dedicated set-up	Front Panel
Second	Punched cards, Batch processing	Countertop
Third	Timesharing	Terminal
Fourth	Menu systems	Menu Hierarchy
Fifth	Graphical controls, WIMP interface (Windows, Icon, Mouse, Pointer)	Screen
Sixth	Cyberspace	?

Figure 3.1 : Generations of User Interaction

3.3 Specific Studies Similar to the Current Project

Presently, no studies of a similar nature to this project are known to exist. Daniel Lau, an honours student at Curtin University in Western Australia has been doing some research into using a PowerGlove with a Silicon Graphics Workstation, however, he is still currently doing this research, and no real information has been available except for his honours proposal (Lau, 1994).

3.4 *Literature on Methodology*

The prototyping approach is a well defined methodology for the design and development of software. Pressman (1992) is an excellent reference book for all aspects of software design and development, covering approaches to system and software analysis, design and implementation of software, aspects of quality assurance, and software testing strategies.

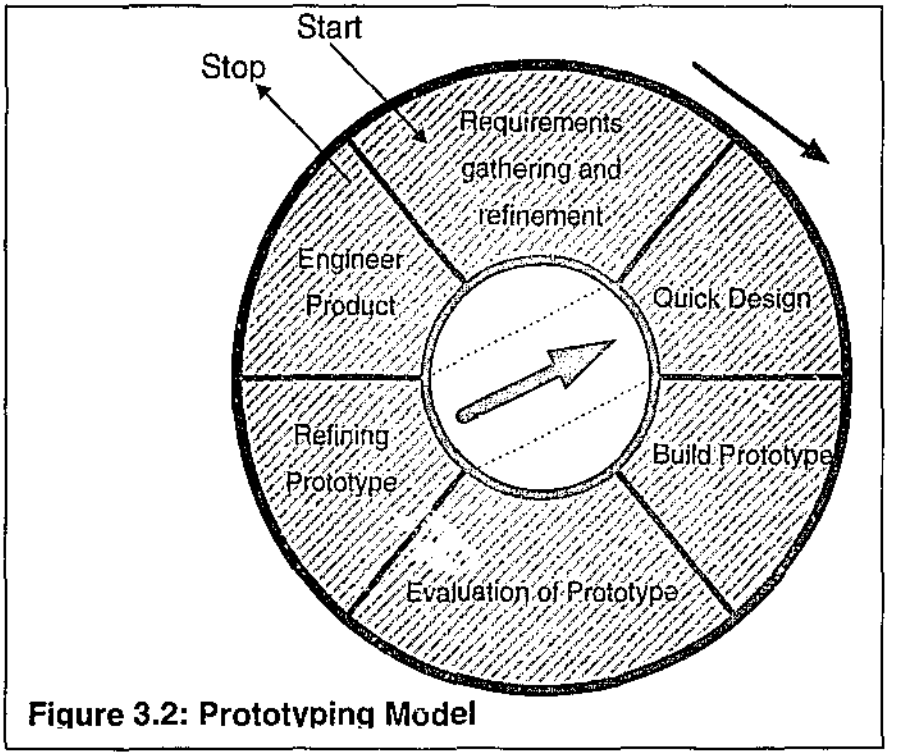

Figure 3.2: Prototyping Model

The project has been developed using a prototyping approach to application design. This approach has a number of advantages over other approaches in that it allows continual re-evaluation of the interface to the application, providing feedback to the progress and success of the project through the implementation phases. Figure 3.2 illustrates the traditional prototyping model for software development.

The prototyping approach to development consists of the following steps:

1. Producing an initial prototype for the system; in effect a shell to which the functionality of the system will be added,

2. Refining and evaluating the system, and making any changes to design that are seen as needed,

3. Adding functionality to the system in a modular fashion, and repeating steps two and three until the full system has been developed, and no further revisions of the system are needed.

One common problem with this development methodology is that the scope of a project gradually expands if not constrained and managed properly. In order to avoid this "expansion" of scope, strict design goals were set early in the design phase of the project, and these were not allowed to change.

4 ANALYSIS AND DESIGN

4.1 Design Overview

The development of any software package requires considerable effort to be spent on the design phase. This is especially true with the project, as for any system where the software empowers the user with the ability to create. The Virtual Construction Site must therefore be carefully planned and guided, and in order to be able to do this, a detailed description of what needs to do has to be performed.

As described in earlier chapters, the Virtual Construction Site is to provide the following:

- Offer a seamless user interface to the virtual environment. Users should be able to navigate around the environment with ease, and be able to manipulate the environment with an intuitive user interface;

- A transparent interface for two and three dimensional input devices, will also be provided, enabling users to use whatever input device they feel comfortable with. Support for the mouse, joystick, keyboard and PowerGlove will be provided through this interface; and,

- Allow the user to to utilise basic building blocks and a drag and drop style of interface to create a virtual environment.

The Virtual Construction Site is to be developed using the VR-386 library, which will provide the basic functionality for rendering and manipulating a virtual world. The VR-386 library has been written for the IBM PC platform, and the full source code is freely available. The following chapters describe the design and layout of this package, and illustrate how the Virtual Construction Site will be incorporated into this package.

4.2 System Model

The VR-386 package is designed to be a large set of programming modules which are incorporated into a single application programming interface (API). Designed to allow the building of virtual-reality applications through the use of this API, the VR-386 library of routines provides a number of modules in order to achieve this.

On the lowest level of support, the modules provide a sophisticated rendering engine, an integer mathematics library, file system and device support, and memory management. On top of this layer of building blocks, a number of more sophisticated modules are built, providing support for the manipulation of objects, generic pointer device layers (including support for input devices such as the mouse, joystick and PowerGlove), management of

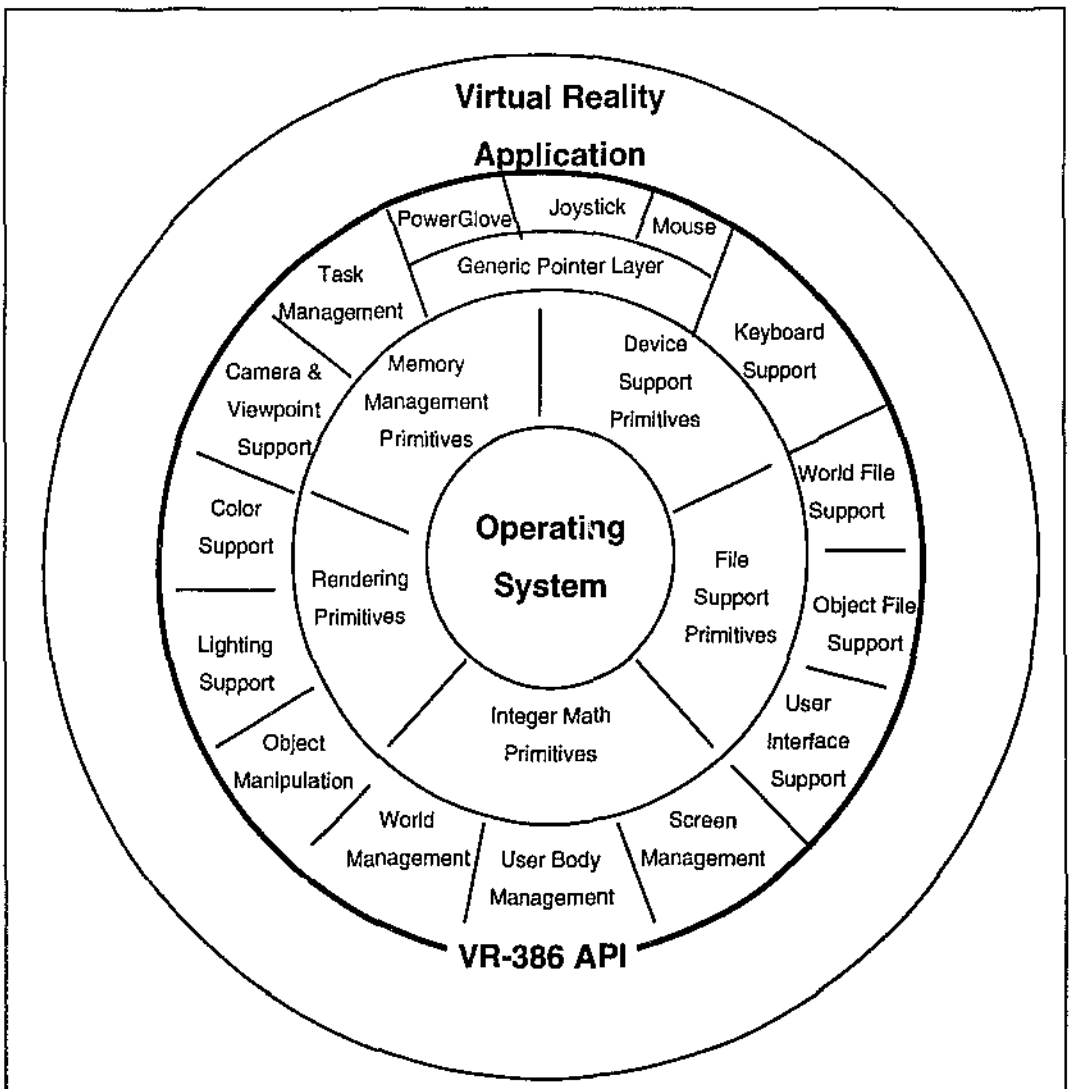

Figure 4.1: Current System Model for VR-386

the virtual world, support for managing cameras and viewpoints, colour, lighting and the user's body within the virtual environment.

Support for loading and saving objects and worlds to the file system, basic task management for animating objects, as well as routines for managing the screen and enabling user interaction are also provided in this layer.

The system model given in Figure 4.1 illustrates the organisation of the various modules within these layers. Each module has been developed using the primitives provided by the layers below it, or through integration with modules on the same layer as the module. All of these layers are accessible through the API, giving the programmer a fairly complete interface with which to build a virtual reality application.

The VR-386 package is designed to be primarily a programming interface, and although a skeleton application is included, it provides only basic support for the features of the package.

4.3 System Integration Plan

The Virtual Construction Site (VCS) aims at integrating with the existing application programming interface, and provide additional functionality to the system, as well as improve the user interface. In order to achieve this, particular emphasis needs to be placed on how the new features will integrate with the existing system in order for it to achieve the desired functionality.

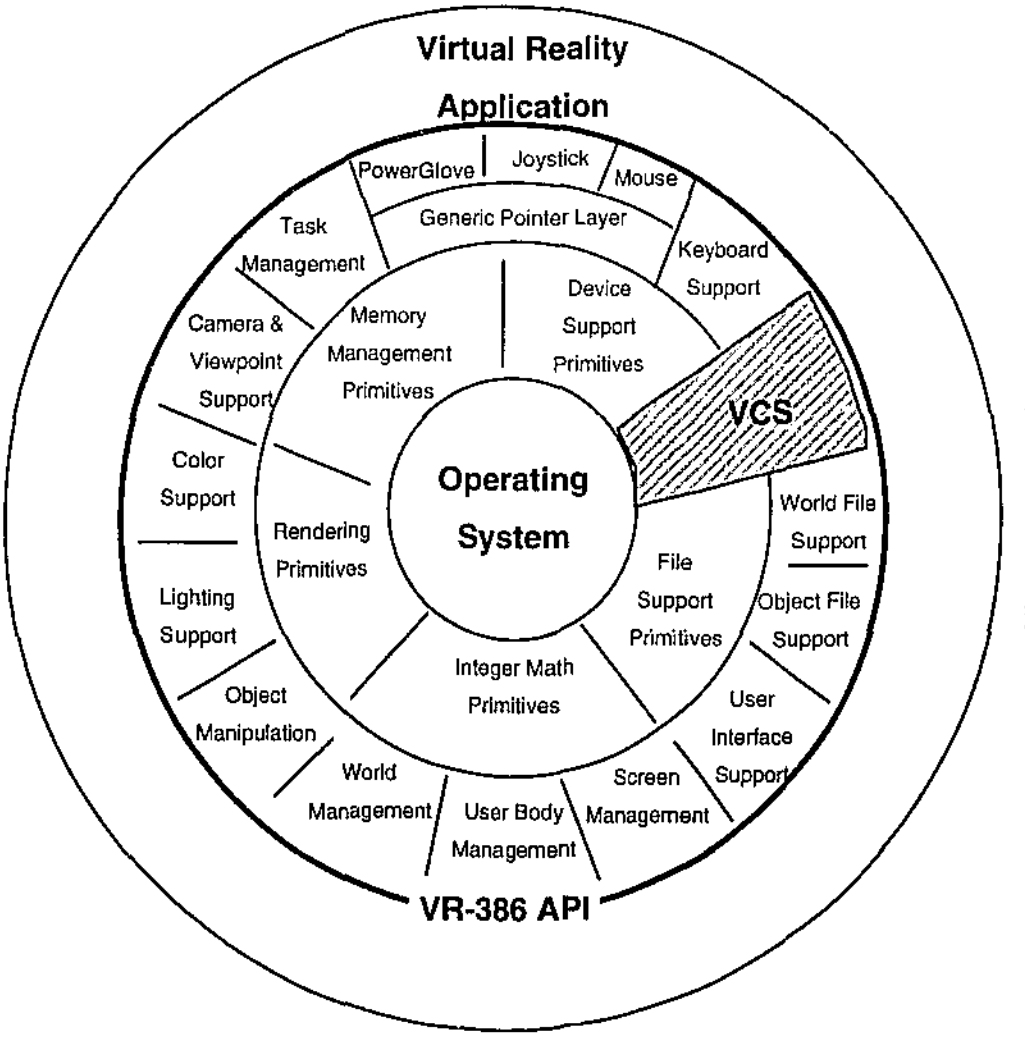

Figure 4.2: System Integration Model

A system integration model is given in Figure 4.2, which illustrates, through the previously described system model, how the VCS module will integrate with all layers of the system.

In order for the Virtual Construction Site to be able to integrate smoothly with the VR-386 library, a number of "hooks" have been used inside the existing VR-386 library code. These "hooks" are effectively a form of linking between the system and the new VCS module, allowing it to take control of the system at particular moments during execution.

A list of the "hooks" present in the VR-386 library, and the associated VCS routines they are linked is given in *Chapter 5.3 - System Structure.*

4.4 User Interface Design

As the user interface was a major design goal for this project, considerable emphasis has been placed on the design of the interface, to ensure that an effective and intuitive final interface to the package was produced.

This section will discuss the various issues involved with each of the elements of the interface, and describe the reasons behind the choices made.

Interaction Methods

Due to the number of devices supported by the package, there are several different methods for interaction available at any one time to the user. The user may decide to point with the glove to select an object, or the mouse may be placed over an object and the left mouse button clicked to select the object, or the joystick or keyboard may be used to select the object.

To provide a smooth transition between all of these forms of interaction, the interface style has been designed similarly for each device. By designing a similarity between the interface styles, users may change between input devices as they feel appropriate with minimal differences in the functionality or principles behind the interface.

Interface Modes

There are two distinct modes which may be used in the program, Navigation Mode and Placement Mode. The user interface style changes dramatically between these two modes.

In *Navigation Mode*, the interface is designed to allow the user to explore the environment, and all actions, whether they be movement on the mouse, or gestures with the glove, are interpreted to be movement within the environment.

In this mode the glove functions similar to a helicopter, allowing the user to use simple intuitive gestures to move around the world, while controlling the speed through flexing of the glove; while the hand is flat the user is motionless, and speed will gradually increase as soon as fingers are closed, achieving maximum speed if the hand is clenched into a fist gesture. If using a mouse or joystick, this mode works similarly, save for the method of controlling movement. Movement of the pointer to the top and bottom of the screen cause movement either in a forward or backward direction - the further away from the center point of the screen, the faster this movement would be.

Any movement of the pointer to the sides of the screen will cause the user to rotate in that direction, the further away from the center point of the screen the pointer is located, the faster the rotation in that direction.

If forward motion is achieved, then the direction of the movement is always in front of the user, thus any turning or rotation while moving will cause a change in direction and the user will start to move in that direction.

In addition, any objects that are "grabbed" before this mode is selected are also moved along with the user until the mode is changed and the objects are released.

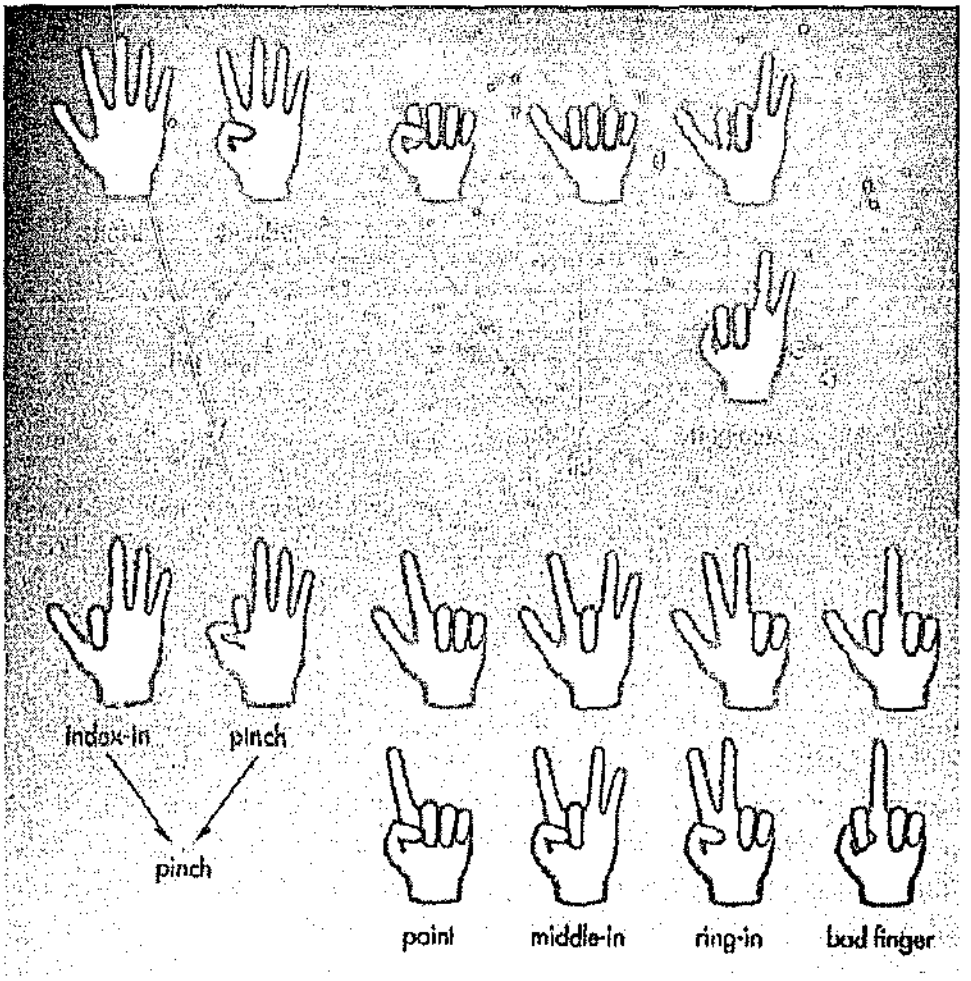

Figure 4.3: PowerGlove Gesture Table (reproduced from Stampe et al.)

In *Placement Mode*, the default mode, the user is kept stationary, while the glove is used to "grab" or "release" objects. The ability to manipulate the selected object(s) is also provided through this mode.

Of the eleven basic gestures that can be detected from the PowerGlove's finger sensor data, only seven can be reliably recognised due to finger and thumb calibration problems. Figure 4.3, reproduced from Stampe et al. (1993), shows the eleven gestures, and the arrows indicate how the gestures are combined.

There are a number of basic tasks that can be used in Placement Mode, some of these having direct gesture equivalents, other more complex tasks do not have intuitive gesture-based equivalents, and must be accessed through the use of context-sensitive menus via the keyboard.

Figure 4.4 demonstrates the equivalent PowerGlove and Mouse actions necessary in order to perform a number of tasks ranging from the simple to more complex.

Task	PowerGlove	Mouse
Selecting an Object	Point	Left Mouse Button
Grabbing an Object	Clench Fist	Right Mouse Button, Select Grab
Releasing an Object	Release Fist	Right Mouse Button, Select Release
Rotating an Object	Pinch	Right Mouse Button, Select Rotate
Move an Object	Grab Object, Move Hand	Grab Object, Move Mouse/Joystick
Change Interface Mode	Button A	Right Mouse Button, Select Change Mode
Change an Object's Colour	Must use keyboard	Select Object, Right Mouse Button, Change Colour
Destroy an Object	Must use keyboard	Select Object, Right Mouse Button, Destroy Object

Figure 4.4: Tasks and Equivalent Actions

Palettes

The concept of a palette is derived from the typical artist's tool to help them build their scenes.

Palettes used in the Virtual Construction Site, however, are used to hold objects, rather than colours. Objects from these palettes may be selected and dragged into the user's environment, enabling users to use these objects to construct complex worlds using a number of basic objects.

In order to utilise palettes, a description of how the interface is designed should first be covered.

Palettes are accessed through the use of a concept called *transit points.* These transit points are sections of the screen which are devoted to transferring the user in between the user's workspace and the palettes.

Figure 4.5 illustrates where the transit points are located in the Virtual Construction Site.

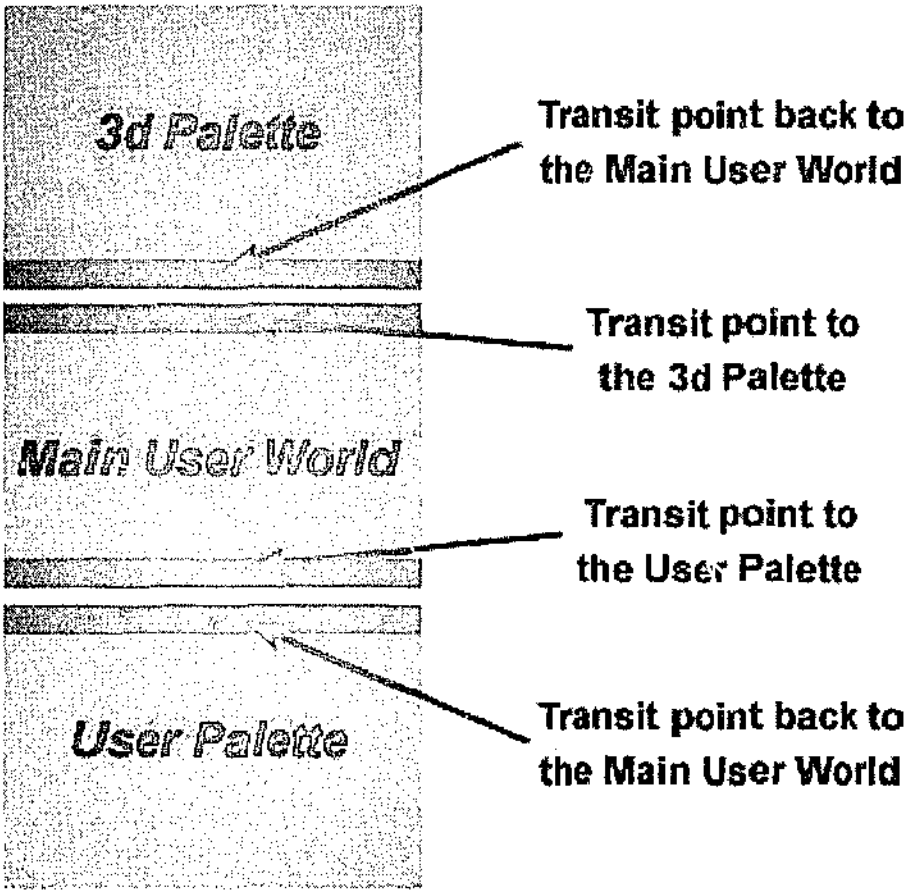

Figure 4.5: Transit Points in the Virtual Construction Site

Two palettes are available for use, the 3d Palette and the User Palette. Using these palettes, and a drag-and-drop style of visual interface, users are able to create virtual environments in a very intuitive manner, and with much greater ease of use than with a textual method of interaction.

"Dragging" and "dropping" three-dimensional objects from a palette is a three-dimensional interaction technique. Based on a metaphor akin to the one humans use to move objects around in real life, this method translates easily to gesture-based input devices, as well as to two-dimensional input devices.

Each of the palettes are described in greater detail below.

3d Palette

This palette holds a collection of basic three-dimensional objects, which may be reused (akin to a colour paint palette).

The objects in this palette are:

- Cube;
- Sphere;
- Pyramid;
- Cylinder; and,
- Cone.

These objects may be used in the construction of the user's environment simply by selecting and dragging the appropriate object back into the user's world.

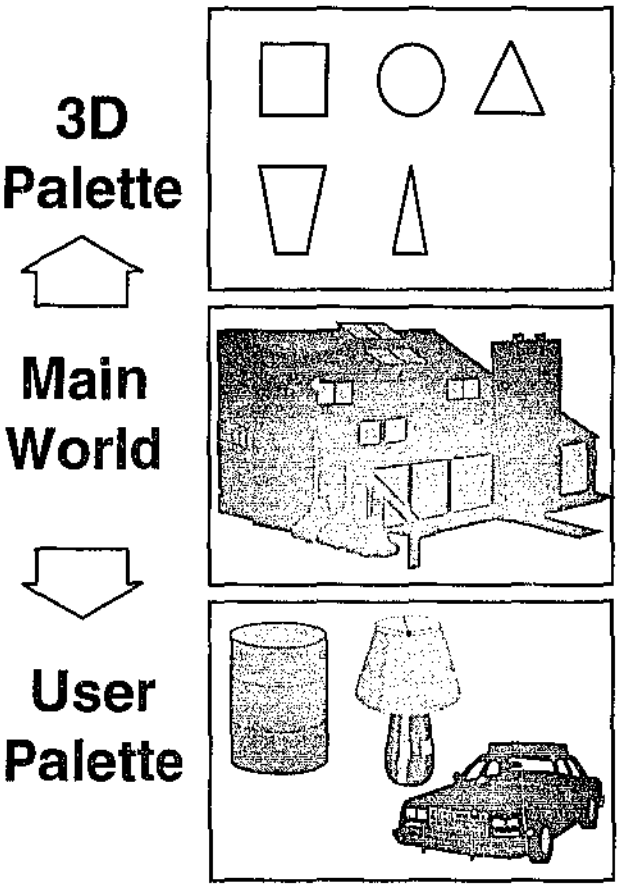

Figure 4.6 : Palettes - Extensions to the User's workspace

User Palette

Similar to a mixing palette, the User Palette holds commonly used objects available for reuse. The User Palette can be envisioned as a small world or room in which the user might store blueprint objects that ready for reuse by the user, and/or are convenient to have on hand.

The user may "grab" objects from within the palette, and "drag" them into the Main User World. The object actually dragged is merely a copy of the object, and not the original blueprint, therefore allowing it to be reused more than once.

An object may also be dragged from the Main User World into the User Palette, thereby adding an object to the palette, and making it a blueprint. An object may also be loaded directly into the User Palette, and this will also cause it to become a blueprint.

Context Sensitive Actions

Not all tasks can be represented in an intuitive fashion, and for these the aim is not to make the action more intuitive, but to make it easier for the user to be able to perform it. In the Virtual Construction Site, this is achieved through the use of context sensitive menus. The user may invoke a context sensitive menu at any time, and the menu hierarchy that appears depends on what the user is doing, and the current state of the program. If no objects are selected, the following options are displayed:

- Navigation Mode;
- Help;
- Information;
- Goto Location;
- Save World; and,
- Quit.

If a single object is selected, then this menu becomes:

- Save Object;
- Paint Object;
- Resurface Object;

- Destroy Object;
- Twirl Object;
- Make Object Fixed; and,
- Information.

While if more than one object is currently selected, then the menu becomes:

- Paint All Objects;
- Resurface All Objects; and,
- Destroy All Objects.

By altering the menu structure according to the user's actions, the actions which can be performed at that state become both easier to access, and also more intuitive.

4.5 Summary

The design processes and priorities salient throughout the project have been examined in this chapter. Discussion now turns to the implementation of these designs.

5 IMPLEMENTATION

5.1 Implementation Overview

In order to design and implement the project, a number of implementation decisions needed to be made. These included such issues as the platform the software would be designed to run on, the hardware and software needed to design and implement the package, and what language and development environment would be used to build the project. These issues will be discussed in order in this chapter, together with the reasons for these decisions.

Platform

In order to enable the most number of users to use this package, the IBM PC platform was chosen as the development platform. This was further compounded by availability of the VR-386 library, which only runs on the PC platform. Unfortunately this also restricted the development to DOS based applications, as the VR-386 library only supported DOS video modes.

Another strong reason behind this choice was the availability of documentation for connecting the PowerGlove to the IBM PC's Parallel Port. Other options were evaluated, however, the Minimal Reality Toolkit, obtained from the University of Alberta, Canada, was also a viable option, and this ran under UNIX with X-Windows and OpenGL. However, due to lack of computing resources, it was not possible to get access to a machine with the capabilities or configuration necessary to enable any further evaluation of this software. This platform also limited the range of users who could possibly utilise the software also, as UNIX machines of the desired configuration make up only a small part of the computing market.

Hardware Used

The following hardware was used to design and implement the project:

- A 486DX4 running at 100MHz was used as the development platform, this was equipped with a parallel port and game port for connection of the PowerGlove (see below),
- A customised PowerGlove was used in developing and testing the system. This was modified* to be connected via the parallel port on a conventional PC, and a 5V power source was obtained through the game port on the PC.

Software Used

In addition to the hardware used, a number of software packages needed to be used to design and implement the project, the packages used were as follows:

- Borland C/C++ 3.1 (DOS) as the development software. Borland C/C++ 4.0 (Windows) was evaluated as a development platform, but there were problems with the PowerGlove under Windows 3.1 which rendered it useless; this was due to timing conflicts between the PowerGlove device driver and Windows,
- DOS 6.2,
- The VR-386 source code,
- Microsoft Windows 3.1,
- Microsoft Office - Microsoft Word 6.0 was used to write all documentation, Microsoft Project was used for project management and scheduling, while Microsoft PowerPoint was used to produce overheads for the final presentation.

* This was modified along the lines of the specifications given by Eglowstein's BYTE article (Englowstein, 1990, p. 288)

5.2 Development Standards and Procedures

When maintaining code, the majority of a programmer's time is spent both familiarising themselves with the code in question. In order to assist this process, a conventional coding style will be followed. This provides a standard framework for issues such as naming style, internal documentation, and code layout that will enable a programmer unfamiliar with the code to become familiar with it quickly and easily.

VCS_ Prefix

All Virtual Construction Site procedures, functions, constants and variables that were visible externally were prepended with the **VCS_** prefix. This notation ensures that all VCS code is easily visible within the VR-386 framework.

Smalltalk-like naming style

All variable names after the prefix have been based on the Smalltalk style of shifted case with no further underscores. Capitals are used to denote the beginning of new words in the name.

Thus, a name like `VCS_myvariablestring` would become `VCS_MyVariableString`, making it clearer to read and understand than if all in lower case.

Internal Documentation

All routines should have comments immediately after the function header. These comments should explain in detail what the routine does, and if a complex routine, should go into some depth. These comments should be written using the `//` syntax (C++ comments) for code documentation, this style making it easier should a programmer wish to comment out a large section of the code using the conventional C form of comments (`/* */`).

Coding Layout

Code will be structured using a aligned-brace method of code layout. This allows all braces to be lined up easily and ensures that braces that are missing are immediately obvious to the programmer. An example of the aligned-brace method is given below. (NB. dashed lines are given to show how the braces line up, and are not actually in the code).

```
void VCS_ExampleRoutine(int MyVariable)
// This routine basically checks to see if MyVariable is equal
// to True, and if so, then calls VCS_DoRoutineA().  Regardless
// of whether this occurs or not, VCS_DoRoutineB() is called
// immediately afterwards.
{
    if (MyVariable)
    {
        VCS_DoRoutineA();
    }
    VCS_DoRoutineB();
}
```

5.3 System Structure

This section contains a breakdown of the routines implemented in the VCS module, along with the modified functionality added to the original VR-386 library. Below you will find an overview of the extensions added to the VR-386 library, a description of the system hooks into the VR-386 and where each module links into the system, along with a comprehensive listing of the VCS routines and the functionality they provide.

Extensions to the VR-386 library

In developing the Virtual Construction Site application using VR-386, there were a number of gaps found in the library's support for various interaction methods and, in some cases, basic functionality. A number of additional routines have been added to the library, thereby extending the library's functionality overall.

The features added include:

- The ability to duplicate existing objects without needing to load objects from disk. Previously the only possible way of creating an object was to either load one from disk, or to create one from scratch. With the additional capability of being able to duplicate objects, additional functionality and features can be implemented to take advantage of this feature, including the ability to drag and drop copies of an object,
- A more intuitive method for navigation in the virtual environment. Akin to helicopter movement in many ways,
- Point-and-click interface style for working with objects,
- A structured, more layered approach to working with different input devices transparently,
- A more consistent and seamless user interface style throughout.

System Hooks

The following list outlines the list of "hooks" made to the VR-386 source code. Each hook involves a link from the original VR-386 source code to a VCS routine, combined with some changes in the functionality of the VR-386

routine in the given file in order for VCS to function correctly. Each such modified routine is listed along with the appropriate filename, the VCS routine that it is now hooked to, and a short description of the reasons behind this change of functionality. For more information on what each VCS routine actually does, see the section ***VCS routines*** below.

File MAIN.C:

VCS_Initialise()

This hook activates the initialisation code present in the VCS Module, allowing it to initialise and configure itself.

File CURSGLOV.C:

VCS_MakeObjectProtected()

This hook works similarly to that in CURSOR3D.c; protecting the glove cursor from the object shuffling that takes place when changing in between worlds.

VCS_ProcessGlove()

This hook transfers control for all glove actions across to the VCS module, allowing it to control and maintain the glove user-interface that has been implemented in the VCS module.

File CURSOR3D.C:

VCS_MakeObjectProtected()

This hook was added in order to make the 3d-cursor objects protected from the object shuffling that takes place when changing in between worlds.

File CURSOR2D.C:

VCS_ProcessMouse()

This hook was added in order for the interaction with the mouse to be controlled from the VCS module.

VCS_SelectObject()

This hook was added to provide support for the new form of object selection added by the VCS module. The Shift key can now be used in combination with the selection action (point or click depending on your input device) to select multiple objects, which can then be acted upon in concert.

File KEYBOARD.C:

VCS_GetControlMode()

VCS_GetWorldMode()

Both of these hooks were added to enhance the functionality of the 'information' menu option. The 'information' command now displays the current control mode and world mode, and uses these two routines to get that information.

VCS_ProcessKeys()

This hook passes control of the keyboard interaction routines over to the VCS module, allowing it to override the keyboard interface to allow the addition of the new features added. These include switching between control modes, additional menu items, and tranferring between the main user world and the additional palettes.

VCS_Quit()

This hook was added to enable the VCS module to save its state and preferences before the normal shutdown procedure was run.

File USCREEN.C:

VCS_DisplayPreRenderHook()

VCS_DisplayPostRenderHook()

These hooks were added to enable the VCS module to be able to display information on the screen before and after the rendering process. It is through these that the transit points are displayed, as well as the status of control and world modes, should these display options be activated.

VCS Routines

The VCS module is made up of a number of small routines, each offering additional functionality or working in concert with other routines to do so. Below you will find a comprehensive list of all the extra routines added by the VCS module, along with a short description of how these routines fit into the overall scheme of things.

Initialisation Functions

void VCS_Initialise(void)

This must be called first so VCS may setup everything it needs to run. It initialises the viewpoint of the user, loads the objects needed for the 3d Palette, and configures the program's state

Control Mode Functions

These routines let you change between the possible control modes:

Navigation Mode - Where the pointer is used to allow the user to fly around the environment.

Placement Mode - Where the pointer is used to allow local object manipulation, so the user may select, grab, and move objects.

void VCS_ChangeControlMode(int NewControlMode)

This routine accepts either one of the following:

VCS_NavigationMode

VCS_PlacementMode

and changes to that equivalent mode. If that mode is already active, then this function is ignored.

*char *VCS_GetControlMode(void)*

This routine returns a string corresponding to the current control mode. This function is primarily used for reporting the status of the current control mode.

Object Protection Functions

Object Protection basically means that the objects are marked as system owned and aren't changed at all when switching world modes. This is done by default for all glove and 3d cursors loaded, and may be used for other objects if desired. Functions are provided to make objects, and object lists protected, and unprotected.

*void VCS_MakeObjectProtected(OBJECT *obj)*

This routine takes the object passed and sets the protected flag on it.

*void VCS_MakeObjlistProtected(OBJLIST *objlist)*

This routine takes the object list passed, and makes all the objects in that list protected.

*void VCS_MakeObjectUnprotected(OBJECT *obj)*

This routine takes the object passed, and removes the protected flag from it.

*BOOL VCS_is_object_protected(OBJECT *obj)*

This routine takes the object passed, and returns a Boolean value indicating whether the object is currently protected.

Object Logging Functions

These functions let you dump a particular object's details into the log file, or let you do a total dump of the current system state and all object lists. This is useful for debugging purposes.

*void VCS_Log_DumpObject(OBJECT *obj)*

This routine takes the object passed, and outputs all the details of that object into the VCS log file.

void VCS_Log_DumpState(void)

This routine outputs the details of the current program state to the VCS log file. This includes the overall world structure, details on all objects, including which world they are associated with, what flags are currently set on each object, etc.

*void VCS_DoObjectCount(OBJECT *obj)*

void VCS_CountSelectedObjects(void)

These two routines are used for object counting for status information. Due to the nature of traversing object lists, object counts are done in the background. By calling VCS_CountSelectedObject(), the variable VCS_ObjectCount is automatically updated to reflect the number of selected objects.

void VCS_SaveSelectedObject(void)

This function will save the details of a selected object to a file. Upon calling of this function, a file name will be requested from the user in the form of a file dialog box, and the selected object will be saved to that file.

Display Option Functions

These functions allow you to turn on/off the compass and horizon display. This is more for use internally (as these are switched off when changing worlds), but the functions are given for future use as well.

void VCS_TurnOffCompass(void)

void VCS_TurnOnCompass(void)

void VCS_TurnOffHorizon(void)

void VCS_TurnOnHorizon(void)

State Saving Primitives

These primitives are used to juggle the objects in the current world so as to emulate multiple worlds.

*void VCS_SaveMainWorldObjects(OBJECT *obj)*

*void VCS_Save3dPaletteObjects(OBJECT *obj)*

*void VCS_SaveUserPaletteObjects(OBJECT *obj)*

*void VCS_MoveObjectInFrontOfPoint(POSE centre, OBJECT *obj, COORD distance)*

World Mode Switching Primitives

These are internal routines used for initialising the new world as the current world is changed.

*OBJECT *VCS_CopyObject(OBJECT *obj)*

This routine is designed to be a primitive for copying an existing object, as opposed to loading one in from disk (which is the only way an object may be loaded with the existing library of routines).

A pointer to an exact clone of the object passed is returned. The user must still place this object, and MUST perform at least a compute_object() on it once the object is in place.

void VCS_Setup3dPalette(void)
void VCS_SetupMainWorld(int PreviousWorldMode)
*static void VCS_ShiftObjectsIntoPosition(OBJECT *obj)*
void VCS_SetupUserPalette(void)

World Mode Functions

void VCS_ChangeWorldMode(int NewWorldMode)

This routine allow you to change the current world mode between:

VCS_3dPalette - A Collection of Basic 3d Objects
VCS_MainWorld - The Main User World
VCS_UserPalette - A Collection of User Configured Objects

*char *VCS_GetWorldMode(void)*

This routine is provided to enable you to get a string holding the current world mode, this is primarily to be used for status information.

Hooks for Rendering and Input Devices

These routines are linked in with the internal engine of VR-386, and will be called at various times throughout the package.

This is useful for displaying information, or catching input device information directly.

void VCS_DisplayPreRenderHook(void)

This routine is linked into the prerender_process() in USCREEN.C.

void VCS_DisplayPostRenderHook(void)

This small bit of code should display the world buttons on the screen, and should put names on them. This is called from postrender_process() in USCREEN.C.

*void VCS_ProcessGlove(POINTER *Glove)*

This is the control routine for the PowerGlove driver and updates the information gained from the glove regarding position and posture, and puts this in a generic form that it then passes to VCS_RegisterMovement().

long VCS_ScaleGloveXCoord(long value)

long VCS_ScaleGloveYCoord(long value)

These routines are used by VCS_ProcessGlove() to scale the glove's movement to that of a generic coordinate reference. This is then used when calling the VCS_RegisterMovement() routine.

void VCS_ProcessMouse(PDRIVER *Mouse)

This is the control routine for the mouse driver, and updates the information gained from the device, then puts it into a generic form which is then passed along to VCS_RegisterMovement().

void VCS_RegisterMovement(int PointerType, long x, long y, long z, int Special)

This routine handles all interface control between the package and the various devices. Interface control is centralised in order to create a truly transparent user interface style, with only the details of the actual devices from which this information came from being used for button and state handling.

PointerType may be either of the following:

VCS_Mouse

VCS_Glove

The Special field is used to contain either button configuration, or the current Gesture ID - as is appropriate for given PointerType.

BOOL VCS_ProcessKeys(unsigned KeyPressed)

This routine is used to handle all additional keyboard functionality provided by the package. It hooks into KEYBOARD.C and is passed the unsigned value of the key pressed. If the given key is handled in this routine, the function must return TRUE, otherwise FALSE is returned.

Miscellaneous Functions

BOOL VCS_IsShiftKeyPressed(void)

This routine, used internally mostly, is designed to simply check whether the shift key is currently pressed.

*void VCS_MoveObjectXY(OBJECT *obj)*

This routine is used to move objects around after they are grabbed with the mouse, thereby simulating a drag-and-drop method of interaction. The object is moved according to the difference between the current mouse or glove position, and the last recorded position. This enables the object to be moved relative to the user's movement.

void VCS_CreateSelectedObject(void)

This routine is used to create a duplicate of a currently selected object.

*int VCS_SelectObject(OBJECT *obj)*

This routine is used to mark a given object as being selected, and utilises the new shift-key functionality added to the package. Using the shift key, a user can select multiple objects which can then be worked on as if a group.

*void VCS_GrabObject(OBJECT *obj)*

This function provides a simple primitive for grabbing and attaching an object to the user.

*void VCS_UngrabObject(OBJECT *obj)*

This function provides the ability to detach an object from the user, in effect dropping it.

void VCS_Quit(void)

This routine is used to catch the shutdown of the package, and allow the VCS module to save any configuration details and shut down as appropriate.

5.4 Evaluation and Testing

In order to ensure a quality system was produced, there were a number of evaluation and testing methods used throughout the development phase. These include:

- **Module Testing**
  This involves putting a variety of values into each variable of a particular function or procedure, and verifying the results;

- **Internal Logging**
  A number of specialist debugging functions were added, including VCS_DumpObject() and VCS_DumpState() These were used to extensively test the various routines as they were added to the system, and used to verify that each new feature worked in coordination with existing code; and,

- **Exhaustive Testing**
  Simply testing all possible number of things in trying to use different possible combinations. This also involved an Alpha and Beta test phase, with constant updating in between.

5.5 Summary

This chapter has presented the decisions made in the implementation of the project, as well as the rationale behind these decisions. The importance of standards, and how they have been applied to the development of the project has also been discussed. Finally, the structure of the new system was covered, describing in detail how the project integrated with the existing system, and the functionality was added to the existing system.

The next chapter discusses the findings of the project, and the problems encountered in throughout the design and development of the project.

6 FINDINGS

6.1 Discussion

To best describe the findings and results of the project, the original goals of the project will be reviewed and discussed in terms of the problems encountered and the solutions used to overcome them.

Virtual Construction Site (VCS)

The first and foremost goal of the project was defined as being: *"the means by which the user may construct a virtual environment using a consistent and intuitive interface"*. Designing a system like this is no easy feat, and required one to be very careful in defining exactly what constituted virtual environment that a user would wish to construct.

The concept of a construction site was seen as the closest real-world analogy, and thus this concept became the driving force behind the design of the project; the user would be given the ability to build the environment using basic building blocks - the most elementary of these being simple three-dimensional shapes such as cubes, cylinders, and spheres. Later this was defined as a separate goal of the project, *"the ability to select simple three-dimensional objects from a defined set, allowing the user to incorporate these into the virtual environment being created"*.

Furthermore, it was seen that in addition to the ability to use elementary three-dimensional objects to build the environment, the package should also provide *"facilities for storing commonly used objects, allowing them to define a set of common objects that they may reuse to create their environment"*.

The ability to store objects and reuse objects as specified by the user was seen as important as it would provide users with the means to design their own sets of primitive objects for the task at hand, thus enabling them to create more advanced and detailed environments.

The design of the application has been useful as a test-bed for new concepts in user-interface design, and this was further expressed by the decision to provide *"support for a gesture-based interface, as well as a*

conventional interface for traditional forms of input such as the mouse, keyboard, and joystick". The decision to support such devices, however, when combined with the earlier decision to provide a consistent and intuitive user interface caused many problems with the design and implementation of the project.

Fundamental problems concerning the accuracy of the PowerGlove caused a number of issues to be raised concerning the possibility of designing a consistent and intuitive interface utilising both two-dimensional and three-dimensional forms of input. These issues are discussed in greater detail in a section on the study of three-dimensional interaction techniques which can be found below.

Conversion of the PowerGlove

As discussed in Chapter Two, modification of the PowerGlove was necessary in order to utilise in combination with an IBM PC compatible platform. Details on how to perform this modification were readily available from a number of sources (Englowstein, 1990; Roehl et al., 1993; Gradeki, 1994), helping minimise the problems encountered in this phase.

A source of power was an issue, however, as the PowerGlove required 5 volts in order to function and the IBM PC parallel port, to which the PowerGlove would be connected with the appropriate modifications, offered no voltage lines. The 15-pin game port, almost a standard on personal computers today, was found to provide a steady supply of 5 volts, and so an extra connector was wired to the PowerGlove to utilise that as a source of power.

Study of three-dimensional interaction techniques

In many ways the exploration of three-dimensional interaction techniques was seen as a major goal of the project; to this end, considerable effort was spent in the research and development of techniques to achieve this goal.

Existing methods for interaction were studied in many forms, however, the nature of the interface demanded interaction forms that were both intuitive and functional in a three-dimensional environment. Conventional

two-dimensional forms of interaction, such as the classic WIMP (Windows Icons, Mouse, Pointer) interface, tended to cover up important elements of the environment and were deemed as conflicting with the inherent structure of the environment. Therefore, it was decided the most appropriate interface style was one that would merge with the three-dimensional environment, and provide a natural addition to the environment.

The concept of a *palette* provided the answer to the posed problem. Designed to extend the user's workspace, a *palette* offers a form of interaction with the user that is much less intrusive than any conventional forms of interaction. The analogy provided by the *palette* also helps to merge it with the interface; that of a typical artist's tool used to aid in rendering a scene. Taking the analogy one step further, the concept of the palette can be broken up into two distinct parts; a "colour" palette, and a "mixing" palette. Within the Virtual Construction Site, however, the colour palette would become a three-dimensional object palette, while the mixing palette would become an area where the user can store objects for later reuse.

Dragging and dropping items, a concept pioneered by the Macintosh Operating System (Mountford, 1994), enabled the user to function intuitively between these additional workspaces and the main environment, allowing users to transfer objects from a palette and place them into the world they are building. In addition, the concept of dragging and dropping items is intuitive in both a two-dimensional and three-dimensional environment, allowing the same interface style to be used with both devices smoothly.

For a task such as selecting or moving objects, gestures provide a very natural means of interaction, the actions map the same in the real world as they do in the environment, providing an interface that is far more intuitive than any two-dimensional form of interaction performing the same task. However, interaction using the PowerGlove becomes far more difficult when attempting to find mappings for more complex tasks, such as changing the colour of an object.

This problem is further compounded by the problems associated with determining reliable gestures for the PowerGlove. The glove can reliably

detect up to eleven basic gestures (these were displayed in Figure 4.3), which, due to finger and thumb calibration problems, is further reduced to seven basic gestures. Of these seven gestures, four of the most intuitive gestures (Bordegoni et al, 1993) were used in Placement Mode, these being: *flat* (no action/release object), *fist* (grab object), *point* (select object), and *pinch* (rotate object). The remaining gestures were deemed too awkward and impossible to use intuitively (except for the gesture *bad finger* which could have been used for quitting the application, however, this was decided to be too offensive to actually be used).

It was found that two-dimensional devices, such as the mouse and joystick, though lacking in depth with regards to interfacing to three-dimensional environments, are more than adequate for the forms of interaction used in the project. These devices have a distinct advantage over the PowerGlove in that they are highly accurate. The PowerGlove relies on the emission of ultrasonic frequencies to determine the location of the glove in three-dimensional space in front of the computer, and any number of things can interfere with this method of transmission - sharp or reflective objects around the computer, environmental noise, and a number of other factors combine to decrease the level of accuracy of the glove dramatically.

As a result, the feedback on the location of the PowerGlove tended to be erratic and this caused severe problems when designing a user interface around it. The exact location of the PowerGlove was deemed too inaccurate and sporadic for reliable translation of user intent, and the hand's movement could be guaranteed only in terms of significant changes of location in three-dimensional space.

The concept of *transit points* was therefore conceived, in which an extreme section of the screen is marked as being a point of transference between the current state and another state. Transit points were then used to enable the user to switch back and forth between the 3d and User Palettes.

In order to enable the user to carry out more complex actions upon objects, the two-dimensional interface design concept of the menu was employed within the interface; allowing tasks such as the changing of an

object's colour, or the loading or saving of an object to be performed. As efficiency of design was an issue, it was seen more productive to design these menus to be context-sensitive, i.e. for them to adapt to the current environment, and offer choices to the user as appropriate. The use of these menus is described in more detail in Chapter 4.4 - User Interface Design.

The last issue to be discussed in this section, is the method of navigating around the virtual environment while in Navigation Mode. After experimentation with different forms of navigation using the PowerGlove, the most intuitive method found was for the glove to act similar to a helicopter joystick. Using this method of movement, motion towards the left or right of the screen would cause the user to be rotated along the vertical axis (Y-axis), while speed was controlled by the amount of flexing of the glove; a flat hand indicating no speed, while a clenched fist indicating full speed in the current direction.

6.2 Application and Use of the Project

The project has been aimed, as described in earlier chapters, at producing a software package that can be used to create virtual environments using an interactive three-dimensional environment. The possibilities for use of this project are numerous, however, several possible uses are listed below.

Architecture

Architecture is one field where virtual reality has had a large amount of acceptance and commercial use. This project offers not only the ability to construct buildings from plans, but to allow navigation around the designs and experience them first hand. This allows architects to see exactly how a building is going to look before the first brick has been laid, and can prove invaluable for detecting possible flaws in the design before construction.

Another possible use along these lines of thought is as a means of conveying locational data to the user. By modeling a building such as a university, the project might be used to construct a three-dimensional model

of that model, and then allow students to familiarise themselves with it without ever having set foot on campus.

Education

This package offers all streams of education the ability to visualise information in a way that extends itself to all areas of research. Utilitising the package as a teaching tool, it is possible to explain concepts through the use of a three-dimensional environment that are best understood by experiencing it visually. *"If a picture is worth a thousand words, then a three-dimensional image can speak over a million words"* (Roehl et al., 1993).
Two examples of use in the field of education include:

- **Astronomy** - A model of the solar system could be used to demonstrate the distances between the orbits of the various planets, and how they interact with each other; and,
- **History** - Historical buildings or structures that existed hundreds of years ago could easily be modeled using the project. This allows students to explore and interact with such structures, as opposed to just reading about them.

Entertainment

In addition to educational use, this project could also be used for all sorts of entertainment purposes. Even now, we are seeing a changing trend in video games towards those with a first-person point of view and detailed environments. Using this project, one can create an environment totally of their own design and merely creating and exploring a created environment could be an adventure unto itself.

6.3 Further Research

There are a number of areas where the research gained through this project could be expanded and enhanced to include provide additional functionality to the user. Below is a short list of some of the topics for that might be used for further research.

- Multiple-user environments - whereby a number of users may occupy and interact within the same virtual space;
- Linking between multiple environments - allowing users to jump from one world to another;
- Distributed/networked environments - this could enable the overhead of rendering complex environments to be shared or distributed across a network, or for multiple environments to be interlinked via networks;
- Support for three-dimensional audio - adding the sense of sound to the environment would enhance the level of immersion provided by the environment considerably; and,
- Support for interaction and navigation using head-mounted displays (Limited support for such devices is is currently provided by the VR-386 package, but not fully supported by the Virtual Construction Site).

6.4 Conclusion

Virtual reality is continuing to be a growing field of interest, and current trends seem to indicate that this will continue to grow. Arthur C. Clarke was once noted to have said *"Virtual Reality won't merely replace Television, it will eat it alive!"* (Rheingold, 1991) - a reflection on the possible future impact of this field on current mediums.

An incredibly versatile area of research, virtual reality has already had considerable impact on a number of fields, most notably with that of architecture, medicine and engineering. Despite this, there are still considerable gaps in the information and research available in particular streams of virtual reality.

One of those streams, interaction methods within three-dimensional environments, formed the basis for the research in this project. Only through experimentation could such interaction methods be developed, and this became the goal of the project, to produce a virtual reality application which would utilise the new forms of interaction and provide a test-bed for development in this regard.

The project also fulfils another goal, that of expanding the field of virtual reality to include those users who might not have the money to spend on expensive packages to create virtual environments, and yet still wish to explore this medium. The project does this by providing a tool which can be used to effectively and intuitively create virtual environments using a number of basic building blocks.

It is hoped that through the knowledge gained from the project, we will not only be one step closer to realising the goal of intuitive and effective interaction methods, but we will also have a useful toolkit with which we can build the future of virtual reality.

REFERENCES

Aukstakalnis, Steve, Blatner, David, Roth, Stephen F. (Ed.) (1992). Silicon Mirage: The Art and Science of Virtual Reality Berkeley, CA: Peachpit Press.

Benedikt, Michael (Ed.) (1993). Cyberspace: First Steps (5th Printing) Massachusetts: MIT Press.

Bordegoni, Monica, Hemmje, Matthias (1993). A Dynamic Gesture Language and Graphical Feedback for Interaction in a 3D User Interface. In Hubbold, R.J., Juan, R. (Eds.) Computer Graphics Forum - Eurographics '93 12(3), pp. 1-11.

Bricken, Meredith (1993). Virtual Worlds: No Interface to Design. In M. Benedikt (Ed.), Cyberspace: First Steps (5th printing). Massachusetts: MIT Press.

Englowstein, Howard (1990, July). Reach Out and Touch Your Data. BYTE Magazine 4(2), pp 283-290.

Fisher, Scott S. (1994). Virtual Interface Environments. In Brenda Laurel (Ed.), The Art of Human-Computer Interface Design (8th printing). Reading, Massachusetts: Addison-Wesley.

Foley, James D., van Dam, Andries, Feiner, Steven K., Hughes, John F. (1991). Computer Graphics: Principles and Practice (2nd Edition). Reading, Massachusetts: Addison-Wesley.

Gradeki, Joe (1994). The Virtual Reality Programmers Kit New York: John Wiley & Sons.

Kurtenbach, Gordon, Hulteen, Eric A. (1994). Gestures in Human-Computer Communication. In Brenda Laurel (Ed.), The Art of Human-Computer Interface Design (8th printing). Reading, Massachusetts: Addison-Wesley.

Latta, John N., Oberg, David J. (1994, January). A Conceptual Virtual Reality Model. IEEE Computer Graphics and Applications, 14(1), pp 23-29.

Lau, Daniel (1994, June). Investigation and Application of Virtual Reality technology to low cost graphics workstations Unpublished honours dissertation proposal, Curtin University, Perth, Western Australia.

Laurel, Brenda (Ed.) (1994). The Art of Human-Computer Interface Design (8th printing). Reading, Massachusetts: Addison-Wesley.

McGuinness, Barry, Meech, John F. (1992). Human Factors in virtual worlds. 1. Information structure and representation. IEE Colloquium on 'Using Virtual Worlds' (Digest No. 093) (pp 3/1-3). London, UK: IEE.

Mountford, S. Joy (1994). Tools and Techniques for Creative Design. In Brenda Laurel (Ed.), The Art of Human-Computer Interface Design (8th printing). Reading, Massachusetts: Addison-Wesley.

Myers, BA (1989). User-Interface Tools: Introduction and Survey. IEEE Software 6(1), pp. 15-23.

Penguin English Dictionary (1974). The Penguin English Dictionary (2nd Edition). Middlesex: Penguin Books Ltd.

Pressman, Roger S. (1992). Software Engineering: A Practitioner's Approach (3rd Ed). New York: McGraw-Hill.

Quinnell, Richard (1993, November). Software simplifies virtual-world design. EDN 38(24), pp 47-54.

Rheingold, Howard (1991). Virtual Reality London: Mandarin Paperbacks.

Roehl, Bernie, Stampe, Dave, Eagan, John (1993). Virtual Reality Creations Corte Madera, CA: Waite Group Press.

Sittas, E. (1991, June). 3D design reference framework Computer Aided Design, 23(5), pp 380-384.

Sturman, David J., Zeltzer, David (1994, January). A Survey of Glove-based Input. IEEE Computer Graphics and Applications, 14(1), pp 30-39.

Vince, John (1992). 3-D Computer Animation Wokingham, England: Addison-Wesley.

Walker, John (1994). Through the Looking Glass. In Brenda Laurel (Ed.), The Art of Human-Computer Interface Design (8th printing). Reading, Massachusetts: Addison-Wesley.

Watt, Alan (1989). Fundamentals of Three-Dimensional Computer Graphics Wokingham, England: Addison-Wesley.

Webster's Dictionary (1981). Webster's Third New International Dictionary USA: G. & C. Merriam Co.

APPENDIX: IMPLEMENTATION SOURCE CODE

The following source code comprises of the VCS module (vcs.h and vcs.c) that was added to the VR-386 library. The VR-386 source code library can be obtained via anonymous ftp to **psych.utoronto.ca** in the /pub/vr-386 directory, or through the World Wide Web via the Universal Resource Location: **ftp://psych.utoronto.ca/pub/vr-386/**

VCS.h - Virtual Construction Site Header File

```
// This file contains various definitions needed for the Virtual
// Construction Site, which is built upon the VR-386 library.
//
// By Jesse Kinross-Smith
//
// March 1996

#ifndef PTRDEF
typedef void POINTER
#endif

// Define if you wish to enable debugging options (ie. F5 to dump state info)
#define DEBUG

// Control Modes Available
#define VCS_NavigationMode      0
#define VCS_PlacementMode       1

// World Modes Available
#define VCS_MainWorld           0
#define VCS_3dPalette           1
#define VCS_UserPalette         2

// Definitions for the special borders
#define VCS_BorderSize          15    // 15 pixels wide/high
#define VCS_BorderColour        15*16+12 // Grey Box
#define VCS_BorderText          15    // White Text
#define VCS_BorderOutline       15*16+8     // Darker Gray Edge
#define VCS_TextOffset          (screeninfo->xmax/2)-(8*9+4)
#define VCS_3dPaletteLabel      " 3d Object Palette "
#define VCS_UserPaletteLabel    "User Object Palette"
#define VCS_MainWorldLabel      "  Main User World  "

// Definitions for location of the VCS mode display information
#define VCS_DisplayColour       15
#define VCS_DisplayWorldLocX    5
#define VCS_DisplayWorldLocY    20
#define VCS_DisplayControlLocX  5
#define VCS_DisplayControlLocY  30

// These are returned by VCS_GetWorldMode() and VCS_GetControlMode()
#define VCS_Text3dPalette       "3d Palette"
#define VCS_TextMainWorld       "Main World"
#define VCS_TextUserPalette     "User Palette"
#define VCS_TextNavigation      "Navigation Mode"
#define VCS_TextPlacement       "Placement Mode"
#define VCS_TextUnknown         "Unknown"

// Navigation Mode - Movement Speeds Available
#define VCS_None                0
#define VCS_Slow                1
```

```
#define VCS_Medium              2
#define VCS_Fast                4
#define VCS_VeryFast            8

// Name of the file that is loaded for configuration details
#define VCS_ConfigFileName      "VCS.cfg"
#define VCS_LogFileName         "VCS.log"

// Needed to be able to define the pointer type used
#define VCS_Mouse               0
#define VCS_Glove               1

// Definitions of what values of Special equal what buttons
#define VCS_NoMouseButton       0
#define VCS_LeftMouseButton     1
#define VCS_RightMouseButton    2

// This is used for calculating movement in Navigation Mode
#define VCS_XCenterpoint        screeninfo->xmax/2
#define VCS_YCenterpoint        screeninfo->ymax/2
#define VCS_ZCenterpoint        0
#define VCS_CenterpointColour   15

// Used to reduce the speed at which Navigation Mode rotates
#define VCS_NavigationStep      10
#define VCS_GloveStep           20

// Used to calculate the movement of the mouse in Navigation Mode
#define VCS_NavigationSpeed     10

// Used similarly, except this is used in Placement Mode for rotation
#define VCS_RotationStep        5

// 3d-Objects Available in the 3d Palette

// These basic objects can actually be replaced by changing the plg file
// Please note that there MUST be no errors with the objects or else a
// crash might occur.  Check the objects separately before using them here.
#define VCS_3dFile_1            "vcs3d_1.dat"
#define VCS_3dFile_2            "vcs3d_2.dat"
#define VCS_3dFile_3            "vcs3d_3.dat"
#define VCS_3dFile_4            "vcs3d_4.dat"
#define VCS_3dFile_5            "vcs3d_5.dat"

// These define the layout and width of the various palettes
#define VCS_3dObjectGap         1150
#define VCS_3dPaletteDistance   3500
#define VCS_UserPaletteDistance 10000
#define VCS_NewObjectDistance   10000

// All of this stuff is defined externally in VCS.c
// to be linked in at compilation.  See the comments in VCS.C for more
// information on how these are used.
```

```
// Global Variables defined in VCS.c
extern int VCS_ControlMode;
extern int VCS_WorldMode;
extern int VCS_DisplayWorldMode;
extern int VCS_DisplayControlMode;
extern int VCS_ModeChangeDelay;

extern OBJLIST *VCS_Objlist_3dPalette;
extern OBJLIST *VCS_Objlist_MainWorld;
extern OBJLIST *VCS_Objlist_UserPalette;

extern char *VCS_NavigationGestures[];
extern char *VCS_PlacementGestures[];

// Functions defined in VCS.c

extern void VCS_Initialise(void);

extern void VCS_ChangeControlMode(int NewControlMode);
extern char *VCS_GetControlMode(void);

extern void VCS_ChangeWorldMode(int NewWorldMode);
extern char *VCS_GetWorldMode(void);

extern void VCS_DisplayPreRenderHook(void);
extern void VCS_DisplayPostRenderHook(void);
extern void VCS_ProcessGlove(POINTER *Glove);
extern void VCS_ProcessMouse(MDRIVER *Mouse);
extern BOOL VCS_ProcessKeys(unsigned KeyPressed);

extern void VCS_RegisterMovement(int PointerType, long x, long y, long z, int Special);

extern unsigned VCS_HorizonColours[];

extern void VCS_MakeObjectProtected(OBJECT *obj);
extern void VCS_MakeObjlistProtected(OBJLIST *objlist);
extern void VCS_MakeObjectUnprotected(OBJECT *obj);
extern BOOL VCS_is_object_protected(OBJECT *obj);

extern int VCS_SelectObject(OBJECT *obj);

extern void VCS_GrabObject(OBJECT *obj);
extern void VCS_UngrabObject(OBJECT *obj);

extern void VCS_Quit(void);
```

VCS.c - Virtual Construction Site Source Code

```
// This file contains the bulk of the code developed for the Virtual
// Construction Site.  Some other files have also been modified so as
// to use the routines in here.
//
// All code designed and implemented by Jesse Kinross-Smith.
//
// March, 1996

#include <stdio.h>

#include "vrconst.h"
#include "pointint.h"
#include "vr_api.h"
#include "pcdevice.h"

#include "VCS.h"

// Some local definitions to make the code more readable.
#define XMin                    screeninfo->xmin
#define XMax                    screeninfo->xmax
#define YMin                    screeninfo->ymin
#define YMax                    screeninfo->ymax
#define ModeChangePossible      (VCS_ModeChangeDelay == 0)
#define ModeHasBeenChanged      20
// Equivalent to the number of times VCS_ProcessGlove has to be called
// before mode changes will be allowed again.
#define Read_Only               "r"
#define ReadWrite               "w"
#define Append                  "a"

static char *VCS_ObjectMenu[] = {
   "Save object",
   "Paint object",
   "Resurface object",
   "Destroy object",
   "Twirl object",
   "Fixed/Movable",  // This option is changed dynamically
   "Information",
   NULL
};

static char *VCS_ObjectsMenu[] = {
   "Paint All Objects",
   "Resurface All Objects",
   "Destroy All Objects",
   NULL
};

static char *VCS_MainMenu[] = {
   "Navigation/Placement",    // This option is changed dynamically
   "Help",
```

```
    "Information",
    "Goto Location",
    "Save World",
    "Quit",
    NULL
};

static char VCS_ToggleFixedString[]        = "make object Fixed";
static char VCS_ToggleMoveableString[]     = "make object Movable";
static char VCS_ChangetoNavigationString[] = "Navigation mode";
static char VCS_ChangetoPlacementString[]  = "Placement mode";

// These store the current Control and World Modes which VCS is in,
// as well as the delay needed before World Mode can be changed again.
int VCS_ControlMode;
int VCS_WorldMode;
int VCS_DisplayControlMode;
int VCS_DisplayWorldMode;
int VCS_ModeChangeDelay = 0;

// These three structures hold the three worlds that will be used to
// hold the details and objects needed for the different World Modes.

OBJLIST *VCS_Objlist_3dPalette;
OBJLIST *VCS_Objlist_MainWorld;
OBJLIST *VCS_Objlist_UserPalette;


char *VCS_NavigationGestures[] = {
    "None", "None", "Slow", "Slow",
    "Slow", "Medium", "Very Fast", "Fast",
    "Medium", "Medium", "Medium", "None" };

int VCS_NavigationSpeeds[] = {
    VCS_None, VCS_None, VCS_Slow, VCS_Slow,
    VCS_Slow, VCS_Medium, VCS_VeryFast, VCS_Fast,
    VCS_Medium, VCS_Medium, VCS_Medium, VCS_None };

char *VCS_PlacementGestures[] = {
    "", "", "", "",
    "", "Rotate", "Grab", "",
    "Select", "", "", "" };

unsigned VCS_HorizonColours[16] = { 0xaf, 0xae, 0xad, 0xac, 0x79, 0x7a, 0x7b, 0x7c };
unsigned VCS_NoHorizonColours[16] = { 0x00 };

int VCS_CompassState;
POSE VCS_SavedPose;
LIGHT *VCS_PaletteLights[1];
OBJECT *VCS_SelectedObject;
OBJECT *VCS_ParentObject;
BOOL VCS_SelectedObjectFlag;
int VCS_ObjectCount = 0;
OBJECT *VCS_LastSelectedObject = NULL;
```

```
OBJECT *VCS_LastObjectInList = NULL;
BOOL VCS_MenuActive = FALSE;

COORD VCS_MoveObjectX, VCS_MoveObjectY;
COORD VCS_MoveObjectLastX, VCS_MoveObjectLastY;

// This shouldnt be hardcoded, but it is for the moment
// This should vary according to the distance of the object from the viewer
int VCS_PerspectiveDistance = 35;

BOOL VCS_LeftButtonHeld = FALSE;
BOOL VCS_RightButtonHeld = FALSE;

int VCS_MinGloveX = 32767;
int VCS_MaxGloveX = -32767;
int VCS_MinGloveY = 32767;
int VCS_MaxGloveY = -32767;

FILE *VCS_LogFile;

extern int do_horizon;
extern int show_compass;

extern struct Screeninfo *screeninfo;

static int min(int a, int b)
{
   if (a < b)
      return a;
   return b;
}

static int max(int a, int b)
{
   if (a > b)
      return a;
   return b;
}

// ########## VCS Initialisation Function ##########
// This must be called first so VCS can setup everything it needs to run.

void VCS_Initialise(void)
{
   FILE *datafile;
   POSE ObjectPose = ZERO_POSE;

   VCS_ControlMode = VCS_NavigationMode;
   VCS_WorldMode = VCS_MainWorld;
   VCS_DisplayControlMode = TRUE;
   VCS_DisplayWorldMode = TRUE;
   VCS_SelectedObject = FALSE;

   VCS_Objlist_3dPalette = new_objlist();
```

```
    VCS_Objlist_MainWorld = new_objlist();
    VCS_Objlist_UserPalette = new_objlist();

    ObjectPose.z = (COORD) VCS_3dPaletteDistance*2;

    // Load the objects for the 3d Palette and add them to the objlist for it
    ObjectPose.x = (COORD) -VCS_3dObjectGap;
    ObjectPose.y = (COORD)  0;
    datafile = fopen(VCS_3dFile_1, Read_Only);
    load_plg_to_objlist(datafile, VCS_Objlist_3dPalette, 1, &ObjectPose, 1,1,1, 0);
    fclose(datafile);

    ObjectPose.x = (COORD)  0;
    ObjectPose.y = (COORD)  0;
    datafile = fopen(VCS_3dFile_2, Read_Only);
    load_plg_to_objlist(datafile,VCS_Objlist_3dPalette, 1, &ObjectPose, 1,1,1, 0);
    fclose(datafile);

    ObjectPose.x = (COORD)  VCS_3dObjectGap;
    ObjectPose.y = (COORD)  0;
    datafile = fopen(VCS_3dFile_3, Read_Only);
    load_plg_to_objlist(datafile, VCS_Objlist_3dPalette, 1, &ObjectPose, 1,1,1, 0);
    fclose(datafile);

    ObjectPose.x = (COORD)  VCS_3dObjectGap;
    ObjectPose.y = (COORD)  VCS_3dObjectGap;
    datafile = fopen(VCS_3dFile_4, Read_Only);
    load_plg_to_objlist(datafile, VCS_Objlist_3dPalette, 1, &ObjectPose, 1,1,1, 0);
    fclose(datafile);

    ObjectPose.x = (COORD)  0;
    ObjectPose.y = (COORD)  VCS_3dObjectGap;
    datafile = fopen(VCS_3dFile_5, Read_Only);
    load_plg_to_objlist(datafile, VCS_Objlist_3dPalette, 1, &ObjectPose, 1,1,1, 0);
    fclose(datafile);

    //VCS_PaletteLights[0] = create_light(NULL, POINT_LIGHT, 128);
    //position_pointlight(VCS_PaletteLights[0], &ObjectPose);

    // Lets initialise the log file for this run by writing a header to it.
    // All subsequent writes to this file will be using Append.
    VCS_LogFile = fopen(VCS_LogFileName, ReadWrite);
    fprintf(VCS_LogFile, "##### ##### ##### Virtual Construction Site - Runtime Log #####
##### #####\n\n");
    fclose(VCS_LogFile);
}


// ########## Control Mode Functions ##########
// These routines let you change between the possible control modes:
// Navigation Mode   Where the pointer is used to allow the user to fly
//                   around the environment.
// Placement Mode - Where the pointer is used to allow local object
//                  manipulation, so the user can select, grab, and move
```

```
//                  objects.

void VCS_ChangeControlMode(int NewControlMode)
{
   if (VCS_WorldMode != VCS_MainWorld)
   {
   // Can only change to Navigation Mode when in the Main World.
   // Ignore keypresses otherwise.
      return;
   }

   VCS_ControlMode = NewControlMode;
}

char *VCS_GetControlMode(void)
{
   switch(VCS_ControlMode)
   {
      case VCS_NavigationMode:   return VCS_TextNavigation;
      case VCS_PlacementMode:    return VCS_TextPlacement;
      default:                   return VCS_TextUnknown;
   }
}


// ########## Object Protection Functions ##########
// Object Protection basically means that the objects are marked as
// system owned and aren't changed at all when switching world modes.
// This is done by default for all glove and 3d cursors loaded, and
// can be used for other objects it desired.
// Functions are provided to make objects, and object lists protected,
// and unprotected.

void VCS_MakeObjectProtected(OBJECT *obj)
// This sets the SYSTEM_OWNED flag for the appropriate object.
{
   set_object_flags(obj, SYSTEM_OWNED, TRUE);
}

void VCS_MakeObjlistProtected(OBJLIST *objlist)
{
   walk_objlist(objlist, VCS_MakeObjectProtected);
}

void VCS_MakeObjectUnprotected(OBJECT *obj)
{
   set_object_flags(obj, SYSTEM_OWNED, FALSE);
}

BOOL VCS_is_object_protected(OBJECT *obj)
{
   return is_system_owned(obj);
}
```

```
// ########## Object Logging Functions ##########
// These functions let you dump a particular object's details into the log
// file, or let you do a total dump of the current system state and all
// object lists.  This is useful for debugging purposes.

void VCS_Log_DumpObject(OBJECT *obj)
{
   int numverts;
   int numpolys;
   POSE p;

   get_obj_info(obj, &numverts, &numpolys);
   get_object_world_pose(obj, &p);

   fprintf(VCS_LogFile, "  ObjDump - Vertices: %d, Polys: %d (%ld,%ld,%ld) -
%s%s%s%s\n",
         numverts, numpolys, p.x, p.y, p.z,
      (is_object_moveable(obj)   ? "Moveable"      : "Fixed"),
      (is_object_visible(obj)    ? ", Visible"     : ", Invisible"),
      (is_object_selected(obj)   ? ", Selected"    : ""),
         (VCS_is_object_protected(obj) ? ", Protected"     : ""));
}

void VCS_Log_DumpState(void)
{
   VCS_LogFile = fopen(VCS_LogFileName, Append);

   fprintf(VCS_LogFile, "Current World Mode: %s\n", VCS_GetWorldMode());
   fprintf(VCS_LogFile, "World Contents\n");
   do_for_all_objects(VCS_Log_DumpObject);

   fprintf(VCS_LogFile, "Inactive Objlist Contents\n");
   walk_objlist(inactive_object_list, VCS_Log_DumpObject);

   fprintf(VCS_LogFile, "3dPalette Objlist Contents\n");
   walk_objlist(VCS_Objlist_3dPalette, VCS_Log_DumpObject);

   fprintf(VCS_LogFile, "Main World Objlist Contents\n");
   walk_objlist(VCS_Objlist_MainWorld, VCS_Log_DumpObject);
   fprintf(VCS_LogFile, "\n");
   fclose(VCS_LogFile);
}

void VCS_DoObjectCount(OBJECT *obj)
{
   VCS_ObjectCount++;
   VCS_LastObjectInList = obj;
}

void VCS_CountSelectedObjects(void)
{
   VCS_ObjectCount = 0;
```

```
   VCS_LastObjectInList = NULL;
   do_for_all_selected(VCS_DoObjectCount);
}

void VCS_SaveSelectedObject(void)
// This routine will save a current single selection to a specified file
{
   char SaveFileName[8];
   char message[40];
   char prompt[] = "Name of Save File: ";
   FILE *SaveFile;

   VCS_CountSelectedObjects();
   if (VCS_ObjectCount < 1)
      sprintf(message, "No Objects Selected to Save");
   if (VCS_ObjectCount > 1)
      sprintf(message, "Too Many Objects Selected to Save");
   if (VCS_ObjectCount == 1)
   {
      askfor(prompt, SaveFileName, 11);
      SaveFile = fopen(SaveFileName, ReadWrite);
      if (SaveFile == NULL)
         sprintf(message, "Unable to Write to Save File");
      else
      {
         save_plg(VCS_LastObjectInList, SaveFile, 1);
         sprintf(message, "Object Saved as %s Successfully", SaveFileName);
         fclose(SaveFile);

         VCS_LogFile = fopen(VCS_LogFileName, Append);
         fprintf(VCS_LogFile, "* Saving PLG file for an Object *\n  Object Info: ");
         VCS_Log_DumpObject(VCS_LastObjectInList);
         fprintf(VCS_LogFile, "  %s\n* End of Saving Object *\n\n", message);
         fclose(VCS_LogFile);
      }
   }
   popmsg(message);
   tdelay(600);
   world_changed++;
}

// ########## Display Option Functions ##########
// These functions allow you to turn on/off the compass and horizon display.
// This is more for use internally (as these are switched off when changing
// worlds), but the functions are given for future use as well.

void VCS_TurnOffCompass(void)
// This saves the current compass state and ensures the compass display is off
{
   VCS_CompassState = show_compass;
   show_compass = FALSE;
}

void VCS_TurnOnCompass(void)
```

```
// This doesnt actually turn the compass on, so much as return it back to its
// previous state.
{
   show_compass = VCS_CompassState;
}

void VCS_TurnOffHorizon(void)
// This turns off the horizon, replacing it with a black background.
{
   do_horizon = FALSE;
   set_horizon(1, VCS_NoHorizonColours, 1);
}

void VCS_TurnOnHorizon(void)
// This turns on the horizon, displaying a gradiated blue sky and green ground
{
   do_horizon = TRUE;
   set_horizon(8, VCS_HorizonColours, 48);
}


// ########## State Saving Primitives ##########
// These primitives are used to juggle the objects in the current world so
// as to emulate multiple worlds.

void VCS_SaveMainWorldObjects(OBJECT *obj)
{
   if (!VCS_is_object_protected(obj))
   {
      remove_object_from_world(obj);
      add_to_objlist(VCS_Objlist_MainWorld, obj);
   }
}

void VCS_Save3dPaletteObjects(OBJECT *obj)
{
   if (!VCS_is_object_protected(obj))
   {
      remove_object_from_world(obj);
      add_to_objlist(VCS_Objlist_3dPalette, obj);
   }
}

void VCS_SaveUserPaletteObjects(OBJECT *obj)
{
   if (!VCS_is_object_protected(obj))
   {
      remove_object_from_world(obj);
      add_to_objlist(VCS_Objlist_UserPalette, obj);
   }
}

void VCS_MoveObjectInFrontOfPoint(POSE centre, OBJECT *obj, COORD distance)
```

```
{
   POSE TranslationPose = DONTCARE_POSE,
        RotationPose    = DONTCARE_POSE,
        RotationPose2   = DONTCARE_POSE;

   set_object_world_pose(obj, &centre);
   global_update_object(obj);

   VCS_GrabObject(obj);

   // Shift object back so its in line with the origin axis

   RotationPose.rx = (ANGLE) 0;
   RotationPose.ry = (ANGLE) 0;
   RotationPose.rz = (ANGLE) 0;
   set_object_pose(obj, &RotationPose);

   // Make the Z Translation
   get_object_pose(obj, &TranslationPose);
   TranslationPose.z += (COORD) distance;
   set_object_pose(obj, &TranslationPose);

   // Shift the object back to its original axis
   RotationPose2.rx = (ANGLE) centre.rx;
   RotationPose2.ry = (ANGLE) centre.ry;
   RotationPose2.rz = (ANGLE) centre.rz;
   set_object_pose(obj, &RotationPose2);

   VCS_UngrabObject(obj);
   global_update_object(obj);
   world_changed++;
}

// ########## World Mode Switching Primitives ##########
// These are internal routines used for initialising the new world as the
// current world is changed.

OBJECT *VCS_CopyObject(OBJECT *obj)
// This routine is designed to be a primitive for copying an existing object,
// as opposed to loading one in from disk (which is the only way an object)
// can be loaded with the existing library of routines.
// A pointer to an exact clone of the object passed is returned.  The user
// must still place this object, and MUST perform at least a compute_object()
// on it once the object is in place.
{
   COORD x, y, z;
   unsigned colour;
   POLY *poly;
   int numpolys, numverts, tpverts;
   WORD polyverts;
   WORD polycount, vertexcount;
   int vertexnum;
   OBJECT *newobj;
   POSE pose;
```

```
    get_obj_info(obj, &numverts, &numpolys);
    tpverts  = (int) total_object_pverts(obj);
    newobj = create_fixed_object(numverts, numpolys, tpverts);
    set_representation_size(newobj, get_representation_size(obj));
    for (vertexcount = 0; vertexcount < numverts; vertexcount++)
    {
       get_vertex_world_info(obj, vertexcount, &x, &y, &z);
       add_vertex(newobj, x, y, z);
    }

    for (polycount = 0; polycount < numpolys; polycount++)
    {
       get_poly_info(obj, polycount, &colour, &polyverts);
       poly = add_poly(newobj, colour, polyverts);
       if ((polyverts > 0) && (poly != NULL))
       {
          for (vertexcount = 0; vertexcount < polyverts; ++vertexcount)
          {
             vertexnum = (int) get_poly_vertex_index(obj, polycount, vertexcount);
             add_point(newobj, poly, vertexnum);
          }
       }
    }

    set_obj_flags(newobj, get_obj_flags(obj));
    get_object_pose(obj, &pose);
    set_object_pose(newobj, &pose);
    get_object_world_pose(obj, &pose);
    set_object_world_pose(newobj, &pose);
    set_object_sorting(newobj, get_object_sorting(obj));

    return newobj;
}

void VCS_Setup3dPalette(void)
{
    POSE NewBodyPose = ZERO_POSE;

    do_for_all_objects(VCS_SaveMainWorldObjects);
    add_objlist_to_world(VCS_Objlist_3dPalette);

    VCS_SavedPose = *body_pose;
    NewBodyPose.z += (COORD) VCS_3dPaletteDistance;
    NewBodyPose.y -= (COORD) VCS_3dObjectGap/2;
    *body_pose = NewBodyPose;

    VCS_SelectedObject = NULL;
    VCS_PerspectiveDistance = 5;
    VCS_TurnOffCompass();
    VCS_TurnOffHorizon();
}

void VCS_SetupMainWorld(int PreviousWorldMode)
```

```
{
   *body_pose = VCS_SavedPose;

   // Protect the selected object, so it gets kept when we move to the
   // Main World.
   if (VCS_LeftButtonHeld)
      VCS_MakeObjectProtected(VCS_SelectedObject);

   if (PreviousWorldMode == VCS_3dPalette)
      do_for_all_objects(VCS_Save3dPaletteObjects);
   else
      do_for_all_objects(VCS_SaveUserPaletteObjects);
   add_objlist_to_world(VCS_Objlist_MainWorld);

   // Unprotect the object and shift it into position so it is in
   // front of the user
   if (VCS_LeftButtonHeld)
   {
      // Unprotect the object and shift it into position so that it is in
      // front of the user
      VCS_MakeObjectUnprotected(VCS_SelectedObject);
      VCS_MoveObjectInFrontOfPoint(*body_pose, VCS_SelectedObject, (COORD)
VCS_NewObjectDistance);
      VCS_MoveObjectLastX = VCS_XCenterpoint;
      VCS_MoveObjectLastY = VCS_YCenterpoint;
   }
   VCS_PerspectiveDistance = 35;
   VCS_TurnOnCompass();
   VCS_TurnOnHorizon();
}

static void VCS_ShiftObjectsIntoPosition(OBJECT *obj)
{
   POSE NewPose = ZERO_POSE;
   VCS_MakeObjectUnprotected(obj);
   NewPose.y += VCS_UserPaletteDistance;
   set_object_pose(obj, &NewPose);
}

void VCS_SetupUserPalette(void)
{
   POSE NewBodyPose = ZERO_POSE;

   // Lets only grab the last selected object

   if (VCS_LeftButtonHeld)
   {
      VCS_CountSelectedObjects();
      VCS_SelectedObject = VCS_LastSelectedObject;
      VCS_MakeObjectProtected(VCS_SelectedObject);
   }
   do_for_all_objects(VCS_SaveMainWorldObjects);
   add_objlist_to_world(VCS_Objlist_UserPalette);
```

```
   VCS_SavedPose = *body_pose;
   NewBodyPose.z += (COORD) VCS_3dPaletteDistance;
   NewBodyPose.y -= (COORD) VCS_3dPaletteDistance*2;
   *body_pose = NewBodyPose;

   if (VCS_LeftButtonHeld)
   {
      VCS_ShiftObjectsIntoPosition(VCS_SelectedObject);
      VCS_MoveObjectLastX = VCS_XCenterpoint;
      VCS_MoveObjectLastY = VCS_YCenterpoint;
   }

   VCS_PerspectiveDistance = 20;
   VCS_TurnOffCompass
   VCS_TurnOffHorizon();
}


// ########## World Mode Functions ##########
// These routines allow you to change the current world mode between:
//    VCS_3dPalette   - A Collection of Basic 3d Objects
//    VCS_MainWorld   - The Main User World
//    VCS_UserPalette - A Collection of User Configured Objects
// Functionality is also provided for getting the current world mode.

void VCS_ChangeWorldMode(int NewWorldMode)
{
   if (VCS_WorldMode == NewWorldMode)
      return; // Lets ignore changes to the same state as it is currently.

   if (VCS_ControlMode == VCS_NavigationMode)
   {
      // This should put the user into Placement Mode automatically.
      VCS_ChangeControlMode(VCS_PlacementMode);
   }

   switch(NewWorldMode)
   {
      case VCS_3dPalette:    VCS_Setup3dPalette(); break;
      case VCS_MainWorld:    VCS_SetupMainWorld(VCS_WorldMode); break;
      case VCS_UserPalette:  VCS_SetupUserPalette(); break;
      default:  // Obviously not a valid world mode, so lets ignore it.
         return;
   }
   display_changed++;
   world_changed++;
   VCS_WorldMode = NewWorldMode;
}

char *VCS_GetWorldMode(void)
{
   switch(VCS_WorldMode)
   {
      case VCS_3dPalette:    return VCS_Text3dPalette;
```

```
      case VCS_MainWorld:     return VCS_TextMainWorld;
      case VCS_UserPalette:   return VCS_TextUserPalette;
      default:                return VCS_TextUnknown;
   }
}

BOOL VCS_IsShiftKeyPressed(void)
// Returns TRUE if either of the shift keys are being pressed.
// Otherwise FALSE is returned.
{
   int keystatus;

   // The left and right shift keys use bits 0 and 1 respectively.
   // Thus by using a mask of 3 (00000011) we isolate only the shift keys for
   // reading.  Anything other than 0 means the shift key is being pressed!
   keystatus = bioskey(2) & 3;
   if (keystatus != 0)
      return TRUE;
   else
      return FALSE;
}

void VCS_MoveObjectXY(OBJECT *obj)
{
   // This routine is used to move objects around after they are grabbed
   // with the mouse (simulating a drag-and-drop method of interaction).

   COORD xTranslation, yTranslation;
   ANGLE xAngle, yAngle, zAngle;
   POSE WorldPose, OldPose,
        TranslationPose = DONTCARE_POSE,
        RotationPose = DONTCARE_POSE;

   // Calculate the differences from the last movement location
   xTranslation = (COORD) (VCS_MoveObjectLastX - VCS_MoveObjectX);
   yTranslation = (COORD) (VCS_MoveObjectLastY - VCS_MoveObjectY);

   // Attach the object to the user.
   VCS_GrabObject(obj);

   // Rotate the object around the user's axis according to the cursor's
   // new movement on the screen.

   get_object_pose(body_seg, &WorldPose);

   xAngle = WorldPose.rx;
   yAngle = WorldPose.ry;
   zAngle = WorldPose.rz;

   // Shift object back so its in line with the origin axis (-Z)
   get_object_pose(obj, &OldPose);
   RotationPose.rx = OldPose.rx - xAngle;
   RotationPose.ry = OldPose.ry - yAngle;
   RotationPose.rz = OldPose.rz - zAngle;
```

```
    set_object_pose(obj, &RotationPose);

    // Make the X,Y Translation
    // (X is reversed because of the orientation of those world modes)
    get_object_pose(obj, &OldPose);
    TranslationPose.x = OldPose.x - (xTranslation * VCS_PerspectiveDistance);
    TranslationPose.y = OldPose.y + (yTranslation * VCS_PerspectiveDistance);
    set_object_pose(obj, &TranslationPose);

    // Shift the object back to its original axis
    get_object_pose(obj, &OldPose);
    RotationPose.rx = OldPose.rx + xAngle;
    RotationPose.ry = OldPose.ry + yAngle;
    RotationPose.rz = OldPose.rz + zAngle;
    set_object_pose(obj, &RotationPose);

    // Update the object afterwards - this should've translated the object
    // using the camera view as the axis. *fingers crossed*
    update_object(obj);

    // Remove the connection between the object and the user.
    VCS_UngrabObject(obj);

    VCS_MoveObjectLastX = VCS_MoveObjectX;
    VCS_MoveObjectLastY = VCS_MoveObjectY;
    world_changed++;
}

// ########## Hooks for Rendering and Input Devices ##########
// These routines are linked in with the internal engine of VR-386,
// and will be called at various times throughout the package.
// This is useful for displaying information, or catching input device
// information directly.

void VCS_DisplayPreRenderHook(void)
// This routine is linked into the prerender_process() in USCREEN.C.
{
    if (VCS_WorldMode != VCS_MainWorld)
        setup_lights(VCS_PaletteLights, 1);
}

void VCS_DisplayPostRenderHook(void)
{
    // This small bit of code should display the world buttons on
    // the screen, and should put names on them.  This is called from
    // postrender_process() in USCREEN.C.

    char buff[30];

    if (VCS_DisplayWorldMode)
        user_text(VCS_DisplayWorldLocX, VCS_DisplayWorldLocY, VCS_DisplayColour,
VCS_GetWorldMode());
    if (VCS_DisplayControlMode)
```

```
      user_text(VCS_DisplayControlLocX, VCS_DisplayControlLocY, VCS_DisplayColour,
VCS_GetControlMode());

   switch(VCS_ControlMode)
   {
      case VCS_NavigationMode:
      {
         // Display the centerpoint on the screen for reference
         vgapoint(VCS_XCenterpoint, VCS_YCenterpoint, VCS_CenterpointColour);
         break;
      }
      case VCS_PlacementMode:
      {
         switch(VCS_WorldMode)
         {
            case VCS_3dPalette:
            {
               // Nothing up the Top

               user_box(XMin, YMax-VCS_BorderSize+1, XMax, YMax, VCS_BorderColour);
               user_box(XMin, YMax-VCS_BorderSize, XMax, YMax-VCS_BorderSize+1,
VCS_BorderOutline);
               user_box(XMin, YMax-VCS_BorderSize+1, XMax, YMax-VCS_BorderSize+2,
VCS_BorderOutline+1);
               sprintf(buff, VCS_MainWorldLabel);
               user_text(XMin+VCS_TextOffset, YMax-VCS_BorderSize+5, VCS_BorderText,
buff);
               break;
            }
            case VCS_MainWorld:
            {
               user_box(XMin, YMin, XMax, YMin+VCS_BorderSize+1, VCS_BorderColour);
               user_box(XMin, YMin+VCS_BorderSize-1, XMax, YMin+VCS_BorderSize,
VCS_BorderOutline+1);
               user_box(XMin, YMin+VCS_BorderSize, XMax, YMin+VCS_BorderSize+1,
VCS_BorderOutline);
               sprintf(buff, VCS_3dPaletteLabel);
               user_text(XMin+VCS_TextOffset, YMin+3, VCS_BorderText, buff);

               user_box(XMin, YMax-VCS_BorderSize+1, XMax, YMax, VCS_BorderColour);
               user_box(XMin, YMax-VCS_BorderSize, XMax, YMax-VCS_BorderSize+1,
VCS_BorderOutline);
               user_box(XMin, YMax-VCS_BorderSize+1, XMax, YMax-VCS_BorderSize+2,
VCS_BorderOutline+1);
               sprintf(buff, VCS_UserPaletteLabel);
               user_text(XMin+VCS_TextOffset, YMax-VCS_BorderSize+5, VCS_BorderText,
buff);
               break;
            }
            case VCS_UserPalette:
            {
               user_box(XMin, YMin, XMax, YMin+VCS_BorderSize+1, VCS_BorderColour);
               user_box(XMin, YMin+VCS_BorderSize-1, XMax, YMin+VCS_BorderSize,
VCS_BorderOutline+1);
```

```
                user_box(XMin, YMin+VCS_BorderSize, XMax, YMin+VCS_BorderSize+1,
VCS_BorderOutline);
                sprintf(buff, VCS_MainWorldLabel);
                user_text(XMin+VCS_TextOffset, YMin+3, VCS_BorderText, buff);

                // Nothing down the Bottom
                break;
             }
          }
       }
    }
    display_changed++;
}

void VCS_ProcessGlove(POINTER *Glove)
{
    switch(Glove->keys)
    {
       case G_AKEY :  VCS_ChangeControlMode(VCS_NavigationMode); break;
       case G_BKEY :  VCS_ChangeControlMode(VCS_PlacementMode); break;
    }
    VCS_RegisterMovement(VCS_Glove, Glove->x, Glove->y, Glove->z, Glove->gesture);
}

long VCS_ScaleGloveXCoord(long value)
{
    // The boundaries for glove coordinates seems to be -625 to +625.
    // I'll convert this to screen coordinates (0 - XMax)
    float ratio;

    value += 625;
    if (value == 0)
       return 0;
    ratio = (float) value / 1250;
    return (long) (ratio * XMax);
}

long VCS_ScaleGloveYCoord(long value)
{
    // The boundaries for glove coordinates seems to be -625 to +625.
    // I'll convert this to screen coordinates (0 - YMax)
    float ratio;

    value += 625;
    if (value == 0)
       return 0;
    ratio = (float) value / 1250;
    return (long) (ratio * YMax);
}

void VCS_ProcessMouse(PDRIVER *Mouse)
{
    int x, y, z = 0, Buttons;
    mouse_last(Mouse, &x, &y, &Buttons);
```

```
   VCS_RegisterMovement(VCS_Mouse, (long) x, (long) y, (long) z, Buttons);
}

// Externs for the painting and surfacing stuff.
extern unsigned stype[];
extern unsigned paint;
extern unsigned surface;
extern unsigned paintcolor;
extern void surf_it(OBJECT *obj);
extern void color_it(OBJECT *obj);
extern unsigned int get_surface(void);
extern BOOL can_point_2D(void);
extern int manip_2D_avail;
extern PDRIVER *cursor_device;

void VCS_RegisterMovement(int PointerType, long x, long y, long z, int Special)
{
   // PointerType is can be either of the following:
   //    VCS_Mouse
   //    VCS_Glove
   // The Special field is used to contain either button configuration,
   // or the current Gesture ID.

   if (VCS_ModeChangeDelay != 0)
      VCS_ModeChangeDelay--;

   switch(VCS_ControlMode)
   {
      case VCS_PlacementMode:
      {
         switch(PointerType)
         {
            case VCS_Mouse:
            {
               if ((Special == VCS_RightMouseButton) && (VCS_MenuActive == FALSE))
               {
                  char user_buffer[50];
                  OBJECT *invis_seg;
                  WORD SelectionX, SelectionY;

                  VCS_MenuActive = TRUE;
                  save_screen();
                  VCS_CountSelectedObjects();
                  switch(VCS_ObjectCount)
                  {
                     case 0:
                        if (VCS_ControlMode == VCS_NavigationMode)
                           VCS_MainMenu[0] = VCS_ChangetoPlacementString;
                        else
                           VCS_MainMenu[0] = VCS_ChangetoNavigationString;

                        switch(menu(VCS_MainMenu))
                        {
                           case 'N':
```

```
                        VCS_ChangeControlMode(VCS_NavigationMode);
                        break;
                     case 'P':
                        VCS_ChangeControlMode(VCS_PlacementMode);
                        break;
                     // Help
                     case 'H':
                        process_a_key('H');
                        break;
                     // Information
                     case 'I':
                        process_a_key('I');
                        break;
                     // Goto location
                     case 'G':
                        askfor("X,Y,Z: ", user_buffer, 50);
                        if (user_buffer[0])
                           sscanf(user_buffer, "%ld,%ld,%ld", &body_pose->x,
&body_pose->y, &body_pose->z);
                        position_changed++;
                        break;
                     // Save world
                     case 'S':
                        break;
                     // Quit
                     case 'Q':
                        process_a_key('Q');
                        break;
                  }
                  break;

               case 1:
                  if (is_object_moveable(VCS_LastObjectInList))
                     VCS_ObjectMenu[5] = VCS_ToggleFixedString;
                  else
                     VCS_ObjectMenu[5] = VCS_ToggleMoveableString;
                  switch(menu(VCS_ObjectMenu))
                  {
                     // Save object
                     case 'S':
                        VCS_SaveSelectedObject();
                        break;
                     // Paint object
                     case 'P':
                        if (!can_point_2D()) break;
                        if (!manip_2D_avail) break;
                        disp_palette();
                        do
                        {
                           move_till_click(cursor_device, 1, &SelectionX,
&SelectionY);
                        } while (SelectionY > 128 || SelectionX > 160);
                        paintcolor = 16 * (SelectionY / 8) + SelectionX / 10;
                        if (surface == 0)
```

```
                              paint = paintcolor;
                           else
                              paint = (surface | ((paintcolor << 4) & 0x0FF0) + 10);
/* hue, brightness *16 */
                           world_changed++;
                           color_it(VCS_LastObjectInList); // In Keyboard.c
                           break;
                        // Resurface object
                        case 'R':
                           if (get_surface() != 1)
                           {
                              surf_it(VCS_LastObjectInList);
                              world_changed++;
                           }
                           break;
                        // Delete object
                        case 'D':
                           delete_object(VCS_LastObjectInList);
                           world_changed++;
                           break;
                        // Twirl object
                        case 'T':
                           break;
                        // Toggle Fixed/Movable
                        case 'F':
                        case 'M':
                           if (is_object_moveable(VCS_LastObjectInList))
                           {
                              VCS_LastObjectInList =
make_moveable_object_fixed(VCS_LastObjectInList, &invis_seg, TRUE);
                              delete_object(invis_seg);
                              global_update_object(VCS_LastObjectInList);
                           }
                           else
                           {
                              VCS_LastObjectInList =
make_fixed_object_moveable(VCS_LastObjectInList, NULL);
                              global_update_object(VCS_LastObjectInList);
                           }
                           break;
                        // Information about the object
                        case 'I':
                           seg_info(0);
                           get_response(1);
                           break;
                     }
                     break;

                  default:
                     switch(menu(VCS_ObjectsMenu))
                     {
                        // Paint all objects
                        case 'P':
                           if (!can_point_2D()) break;
```

```
                        if (!manip_2D_avail) break;
                        disp_palette();
                        do
                        {
                           move_till_click(cursor_device, 1, &SelectionX,
&SelectionY);
                        } while (SelectionY > 128 || SelectionX > 160);
                        paintcolor = 16 * (SelectionY / 8) + SelectionX / 10;
                        if (surface == 0)
                           paint = paintcolor;
                        else
                           paint = (surface | ((paintcolor << 4) & 0x0FF0) + 10);
/* hue, brightness *16 */
                        world_changed++;
                        do_for_all_selected(color_it); // In Keyboard.c
                        break;

                     // Resurface all objects
                     case 'R':
                        if(get_surface()) break;
                        do_for_all_selected(surf_it);
                        world_changed++;
                        break;

                     // Destroy all objects
                     case 'D':
                        sprintf(user_buffer, "Delete %d object%s!  Are you sure?",
                           VCS_ObjectCount, (VCS_ObjectCount > 1) ? "s" : "");
                        popmsg(user_buffer);
                        if (toupper(get_response(1)) == 'Y')
                        {
                           do_for_all_selected(delete_visobj);
                           world_changed++;
                        }
                        break;
                  }
               }
               while (get_response(0) != 0);
               restore_screen();
               VCS_MenuActive = FALSE;
            }

            if (!VCS_LeftButtonHeld && Special == VCS_LeftMouseButton)
            {
               // Right Button has just been pressed initially
               VCS_LeftButtonHeld = TRUE;
               VCS_MoveObjectLastX = x;
               VCS_MoveObjectLastY = y;

               // Actual Object Selection is done using VCS_SelectObject()
               // which is called just after this routine.
            }
            else
            {
```

```
   if (VCS_LeftButtonHeld && Special == VCS_LeftMouseButton)
   // Button is still being pressed
   {
      // if VCS_SelectObject() hasnt selected anything,
      // then turn off the held status for the button.
      if (VCS_LastSelectedObject == NULL)
         VCS_LeftButtonHeld = FALSE;
      else
      {
         // Otherwise move the object according to the cursor's
         // current location
         VCS_MoveObjectX = x;
         VCS_MoveObjectY = y;
         if (VCS_WorldMode == VCS_MainWorld)
            do_for_all_selected(VCS_MoveObjectXY);
         else
            if (VCS_SelectedObject != NULL)
               VCS_MoveObjectXY(VCS_SelectedObject);
      }
   }
   else
   {
      if (VCS_LeftButtonHeld && Special == VCS_NoMouseButton)
      {
         if (VCS_WorldMode == VCS_3dPalette)
            if (VCS_SelectedObject != NULL)
            {
               delete_object(VCS_SelectedObject);
               VCS_SelectedObject = NULL;
            }

         if (VCS_WorldMode == VCS_UserPalette)
            if (VCS_SelectedObject != NULL)
            {
               if (VCS_ParentObject != NULL)
               {
                  delete_object(VCS_ParentObject);
                  VCS_ParentObject = NULL;
               }
               unhighlight_object(VCS_SelectedObject);
               VCS_SelectedObject = NULL;
            }

         // Button has just been released..
         if (!VCS_IsShiftKeyPressed() && (VCS_WorldMode == VCS_MainWorld))
         {
            // As long as the shift key wasnt held down
            if (VCS_LastSelectedObject != NULL)
               unhighlight_object(VCS_LastSelectedObject);
         }
         VCS_LeftButtonHeld = FALSE;
      }
   }
}
```

```
            }
            break;

            case VCS_Glove:
            {
               x = VCS_ScaleGloveXCoord(x);
               y = VCS_ScaleGloveYCoord(y);
               VCS_MinGloveX = min(VCS_MinGloveX, x);
               VCS_MaxGloveX = max(VCS_MaxGloveX, x);
               VCS_MinGloveY = min(VCS_MinGloveY, y);
               VCS_MaxGloveY = max(VCS_MaxGloveY, y);
               switch(Special)
               {
                  case G_FIST :  do_3D_manip(GRASP_DO); break;
                  case G_PINCH:  do_3D_manip(ROTATE_DO); break;
// Pinch is not really supported in the initial VCS design - Should it be?
                  case G_POINT:  do_3D_manip(SELECT_DO); break;
                  default:       do_3D_manip(FREE_DO); break;
               }
               break;
            }
         }
         // Check Border Areas for selection, but dont allow multiple
         // mode changes straight after one another.
         if (y <= YMin+VCS_BorderSize)
         {
            // Change World Mode to 3d Object Palette
            if ((VCS_WorldMode == VCS_MainWorld) && ModeChangePossible)
            {
               VCS_ChangeWorldMode(VCS_3dPalette);
               VCS_ModeChangeDelay = ModeHasBeenChanged;
            }

            if ((VCS_WorldMode == VCS_UserPalette) && ModeChangePossible)
            {
               VCS_ChangeWorldMode(VCS_MainWorld);
               VCS_ModeChangeDelay = ModeHasBeenChanged;
            }
         }

         if (y >= YMax-VCS_BorderSize)
         {
            // Change World Mode to User Object Palette
            if ((VCS_WorldMode == VCS_MainWorld) && ModeChangePossible)
            {
               VCS_ChangeWorldMode(VCS_UserPalette);
               VCS_ModeChangeDelay = ModeHasBeenChanged;
            }
            if ((VCS_WorldMode == VCS_3dPalette) && ModeChangePossible)
            {
               VCS_ChangeWorldMode(VCS_MainWorld);
               VCS_ModeChangeDelay = ModeHasBeenChanged;
            }
         }
```

```
   if (x <= XMin+VCS_BorderSize)
   {
      // Rotate Viewpoint Negatively around the Y axis.
      if (VCS_WorldMode == VCS_MainWorld)
         body_pose->ry -= float2angle((VCS_BorderSize-x)/VCS_RotationStep);
   }
   if (x >= XMax-VCS_BorderSize)
   {
      // Rotate Viewpoint Positively around the Y axis.
      if (VCS_WorldMode == VCS_MainWorld)
         body_pose->ry -= float2angle((XMax-VCS_BorderSize-x)/VCS_RotationStep);
   }
   break;
}

case VCS_NavigationMode:
{
   // Navigation Mode allows easy manipulation of the user's
   // viewpoint through movement of the hand in 3D space, and
   // simple gesture control.
   // Left or right from the centerpoint indicates minor rotation
   // around the Y axis, while movement above or below the
   // centerpoint indicates movement into or away from the screen.
   // Both varying according to the degree from which the hand
   // is moved away from the centerpoint of the screen.
   // If the hand is clenched, then the viewpoint is moved quickly
   // towards the direction the viewpoint is facing, a flat hand
   // indicating no movement, and gradients in between giving
   // varying speeds of movement forward.
   // In the case of a mouse, the buttons are used instead.
   // The left mouse button indicates steady movement in a
   // positive direction towards the screen, while the right mouse
   // button indicates steady movement away from the screen.


   switch(PointerType)
   {
      case VCS_Mouse:
      {
         key_set_direct((int)((x-VCS_XCenterpoint)/VCS_NavigationStep),
                        (int)((y-VCS_YCenterpoint)/VCS_NavigationStep), 0);

         if (Special != 0)
         {
            // Left button indicates positive movement
            if (Special == VCS_LeftMouseButton)
               key_set_move(0,-1,0,1);
            // With the Right Button indicates negative movement
            if (Special == VCS_RightMouseButton)
               key_set_move(0,1,0,1);
         }
      }
      break;
```

```
            case VCS_Glove:
            {
               // I should factor in the z coordinate here somehow too...
               // Should I allow finger gestures, and just use a 3D centerpoint
               // as the basis for movement?  Will this be intuitive enough?

               x = VCS_ScaleGloveXCoord(x);
               y = VCS_ScaleGloveYCoord(y);

               VCS_MinGloveX = min(VCS_MinGloveX, x);
               VCS_MaxGloveX = max(VCS_MaxGloveX, x);
               VCS_MinGloveY = min(VCS_MinGloveY, y);
               VCS_MaxGloveY = max(VCS_MaxGloveY, y);
               key_set_direct((int)((x-
VCS_XCenterpoint)/VCS_GloveStep*VCS_NavigationSpeeds[Special]),
                              (int)((y-
VCS_YCenterpoint)/VCS_GloveStep*VCS_NavigationSpeeds[Special]), 0);
               //key_set_direct(0,-VCS_NavigationSpeeds[Special], 0);
            }
            break;
         }
         break;
      }
   }
   position_changed++;
}

// ########## Customised Object Selection Routine ##########

void VCS_CreateSelectedObject(void)
{
   OBJECT *NewObject;
   POSE OldPose;

   get_object_pose(VCS_SelectedObject, &OldPose);

   NewObject = VCS_CopyObject(VCS_SelectedObject);
   add_object_to_world(NewObject);

   NewObject = make_fixed_object_moveable(NewObject, NULL);

   highlight_object(NewObject);
   VCS_SelectedObject = NewObject;
   VCS_LastSelectedObject = NewObject;
   global_update_object(NewObject);
   world_changed++;
}

int VCS_SelectObject(OBJECT *obj)
{
   // This is called by move_and_select_2D() in Cursor2d.c
   // Return value of 0 = no action, 1 = new object selected
```

```
// We want to ignore mouse clicks in this mode, they are catered for elsewhere.
if (VCS_ControlMode == VCS_NavigationMode)
   return 0;

if (VCS_WorldMode == VCS_MainWorld)
   if (!VCS_IsShiftKeyPressed())
      do_for_all_selected(unhighlight_object);

VCS_LastSelectedObject = NULL;

if (obj && is_object_selectable(obj))
{
   // If the shift key is pressed, then allow multiple highlighted objects,
   // otherwise clear the previous selections.
   // This is only for the main world though.

   if(is_object_selected(obj))
   {
      if (VCS_WorldMode == VCS_3dPalette)
      {
         if (VCS_SelectedObject != NULL)
         {
            delete_object(obj);
            VCS_SelectedObject = NULL;
         }
         else
            unhighlight_object(obj);
      }
      else
         unhighlight_object(obj);
   }
   else
   {
      // Lets only allow the user to select one object from the 3d Palette
      // at a time, and turn off any old choices as a new one is selected.

      if (VCS_WorldMode != VCS_MainWorld)
      {
         if ((VCS_SelectedObject != NULL) && (VCS_WorldMode == VCS_3dPalette))
         {
            delete_object(VCS_SelectedObject);
            VCS_SelectedObject = NULL;
         }

         if (VCS_IsShiftKeyPressed())
         {
            highlight_object(obj); // Just highlight it
         }
         else
         {
            VCS_SelectedObject = obj;
            VCS_ParentObject = obj;
            VCS_CreateSelectedObject();
         }
```

```
            }
            else
            {
               highlight_object(obj);
               VCS_LastSelectedObject = obj;
            }
         }
         world_changed++;
         return 1;
      }
      return 0;
}

void VCS_GrabObject(OBJECT *obj)     // Grab an object
{
   if(is_object_child_of(body_seg, obj)) return;  // can't grab body part!
   attach_object(obj, body_seg, 1);
}


void VCS_UngrabObject(OBJECT *obj) // Drop an object
{
   if(is_object_child_of(body_seg, obj))
   detach_object(obj,1);
}


BOOL VCS_ProcessKeys(unsigned KeyPressed)
{
   switch (KeyPressed)
   {
      // Just a few debug options so I can see what's going on.
#ifdef DEBUG
      case F5:
         VCS_Log_DumpState();
         break;
      case F6:
         VCS_SaveSelectedObject();
         break;
#endif

      // Extensions for VCS - F7 toggles VCS display stuff, and
      //     F8 will toggle between Navigation and Placement Mode
      case F7:
         if (VCS_DisplayControlMode)
         {
            VCS_DisplayControlMode--;
            VCS_DisplayWorldMode--;
         }
         else
         {
            VCS_DisplayControlMode++;
            VCS_DisplayWorldMode++;
         }
```

```
            break;
         case F8:
            if (VCS_ControlMode == VCS_NavigationMode)
               VCS_ChangeControlMode(VCS_PlacementMode);
            else
               VCS_ChangeControlMode(VCS_NavigationMode);
            break;

         // Extensions for VCS - F9 and F10 allow switching between World Modes
         case F9:
            switch(VCS_WorldMode)
            {
               case VCS_3dPalette:     break;
               case VCS_MainWorld:     VCS_ChangeWorldMode(VCS_3dPalette); break;
               case VCS_UserPalette:   VCS_ChangeWorldMode(VCS_MainWorld); break;
            }
            break;
         case F10:
            switch(VCS_WorldMode)
            {
               case VCS_3dPalette:     VCS_ChangeWorldMode(VCS_MainWorld); break;
               case VCS_MainWorld:     VCS_ChangeWorldMode(VCS_UserPalette); break;
               case VCS_UserPalette:   break;
            }
            break;

         default:
            return FALSE;
      }
      return TRUE;
}

// ########## Hook to the quit routine so VCS can close down nicely ##########

void VCS_Quit(void)
{
#ifdef DEBUG
   VCS_LogFile = fopen(VCS_LogFileName, Append);

   fprintf(VCS_LogFile, "Maximum and Minimum Glove Values:\n----------------------------
-----\n");
   fprintf(VCS_LogFile, "X: %d - %d\nY: %d - %d\n\n",
      VCS_MinGloveX, VCS_MaxGloveX, VCS_MinGloveY, VCS_MaxGloveY);
   fclose(VCS_LogFile);
#endif
}
```