Table of Contents

Change History

Configuration Control

Project:	OOVE
Title:	Configuration Management Plan
Doc. Reference:	WB 006
Created by:	Warren Blumenow
Creation Date:	17 April 1995

Document History

Version	Date	Status	Who	Saved as:
0.01	95/04/17	Draft	WB	\1995_07\MP\WB006WW.001
1.00	95/10/11	Approved	WB	\1995_07\MP\WB006WW.100
1.01	98/03/18	Approved	WB	\1995_07\MP\WB006WW.101
1.02	98/09/02	Approved	WB	\1995_07\MP\WB006WW.102

Revision History

Version	Date	Changes
0.01	95/04/17	New Document created using QST12870.101
0.01	95/06/12	Made changes as specified in project initiation audit report.
0.01	95/08/14	Added file naming conventions for project audit reports and minutes.
1.00	95/10/11	Document baselined at Version 1.00 after project manager approval. Status changed to Approved.
1.00	95/12/04	Updated section 5.2 to reflect project document backup to SEAL FTP server.
1.00	96/07/29	Updated section 4.2.2 to reflect location of program code fragments.
1.01	98/03/18	Miscellaneous minor changes throughout.
1.02	98/09/02	Final corrections and formatting.

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	95/10/11	Approved	WB 505

1 Scope

1.1 Introduction

This document describes the configuration management plan applying to all artefacts supporting this project.

Such artefacts will include hardcopy and electronic representations of documents, as well as the source code files which comprise the software implementation.

This document reviews the procedure which applies to the management and storage of these artefacts.

1.2 Purpose

Configuration items are individual documents, forms and source code modules in electronic and hardcopy format. They comprise:

- Management Products and procedures
- Technical Products, and
- Quality Assurance products

1.3 Applicability

To be referenced by all individuals who engage in product development for this project.

1.4 Audience

The following comprise the audience for this document:

a. The developer of this product, namely Warren Blumenow

b. The project manager, namely Professor Barry Dwolatzky

c. The members of the SEAL Management Board

d. The Head of Department of Electrical Engineering

e. Individuals who perform internal and external audits on projects undertaken within the SEAL Quality Management System.

1.5 Applicable Documents

1.5.1 Standards

a. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 1.00, 3 October 1994.

1.6 Requirements Traceability

a. ISO9001 (1994) 4.5 Document Control

b. ISO9001 (1994) 4.8 Product Identification and Traceability

c. ISO9001 (1994) 4.13 Control of Non-Conforming Product

d. ISO9001 (1994) 4.16 Quality Records

2 Documentation Structure

The documents are categorised to reflect the nature of their content. The key categories of documents include:

- Management Products,
- Technical Products, and
- Quality Assurance products.

3 Artefact Identification

This section defines a consistent referencing system that shall be used for all artefacts. All artefacts are identified by both an artefact filename followed by an artefact file extension in the following format, as described in the following paragraphs.

<name>.< extension>

3.1 File Naming Convention

3.1.1 Project Products—Multiple word processor types present

The file name will be structured as follows:

WBXXXYY.ZZZ

Where WB is the 2-letter acronym for this project,

XXX is the 4, 3, or 2 digit sequence number of this document,

YY is a file type identifier as follows:

- WW — Microsoft Word for Windows Document
- WT — Microsoft Word for Windows Document Template
- TX — LaTeX
- AS — Pure ASCII Text
- HC — Hard Copy File (No Source Present)
- PS — PostScript File
- PP — Microsoft PowerPoint Presentation

ZZZ is the version number of this file, starting from 001 as the initial revision. When a document is baselined as the first version, the most significant digit will represent the version number.

Example: The User Reference Manual for the SQM project has the filename SQM001WW.001. It is interpreted as the document SQM001 of type Word for Windows and is the first version in the sequence. This is the initial revision of this document. The approved first baseline version of this

document will have the revision number 100, and will be named SQM001WW.100.

3.1.2 Code / Line Art / Image Files

The file name will be structured as follows:

WBXXXYY.BBB

where WB is the 2-letter acronym for the project,

XXX is a 3 digit serial number

YY is a two digit version number, in the range 01 to 99, and starting from 01.

BBB is the file extension and will be whatever the particular compiler/tool demands.

The BBB file extensions that are used will include:

- CDR — Corel Draw Image File
- MDL — Rational Rose Petal File
- CPP, H — C++ Source Code Files
- EDG — Edge Diagrammer Flowcharts
- PCX, GIF, BMP — Bitmap Image Files
- AVI — Video Sequence Files

Example: The third display module (Version 2.4) for the CART project could have the following filename: CRT00324.CPP, if it was written in C++.

3.1.3 Project Audit Reports

The file name will be structured as follows:

9507AASS.PAR, where

95 is the 2 digit serial number representing the year of project initiation

07 is the 2 digit serial number representing the project ID according to the SEAL Projects Register (QS 199)

AA is a two letter acronym representing the audit type: PI = Project Initiation; IP = Project In-Process; PC = Project Closure

SS is a 2 digit serial number in the range 01 to 99, and starting from 01. SS only applies to audits of type 'IP'

PAR is the file extension, which stands for Project Audit Report

Example: The third Project In-Process Audit Report for this project initiated in 1995, with an ID of 07 would have the following filename: 9507IP03.PAR.

3.1.4 Minutes

The file name will be structured as follows:

YYMMDD-X.BBB

where YYMMDD represents the date of the meeting at which the particular minutes were taken. YY represents the year, MM represents the month and DD the day of the particular meeting.

X is a 1 digit meeting sequence number.

MIN is the file extension, which stands for Minutes.

Example: The minutes of the first meeting held on the 4th of March 1995 would be saved in a file with the following filename: 950304-1.MIN.

3.2 Project document sequence numbering

3.2.1 Where the document sequence number comprises 3 digits

- The MP series will be from 001–099
- The TP series will be from 100–499
- The QA series will be from 500–999

4 Artefact Management

The controls adopted for this project are as follows:

4.1 Active Machine used for supporting the files

SEAL Lab Computer System S–41 is the resident computer.

4.2 Directory Structure

4.2.1 Drive Partition

The E Drive partition is used.

4.2.2 Project Directory

- \1995_07\MP - for the storage of the management series documents
- \1995_07\TP - used for the storage of technical products, as follows:
  - \1995_07\TP\APP - Application Description
  - \1995_07\TP\BOOCH - Rational Rose Booch Diagrams
  - \1995_07\TP\CODE - Program Code Fragments.
  - \1995_07\TP\CONCEPT - Conceptual System Design
  - \1995_07\TP\CONCLUDE - Dissertation Conclusions
  - \1995_07\TP\DIAGRAMS - Figures And Diagrams
  - \1995_07\TP\FLOWCHT - Algorithm Flowcharts
  - \1995_07\TP\HLD - High Level Software Design
  - \1995_07\TP\JTPAPER - Joint Paper in conjunction with George Spanellis
  - \1995_07\TP\LLD - Low Level Software Design
  - \1995_07\TP\LTSURVEY - Literature Survey
  - \1995_07\TP\MISC - Various documents including dissertation front pages

- \1995_07\TP\PAPER - MSc Paper
- \1995_07\TP\PFS - Product Functional Specification
- \1995_07\TP\PROGRESS - Progress Reports.
- \1995_07\TP\PROPOSAL - Project Proposal.
- \1995_07\TP\REFS - References and Bibliography
- \1995_07\TP\RESOURCE - Various Resources
- \1995_07\TP\URM - User Reference Manual
- \1995_07\TP\VRAIS96 - documents relating to the VRAIS 1996 symposium in Santa Clara.
- \1995_07\TP\VRST97 - documents relating to the VRST 1997 conference in Switzerland.

- \1995_07\QA - used for the storage of quality assurance products including minutes of review meetings, audits of the system, and correspondence.

5 Archiving of Artefacts

5.1 Electronic Artefacts

On project completion an archive copy of all electronic artefacts supporting this projects will be supplied on recordable optical disk. The optical disk will be kept in the official SEAL plastic folders immediately inside the SEAL Project Binder.

5.2 Backup procedure on computers in the SEAL Post-Graduate Laboratory Facility

Computers in the SEAL Laboratory Facility are subject to the following backup procedure:

a. **Daily Backup:** Project files are archived to the stiffy diskettes kept inside the SEAL Project Binders, using the 32-bit ARJ utility. (Risk covered: Failure on the hard disk or inadvertent loss or corruption of data.)

b. **Weekly Backup**: Project files are archived to the SEAL FTP server in the /home/SEALPROJ/1995_07 directory using the ARJ utility. This action ensures that these project artefacts are archived to a site outside the SEAL Lab. (Risk covered: Destruction or loss of the computers in the SEAL Lab by fire, theft, or other disaster.)

5.3 Hardcopy Records

Are maintained in an official SEAL Project Documentation Binder in the Project File in Room CM2.221 (The SEAL Laboratory Facility).

6 Configuration Status Accounting

The Project Manager is responsible for ensuring that status of artefacts recorded in the Master Document List corresponds to the:

- identity of the artefacts in the file directories, and
- the revision numbers are correct.

Appendix Group II

Technical Products

Conceptualisation and Analysis

OOVE

Literature Survey

Technical Product

Version 1.03

Document Status: Approved

Table of Contents

Change History

Configuration Control

Project:	OOVE
Title:	Literature Survey
Doc. Reference:	WB 101
Created by:	Warren Blumenow
Creation Date:	28 September 1995

Document History

Version	Date	Status	Who	Saved as:
0.01	95/09/28	Draft	WB	\1995_07\TP\LTSURVEY\WB101WW.001
0.02	95/11/08	Draft	WB	\1995_07\TP\LTSURVEY\WB101WW.002
0.03	96/02/17	Draft	WB	\1995_07\TP\LTSURVEY\WB101WW.003
0.04	96/05/02	Draft	WB	\1995_07\TP\LTSURVEY\WB101WW.004
1.00	97/06/11	Approved	WB	\1995_07\TP\LTSURVEY\WB101WW.100
1.01	98/02/13	Approved	WB	\1995_07\TP\LTSURVEY\WB101WW.101
1.02	98/07/10	Approved	WB	\1995_07\TP\LTSURVEY\WB101WW.102
1.03	98/09/01	Approved	WB	\1995_07\TP\LTSURVEY\WB101WW.110

Revision History

Version	Date	Changes
0.01	95/09/28	New document created using WB002.
0.02	95/11/08	Produced draft form of Sections 2,3 and 4.
0.03	96/02/17	Updated Sections 2, 3, 4 and 5.
0.04	96/05/02	Additions to Section 2 (VR Background) and 5 (Models, Architectures and Design Strategies).
1.00	97/06/11	Document status changed to Approved. Minor corrections made to Sections 2, 3, and 5.
1.01	98/02/13	Section 6 (Summary of Relevant VR Systems) added.
1.02	98/06/14	Miscellaneous small changes after meeting of 98/06/10— Revisited referencing in Section 2, added diagram for DSM, MVC discussion updated, other small corrections.
1.03	98/09/01	Final cosmetic editing and formatting.

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	97/06/09	Approved	WB 523

Change Forecast

This document will be updated whenever a review of additional literature relevant to the project needs to be included.

1 Scope

1.1 Introduction

This document presents a survey of the relevant literature pertaining to Virtual Reality technology, Virtual Environment systems and architectures, as well as other aspects that are relevant to the project at hand.

A general background to Virtual Reality is provided, including a discussion of the key features which characterise a Virtual Reality system, a description of specialised hardware devices, and a review of applications of the technology.

Some of the important design issues related to VR system design are highlighted, and several contributory technologies which facilitate addressing these design issues are discussed. A number of architectural approaches and design strategies which have been encountered in the literature are presented.

1.2 Purpose

The purpose of this document is to furnish the reader with the necessary background to the technologies being dealt with, and to provide insight into the important strategic issues that must be addressed by the system. The document is intended to summarise the main features of existing systems and highlight key issues so that the design choices which have been made in formulating the architecture of the system at hand can be understood in their proper context.

1.3 Definitions

1.3.1 Abbreviations

VR = Virtual Reality

VE = Virtual Environment

MVC= Model–View–Controller

DSM = Decoupled Simulation Model

BOOM = Binocular Omni-Oriented Monitor

CAVE = Cave Automatic Virtual Environment

1.4 Audience

The audience for this document includes the project manager, members of the SEAL management board, the external examiner, and any other interested parties.

1.5 Applicable Documents

1.5.1 Standards

a. SEAL QMS Document Creation Template, QS 002, Revision 0.02, 3 October 1994

b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 1.00, 3 October 1994

2 Background to Virtual Environments / Virtual Reality

2.1 Definition

Blau, et al. [1992] describe a Virtual Environment as a realtime simulation in which the user is immersed in a three dimensional graphical space. The user is given freedom of movement within the space, and is able to interact directly with the simulation. Shaw, et al. [1992] refer to Virtual Reality as the highly interactive three dimensional control of a computational model, in which application data is manipulated by means of natural three dimensional interaction techniques. According to Blake [1994], the user is placed inside a virtual space, within which he can seem to change his viewpoint and can interact with and manipulate objects in the space.

From these descriptions and a broad review of the Virtual Reality Literature (Shaw et al. [1993], du Pont [1993], Loftin and Kenney [1995], inter alia), a number of key features can be identified as comprising the critical aspects of Virtual Reality. One of the most important characteristics of Virtual Reality is that the user interface is interactive. The user's interaction with the virtual world involves three distinct components: *viewpoint*, *navigation* and *manipulation*. In order to provide the underlying application with which the user interacts, *modelling* and *simulation* must be present. Finally, *immersion* refers to deceiving the user's senses so that the user believes that he is physically in the virtual world. Each of these features is discussed briefly below:

Viewpoint/Aspect The ability of the user to change his point of view in observing a three dimensional scene is probably the most distinguishing feature of what constitutes a Virtual Reality user interface. This is the most fundamental way in which a user can interact with a virtual space—by "looking around" or facing in a particular direction.

Navigation Providing freedom of movement within the virtual space allows the user to view and interact with objects in the scene from any chosen angle. In addition, the user may move closer to objects of interest in order to view them in greater detail, or move backwards in order to obtain a broader perspective and perceive the spatial and hierarchical relationships which exist between the objects in the world.

Manipulation The use of natural three dimensional interaction mechanisms allows the user to directly manipulate objects in the virtual world. The user may grasp or release objects, or provide semantic input into the simulation through the use of gesture.

[Note: Whenever the user changes his viewpoint, navigates through the virtual space or manipulates an object, it is critical that correlation between user input and displayed image is maintained. The interactivity of the system will be compromised if there is lag between the user's actions and the resulting change in the visual display. This is an important consideration in the design of a Virtual Reality system, and is discussed further in Section 3.]

Modelling In order to present the user with a logical and consistent environment, the VR system must maintain some model of the objects which are contained in the virtual world and with which the user interacts. Because it is such a fundamental feature of Virtual Reality, some form of modelling is implicit in any VR application.

Simulation A useful Virtual Reality system will invariably incorporate some underlying process with which the user interacts. This process usually takes the form of a continuously running simulation which provides the *non-graphical* functionality of the system. It is often advantageous to decouple the user interface from the application-level functionality of the system. This separation allows the system to interface easily with existing applications and, possibly more significantly, eliminates the dependency of the graphics rendering rate on the speed of the simulation. The latter consideration is covered in Section 5.2.

Immersion Immersion refers to the illusion of being inside a virtual space. An immersive VR system deceives the user's senses, causing the user to feel as though he is physically located in the synthetic environment which is presented. Immersion is a key benefit of the Virtual Reality user interface paradigm, since gives the user the sensation of interacting directly with the application, and not using a computer at all.

Specialised VR hardware is used to create the illusion of immersion. Typically, this includes a head-mounted display or other immersive display device, and head tracking hardware which can determine the position and orientation of the user's head. By means of these devices, the system is able to present the appropriate synthetic images to each eye based on the user's current viewpoint. The use of natural three dimensional input devices allows for direct manipulation. These and other VR hardware devices are discussed in the following section.

2.2 Hardware Devices

Various types of specialised hardware exist to support Virtual Reality applications. These include [de Waal and Olckers, 1994; Blake, 1994]:

Output Devices:

- stereoscopic head mounted displays (HMDs)
- tactile feedback devices
- virtual acoustic displays
- other output devices

Input Devices:

- six degree of freedom tracking devices
- glove-based input devices (datagloves)
- other input devices

Each of these will be discussed here briefly. For a more detailed treatment of these technologies, the reader is referred to de Waal and Olckers [1994] and Blake [1994], as well as various books which deal with general aspects of Virtual Reality, such as one by Hamit [1993] and one by Wodaski [1993] which are listed in the bibliography.

2.2.1 Output Devices

Stereoscopic Head Mounted Displays A Head Mounted Display (HMD) (also referred to as a "Helmet Mounted Display" in military or aviation applications) is a display device that is mounted on the head of the user. Small CRT or LCD display screens within the device present computer-generated images directly to the user's eyes [de Waal and Olckers, 1994]. The user is able to move his head freely, and the display moves along with it. The images which are presented to the user's eyes are correlated to the user's head movements through the use of a head tracking device (refer to the discussion of six degree of freedom tracking devices below).

Critical features which need to be taken into account in the design or selection of an HMD are the physical size and weight of the device, the field of view (horizontal and vertical), and the display resolution and optical performance of the device. There are generally trade-offs between these parameters such as field of view vs. optical performance. The higher the field of view the lower the quality of the displayed image is likely to be [Zwern, 1995]. Most HMDs are stereoscopic; the image which is calculated for the left eye differs slightly from that presented to the right eye, resulting in the realtime illusion of depth.

Tactile Feedback Devices Haptic or tactile perception refers to providing the user with a physical sensation of the virtual world through the sense of

touch. Devices in this category supply kinaesthetic information about the environment which can take the form of mechanical, thermal, electrical or chemical stimuli. Though the use of such devices, the user is able to perceive, in a tactile manner, the shape, texture, temperature, elasticity and force reflecting characteristics of objects in the virtual world. In this way, the user is given feedback concerning his interaction with the system [Hettinger, 1997].

Virtual Acoustic Displays The purpose of virtual sound in Virtual Reality is to present the user with an auditory representation of the virtual world which closely emulates real-world acoustic behaviour. Auditory information which is presented in this way is able to be intuitively interpreted by the user. Humans have an innate ability to selectively concentrate on one particular sound source in the midst of noise; auditory perception of the environment can greatly enhance the user's interaction with the system. For example, the user may be engaged in one area of the virtual environment, and have his attention attracted by potentially relevant auditory information originating in another area, potentially not within the current visible region.

Virtual acoustic displays are primarily concerned with the production of externalised three dimensional sound. An acoustic image is said to be *externalised* if the sound source appears to the listener to lie outside the head [Durlach et al., 1992]. *Three dimensional* sound refers to sounds which are able to be localised in space. Auditory localisation involves the concept of a *head-related transfer function*, which models the spatial characteristics of a particular individual's head and outer ear. Sound delivered by means of headphones effectively bypasses the filtering effect of the user's outer ear. By simulating the acoustic properties of the user's auditory apparatus, the head-related transfer function, when applied to the signal, allows sound to be produced, the origin of which appears to be localised in space.

Other Output Devices Various output devices other than those discussed above have been developed, usually for use in specialised applications, or in situations with particular requirements.

The *BOOM* (Binocular Omni-Oriented Monitor) is a stereoscopic monitor mounted onto a series of counterbalanced kinematic linkages. It uses mechanical tracking to determine position and orientation, by means of shaft encoders which measure the relative positions of the support arms [Blake, 1994; de Waal and Olckers, 1994].

Various devices exist which augment the display produced by a conventional computer screen. These include the *Cyberscope* [Blake, 1994], which attaches to the computer screen and into which the user

peers; and various types of *shutter-based glasses* which add stereoscopic capabilities to a standard display monitor.

The *CAVE* (Cave Automatic Virtual Environment) uses three projectors to display images onto two walls and the floor of a ten-foot cube inside which the user stands. The user wears stereoscopic shutter glasses, and the user's head position and orientation are tracked so that the images can be rendered in correct viewer-centred three dimensional perspective [Disz et al., 1995].

Head-mounted displays which overlay computer-generated images onto the real world are termed *augmented reality* devices. The principles underlying the operation of these devices are generally similar to those pertaining to ordinary HMDs.

2.2.2 Input Devices

Six Degree Of Freedom Tracking Devices Tracking devices are used to measure the position and orientation of a device or of some part of the user's body. For example, a head tracker is generally used in conjunction with a head mounted display to determine the position and orientation of the user's head. Based on this information, images are generated for each eye so that the view presented to the user is consistent with his head movement. Position and orientation trackers are also used for tracking data gloves and other input devices.

In general, six degrees of freedom are both necessary and sufficient to fully specify position and orientation in three dimensional space. The six degrees of freedom referred to are the x, y and z co-ordinates of the tracker's position in three dimensional space, relative to some static reference point, and the roll, pitch and yaw values which define the orientation of the tracker.

There are various methods of performing position and orientation tracking. These include magnetic, optical, physical and acoustic tracking. One of the most successful and widely used tracking methods involves the interaction of magnetic fields. This is the method employed by the popularly used Polhemus 3-space tracker, as well as by trackers manufactured by Ascension Technologies. A pulsed magnetic field is radiated from a stationary source. A sensor, which can be attached to any object, reports it's position and orientation in space relative to the source by measuring the magnetic field interaction [Sturman and Zeltzer, 1994]. Optical tracking relies on a system of markers, and is performed by means of cameras. This method is computationally demanding, and is often unable to be performed in real time. Furthermore, optical trackers are unable to resolve markers that are in close proximity to one another. Physical tracking devices involve sensors which are physically attached to

the device being tracked. Examples include tilt sensors, accelerometers, and compasses. Acoustic trackers generate pings of high frequency sound and use triangulation to determine the position of the source. These systems are affected by noise and acoustic reflections. Both optical and acoustic tracking are restricted to line-of-sight operation; this limitation is not applicable to magnetic trackers.

Glove Based Input Devices The use of sensing gloves in Virtual Reality allows the dexterity and naturalness of gesture and hand motion to be exploited as an input modality. Glove based input devices are often referred to generically as datagloves. The name 'DataGlove' is in fact the tradename of a particular glove device manufactured by VPL Research.

Datagloves are generally able to measure both handshape and hand motion. Handshape is typically measured by means of electromechanical devices which fit over the hand and fingers, such as fibreoptic sensors or flexible strain gauges. Hand motion may be obtained optically through the use of a camera, or by means of an acoustic or magnetic tracking device as discussed above [Sturman and Zeltzer, 1994].

The interpretation of gestures is generally achieved in software, using the raw handshape and hand motion information captured from the dataglove as input, and employing, inter alia, filtering, rule-based and neural network approaches to process the transduced data.

Other Input Devices

In addition to those discussed above, there are various other input devices which have been developed to support VR applications.

The concept of a dataglove as discussed above can be extended so that in principle any part of the user's body can be tracked. The culmination of this is a full *body suit* which monitors the position and orientation of the user's entire body [de Waal and Olckers, 1994].

The *SpaceBall* is a six degree of freedom tracker which consists of a fixed ball connected to a base unit. The ball is grasped by the user and it measures the applied pressure and tortional forces to which it is subjected. The user performs three dimensional manipulations by twisting or moving the ball.

Position and orientation tracking devices are often attached to objects other than the user's head or hand. Examples include baseless joysticks, 3D mice, and the hand-held wand device employed by Taylor, Stevens and Canfield [1995].

2.3 Applications

Numerous applications for Virtual Reality technology have been described in the literature. What characterises the majority of these application areas is that they represent tasks that are difficult to perform using traditional two dimensional user interfaces. Three dimensional immersive user interfaces and Virtual Reality have a number of unique characteristics that distinguish them from familiar text-based or window-based systems—most significantly natural interaction mechanisms and direct manipulation [Shaw et al., 1993]. The most suitable candidate applications for the technology are those that are able to exploit these unique features to solve problems in areas where traditional user-interfaces fall short.

West et al. [1992] describe several VR application areas, including three-dimensional Computer Aided Design (CAD) and information management. De Waal and Olckers [1994] comment on the use of Virtual Reality in architectural design, entertainment and medical applications. The technology has also been used successfully for scientific visualisation [Taylor, Stevens and Canfield, 1995], military simulation [Zyda et al., 1992] and education and training [Loftin and Kenney, 1995].

The potential range of applications for VR technology is clearly very broad, and a detailed discussion of applications such as CAD, medical imaging, architecture and entertainment will not be undertaken here. Of particular relevance to the project at hand is the application of VR in the field of information visualisation.

The use of VR for information visualisation recognises the fact that humans have inherent visual perception and spatial skills that are largely ignored in conventional approaches to the presentation of information [West et al., 1992:2]. The volumes of information needing to be dealt with are increasing rapidly, and traditional data processing and analysis approaches are often unable to cope with the sheer volume of data. Interactive visualisation techniques are able to accelerate the comprehension and interpretation of information, and enable intuitive and rapid access to required information [Risch et al., 1996].

3 Design Issues

There are a number of issues which need to be considered in the design of interactive three dimensional user interfaces. Some of these issues, including visual update rate and lag, are related to system performance Other considerations are cost and human factors.

3.1 Visual Update Rate

The overall performance of any Virtual Reality application is to a large extent determined by its ability to maintain a high visual update rate. Smoothly animated images facilitate interaction with the system, and contribute significantly to the illusion of immersion that the user experiences.

In order for the user to perceive the individual frames being generated as smooth motion, a high frame update rate must be maintained. According to Shaw et. al [1993:4] to appear smooth, the visual update rate must exceed 10 updates per second. Schaufler [1996:95] indicates that frame rates as high as twenty frames per second are required. Hubbold et. al [1993:5] concur that at least 20 frames per second (fps) are generally required for the illusion of continuous smooth motion. They note that only at frame rates approaching 60 fps, are all artefacts such as ghosting and aliasing completely eliminated. By way of comparison, the video formats PAL (Phase-Alternation Lino)[1] and NTSC (National Television System Committee)[2] provide frame rates of 25 and 29.97 frames per second respectively [Shnier, 1996]. Shnier notes that because of its lower frame rate, PAL often shows noticeable flicker as compared to NTSC.

Various techniques for improving the rendered frame rate are described in the literature. All of these methods involve decreasing in some manner the amount of computation needing to be performed between display updates. The approaches that have been encountered include:

- decoupling of the visual update loop from simulation computation, to eliminate temporal dependency between rendering and non-graphical processing [Shaw et al., 1992] (discussed fully in Section 5.2)

[1] PAL is the broadcast television format used in Western Europe (including the United Kingdom) and Australia. It is not well standardised (certain countries use slightly different implementations).

[2] NTSC is the broadcast television format used in North America and Japan, standardised in 1953.

- time critical computing, in which computation accuracy is sacrificed in favour of computation speed [Taylor, Stevens and Canfield, 1995]

- adaptive rendering schemes which simplify, in various ways, tl geometric models being dealt with, thereby reducing graphics computation. Examples of this include scene complexity reduction [Taylor, Stevens and Canfield, 1995], re-use of parts of previously rendered frames [Schaufler, 1996], and level of detail reduction in displaying distant or temporarily unimportant objects [Ohshima, Yamamoto and Tamura, 1996]

- the use of distribution and parallel processing, which has the net effect of increasing the available computing resources, thereby reducing the computational load on each processor (this aspect is dealt with fully in Section 4.2)

3.2 Interaction Lag

One of the key requirements of Virtual Reality is interactivity. In order for a VR system to be interactive, the images presented to the user must respond rapidly to the user's actions [Shaw et. al, 1993].

If the lag between the user's actions and the system response is too long, the user has a tendency to compensate by modifying his behaviour. Examples of this which have been identified by Shaw et. al [1993] include the user slowing down his movements, and the adoption of a *move and wait* strategy. When this occurs, the interactivity of the system is adversely affected. In addition, interaction lag may have other consequences for the user, such as disorientation and cybersickness (see Section 3.4).

Pausch [1991] notes that low interaction latency is more important than graphics quality—a user is more willing to tolerate low display resolution than high interaction lag. The usability of a system which suffers from low graphics quality but provides realtime response to the user will be far greater than a system which renders photo-realistic images at the expense of interactivity. Hettinger [1997] suggests that "The success or failure of a particular VE system is not necessarily a function of how 'realistic' it is. Rather it is a function of the extent to which the [human] behavioural goals of the system have been met".

In order for a VR system to be responsive, there needs to be a tight coupling between the user's actions and the system's response. The system must provide rapid feedback to the user and an acceptable closed loop response time. According to Shaw et al. [1993], for the system to be regarded as responsive, the maximum tolerable image lag between movement and image update is 100 milliseconds. This is especially true when a head mounted display with head tracking is used.

Various methods for reducing temporal-spatial distortion due to lag between user input and visual feedback have been investigated. Many of the techniques described above for improving the visual update rate will have the concomitant effect of reducing lag by decreasing the amount of computation needing to be performed between obtaining input from the user and producing the final rendered result. In addition, there exist strategies which are specifically aimed at reducing lag:

- Liang, Shaw and Green [1991] designed a predictive Kalman filter to compensate for delay in head tracker orientation data which they determined to be responsible for much of the lag experienced by the user of an HMD. The system predicts future input data from the head tracker by extrapolating from known previous data.

- In a distributed VR system, effective use can be made of a technique called "dead reckoning". In essence, dead reckoning involves extrapolation of an object's current position and orientation based on its previous position and orientation, combined with knowledge of its motion characteristics. For example, if the location and velocity vector of an object at a particular time is known, then an approximation of that object's position at some future time can be calculated. This reduces lag since the rendering subsystem can use the locally calculated approximation and does not need to wait for an update before rendering the object. The Distributed Interactive Simulation (DIS) protocol (described in [Roehl, 1996]) uses this technique. Each node calculates the state of any objects in the simulation for which it is responsible, periodically sending updates to other affected nodes. All nodes run the dead-reckoning model: remote nodes use dead-reckoning between updates; the local node determines how frequently to send updates by monitoring the deviation of each local object's behaviour from the dead-reckoning model.

- Where network communication is involved, network latency can contribute significantly to the overall system lag. Increasing the available network bandwidth is one method for dealing with this [Taylor, Stevens and Canfield, 1995]. A second approach is to improve the network communications protocols which are used [Kessler and Hodges, 1996]. Lastly, the use of dead reckoning as described above also reduces network latency by reducing the frequency of network traffic—updates are not sent over the network if the dead-reckoning model is sufficiently accurate.

3.3 Cost Considerations

Typically, the cost of implementing a Virtual Reality system is high. The two main contributing factors are the computationally intensive nature of the task, leading to a need for powerful and hence expensive computer

hardware; and the substantial cost of specialised VR input and output devices.

The latter problem may be alleviated through the use of so-called "fishtank" VR systems, which make use of a standard display monitor for the presentation of graphics, and standard input devices, such as a mouse or keyboard. Koller et al. [1995] for example, describe a Virtual GIS system which features both a workstation window-based interface and an immersive head mounted display-based interface. Although fishtank implementations are not as immersive as glove and HMD-based VR systems, the substantial benefits of three dimensional presentation of information are not lost. Specific devices which provide significant utility in the context of the particular application area being addressed can be added as required, or low cost alternatives can be considered [Pausch, 1991]

There are several ways to reduce the computational resource requirements of a VR system:

- The rendering algorithms which are used can be optimised. A number of approaches for improving the visual update rate were discussed in Section 3.1 above. Seen another way, if the frame rate remains constant, the use of such techniques reduces the overall processing requirements of the system. The system is therefore able to provide the same level of performance on less powerful hardware.

- The functionality of the system should be matched to the requirements of the application. Pausch [1991] notes that for many applications, extremely high spatial resolution or stereoscopic display may not be necessary.

- Distributed processing can be used to divide the computational load between a number of low-cost machines. Kyriazis [1990:23] points out that the cost of a single sequential machine which provides the same combined level of performance as a system which makes use of parallel processing, is in general significantly higher. The use of distributed processing for Virtual Reality is discussed further in Section 4.2.

3.4 Human Factors

The design of any user-interface must take into account various human factors such as usability and ergonomics. There is increasing recognition of the role of cognitive science in human-computer interface design. Virtual Reality systems in particular introduce issues which are specifically related to the immersive nature of the interface and its effects on human perception.

The need for adequate frame rate and low lag have already been discussed, both of which contribute significantly to the usability of the system and its acceptance by the user.

One of the main problems with Virtual Reality user interfaces is their tendency to induce motion sick ess. The terms "simulator sickness" and "cybersickness" are often used to describe the side effects resulting from the use of VR systems. Cybersickness is, according to Hettinger [1997], "primarily visually-induced motion sickness involving the occurrence of symptomatology in physically stationary individuals". Symptoms may include disorientation, nausea and oculomotor problems [Loftin and Kenney, 1995].

4 Contributory Technologies

4.1 Object Orientation

This section should be read in conjunction with Section 2 of the High Level Software Design (WB 105), which deals with object-oriented software development.

4.1.1 Overview

The applicability of object orientation as a software architecture for high performance graphics applications, and for interactive, realtime three dimensional user interfaces and Virtual Reality in particular, has been highlighted by a number of researchers.

In his study on architectures for high performance graphics systems, Kyriazis [1990:23–24] assesses the ability of object oriented systems to efficiently perform tasks related to graphics algorithms. He concludes that object orientation is well suited to high performance graphics applications. According to Kyriazis, object orientation provides a means for developing systems which are both upgradeable and parallelizable. Graphics algorithms are usually easily parallelizable, and decomposition of the system into multiple functional units through the use of objects allows this parallelism to be exploited (refer to Section 4.2 below).

The AVIARY system incorporates the object oriented paradigm in its conceptual software model. [Snowdon, West and Howard, 1993]. Once more, one of the key advantages mentioned is the facilitation of parallel processing. The division of the system into objects provides "convenient units of parallelism" which allows the exploitation of parallel hardware or distributed systems. Additionally, objects are conceptually easy to deal with, since they typically represent tangible entities.

4.1.2 Systems

The AVIARY system comprises a hierarchy of virtual world classes [West, et al., 1992]. Each world class in the hierarchy defines the properties and laws which govern objects in that world. Laws are implemented by means of program code in the methods of the world class. The classes that are used to define the objects or entities in the world are derived from the world class. Each entity class inherits the properties of the world class from which it is derived, and adds functionality which is specific to that entity.

Other object oriented VR systems include Alice [Pausch et al., 1995], Bricknet [Singh et al., 1994] and VERN [Blau et al., 1992]. A significant

feature of VERN is that all code relating to communications and process control is encapsulated in the highest level classes of the class hierarchy.

4.2 Distributed Processing

4.2.1 Overview

Virtual Environment systems described in the literature make use of distributed processing for two distinct reasons—the distribution of the computational load, and the sharing of experiences (by facilitating interaction between more than one participant). Singh et al. [1994] refer to these categories as single-user and multi-user network-based virtual worlds, respectively. The former has greater significance in the context of the project at hand, since the scope of this work does not include interaction between multiple participants. However, many of the techniques used in the design of multi-user distributed virtual environments can offer to the design of systems that employ distributed processing for load distribution.

Distributing the processing load over a number of workstations improves the computational capabilities available to the application. Hubbold et al. [1993:9] note that virtual environments are inherently concurrent, and require the various input processing, rendering and simulation tasks all to occur simultaneously. It is advantageous to exploit this concurrency by allowing components of the system to execute asynchronously on separate processors. In this way, better overall performance can be achieved through the utilisation of the combined processing power of a number of workstations [Singh et al., 1994].

Distribution of processing additionally improves the reliability and fault tolerance of the system. If a single node fails, only certain processes will be affected, and the possibility exists for dynamically reallocating those processes to other processors.

4.2.2 Systems

VR systems that make use of distributed processing in order to improve performance include the MR Toolkit [Shaw et al., 1993], Alice [Pausch et al., 1995], DIVER [Gossweiler et al., 1993], AVIARY [West et al., 1992] and dVS [Benford et al., 1994]. All of these systems use a multi-process architecture to facilitate distribution.

The MR Toolkit makes use of a master/slave process organisation in which a centralised process performs the geometric modelling, and manages communication between subordinate processes. It collects device and simulation data from device servers and computation processes, and despatches it to any slave processes that need it. In Alice

and DIVER, the simulation process spawns remote rendering and device handling processes. The simulation provides the renderer with regular updates; device handlers send updates to both the simulation and the rendering processes. AVIARY uses a dedicated communications subsystem through which loosely connected processes interact on a peer-to-peer basis.

Multi-user distributed VR systems include DIVE , Bricknet [Singh et al., 1994], NPSNET [Zyda et al., 1992] and VERN [Blau et al., 1992].

4.2.3 Strategies for Distribution

There are a number of mechanisms that may be used to facilitate distributed processing. Two of the most significant techniques are remote procedure calls and distributed objects.

Remote Procedure Calls

The term Remote Procedure Call (RPC) refers to a protocol that allows a program on one computer (client) to execute a program on another computer (server). The client program sends a message to the server with appropriate arguments and the server returns a message containing the results of the program executed. The semantics of a remote procedure call are similar to those of a standard local procedure call. Sun Microsystems developed the first widely used RPC protocol as part of their Open Network Computing (ONC) architecture in the early 1980s. [Mecklermedia, 1998]

Distributed Objects

A number of object oriented methods for programs to communicate with each other have also been developed. In general, they provide the same types of capabilities as traditional RPCs, but take advantage of the unique features of object orientation to hide the details of process distribution from the application level processes. The most well known technologies are CORBA and DCOM.

CORBA (Common Object Request Broker Architecture) is a specification that enables objects to communicate with one another irrespective of what programming language they were written in, what operating system they run on, or where they are physically located [Mowbray and Zahavi, 1995]. It is aimed primarily at application interoperability between heterogeneous systems and platforms.

CORBA was developed by an industry consortium known as the Object Management Group (OMG). There are several implementations of CORBA, the most widely used being IBM's SOM and DSOM architectures. CORBA has also been incorporated into Netscape's ONE (Open Network

Environment) platform. Microsoft's COM and DCOM and Sun Microsystems' RMI are CORBA's main competitors [Mecklermedia, 1998].

The main focus of the abovementioned object oriented distributed processing technologies is on interoperability between widely disparate systems. As a result, they must necessarily provide a broad range of functionality and flexibility. One of the consequences of this is that performance may be impacted. In an application such as Virtual Reality, realtime performance is a key consideration.

According to McGoogan [1996]:

"CORBA does not currently specifically address realtime computing"

In contrast to the multitudinous array of hardware and software targeted by standards such as CORBA, a single-user distributed Virtual Reality system will typically run in a relatively predictable environment. As a result, the complete CORBA specification is not very well matched to the requirements of distributed VR.

The Realtime Platform Special Interest Group (RTSIG) of the Object Management Group is currently examining ways of addressing this, including implementing a subset of the CORBA specification—"lightweight CORBA", that is optimised for high speed and low overhead [McGoogan, 1996:22].

5 Models, Architectures and Design Strategies

A number of different architectural approaches to the design of Virtual Reality systems are encountered in the literature. Several methodologies and models which have been deemed relevant to the project at hand are discussed in this section.

5.1 Model-View-Controller

The Model-View-Controller (MVC) user interface paradigm represents a generic framework for structuring interactive graphical applications. It provides a mechanism for separating: the application state and behaviour, the presentation of that state and behaviour to the user, and the means whereby the user interacts with the application [Krasner and Pope, 1988; Burbeck, 1992]. The approach is well suited to the design of object oriented graphics systems [Kyriazis, 1990:22].

An MVC application essentially consists of an input subsystem, an output subsystem, and some model of the underlying application domain. The *Controller* subsystem is responsible for handling input from the user; the *View* subsystem represents the output display perceived by the user. Changes in the View take place as a result of transformations performed on the graphics database contained in the *Model*. The MVC concept is illustrated in Figure WB101–1.

MVC applications are typically characterised by what Krasner and Pope [1988] refer to as a *standard interaction cycle*. When the user takes some input action, the Controller notifies the Model to change itself accordingly. The Model changes its state in response to the received input, and notifies both the View and the Controller of the change. The View responds by updating the display based on the new Model state. The Controller may respond by changing the method of interaction. Provision is also made in the model for communication between the View and Controller; this is primarily used to manage the sharing of input and output devices among multiple applications [Burbeck, 1992].

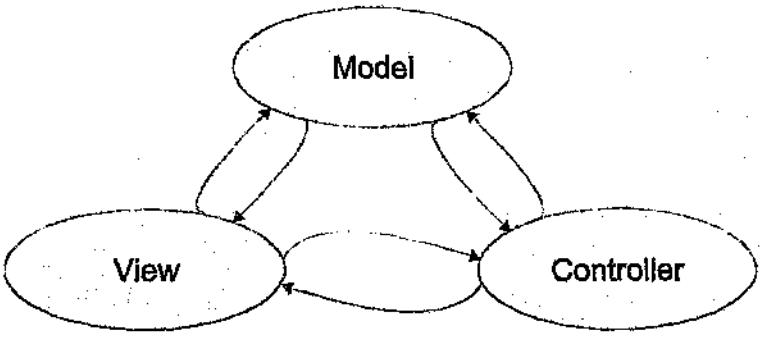


Figure WB101–1: Model View Controller Arrangement

5.2 The Decoupled Simulation Model

Pausch et al. [1995] indicate that a problem with many existing Virtual Reality systems is that the graphics rendering rate is closely coupled to the speed at which the underlying simulation proceeds. The Model–View–Controller metaphor in particular suffers from this limitation. The MVC standard interaction cycle inherently limits updates to the View to the rate at which the Model's state is updated.

From an implementation point of view, there is often a main processing loop which repeatedly executes the following sequence of tasks:

- poll all input devices
- update the simulation
- update the representation displayed to the user

If the user changes his viewpoint, he must wait for a complete iteration of the simulation to be completed before the display is refreshed.

Shaw et al. [1992; 1993] at the University of Alberta have developed the Decoupled Simulation Model (DSM) for Virtual Reality systems. This model provides a means for structuring Virtual Reality applications so that the user perceives smooth animation of the environment, without temporal dependency on the processing rate of the application. The key feature of this model is that the computation or simulation component proceeds independently of the main visual update loop of the system.

The Decoupled Simulation Model divides a VR system into four components, as depicted in Figure WB101–2 [Shaw et al., 1992]:

- Computation
- Geometric Model
- Interaction
- Presentation

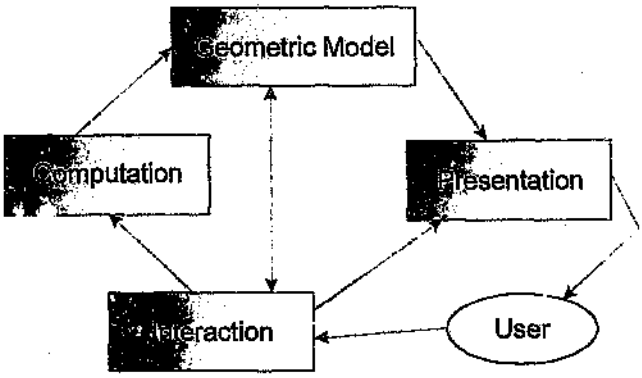


Figure WB101–2: The Decoupled Simulation Model

The Computation component manages all non-graphical computations in the application. It runs essentially independently of the rest of the system, and typically takes the form of a continuously running simulation which evaluates a computational model of some process. The Computation component periodically forwards its output data to the Geometric Model.

The Geometric Model component maintains a high level representation of the data generated by the computation. It performs a mapping from application data to graphical data.

The Interaction component is responsible for handling all input from the user. It provides an interface to the input devices used by the system.

The Presentation component generates the view of the environment seen by the user. This view will typically be visual, but may also include auditory and haptic feedback aspects. In the visual domain, the view rendered by the Presentation component is based on graphical data received from the Geometric Model as well as viewing parameters, such as the current viewpoint, received from the Interaction component.

There are two main loops of information flow in the system. The first of these is the *interaction–geometric model–presentation* loop, which does not include the computation component, and is therefore independent of the rate at which the simulation proceeds. This loop is responsible for visual updates in response to the user's navigation through the environment, and needs to be optimised to ensure low lag and a high visual update rate.

The second, slower loop takes the form of *interaction–computation–geometric model–presentation*, and occurs when the user performs actions of semantic significance to the simulation.

In their work on performance modelling, Taylor, Stevens and Canfield [1995] also distinguish between two categories of information flow. Their immersive visualisation environment makes use of a head tracker and a wand (see Section 2.2 above). In their implementation, each major process of the system runs asynchronously. This inherently decouples the simulation from the visual update of the display. The user's interactions with the system by means of the head tracker do not modify the simulation process. The simulation proceeds independently of changes in the field of view produced by movement of the head tracker. This situation corresponds to the interaction–geometric model–presentation loop in the Decoupled Simulation Model. Conversely, interactions by means of the wand *do* cause modifications to the simulation process, the results of which cause changes in the display. This coincides with the interaction–computation–geometric model–presentation loop in the DSM.

The distinction is made between viewpoint lag, which does not involve the simulation and can be minimised to achieve a fast visual update loop, and interaction lag, which takes into account the results of the simulation.

The Alice and DIVER systems developed by the University of Virginia [Pausch et al., 1995; Gossweiler et al., 1993], also decouple simulation and rendering. This is once again achieved by means of a multiple process architecture. The processes responsible for computing the simulation are separated from those that maintain the geometric database and perform the rendering. These processes are able to run on separate machines and communicate with each other over a local area network. A key aspect of these systems is that the application/rendering separation is performed in a manner that is transparent to the programmer. The programmer is not required to explicitly coordinate communication between processes residing on different machines.

5.3 Players and Ghosts

The concept of *Players* and *Ghosts* has been documented by Blau, et. al [1992]. The context of their work is VERN, a system for distributing virtual environments over a network of graphical workstations. Real world objects participating in a VERN distributed interactive simulation are represented in software by software objects called *Players*. Each Player resides on a particular machine. On every machine other than the Player's home machine, there exists a corresponding Ghost for that Player. Ghosts act as representatives for their associated Players. Thus, for each player there is a representative on every machine. Blau et al. describe the situation as follows:

> *"...in an N Player simulation on M networked machines, each machine is guaranteed to have exactly N objects representing all players."*

A Player is therefore able to communicate with any other player as though it were resident on the local machine. A Ghost is responsible for forwarding messages when necessary to its associated Player.

A VERN Ghost maintains an approximation of the state of its associated Player by means of a dead-reckoning model. Each Player sends updates to its Ghosts whenever its actual state deviates significantly from what is predicted by the dead-reckoning model. Messages received by a Ghost can be of two types: queries and commands. Queries are typically used to request the state of a Player, and are handled by the Ghost directly. Commands, on the other hand, must be passed on to the Player.

6 Summary of Relevant Virtual Reality Systems

6.1 Minimal Reality (MR) Toolkit

Developed By: Department of Computing Science, University of Alberta, Canada

Platform: Silicon Graphics and DEC workstations; Unix

Principal Characteristics:

- Incorporates the Decoupled Simulation Model, which decouples the visual update rate of the application from the update rate of the simulation.
- Support for the distribution of computation over a number of workstations.
- Environment Manager handles communication management, application replication, network configuration and concurrency control.

Key References: Shaw et al. [1993], Shaw et al. [1992], Wang, Green and Shaw [1995], Liang, Shaw and Green [1991].

6.2 AVIARY

Developed By: Advanced Interfaces Group, Department of Computer Science, University of Manchester, United Kingdom

Platform: Essentially platform independent, but targeted at the "ParSiFal T-Rack multicomputer", a high performance transputer-based parallel processing machine developed by their department.

Principal Characteristics:

- Object oriented design incorporating a hierarchy of virtual world classes.
- Users and applications viewed similarly, both manifesting themselves in the virtual world through their interaction with the objects contained within it. Supports multiple users and applications.

Key References: West et al. [1992], Snowdon, West and Howard [1993], Hubbold et al. [1993].

6.3 dVS

Developed By: Division Limited

Platform: "Various platforms"

Principal Characteristics:

- Division of the system into a number of discrete parallel components called Actors, which communicate via a distributed shared database.
- Layered architecture designed to hide the details of distribution of processing from the developer.

Key References: Described in Benford et al. [1994] as well as du Pont [1993]

6.4 Alice, DIVER (Distributed Virtual Environment Research Platform)

Developed By: User Interface Group, Computer Science Department, University of Virginia, USA

Platform: Silicon Graphics, Unix. Plans to port to Windows 95.

Principal Characteristics:

- Decoupling of simulation and rendering through *transparent* process separation
- Interpreted, object oriented development language facilitating rapid prototyping.

Key References: Pausch et al. [1995], Gossweiler et al. [1993].

6.5 CAVE (CAVE Automatic Virtual Environement)

Developed By: Electronic Visualisation Laboratory (EVL), University of Illinois; MCS Department, Argonne National Laboratory; National Centre for Supercomputing Applications, Illinois, USA

Platform: IBM SP-2 Parallel Supercomputer, Silicon Graphics Work-stations

Principal Characteristics:

- Decoupling of viewpoint lag (not involving simulation) from interaction lag (which incorporates the results of the simulation).

- VR used for three dimensional visualisation or large volumes of information.

Key References: Taylor, Stevens and Canfield [1995], Disz et al. [1995], Cruz-Neira et al. [1993], Song and Norman [1993].

6.6 VERN (Virtual Environment Realtime Network)

Developed By: Institute for Simulation and Training & Department of Computer Science, University of Central Florida, USA

Platform: Unix (Sun Solaris/Silicon Graphics)

Principal Characteristics:

- Object Oriented implementation in C++
- The concept of Players and Ghosts, which facilitates communication between parts of the system that reside on separate machines. (Described in Section 5.3 above)

Key References: [Blau et al., 1992]

6.7 BrickNet

Developed By: Institute of Systems Science, National University of Singapore

Platform: Silicon Graphics, Unix

Principal Characteristics:

- Object orientation and distributed processing (client-server architecture)
- Layered Architecture in which the details of lower layers are hidden from higher level application-oriented layers.

Key References: [Singh et al., 1994]

OOVE

Application Description

Technical Product

Version 1.03

Document Status: Approved

Table of Contents

Change History

Configuration Control

Project:	OOVE
Title:	Application Description
Doc. Reference:	WB 103
Created by:	Warren Blumenow
Creation Date:	17 April 1995

Document History

Version	Date	Status	Who	Saved as:
0.01	95/04/17	Draft	WB	.1995_07\TP\APP\WB103.001.DOC
0.02	95/08/13	Draft	WB	\1995_07\TP\APP\WB103.002.DOC
0.03	96/09/21	Draft	WB	\1995_07\TP\APP\WB103.003.DOC
1.00	97/10/25	Approved	WB	\1995_07\TP\APP\WB103.100.DOC
1.01	97/11/14	Approved	WB	\1995_07\TP\APP\WB103.101.DOC
1.02	98/02/03	Approved	WB	\1995_07\TP\APP\WB103.102.DOC
1.03	98/06/19	Approved	WB	\1995_07\TP\APP\WB103.103.DOC
1.10	98/09/02	Approved	WB	\1995_07\TP\APP\WB103.110.DOC

Revision History

Version	Date	Changes
0.01	95/04/17	New document created using WB 002.
0.02	95/08/13	First complete draft version.
0.03	96/09/21	Updated Sections 2 and 5.
1.00	97/10/25	Document status changed to Approved. Updated Sections 4 and 5.
1.01	97/11/14	Added Section 7 and miscellaneous other minor changes.
1.02	98/02/03	Expanded Section 6.
1.03	98/06/19	Miscellaneous changes throughout.
1.03	98/09/02	Minor changes to Section 4. Final cosmetic editing and formatting.

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	97/10/19	Approved	WB 525

1 Scope

1.1 Introduction

This document provides the necessary background to the problem which is being addressed, and offers a functional description of the proposed application. The project investigates the design of an improved Man-Machine Interface for electrical power network monitoring.

The motivating factors—in particular the shortcomings of existing network monitoring methodologies—are discussed, and previous approaches to the problem are described and evaluated.

The solution put forward involves the use of Virtual Reality technology to facilitate network visualisation. Interactive three dimensional user interfaces are well suited to the task of data visualisation, taking direct advantage of the vast capabilities of human perceptual mechanisms. As a result, there is great potential for the use of this technology in electrical network monitoring applications.

1.2 Purpose

The purpose of this document is to place the project in its proper context, and to broadly identify the functional requirements of the application that will be developed.

1.3 Definitions

1.3.1 Abbreviations

SCADA = Supervisory Control And Data Acquisition

MMI = Man-Machine Interface

1.4 Audience

The audience for this document includes the project manager, the external examiner, members of the SEAL management board, and any other interested parties.

1.5 Applicable Documents

1.5.1 Standards

a. SEAL QMS Document Creation Template, QS 002, Revision 0.02, 3 October 1994

b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Rev 00, 3 October 1994

2 Problem Specification

Existing man-machine interfaces used for the monitoring of electrical power networks have some serious limitations. Incoming information about network problems is typically presented to the operator in the form of text-based alarm messages [Blake, 1995a]. The comprehension and interpretation of these messages can place a significant cognitive load on the operator [West et al., 1992]. When a fault occurs, it is critical that the problem be resolved rapidly in order to minimise degradation of the power supply to the consumer.

Improvements in electrical power system data acquisition and control systems have lead to increases in the amount of data which is made available to the control staff. While useful for fault diagnosis, this additional data can prevent the operator from reacting quickly [McDonald, Burt and Young, 1992]. During a disturbance, the number of alarms received can rapidly exceed the ability of the operator to react appropriately to them. The result is a condition which Candy [1995:2] refers to as data overloading (see Section 3.4 below). Despite the overwhelming number of alarm messages received, most of the messages are not directly related to the problem, and as such are regarded as 'noise' and ignored by the operator. The existing text-based methodologies for the presentation of disturbance data are simply no longer adequate when dealing with highly complex systems. Clearly, an improved man-machine interface philosophy for electrical network monitoring is needed.

Solutions such as the use of information filtering and artificial intelligence (specifically rule-based Expert Systems) have previously been proposed as methods of reducing the amount of data having to be dealt with by the operator. Such approaches undoubtedly do have a role to play in network monitoring, but as will be discussed, both of these techniques have their limitations. Furthermore, neither of these approaches addresses the essential requirement from the point of view of the operator, namely rapid visualisation of the available information to obtain a mental picture of the power system state [Candy, 1995].

Hubbold et al. [1993:2] point out that information which is *presented* in a suitable way is able to be processed far more effectively and directly by the user. Visual perception is by far the most powerful of the human senses, and humans have well developed visual perception skills. People are capable of processing visual data extremely quickly. User interfaces which incorporate graphical representation are able to recruit these capabilities, and are far more natural and intuitive than text-based systems. According to Blake [1994], visualisation "greatly enhances human comprehension, learning and communication, especially for highly complex systems". The use of graphics increases the bandwidth between

human operator and machine, relying on the operator's perceptual mechanisms to filter and abstract the information that is presented. This has the effect of transferring much of the load on the operator from the cognitive to the perceptual.

3 Background

3.1 Electrical Power Network Structure

Candy [1995:7] models a power system as a pyramid structure, with each level in the pyramid representing a different level of abstraction of the network. This pyramid model is illustrated in Figure WB103–1. At the lowest level are individual data *elements*, corresponding to individual status indicators. Groups of these elements form *devices*. Device groups are referred to as *electrical objects*. At the next level of abstraction are *substations*, which consist of groups of electrical objects. Substations may be grouped into *regions*, and a number of regions make up the *entire network*.

For each level in the pyramid structure, there are corresponding two dimensional schematic layouts which show the connectivity of the electrical equipment comprising that level.

This hierarchical pyramid model reportedly closely approximates the mental models used by the control staff in analysing problems on the network.

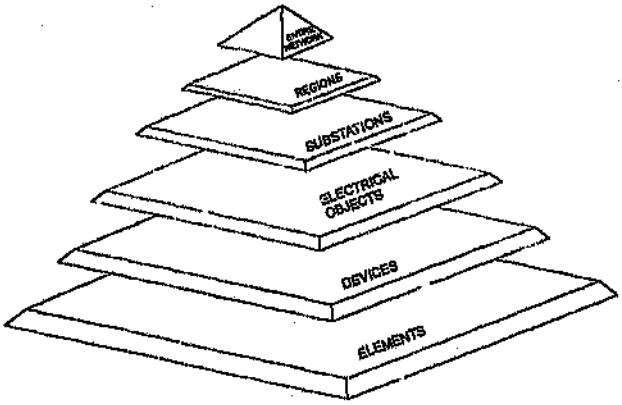


Figure WB103–1: Power System Hierarchical Pyramid Model

3.2 SCADA Systems and Alarm Generation

SCADA (Supervisory Control And Data Acquisition) systems are industrial measurement and control systems that are used to gather data from remotely located pieces of equipment. They are commonly used in the monitoring and control of electrical power systems. A SCADA system essentially consists of three components: a master station, a number of remote terminal units (RTUs) and some form of communications or telemetry subsystem [Wiese, 1997].

Remote terminal units are the devices responsible for the acquisition of SCADA data at the remote site. An RTU is predominantly concerned with reporting changes of state in various *elements*. These elements correspond to the elements which form the lowest level of the pyramid model described in Section 3.1.

According to Wiese [1997], the data obtained from an element can be of three main types. *Analogue* data represents the measured value of a continuously varying quantity. *Digital* data identifies an element as being in one of a number of discrete states. For example a switch may be either on or off. Finally, *pulse* data indicates the cumulative occurrence of some event, for example "number of revolutions".

The data gathered by an RTU is processed to detect alarm conditions based on known criteria, and alarm messages are generated for each alarm condition that is triggered. These alarms are transmitted to the central master station via the communications subsystem. The master station presents the incoming alarm messages to the operator via the man-machine interface.

3.3 Operator Requirements

In designing a man-machine interface for network monitoring, it is important to identify the requirements of such a system from the point of view of the operator.

When a fault occurs, the primary task of the operator initially, is to determine the nature and cause of the problem and evaluate its potential consequences. In order to do this the operator must build a clear mental picture of the power system state. Based on this information, the operator then determines a course of action to correct the problem.

Blake [1995a] states that the primary goal of the MMI should be to provide the operator with the precise information that he requires at any given time. Kirschen et al. [1989] have identified some serious limitations of conventional alarm-based systems, which point to the fact that such systems do not achieve this objective. In particular, alarm messages may contain too much or not enough information, alarms may be unnecessarily repeated, and multiple alarms may be generated for the same condition. In addition, the sheer volume of alarms may be overwhelming.

3.4 Data Overloading

Candy [1995] defines data overloading as:

> *"...a situation that occurs when the number of alarm messages arriving at operator workstations exceeds the ability of the control staff to use, process or react to them."*

Existing network monitoring mechanisms which make use of text-based alarm messages are particularly prone to causing this condition. When a fault occurs, the operator is suddenly faced with a flood of alarm data, much of which is associated with events that are extraneous to the actual cause of the problem. Messages resulting from these secondary effects must be acknowledged and dealt with by the operator even though they do not assist him with the primary task of understanding the nature and extent of the disturbance. Moreover, these messages may obscure other important information.

Clearly, if problems on the network are to be resolved rapidly, data overloading is a condition which should be avoided.

4 Previous Improved Man-Machine Interface Strategies

4.1 Abstraction and Data Reduction through Filtering

One approach to improving the user interface involves the use of filtering techniques to reduce the volume of information being presented to the operator. Techniques which may be used to reduce the amount of data needing to be dealt with include aggregation, averaging over time and thresholding [Becker, Eick and Wilks, 1995].

Classification of events into categories is one means for aggregation of incoming data. Candy [1995] proposes that incoming alarm information be grouped into just four categories, namely health, main protection, backup protection, and information.

4.2 Expert Systems

The use of expert systems to pre-process the data received from the control system has been discussed by a number of researchers [Kirschen et al., 1989; McDonald, Burt and Young, 1992]. These "Intelligent Alarm Processors" make use of predefined heuristic rules to draw conclusions about incoming alarm information, thereby alleviating some of the demands placed on the operator.

The main advantage that expert systems have over filtering techniques is that they employ domain knowledge in the drawing of inferences. This gives them an ability to perform a deeper analysis of the nature of a disturbance. For example, an expert system may have knowledge of the configuration of the network or the behaviour of various pieces of electrical equipment. Through the use of this knowledge, the propagation of the effects of a fault can be simulated, and various diagnostic conclusions can be drawn.

Most expert systems approaches employ a brute force approach in which the input to the expert system is the unfiltered, raw stream of alarms generated by the SCADA system, combined with a priori information about the topology of the network and the mapping of alarm information to actual devices.

The expert systems described by McDonald, Burt and Young [1992] use a reasoning process based on hypothesis. A number of possible hypotheses are generated to account for a particular alarm message. Each subsequent alarm that is received provides additional information which either supports or disproves these hypotheses, until a statistically likely candidate explanation is found.

4.3 Graphical Displays

4.3.1 Network State Visualiser Prototype

An improved network monitoring man-machine interface strategy based on graphical representation of the network state has been described by Blake [1995a]. The philosophy behind this approach is to enable the operator to visually assess the network state without having to deal with large numbers of alarm messages.

Blake's prototype *Network State Visualiser* system consists of a two dimensional split-screen graphical user interface, which presents the network to the operator diagrammatically through the use of tiled schematics. Information about the current network state is superimposed on these schematic representations of the network.

The system essentially provides the operator with a window onto a particular level of the pyramid model described in section 3.1. The operator is able to pan the window (in discrete steps) across the current layer, as well as switch between layers. Sub-windows show how the current view fits into the context of the higher-level layers.

The use of graphics allows alarm information to be presented in a more natural and intuitive manner than is possible with text-based systems.

4.3.2 SeeNet

Many of the problems that are confronted in monitoring electrical power networks—system complexity, network size, extremely high data volumes—are also encountered in *data communications* network monitoring. Becker, Eick and Wilks [1995] have developed a suite of graphical tools called SeeNet, for visualising data communications network data.

Like the Network State Visualiser system described above, SeeNet makes use of a two dimensional graphical representation of the network, with the ability to successively zoom into regions to obtain greater detail or zoom out to establish context.

The system incorporates some novel representation and interaction techniques. *Linkmaps* show the connectivity of the network; *nodemaps* display statistical information for each node by varying the visual characteristics of symbolic glyphs; *parameter focusing* allows the values of various display parameters to be altered by the user; and *direct manipulation* provides the user with continuous visual feedback during interaction with the system.

5 Proposed Solution

The approaches described in Section 4 have a definite role to play in addressing the deficiencies of existing user interfaces for network monitoring. Each of these alternatives, however, introduces problems of its own. The solution that is proposed involves the use of three dimensional user interface technology, and Virtual Reality in particular, as an alternative to these approaches.

5.1 Problems with Previous A[illegible]aches

5.1.1 Filtering

The key disadvantage of filtering techniques such as aggregation or averaging is that they may obscure important information [Becker, Eick and Wilks, 1995]. Superficial processing by means of filtering rules may not be powerful enough to adequately distinguish important information from noise. Furthermore, filtering methodologies may only be applicable to particular types of information. For example, averaging over time is generally only meaningful for analogue element data.

It should also be noted that techniques such as averaging and thresholding are typically already incorporated into the mechanisms that are used to trigger alarm conditions at the RTU in the first place. As a result, the application of these techniques again at the alarm level may be of limited utility.

5.1.2 Expert Systems

The main problem with the use of expert systems in alarm processing is response time. Very extensive rule bases are required when analysing complex systems where the number of possible scenarios is large. A large number of rules coupled with a large number of alarms typically results in slow problem diagnosis and unacceptably long delays before conclusions are reached [Candy, 1995].

In addition, the large number of rules that are needed to cover all possible eventualities typically makes expert systems expensive to implement and maintain.

Candy [1995] contends that implementing expert systems at the RTU is preferable to using a centralised expert system at the master station. He reasons that centralised, brute force expert systems do not take into consideration the underlying problem that the number of alarms being generated in the first place is high. The aim of an expert system at the RTU level would be to reduce the total number of alarms being generated at the source. The parallelism inherent in this approach would to a large

extent alleviate the problem of response time normally associated with expert system alarm processing. However, this approach does not alleviate the cognitive burden that is placed on the operator of interpreting text messages to build a mental picture of the power system state.

5.1.3 Graphical User Interfaces and Navigation

Given the inherent advantages of visual representation of information, graphical user interfaces represent a vast improvement over existing text-based, alarm-centred methodologies. Graphical displays are able to directly recruit the perceptual capabilities of the user to facilitate the rapid comprehension of information. There are problems, however, with traditional two dimensional graphical interfaces, most notably *navigation* and *loss of context*.

Blake [1995a:54] indicates that navigation is a fundamental problem of Cathode Ray Tube (CRT) based plant monitoring systems. At a particular time, a single CRT can only provide the operator with a limited view of the plant. Movement of the view from one area of the plant to another typically requires navigation through a number of separate screens. Users are prone to experiencing loss of orientation when swapping between discrete viewports. As a result, the operator may lose track of the context of the information being presented. West et al. [1992] agree, noting that limitations of screen size and flat display surfaces are significant contributory factors to the difficulty of designing interfaces to complex systems. Becker, Eick and Wilks [1995] report that loss of global context and disorientation are also encountered when performing *zooming* operations.

Blake suggests a number of mechanisms to improve navigation between discrete schematic views of the network. These include hyperlinks, navigation to logically or geographically adjacent "tiles", hierarchical abstraction (a particular tile maps to multiple tiles on a lower level), lists of recently used components, and *navigation fadeout* (recently visited locations remain highlighted in higher level subviews). The author feels that while they undoubtedly improve matters, none of these approaches provides an entirely satisfactory solution to the problem of navigation. Each of these mechanisms places a not insignificant cognitive load on the operator, who must mentally reconcile the "jump" between current and prior views.

5.2 Virtual Reality for Electrical Network Visualisation

Interactive, realtime three dimensional user interfaces are extremely well suited to visualisation applications. Virtual Reality provides a natural interface through which spatial and hierarchical relationships can be effectively represented. It enables the user to focus on analysing the

information being presented rather than his interaction with and manipulation of the user interface. In fact one of the design goals of an immersive Virtual Environment is to give the user the sensation of interacting directly with the application, and not using a computer interface at all.

VR to a large extent overcomes problems of navigation and loss of context by providing natural three dimensional interaction mechanisms which reinforce the user's spatial memory. Spatial memory refers to a person's ability to remember where objects are in three dimensional space [West et al., 1992].

Three dimensional visualisation facilitates rapid comprehension of information by relying on the user's innate three dimensional perceptual capabilities. Human visual processing mechanisms are inherently able to resolve monocular and binocular depth cues to visually isolate objects of interest in a three dimensional scene. The addition of the third dimension of depth consequently allows more information to simultaneously and unambiguously be displayed within the user's field of view.

The most significant advantages that VR technology provides over other user interface styles are:

- functionally unlimited viewing area
- three dimensional spatial representation of relationships
- natural and intuitive three dimensional interaction mechanisms

5.3 Applicability of the Proposed Approach

Fairchild, Poltrock and Furnas [1988] contend that in general a large knowledge base cannot be represented in two dimensions without a large number of the interconnections between nodes intersecting. The presence of these intersections results in unnecessary visual complexity, which may introduce representational ambiguities or impede the tracing of interconnections. In three dimensions, an arbitrary network can be represented without intersections, and the user is able to resolve ambiguities simply by changing his viewpoint.

The pyramid model of an electrical power network that was presented in Section 3.1, maps well to three dimensional navigation over the network. The highest levels in the pyramid represent the network when viewed from some distance, showing the entire network and the regions, but hiding much of the underlying detail. As one moves closer, less of the network is simultaneously visible, and the representation shows more of the detail of the lower levels in the pyramid. Three dimensional navigation coupled with

smoothly changing level of detail adjustment of the model neatly captures the semantics of the pyramid model.

5.4 Previous Work

Virtual Reality has been used extensively for information visualisation applications. Systems described in the literature include:

- Cosmic Explorer, a Virtual Reality environment for visualising numerical cosmology data [Song and Norman, 1993].
- A scientific visualisation system for interacting with supercomputer simulation data, based on the CAVE Automatic Virtual Environment [Taylor, Stevens and Canfield, 1995].
- The CA Unicenter TNG [data communications] network monitoring tool, which features a three dimensional user interface for visualising network components [Stevens, 1997].
- A realtime three dimensional geographic information system, or "Virtual GIS", which facilitates visualisation and understanding of terrain models [Koller et al., 1995].
- A prototype system developed by British Telecom, which uses Virtual Reality to monitor network integrity and detect faults in a large telecommunications network [Andrew and Ellis, 1993].

6 Application Description

This section provides a description of the proposed application from the perspective of the user, and outlines the key functional requirements of the system.

6.1 Overview

The proposed application will take the form of an enhanced Man-Machine Interface (MMI) for the visualisation of electrical power network monitoring data. The envisioned MMI will comprise a Virtual Environment within which the user interacts with a model of the network by means of natural and intuitive navigation and manipulation mechanisms

The user will be presented with a graphical map of South Africa. The electrical network model will be represented as various three dimensional objects displayed on the map. The visual three dimensional characteristics and appearance of these objects will convey dynamic information about the state of the electrical objects they represent. For example, an object's height might be used to convey information about voltage, surface texture might be used to indicate frequency, and colour might be used to demarcate energised zones.

The user will have the capability to navigate freely within the environment in three dimensions. Consequently, the user will be able to observe, from any desired vantage point, the various spatial and hierarchical relationships which exist within the model, as well as inspect any of the model parameters which are graphically depicted. The user will be able to continuously zoom in on (move towards) objects of interest in order to view them in closer detail, or zoom out in order to acquire a more global perspective of the model. As the user's proximity to a particular region of interest increases, the level of detail displayed will change accordingly to reveal only those features which are relevant to the current viewing context.

Changes in the state of the objects which are presented will be driven by a continuous simulation, which will update the displayed view in real time. In a future full scale implementation of the system, the visualisation MMI would interface directly with existing SCADA systems. The simulation data would be updated in real time in response to alarm information received from RTUs.

6.2 Potential Additional Features

A mechanism similar to Becker, Eick and Wilks' parameter focusing [1995], would enable the operator to dynamically vary the information presented by the system depending on his current needs or objectives.

For example, at different times the operator might wish to see only the connectivity of the network or only the loading characteristics. In this way, information not relevant to the task at hand would be suppressed.

The use of three dimensional sound (discussed in the Literature Survey, WB 101) would be of great benefit in alleviating problems such as loss of global context and what may be referred to as *tunnel vision.* The latter condition occurs when the operator observing the network at a high level identifies a fault and zooms into that portion of the network to resolve the problem. Meanwhile, another fault occurs in a portion of the network which is not observable within the operator's current viewport. The result of this is that a small fault might divert attention from and hamper the resolution of a more critical problem. The use of localizable audible alarms could be used to alert attention to portions of the network that are not currently visible. In the same way, sound could be used to give the operator auditory cues for determining his current global position and orientation.

6.3 Prototype System

The objective of the project at hand is by no means to construct a comprehensive full scale system with all of the functionality that has been described above. Rather, the scope of the project is to build a prototype system which effectively demonstrates key concepts and contributory technologies (particularly object orientation and distributed processing). The prototype is intended mainly to provide a sound architectural framework for structuring future applications, and to demonstrate the feasibility of the proposed approach.

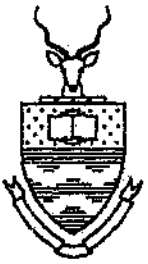
In light of this, a number of simplifying assumptions will be made in implementing the prototype system. In particular:

- A simplified network model will be used in the prototype application, consisting of only a few objects with relatively straightforward dynamics.
- The simulation will not interface with any existing network monitoring systems. Instead, the data feed that is used will be obtained from a simple text file, and the simulation will update the state of the various displayed objects in some predefined manner. The necessary support for interfacing with external systems will, however, be provided by the system architecture.
- Dynamic level of detail adjustment and the additional features described in Section 6.2 above will not be implemented within the scope of this project.

7 Overview of System Requirements

The main functional and non-functional requirements of the system include:

- Acceptable interactive performance in terms of frame rate and lag
- Transparent distribution of processing
- Hardware configuration flexibility
- Platform independence and portability
- Support for a range of input and output devices
- Maintainability and extensibility



OOVE

Product Functional Specification

Technical Product

Version 1.01

Document Status: Approved

Table of Contents

Change History

Configuration Control

Project:	OOVE
Title:	Product Functional Specification
Doc. Reference:	E:\MSC\1995_07\TP\PFS\WB102WW.101.DOC
Created by:	Warren Blumenow
Creation Date:	1996/07/14

Document History

Version	Date	Status	Who	Saved as:
0.01	96/07/14	Draft	WB	\1995_07\TP\PFS\WB102WW.001
0.02	96/11/20	Draft	WB	\1995_07\TP\PFS\WB102WW.002
0.03	97/03/15	Draft	WB	\1995_07\TP\PFS\WB102WW.003
1.00	97/06/11	Approved	WB	\1995_07\TP\PFS\WB102WW.100
1.01	98/07/28	Approved	WB	\1995_07\TP\PFS\WB102WW.101

Revision History

Version	Date	Changes
0.01	96/07/14	New document created using QST324-20.100
0.02	96/11/20	Completed Sections 2 (Problem Statement) and 3 (Environment)
0.03	97/03/15	Draft version completed.
1.00	97/06/11	Document status changed to Approved.
1.01	98/07/28	Small corrections and final formatting.

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	97/06/09	Approved	WB 523

Change Forecast

This document will be updated each time new requirements are identified or changes are made to the project specifications.

1 Scope

1.1 Purpose

This document provides a checklist of questions and answers for identifying the high level functional requirements of the project. It should be viewed in conjunction with the Application Description (WB 103), which provides a functional description of the application.

1.2 Applicability

This document forms part of the series of technical documents developed in support of the project. It builds on the functional requirements identified in the Product Description.

Reference is made in the document to key sections of the Management Products that have been developed to support this project.

1.3 Audience

The audience for this document comprises the various stakeholders of the Software Engineering Applications Laboratory (SEAL), including:

- The product developer, namely Warren Blumenow
- The project manager, namely Professor Barry Dwolatzky
- The project quality assurance (QA) Co-ordinator, namely Professor Alastair Walker
- All full-time and part-time post-graduate students associated with the SEAL
- Members of the SEAL Management Board
- The Head of Department, Electrical Engineering
- Individuals who will perform internal and external surveillance audits of the SEAL Quality Management System

1.4 Applicable Documents

1.4.1 Standards

a. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 1.00, 3 October 1994.

1.5 Assumptions

It is assumed that the reader is familiar with the ISO 9000 series standard for quality systems management.

1.6 Requirements Traceability

This document addresses the following requirements:

a. ISO 9001 (1994) 4.4 Design Control

2 Problem Statement

2.1 Product Title

The title of this MSc Project is: *"An Object Oriented Approach to the Design of Distributed Virtual Reality User Interfaces for Electrical Network Monitoring"*

2.2 What problem is the product meant to solve?

The project investigates the use of advanced man-machine interface technology for electrical network monitoring. Existing user interfaces used for power system monitoring display information about the network state to the operator in the form of text-based alarm messages. During a disturbance, the vast number of messages generated and their manner of presentation place extraordinary demands on the operator. This project proposes the use of interactive three dimensional graphical user interfaces and Virtual Reality as an alternative approach to the presentation of network monitoring information. Virtual Reality is able to recruit the considerable capabilities of the operator's perceptual systems in order to accelerate decision making and fault diagnosis.

Additionally, the product addresses various issues relating to the design of Virtual Reality systems in general—specifically those related to performance and cost. The use of distributed processing in order to provide acceptable real-time performance, while still allowing the system to run on inexpensive hardware, is a key design objective of the system.

2.3 Where does this product fit into the "big picture"?

This work forms part of a broad venture being undertaken by the Software Engineering Applications Laboratory (SEAL), which concerns itself with examining various aspects of Advanced Man-Machine Interface (AMMI) technology, Computer Based Training (CBT) and Virtual Reality. These projects together comprise the Advanced Computer Interfaces (ACI) group within the SEAL.

Certain aspects of the project involve collaboration with other members of the SEAL, including George Spanellis (refer to the MSc project entitled *"A Distributed Processing Approach for Implementing Advanced Man-Machine Interfaces"*, SEAL project reference number 1995_11), and Michael Polyzos (refer to the MSc project entitled *"A Distributed Virtual Reality Approach for Implementing Advanced Man-Machine Interfaces Using Low Cost Personal Computers"*, SEAL project reference number 1996_14).

2.4 Product History

This is a new project, and as such there is no product history. The documentation and software products which are created in the scope of this project are expected to form a basis for future projects within the Software Engineering Applications Laboratory.

3 Environment

3.1 Cost

3.1.1 What is the limit of monetary and time expenditures to be used to create the product?

The principle financial constraint on the project is the cost of the computer hardware which is used for development, the details of which have been identified in WB 005, the Project Management Plan. The product itself has no intrinsic cost associated with its implementation, since it consists entirely of software and documentation artefacts. Additional costs pertaining to the project relate to the training needs which have been identified in WB 005, as well as the costs associated with attendance at national and international meetings. The time allocated to the project will correspond to the time associated with a full-time Masters research project.

3.1.2 What is the limit of monetary and human resources expenditures to be used to maintain the product?

There is no intrinsic monetary cost associated with the maintenance of the product, beyond the cost of the hardware required to perform further development of the software. The human effort required will depend on the nature of the maintenance task undertaken. Since this is a prototype system, it is envisioned that significant further development of the product will take place, requiring a substantial investment of time.

3.1.3 What facilities are available for the product development?

The following facilities are available for development of the project:

- The computer identified as S-41 within the Software Engineering Applications Laboratory postgraduate facility
- Additional computers as required for testing of the system in various distributed processing configurations, sourced from unoccupied cubicles in the Software Engineering Applications Laboratory facility
- Infrastructure, printing, library and associated facilities furnished by the University of the Witwatersrand, the Department of Electrical Engineering and the Software Engineering Applications Laboratory

3.1.4 What personnel are available to support the product development

The product developer, namely Warren Blumenow and the project supervisor, namely Professor Barry Dwolatzky. (refer to Section 4, WB 005 Project Management Plan)

3.1.5 What methods of progress measurement does the customer require to track the development task? (e.g. status reports, progress reports)

Regular project progress meetings are to be held with the project supervisor, during which applicable documents will be tabled for review. In addition, regular progress reports are to be submitted to the Faculty of Engineering.

3.1.6 What are the milestones that shall be met? (i.e. schedules, deliveries)

The major milestones in the development of the project are specified in the Work Breakdown Structure that has been defined in the Project Management Plan (WB 005).

3.2 Hardware/Software

(See also Section 4.6, WB 005 Project Management Plan)

3.2.1 What computers does the product run on? (e.g. makes, models, particular installations).

The system is designed to run on a group of standard IBM PC compatible computers. The minimum specified requirement calls for 80486 class PCs, although Pentium class PCs are recommended. The computers are required to be connected via a standard Ethernet network.

3.2.2 What peripherals are required to interface with the product? (e.g. terminals, hardcopy devices, tape drives, disk drives, displays, etc.)

In the initial implementation, the only peripherals which will be used will be the standard keyboard, mouse and display monitor of each of the computers on which the system runs. This will be a so-called "fish-tank" Virtual Reality system implementation. It is envisioned, though, that in a future incarnation, the system will interface with a number of specialised Virtual Reality input and output devices, such as a dataglove, head-mounted display and head tracking hardware, and the ability to add support for these devices must be provided for in the design of the system. The reader is referred to the Literature Survey (WB 101) for a comprehensive discussion of these technologies.

3.2.3 What unique interfaces does the product need to access? (e.g. cockpit hardware, microcomputer, test benches, etc.)

In the initial implementation, there will be no specialised interfaces. However, in a final full scale implementation of the system (beyond the scope of this project), the software will be required to interface with existing Supervisory Control and Data Acquisition (SCADA) systems (discussed in the Application Description, WB 103), which will provide the

simulation input data streams. In the prototype system, these interfaces will be simulated by reading data contained in standard text files.

3.2.4 On what operating system does the product run? (i.e. operating system ID, revision level).

The target operating system is Microsoft Windows 95 Version 4.00.950a or 4.00.950b, or Windows NT Workstation Version 4.0. The product is designed to run equivalently on both platforms. Recompilation of the source code for each platform is required, but no modification of the source code is necessary. When the system is distributed over a group of networked PCs, a heterogeneous mix of the two operating systems is permissible.

3.2.5 What database methods does this product use? (e.g. Dbase, SQL etc.)

The product does not require any database functionality in the current implementation.

3.2.6 With what other external (foreign to this product) software products does this product interface? (e.g. routines, libraries, product names, manufacturers, suppliers, revision levels, etc.)

The product interfaces with:

- The Microsoft Foundation Classes (MFC) Version 4.2
- Criterion Software's RenderWare, Version 2.0
- Code developed by George Spanellis, as part of the MSc project entitled "A Distributed Processing Approach for Implementing Advanced Man-Machine Interfaces", SEAL project reference number 1995_11
- Linked List template class code developed by Stephen Levitt as part of the MSc project entitled "A Map Symbol Recognition System', SEAL project reference number 1995_10

These products and their relation to the project at hand are discussed fully in the High Level Software Design, WB 105, Section 3.1, in the context of the documentation of the software architecture.

3.2.7 What are the size restrictions for the finished product? (e.g. bytes of code, text, embedded data, etc.)

There are no explicit size restrictions of this nature. However, efficiency and effective resource utilisation will be considered during development of the product.

3.3 Required Design Restrictions

3.3.1 What are the language restrictions for the product? (e.g. language names, versic · or maximum number of languages, etc.)

The product will be deveioped entirely using the C++ programming language. The particular compiler that has been selected is the Microsoft Visual C++ compiler, Version 4.2. The user-interface specific portions of the program source code will make use of the Microsoft Foundation Classes (MFC).

3.3.2 To what standards shall the product conform? (e.g. document titles, sources, revision levels, etc.)

In terms of documentation, the applicable standards are defined in QS 003, the Document Layout, Typesetting and Presentation Standard of the Software Engineering Applications Laboratory Quality Management System. No other specific standards to which the product is to conform have been identified.

3.4 Customers

3.4.1 Who is commissioning this product? (e.g. identify signatures of authority, etc.)

The project has been proposed by the author, by means of a research proposal submitted to the Faculty of Engineering, University of the Witwatersrand.

3.4.2 Who is to accept delivery? (e.g. identify signatures of authority, etc.)

The project supervisor, Professor Barry Dwolatzky.

3.4.3 What is the profile of the typical end user? (e.g. years of experience in some specified job, years of schooling, etc.)

Since the software produced is to be a prototype system, the end user of the product is likely to be another researcher within the Software Engineering Applications Laboratory. However, testing may be undertaken in which the product is given to control room staff in order to assess its efficacy in comparison with existing systems. Such testing would draw on the user's experience of the current user interfaces for electrical network monitoring.

3.4.4 What are the human factors criteria and how shall the success of the product in meeting the criteria be determined? (e.g. non-direct paths, non-standard user-interface, etc.)

The use of Virtual Reality technology introduces several human factors issues related to the immersive nature of the interface and its effects on human perception. Specifically, VR interfaces are predisposed to inducing particular forms of motion sickness. This condition is commonly referred to as "simulator sickness" or "cybersickness". These issues are discussed in more detail in the Literature Survey (WB 101).

3.5 Performance factors

3.5.1 What human support is required to operate the product? (e.g. Identify any runtime operations which the user or support personnel shall physically carry out in order to achieve any specified results, etc.)

The nature of the product as an interactive visualisation tool implies significant human involvement during use of the system. In addition, there will be certain configuration tasks that are required to be performed prior to using the product. Specific information concerning the user interface and system configuration can be found in the User Reference Manual (WB 107).

3.6 Operational Considerations

3.6.1 What methods shall be used to deliver the product? (e.g. data storage or transfer medium, etc.) (See also QSP 335-10 Software Development: Handling, Storage, Packaging and Delivery).

The main deliverable of the project is the Masters dissertation, which will be delivered to the Faculty of Engineering in hard copy as per the requirements of the University and the higher degrees committee. In addition, this project will form part of the projects archive of the Software Engineering Applications Laboratory, and as such, will be stored in electronic format, and archived on the Software Engineering Applications Laboratory FTP server facility. An optical disk copy of the software and documentation will also be supplied and will be contained in the project binder.

3.6.2 What are the restrictions on the installation of the product? How easy shall the product be to install? (e.g. classes of users, numbers of users, access limitations etc.)

Installation of the product will be relatively straightforward, involving mainly the setup of the various configuration files. The use of the same executable program on all machines as described in the Low Level Software Design (WB 113) will simplify installation.

3.6.3 To what degree shall the operation of the product be successfully restricted to authorised users only? (e.g. classes of users, numbers of users, access limitations etc.)

No access restrictions are applicable.

3.6.4 How shall product integrity be ensured? (e.g. data recovery source code recovery, etc.)

The comprehensive strategy that will be used for project artefact archival and backup is described in the Configuration Management Plan (WB 006)

3.7 Maintainability

3.7.1 What support of the product after turnover to the customer is required? (e.g. training support, computer personnel, etc.) (See also QS 139 Servicing).

After submission of the project, the product and all of its artefacts will be managed by the project supervisor and the QA co-ordinator of the Software Engineering Applications Laboratory.

3.7.2 Where might the requirements for the product expand in the future?

The aim of the project at hand is to demonstrate the feasibility of the proposed application by investigating relevant concepts and technologies. The requirements which have been identified consequently represent the requirements of a *prototype* system. It is envisioned that significant future work will be carried out in this particular area of research, and there is therefore significant scope for the functional requirements of the product to change as the product evolves in the future.

4 Documentation

4.1 Requirements

4.1.1 What management documents will be developed to support the product?

Master Document List	WB 001
Document Creation Template	WB 002
Quality Plan	WB 003
Product Description	WB 004
Project Management Plan	WB 005
Configuration Management Plan	WB 006
Binder Labels	WB 008
Project Archive Diskette Labels	WB 009

4.1.2 What technical documents are recommended for the development of the product? (Refer to QS 324 Software Development - Design Control Policy)

Documents that will be developed as part of this project will include but not be limited to:

- a comprehensive survey of the relevant literature
- a document detailing the proposed application
- conceptual system design document
- high level software design document
- low level software design document
- user reference manual

4.1.3 What are the definitions of all the acronyms and technical terms?

These are defined in each specific document in which they are used.

4.1.4 Who has created and revised the requirements document and what are the dates of creation and revision?

This information is provided in the Change Control segment of this document.

4.2 Support

4.2.1 What type of support documentation shall the developer provide with the product?

This is as defined in QS195 Project Documentation and Support Standard.

4.2.2 To what standards shall the support documentation for the product conform?

The supplied documentation will conform to the Quality Management System Requirements of the SEAL Quality Management System, which is traceable in its requirements to ISO 9001 (1994).

Appendix Group III

Technical Products

Design and Implementation

OOVE

Conceptual System Design

Technical Product

Version 1.02

Document Status: Approved

Table of Contents

Change History

Configuration Control

Project:	OOVE
Title:	Conceptual System Design
Doc. Reference:	E:\MSC\1995_07\TP\CONCEPT\WB104WW.102.DOC
Created by:	Warren Blumenow
Creation Date:	8 September 1998

Document History

Version	Date	Status	Who	Saved as:
0.01	95/04/17	Draft	WB	\1995_07\TP\CONCEPT\WB104WW.001
0.02	96/11/21	Draft	WB	\1995_07\TP\CONCEPT\WB104WW.002
0.03	97/03/09	Draft	WB	\1995_07\TP\CONCEPT\WB104WW.003
1.00	97/10/25	Approved	WB	\1995_07\TP\CONCEPT\WB104WW.100
1.01	98/04/13	Approved	WB	\1995_07\TP\CONCEPT\WB104WW.101
1.02	98/08/27	Approved	WB	\1995_07\TP\CONCEPT\WB104WW.102

Revision History

Version	Date	Changes
0.01	95/04/17	New document created using WB 002
0.02	96/11/21	Documented draft specification of Conceptual Virtual Reality System Model
0.03	97/03/09	Added Sections 3 (PAM) and 4 (EM). Updated Section 2.
1.00	97/10/25	Document status changed to Approved. Changed made to Sections 2 and 3. Added Section 5.
1.01	98/06/13	Miscellaneous changes after meeting of 98/06/10
1.02	98/08/27	Final corrections and formatting.

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	97/10/25	Approved	WB 525

1 Scope

1.1 Introduction

This document outlines the design of the system from a conceptual standpoint, providing insight into the functional structure of the proposed solution. The concepts and models presented here comprise the fundamental basis underlying the system architecture.

There are essentially two models that have been developed, and which are described in this document. The first of these is a conceptual model of a generic Virtual Reality system; the second is a model for achieving transparent distribution of processing. The former will be referred to as the *Conceptual Virtual Reality System Model*. The latter has been called the *Process Agent Model (PAM)*.

The document also separately describes the Environment Manager and the Network Communications infrastructure which form part of the Process Agent Model.

1.2 Purpose

The purpose of this document is to give the reader insight into the key conceptual models that have been formulated. These models provide the conceptual framework for the software implementation that has been developed. It is crucial that the reader is familiar with the concepts that are discussed here in order to appreciate in their proper context the design decisions that have been made in the software design of the system (The software design is documented fully in the High Level and Low Level Software Design documents—WB 105 and WB 113)

1.3 Definitions

1.3.1 Abbreviations

OOVE = Object Oriented Virtual Environments (This is the acronym which has been assigned to the project at hand)

MVC = Model–View–Controller

PAM = Process Agent Model

1.4 Audience

The audience for this document includes the project manager, the external examiner, individuals involved in further development of the project,

members of the SEAL management board, and any other interested parties.

1.5 Applicable Documents

1.5.1 Standards

a. SEAL QMS Document Creation Template, QS 002, Revision 0.02, 3 October 1994

b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 1.00, 3 October 1994

1.5.2 References

a. References and Bibliography, WB 114

1.6 Assumptions

Several of the concepts that are presented here have been gleaned from the literature. It is therefore assumed that the reader has consulted the literature survey document (WB 101), and is familiar with the material contained therein.

2 Conceptual Virtual Reality System Model

A generalised conceptual model of a Virtual Reality system has been developed. This model provides a framework for structuring an interactive VR application, by delineating the principal functional components of the system, and specifying how these components interact to provide the required VR system functionality.

2.1 Foundations and Relevant Previous Work

A number of distinct approaches to the design of Virtual Reality systems have been encountered in the literature, and have been discussed in some detail in the Literature Survey (WB 101). The conceptual system model that is presented here, while novel in its own right, incorporates a number of aspects of existing systems, distilled from the literature. Of particular relevance are the Model View Controller (MVC) user interface paradigm [Krasner and Pope, 1988], the Decoupled Simulation Model (DSM) [Shaw et al., 1993], and the AVIARY system's approach to dealing with users and applications. The model that is described combines and integrates some of the most useful features of these systems, and in addition introduces several concepts that have not previously been described.

The Model-View-Controller methodology provides a sound basis for structuring interactive graphics applications [Krasner and Pope, 1988; Kyriazis, 1990]. It is natural to break a VR system down in such a way as to maintain a clear separation between those components dealing with modelling of the underlying application domain, those dealing with input processing and the user's interaction with the system, and those concerned with graphics generation and the presentation of the application state.

The Decoupled Simulation Model ensures that the visual update rate of the system is not temporally dependent on the update rate of the underlying simulation [Shaw et al., 1993]. Simulation decoupling can be achieved within the framework of MVC, by distinguishing those portions of the *Model* that are concerned with graphical manipulation and geometric modelling from those that perform non-graphical computation or simulation. Splitting the Model into two separate components in this way allows the main visual update loop to proceed independently of the simulation.

From the point of view of their influence over the objects in the virtual world, the AVIARY system does not regard users and applications as being fundamentally distinct from one another [West et al., 1992]. An object may exhibit its own internal volitional behaviour or have behaviour

imposed on it by an external source. Both the application and the user are seen as such external forces driving the behaviour of objects within the world. Each interacts with the virtual world through a clearly defined interface.

The only manifestation of the user's interaction with the system is in the behaviour that such interaction invokes within the virtual world. Likewise, the only influence the simulation has over the information that is eventually presented to the user is through *its* interaction with, and influence over the behaviour of, the objects in the virtual world.

2.2 Components and Processes

In light of the above discussion, the model that has been developed distinguishes at the highest level between four main system components: an *Interaction* component, a *Presentation* component, a *Simulation* component and a *Geometric Model*. The overall VR system will contain each of these components, as illustrated in Figure WB104–1. The user has also been represented in this diagram. The system components communicate both with one another and with the user; these relationships have been indicated by means of arrows.

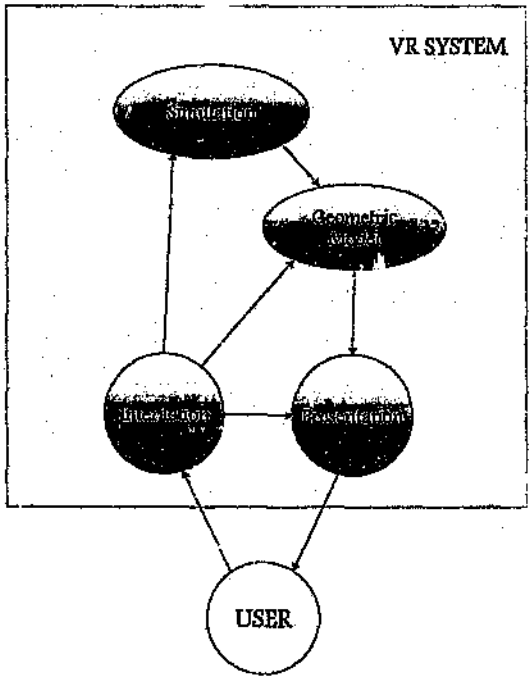


Figure WB104–1: High Level System Components

A concrete realisation of the concept of intercommunicating system components is provided by the notion of *system processes* which pass messages to one another. While there will necessarily be a close correlation between the system components and the corresponding processes by means of which they are realised, each high level system component may consist of more than one process. Consciously decomposing each system component into a number of processes in this way allows for effective distribution of processing. This aspect is discussed further in Section 3.

A process-level view of the Conceptual Virtual Reality System Model is depicted in Figure WB104–2. Corresponding to the four main system components referred to above, the four main types of processes that will be present in a VR system implementation are:

- Input Handler Processes
- Presentation Generator Processes
- Simulation Processes
- Object Processes

The arrows in the diagram indicate the direction of message passing between processes.

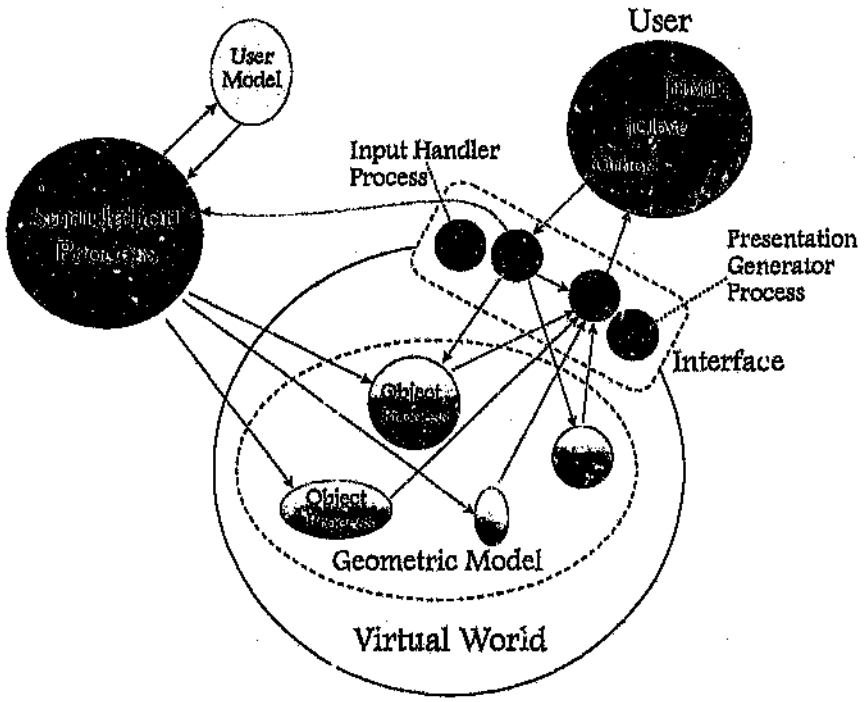


Figure WB104–2: Conceptual Virtual Reality System Model

2.3 Roles and Responsibilities

The user interacts with the virtual world by means of input and output devices. Input devices may include a keyboard, mouse, dataglove or head tracker; output devices might take the form of a display monitor, head-mounted display (HMD) or virtual acoustic display.

User-input which is obtained from the various input devices is dealt with by one or more *Input Handler processes*. An Input Handler process incorporates all the necessary software drivers for interacting with a particular input device, and provides the rest of the system with a high level interface to that device. It is responsible for performing any pre-processing of the input that may be required to extract the semantics of the user's interaction with the system.

Correspondingly, a *Presentation Generator process* is responsible for producing the view which is displayed by each output device[1]. Once more, this process houses all of the low level software that is required to interface with the particular output device. Furthermore, the Presentation Generator process is responsible for performing any viewing transformations that are required to be made to the geometric data that it uses to render the view.

The Presentation Generator and Input Handler processes together comprise the *Interface* between the user and the virtual world.

A number of *Object processes* collectively form the geometric model. The geometric model provides a comprehensive geometric and behavioural representation of the objects which comprise the virtual world. Each Object process is responsible for handling updates to the part of the model that it represents.

Simulation processes are responsible for managing all non-graphical computations. In a general VR system, a Simulation process will typically contain a computational model of some application process and will be responsible for running a continuous simulation of that process. In the case of the project at hand the target application process is an electrical power network model (refer to the Application Description, WB 103).

User modelling is incorporated by means of a *User Model*, which may either be a separate process, or form part of the Simulation process. The

[1] For the purpose of this discussion, it will be assumed that the Presentation Generator renders graphical images to a visual display device, although it should be noted that the output which is generated may equivalently include other modalities such as auditory output (in the case of virtual acoustic displays) or tactile feedback.

User Model is used by the Simulation to tailor the operation and appearance of the system to match the needs or abilities of a particular user. (Although user modelling has been incorporated into the conceptual system model, it has not been included in the initial prototype implementation of the system).

2.4 Message Passing and Information Flow

As indicated in Figure WB104–2 above, there is a fairly complex message passing network in the system. Message passing takes place between system processes for a number of reasons:

In response to received user-input, an Input Handler may initiate changes to the Objects in the geometric model, as well as provide inputs into the simulation. Changes in the user's viewpoint are propagated by an Input Handler directly to any affected Presentation Generator processes.

The objects in the geometric model incrementally communicate net changes in their visual state to each Presentation Generator process. A Presentation Generator uses this data together with information about the user's viewpoint which it has previously received from the Input Handler processes, to render the displayed images.

Simulation processes initiate changes to objects in the geometric model, and may also receive information which is of semantic significance to the simulation from an Input Handler process.

2.4.1 Loops of Information Flow in the System

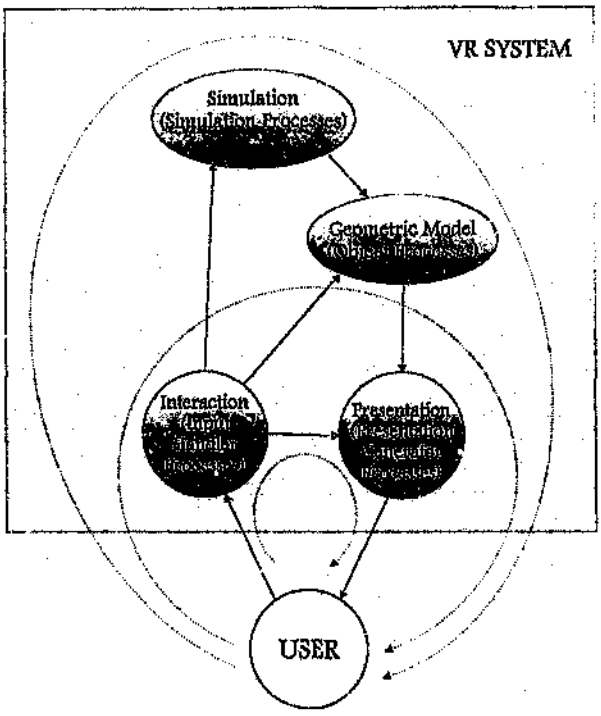
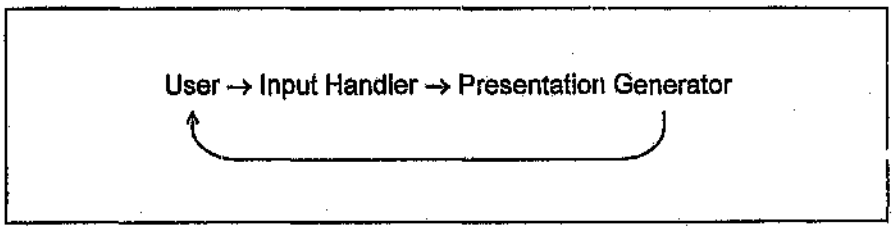


Figure WB104–3: Loops of Information Flow in the System

As indicated in Figure WB104–3, there are three main loops of information flow in the system. For conciseness, these will be diagrammed prototypically, with reference to a single instance of each of the Input Handler, Presentation Generator, Object and Simulation processes.

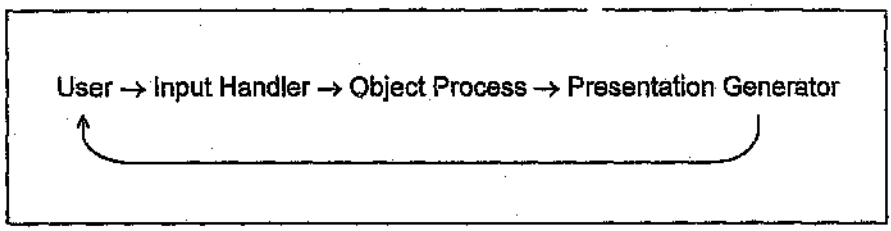
The first loop of information flow is a very tight loop through which information about the user's current viewpoint is transferred:



This allows for a very fast visual update loop as the user navigates through the Virtual Environment. The Presentation Generator uses the latest information that it has about the state of the objects in the geometric

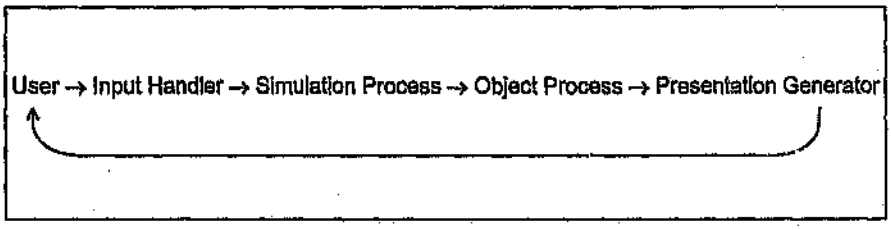
model and performs the necessary viewing transformations, based on the current viewpoint obtained from the Input Handler, to render the view.

The second loop comes into effect when the user interacts with any of the objects in the Virtual Environment:



Here, information flows from the Input Handler process to the affected Object process, which updates its internal geometric and behavioural model accordingly. The object process then furnishes the Presentation Generator with an update regarding its visible state. The Presentation Generator performs the required viewing transformations using the latest known viewpoint in order to render the view. This loop will naturally be slower than the first loop; in addition to the increased message passing overheads, there is now additional computation to be performed by the Object process. The amount of processing performed by Object process will depend on the complexity of the behavioural models that have been defined for the geometric model.

The third and final loop of information flow in the system includes the Simulation process, and comes into play when the user interacts with the simulation:



In response to input received from the Input Handler, the Simulation process makes the necessary changes to the simulation parameters and recomputes the simulation. The Simulation process propagates the results through the rest of the system by effecting changes to the objects in the geometric model. Thereafter, the sequence of events are analogous to the previous case—the objects recompute their state and then update the Presentation Generator, which renders the view.

2.4.2 Open Loop Information Flow

The three main loops of information flow outlined above represent typical patterns of message passing in the system, but are not the only possible sequences. All other possible sequences, however, will take the form of subsets or *open loop* versions of the above loops. Examples of this are where:

- a Simulation process *initiates* changes to the objects in the geometric model, or
- an Object process acts autonomously.

The information flow necessary to realise the first situation consists of the third main loop described above, but with the initial message passing from the Input Handler to the simulation removed:

Simulation Process → Object Process → Presentation Generator → User

Similarly, the second scenario is achieved by the second loop described above, again without intervention from the user:

Object Process → Presentation Generator → User

2.5 Key Characteristics

The principle characteristics of the Conceptual Virtual Reality System Model which has been presented include:

Simulation Decoupling In accordance with the principles of the Decoupled Simulation Model [Shaw et al., 1993], the simulation has been decoupled from the rest of the system. This allows the main visual update loop to proceed independently of any processing done by the simulation. The main motivating factors for this are the maintenance of an acceptable visual update rate and the minimisation of lag.

Navigation Decoupling In addition to decoupling the *simulation* from the user's realtime interaction with the system, the user's *navigation* within the Virtual Environment has furthermore been decoupled from his

manipulation of the objects contained within it. The author proposes that this be referred to by the term "*Navigation Decoupling*". Therefore, the visual update rate during navigation is independent of any computation performed by the Object processes which make up the geometric model.

[Note: The combined effect of Simulation Decoupling and Navigation Decoupling is that the highest possible visual update rate and the lowest possible lag are achieved during navigation—when the user's viewpoint is the only variable that is changing. Fortunately, these are exactly the same circumstances under which the need to maintain a high visual update rate and low interaction lag are the greatest—when the scene observed by the user is changing rapidly. Work by Liang, Shaw and Green [1991] confirms this by demonstrating that the overall perception of lag felt by the user is greatest when head tracker orientation data is delayed—changes in viewpoint orientation during navigation account for the most noticeable changes in the observed scene.

The model which has been developed, therefore, provides the fortuitous circumstances that the highest possible visual update rate is able to be provided exactly when it is required: when the viewer is navigating through the virtual environment and rapidly changing his point of view.

The tasks of navigation and object manipulation are rarely performed concurrently to any large extent. Users will typically manipulate an object only once they have navigated to that object. Object manipulation is therefore typically performed while the display is fairly static. The visual update rate requirements when the display is static are significantly lower, and therefore the additional processing time devoted to object computations can be tolerated.]

Distribution of Processing The model which has been developed facilitates distributed processing by dividing the system into a number of largely independent functional units (processes) which communicate by passing messages to one another. This aspect is dealt with in detail in Section 3.

Device Independence The approach that has been taken to interfacing with particular hardware devices makes the system to a large extent device independent. The specifics of dealing at a low level with a particular input or output device are entirely encapsulated in the respective Input Handler or Presentation Generator process that supports that device. Practically, this means that the system implicitly supports *any* input or output device—including devices that were not available during the

initial implementation of the system. All that is required to add support for a new piece of hardware is to develop the appropriate Interface process (Input Handler or Presentation Generator) to handle that device. The interface presented to the rest of the system will be identical.

3 The Process Agent Model (PAM) for Distributed Processing

The Process Agent Model (PAM) which has been developed, provides a mechanism for the distribution of processing across a number of workstations in a manner that is transparent to the high level system processes. The primary motivation for the use of distributed processing is as a means for improving temporal system performance. *Transparent* distribution of processing implies that the application processes are insulated from having to deal in any way with the details of the underlying network communications.

3.1 Overview and Objectives

In its most generic form, the overall system may fundamentally be regarded as consisting of a number of interacting processes. (Refer to the Conceptual Virtual Reality System Model presented in Section 2 above). These processes communicate with each other by means of asynchronous message passing. Each process is to a large extent self-contained and decoupled from the rest of the system. For this reason, processes recommend themselves as conveniently distributable entities. If the overall processing task is distributed across a number of workstations, then one or more processes will be housed on each machine.

In order for the distribution of processing to be transparent to the individual system processes, it is desirable for the message passing mechanism used between remote machines to be the same as that used between processes which reside on the same machine. Under these conditions, a high level process wishing to communicate with another process, does so in the same way, regardless of whether the target process is local or remote. The process need not be concerned with the details of remote message passing. The Process Agent Model provides a mechanism for achieving this.

3.2 Distributed Processing Considerations

In order to facilitate the distribution of system processing, a network communications infrastructure needs to be put in place to allow processes which reside on different machines to communicate with one another. The message passing requirements from the point of view of the application processes are very different to those of the network. In a standalone system implementation, communication between application processes

typically makes use of some kind of function calling mechanism[2]. Conversely, network communications involves the transfer of *packets* of information, or simple character strings.

Function calls and strings of characters are fundamentally different entities. When a function call is made, the parameters which are passed are placed on the stack, and the thread of execution passes to the beginning of the specified function. In a distributed processing environment (in the absence of shared memory), there is no common stack onto which parameters can be placed. Furthermore, the operating system does not in general provide a mechanism for transferring a thread of execution to a peer process executing on a remote machine. In order to effect message passing between machines, therefore, function calls must be *encoded* into a form which is suitable for transmission over the network.

3.3 Layered Architecture

In order to reconcile the disparate message passing requirements of the application processes and the network, the PAM makes use of a layered architecture (Figure WB104–4), designed to hide the encoding of function calls and the details of the underlying communications from the application processes. A layered approach, in which each layer performs a well-defined function and provides a particular set of services to the layer above, is common in the design of network communications software. According to Halsall [1992], a layered organisation makes such systems more readily testable and maintainable, by separating the responsibilities of each layer. The dVS VR system features a layered architecture, which is designed to hide the distributed software and hardware environment from the developer [du Pont, 1993].

The Process Agent Model distinguishes between three layers: the application processes, the Environment Manager (EM), and the network.

The application processes layer corresponds to the processes which make up the Conceptual Virtual Reality System Model which has been described above. Communication between the application processes is

[2] For reasons that are discussed in the High Level Software Design (WB 105), an object-oriented methodology has been used in the design and implementation of the system at hand, and consequently the system processes have been realised as intercommunicating objects. Therefore, for the purposes of this document, communication between processes is discussed from the point of view of class member function invocation. However, the notion of the use of function calls for the transfer of messages between processes is not limited to the object oriented methodology.

facilitated by the Environment Manager, which in turn utilises the network for the actual transfer of messages. The EM provides the application processes with a high level interface to the functionality provided by network. The application processes interact only with the Environment Manager, without regard for any of the low level implementation detail of the transfer of messages across the network. The Environment Manager is also responsible for performing a number of other tasks, such as system initialisation, resource management and process-to-processor allocation. These tasks are described more fully in Section 4. The network layer is covered in Section 5.

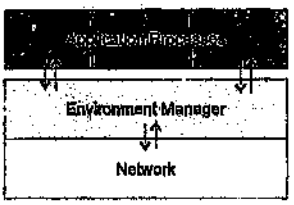


Figure WB104–4: Layered Architecture of the Process Agent Model

A layered architecture alone is not sufficient to make the distribution of processing *transparent* to the application processes. The objective of hiding the details of network communications from the high-level processes is achieved, but a process is still required to communicate with the Environment Manager whenever it wishes to send a message to a remote process. The process must still be *aware* of the fact that messages are being transferred over the network.

For truly transparent process distribution, the application processes must interact with one another as if there were *no network communications* taking place. The processes should be unaware that messages are being transferred across the network at all. In order to achieve this, the concept of an *agent process* is introduced.

3.4 Agents

Each process is unique in the sense that it may by definition exist on only one workstation at any given time. All machines other than the machine that houses the process are considered *remote* from the point of view of that process. For each process, an agent process is created on each remote machine. In other words, either the process itself or an agent for the process exists on each of the networked machines. This situation is

true for all of the processes which make up the system. It may therefore be considered that on each machine, a complete version of the entire system exists: any processes which are absent are represented by agent processes. This is illustrated in Figure WB104–5.

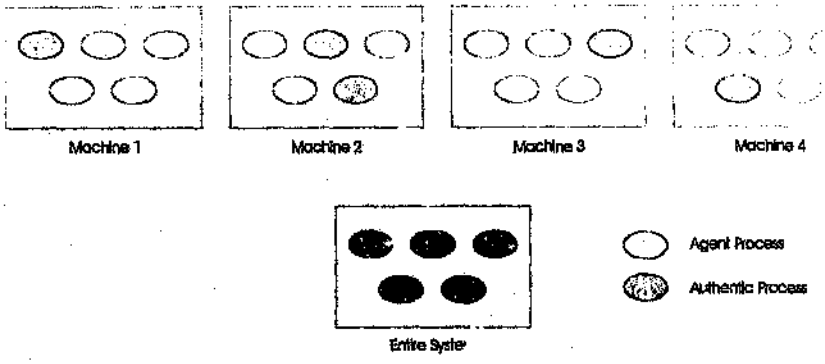


Figure WB104–5: Processes and Agents

An agent process serves as a proxy or representative for the authentic process with which it is associated. It intercepts all messages destined for that process, and routes these messages, by means of the EM and the network, to the machine on which the actual target process exists. The EM on the remote machine in turn forwards these messages to the authentic destination process.

An agent process can be thought of as a "stub" or skeletal version of the authentic process with which it is associated. It is concerned only with guaranteeing the transfer of messages to the associated authentic process, and does not perform any actual application-related computation. As a result, the amount of processing needing to be performed by an agent process and hence its computational resource utilisation are minimal.

The interface presented by an agent process is identical to that of the authentic process. Whether the target process is local or remote is transparent to the calling process—the calling process is unable to distinguish when the target process is local and when it is remote.

The concept of processes and agents is similar to the concept of players and ghosts described by Blau et al. [1992:158].

3.5 Local and Remote Message Passing

A process wishing to send a message to another process simply sends the message to the local agent for the target process. A process acts *as*

its own local agent if the calling process resides on the same machine. This results in the favourable circumstances that no additional overheads are imposed on local message passing.

Where both processes are local, messages are handled by the target process directly—in exactly the same way as they would be handled in the absence of distributed processing. The overheads associated with passing the message down and then back up through the EM and network layers are eliminated.

Where the target process resides on a remote machine, an agent process intercepts the message, and sends it across the network via the EM. Because the interface presented by an agent process is identical to that of the corresponding authentic process, when the calling process sends a message to the local agent, as far as it is concerned, it has sent the message to the authentic target process. Likewise, when the message is received by the target process on the remote machine, that process is not aware that the message did not come from the calling process directly.

3.6 Function Call Encoding and Decoding

In order to be transferred across the network, a function call must be encoded into a form suitable for transmission across the network, and decoded upon arrival.

In order to fully describe a remote function call, the following information needs to be specified: the machine on which the target process resides; the identity of the target process; the function that is to be called; and the parameters that are to be passed to that function. All of this information must be encoded into the data field of the network message in order for the function call to be faithfully reproduced on the destination machine[3].

The encoding of a function call can be regarded as comprising two main tasks. The first of these involves the encoding of the function arguments. Each parameter value that is to be passed to the target function must be represented as a sequence of characters in the body of the network message. Coulouris and Dollimore [1991:86] refer to this process as "argument marshalling". The second task involves the addition of

[3] In general, the *calling* process would also need to be identified, so that the function return value, as well as updates to any parameters that have been passed by reference could be sent back to it. However, for reasons that are discussed in the Low Level Software Design (WB 113, Section 3.4), the current implementation does not cater for return values or for passing of parameters by reference, and therefore the identity of the calling process is not required to be encoded into the message.

successive header information to the message. This header information is used to respectively identify the target process and the target function.

3.6.1 Marshalling of Function Arguments

The use of agent processes provides a convenient mechanism for performing the necessary marshalling of function arguments, and obviates the need for the calling process to even be aware that the function call is being encoded.

When the target process is remote, the calling process is unaware that it is not communicating directly with that process, since the agent process with which it communicates presents it with the same interface as the authentic process. The agent which intercepts the message is responsible for performing the necessary argument marshalling—manipulating the arguments it receives from the calling process into a form suitable for transmission over the network. The calling process is not required to perform any encoding before making the call.

3.6.2 Remote Message Handlers

When the message is received by the Environment Manager on the remote machine, it is forwarded, essentially unmodified, directly to the target process. It is then the responsibility of the target process to perform the necessary decoding of the received message in order for the appropriate function call to be made with the correct parameters.

While it is convenient for each process to perform its own decoding of remotely originating messages in this way, thereby encapsulating this functionality into each individual process class, it is at the same time desirable for the target process to be unaware that the message came from remote process, and to be uninvolved in the decoding process. In order to reconcile these conflicting points of view, it is illustrative to regard each authentic process as containing a "chunk of code" whose purpose is purely to provide it with the ability to receive remotely originating messages—code which does not contribute at all to the application-oriented functionality of the process. It is this "chunk of code" which does the necessary decoding of the function call parameters from the received message, and which then actually calls the appropriate function. This "chunk of code" will be referred to as the *remote message handler* for that process.

While the remote message handler is physically part of the process, it is logically not integral to it. It should be viewed simply as an adjunct that allows the process to receive remote messages, and which is entirely separate from the base functionality of the process.

3.6.3 Message Headers

The addition of header information to a network message is performed by successive layers as a message moves through the architecture. Each layer adds information appropriate to its own level of abstraction. Specifically, the agent process adds the function ID of the function that was called; the Environment Manager adds the process ID of the process from which the message was received; and the network adds the machine ID of the destination machine which has been provided to it by the EM.

It is this final form of the message that is actually sent across the network to the destination machine. The low level network drivers use the machine ID contained in the foremost header to route the message to the correct destination. At the far end, the network layer strips off the machine ID before providing the message to EM. This information is no longer needed since the message has already reached the appropriate machine. The EM strips off the Process ID, which it uses to identify the target process, and then delivers the remainder of the message to the remote message handler of that process. Again once the message reaches the correct process, this information is no longer needed. The remote message handler strips off the last header—the function ID—and uses it to identify which function is to be called. It then decodes the function parameters and calls the appropriate function.

3.7 Graphical Illustration of the PAM

The following diagrams serve to demonstrate graphically the Process Agent Model message passing mechanism. A simple example consisting of three processes distributed over two machines is used for illustration. (See Figure WB104–6). Processes A and B reside on machine 1, while process C runs on machine 2.

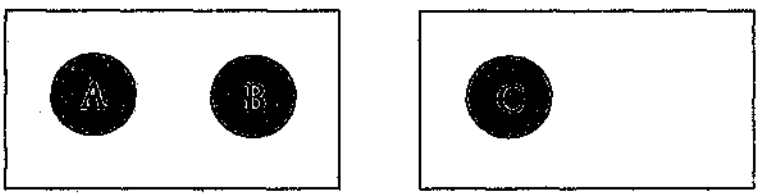


Figure WB104–6: Simple Example System

Figure WB104–7 shows the most trivial case—where the calling process and the target process both reside on the same machine. In the situation depicted, process A wishes to send a message to process B. In this case, the required message passing is entirely local, and takes the form of a

conventional function call invocation. Process A simply invokes one of the functions of B.

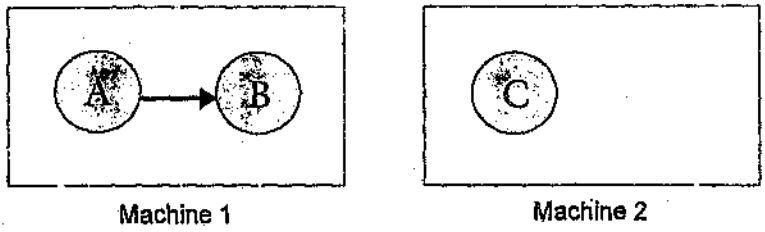


Figure WB104–7: Local Message Passing Mechanism

We now turn our attention to the situation where the target process is remote, such as when process A wishes to send a message to process C. In order to achieve the goal of transparency, it is desirable for the message passing mechanism to appear, to both the calling process and the target process, to be identical to the preceding case. This desired situation is illustrated in Figure WB104–8. If this scenario is realised, then from the point of view of both processes, the message again appears to involve just a simple function call—in this case process A invokes one of th' functions of C. The mechanism whereby the Process Agent Model achieves this in reality is illustrated in Figure WB104–9.

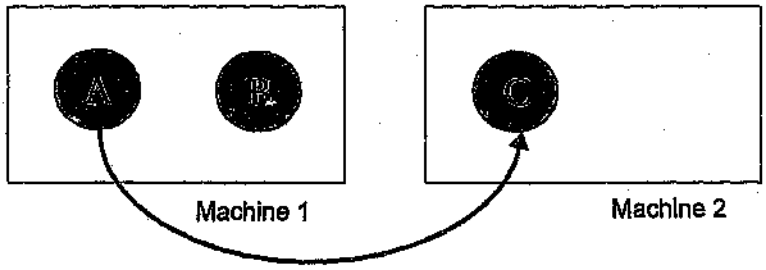


Figure WB104–8: Desired Remote Message Passing Mechanism

Three agent processes, A_{agent}, B_{agent} and C_{agent} are introduced. Each presents the same interface to the calling process as the corresponding authentic process.

When process A wishes to send a message to process C, it simply identifies the local agent for C and calls the appropriate function. Because the interfaces are identical, as far as A is concerned, it has invoked one of the functions of what it perceives to be the authentic process C. In reality, A has invoked the corresponding function of C_{agent}. In this way, the agent process is able to intervene on behalf of the absent process C.

C_{agent} encodes the message and sends it to the Environment Manager. (The details of the precise encoding mechanism that is used can be found in the Low Level Software Design, WB 113, Section 6.3). The Environment Manager maintains a process-to-processor allocation table, from which it determines the location of the actual process C to be machine 2. The EM uses the network to deliver the message to the peer EM on machine 2. Upon receipt of the message, the remote EM delivers it to the remote message handler of the authentic process C. C's remote message handler then decodes the message and invokes the intended function. Once more, the actual function is not aware that it has been invoked by the remote message handler and not by process A directly.

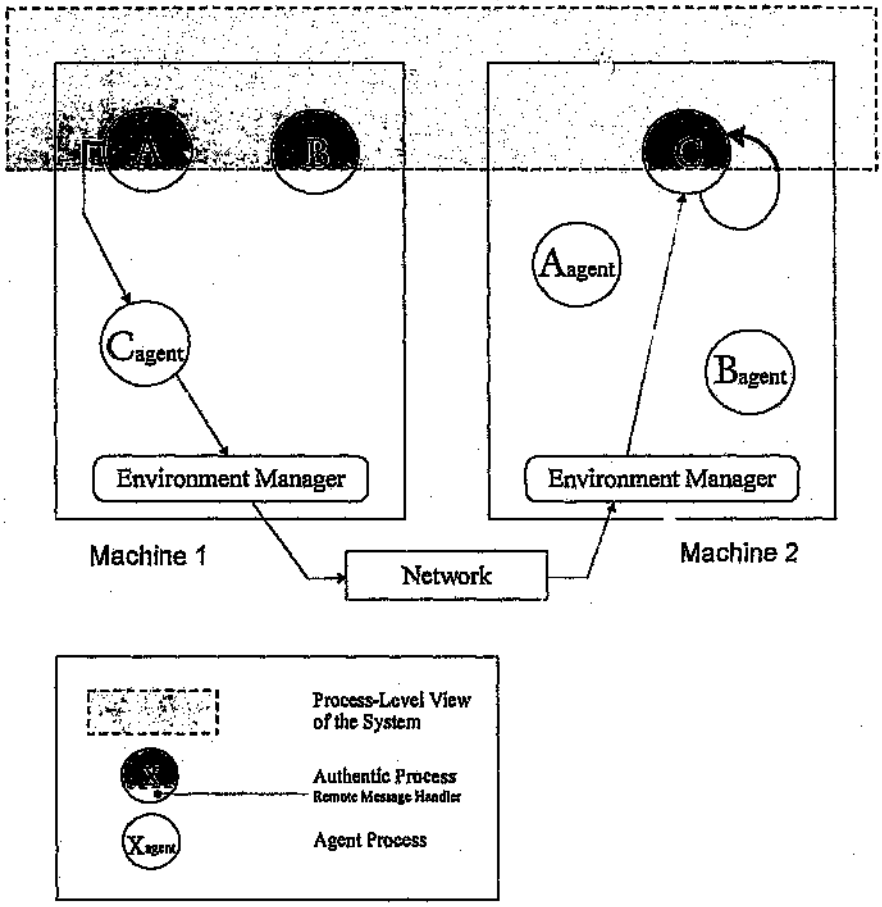


Figure WB104–9: The Process Agent Model Remote Message Passing Mechanism

3.8 Advantages of the Process Agent Model

The following constitute the key advantages of the Process Agent Model:

- The architecture places no additional overheads on local messages. A disadvantage of many distributed processing system architectures, is that *all* messages are handled by a central message handling subsystem. The PAM overcomes this problem by allowing messages to be transferred between local processes without the intervention of the Environment Manager.

  In the limiting case where all system processes run on a single machine, a system making use of the PAM reduces to a standalone version which has practically *no* additional overhead compared to a system that does not support distributed processing at all[4].

- The development of application processes is simplified, since ordinary function calls can be used for inter-process communication, without the application processes needing to be aware of the underlying network communications.

3.9 The Process Agent Model versus Remote Procedure Calls

The Process Agent Model message passing mechanism which has been described above, has features in common with the concept of *Remote Procedure Calls* (RPCs), which have been discussed in the Literature Survey (WB 101), Section 4.2.3. Like the PAM, remote procedure calls allow calls to be made to a procedure which resides on a different computer or in a different address space.

There are, however, some notable disadvantages to the use of RPCs in the context of the project at hand. The main drawbacks of remote procedure calls include the following:

1. **The calling process is suspended until the called procedure returns** [Coulouris and Dollimore, 1991:84]. Shnier [1996] refers to RPC communication as synchronous or blocking—the requester must wait for a response before proceeding. This is the most serious disadvantage of RPCs, since it limits the degree of concurrency of

[4] The addition of a remote message handler to each process does increase the memory footprint of the system. This may have a slight impact on performance. Furthermore, unless the implementation of the system is optimised so that no polling for network messages occurs in the limiting case of a standalone machine, the network polling loop will introduce a further source of overhead. Neither of these factors is particularly significant to the overall performance of the system.

processing that is possible. The calling process is unable to continue processing until it has received a response from the remote machine.

2. **The calling syntax is different.** Typically, the syntax of a remote procedure call is different to that of a local procedure call. Although certain RPC implementations aim to make remote procedure calls as much like local calls as possible [Coulouris and Dollimore, 1991], the use of remote procedure calls is never transparent to the calling process. One approach to the issue of transparency is to use RPCs for *all* message passing in the system. This, however, is not a favourable solution, as is discussed under point 3 below.

3. When RPC is used for all procedure calls, **the overhead for communication between processes on the same machine is high.** In RPC, local communication is treated as an instance of remote communication, and simple local operations are treated in a similar way to complex remote operations. There are therefore significant overheads associated with the use of RPCs for local messaging.

   (This issue has been addressed by Bershad et al. [1990], who have developed the "lightweight remote procedure call" (LRPC). For same-machine communication, LRPC makes use of a "trimmed down" version of the general remote procedure call mechanism. As a result, when used for all inter-process communication, LRPC is able to achieve significantly better overall performance than the use of full-blown RPC)

4 The Environment Manager

As was indicated in Section 3.3, the Environment Manager (EM) forms an intermediary layer between the application processes and the network. The main function of the EM is to provide the application processes and agent processes with a high level interface to the functionality of the network. It is primarily concerned with process-to-processor allocation, message delivery and communications management.

The Environment Manager is responsible for allocating the available processing resources among the various processes. This allocation may be performed automatically at runtime, through determination of the capability of the available machines or may be manually specified based on a priori information about the hardware configuration being used. In addition, process-to-processor allocation may be static in the sense that once decided, is remains the same for the duration of the system's execution, or may be dynamic. Dynamic allocation might include the reallocation of processing tasks to under-utilised resources (load balancing), or catering for machines that become available subsequent to the initial startup of the system.

An agent process wishing to forward a message that it has intercepted to the corresponding remote process simply gives the message to the EM, identifies the destination process, and the rest—including routing of the message to the correct machine and guaranteeing message delivery where appropriate—is done by the EM. At the other end, the message is delivered to the target process by the EM without that process needing to be concerned with the details of network polling and message retrieval.

In terms of communications management, the Environment Manager serves as an intermediary whereby processes obtain access to other processes. This is necessary if the EM is to perform dynamic process-to-processor allocation, since any local reference that a process maintains to another process will no longer be valid when that process is redeployed. Where multithreading is used, synchronisation of this nature also prevents more than one process from obtaining access to the same data simultaneously, which could have undesirable consequences (This aspect is discussed under the heading of Common Tactical Policies, Threads of Execution in section 3.2 of the Low Level Software Design, WB 113).

In addition to these primary responsibilities, the EM also performs various initialisation and setup functions.

4.1.1 Origins and Earlier Work

The term "Environment Manager" has been used previously by Wang, Green and Shaw [1995], as well as by West et al. [1992]. The two systems in question are respectively the MR Toolkit and the AVIARY system, both of which have been discussed in the Literature Survey (WB 101) and previously in this document. The Environment Manager which forms part of the Process Agent Model that has been developed owes not only its name but also many of its characteristics to this earlier work.

In the context of the MR Toolkit, the Environment Manager is a high level tool which runs on top of MR. Its primary objective is to reduce the programming effort involved in constructing networked virtual environments using MR. The MR EM is geared towards distributed multi-user virtual environments where each machine contains a replicated copy of the entire virtual world, and users on different machines interact with one another via the network. It is responsible for application initialisation and replication, network configuration, communication management and concurrency control. It initialises the objects in the virtual world, sets up the various VR hardware devices and manages graphical updates. It also attempts to optimise network bandwidth utilisation. Each user in a multi-user virtual environment runs a local EM which communicates with other EMs across the network. The objects managed by the EM need not be aware that the system is distributed, since all interaction between objects is done via the EM.

The AVIARY system also incorporates a "Virtual Environment Manager" (VEM) as part of its software implementation. The AVIARY system makes use of distributed processing primarily to improve system performance. The AVIARY VEM process is primarily concerned with the allocation of objects to processors, and the maintenance of consistency between these distributed virtual objects.

As was noted in the Literature Survey (WB 101), a distinction needs to be drawn between "multi-user" distributed Virtual Reality and VR systems that make use of distributed processing in order to improve performance. The MR Toolkit is in the former category, while the AVIARY system is in the latter. In this respect, the Environment Manager which forms part of the project at hand has more in common with the AVIARY VEM than the MR EM. However, many of the requirements of these two types of distributed VR are the same, and consequently many of the tasks performed by the MR EM are also managed by the EM in this project.

The placement of the EM within the architecture of the Process Agent Model is different to both the MR and the AVIARY implementations. The MR EM acts as a master process that resides at the highest level; all other system components are subordinate to it. In the AVIARY system, the EM

is merely another system process, and exists at the same level as the application processes. The Process Agent Model places the EM as an intermediate layer between the application processes and the network.

The main difference between the EM in this project and the MR EM is that several of the tasks that are performed by the MR EM have in this system been more appropriately encapsulated in the high level system processes and the network. The tasks of VR device setup and graphical update management are performed by the Input Handler and Presentation Generator processes respectively, and the task of network bandwidth optimisation is performed by the network.

4.1.2 Summary of Responsibilities

The Environment Manager is responsible for performing the following tasks, inter alia:

- **Network initialisation** and the establishment of communications links with all other machines.
- **Machine capability evaluation** on system startup, to determine the strengths and weaknesses of each machine.
- **Process-to-processor allocation**
  Assignment of processes to processors may be based on machine capability or the constraints of physically attached devices. For example, the Input Handler process which supports a particular input device should reside on the machine to which that piece of hardware is attached.
- **System initialisation** including the startup of all system processes and agents.
- **Routing** of messages received from agents to the remote machine which houses the authentic target process.
- **Delivery** of messages received via the network to the appropriate destination process.
- **Guaranteed delivery** of critical messages such as events.
- **Performance Monitoring**.
- Dynamic **load balancing and intelligent re-allocation** of processes based on processor and network utilisation and in the event of the addition or removal of processing resources.

- **Disaster recovery** and object persistence in the event of a machine or process that stops responding.

It should be noted that the initial prototype implementation of the Environment Manager which has been developed in the scope of the project at hand does not provide all of the functionality described above. In particular, automatic machine capability evaluation, performance monitoring, and dynamic load balancing and process re-allocation, are the subject of a separate masters project being undertaken by Michael Polyzos, entitled "A Distributed Virtual Reality Approach for Implementing Advanced Man-Machine Interfaces Using Low Cost Personal Computers".

5 Network Communications Infrastructure

The network communications layer referred to in Section 3.3 is responsible for providing the transport mechanism whereby messages are actually transferred between one machine and another. Its responsibilities include managing the interface between the software and the network communications hardware, providing guaranteed delivery of certain types of messages, and optimising bandwidth utilisation. It shields the Environment Manager from the details of the underlying network-oriented details, including protocols, message segmentation and addressing mechanisms.

The design and implementation of the network layer of the architecture is the subject of a separate masters project being undertaken by George Spanellis, entitled "A Distributed Processing Approach for Implementing Advanced Man-Machine Interfaces". His work includes aspects such as:

- a hybrid protocol in which the transport mechanism used is dependent on the content of the message being transferred. This allows for the use of reliable, managed, connection based protocols for messages whose delivery must be guaranteed, while best-try protocols with lower overheads (such as datagrams) can be used for less critical information such as state updates [Kessler and Hodges, 1996].

- intelligent queueing, whereby messages that have become redundant are intelligently culled before they are transferred across the network.

- a dual queueing mechanism, capable of delivering temporally consistent state update information to the rendering subsystem.

OOVE

High Level Software Design

Technical Product

Version 1.03

Document Status: Approved

Table of Contents

Change History

Configuration Control

Project:	OOVE
Title:	High Level Software Design
Doc. Reference:	WB 105
Created by:	Warren Blumenow
Creation Date:	17 April 1995

Document History

Version	Date	Status	Who	Saved as:
0.01	96/04/17	Draft	WB	\1995_07\TP\LTSURVEY\WB105WW.001
0.02	96/09/12	Draft	WB	\1995_07\TP\LTSURVEY\WB105WW.002
0.03	97/02/03	Draft	WB	\1995_07\TP\LTSURVEY\WB105WW.003
1.00	97/10/24	Approved	WB	\1995_07\TP\LTSURVEY\WB105WW.100
1.01	97/11/18	Approved	WB	\1995_07\TP\LTSURVEY\WB105WW.101
1.02	98/03/04	Approved	WB	\1995_07\TP\LTSURVEY\WB105WW.102
1.03	98/06/12	Approved	WB	\1995_07\TP\LTSURVEY\WB105WW.103

Revision History

Version	Date	Changes
0.01	96/04/17	New document created using WB 002
0.01	96/06/10	Initial Draft of section 2
0.01	96/08/22	Initial Draft of section 3
0.02	96/09/12	Updated section on Object Oriented Design
0.03	97/02/03	Changes to structure of logical architecture section
1.00	97/10/24	Booch notation documented
1.01	97/11/18	Updated section 3.3
1.02	98/03/04	refined sections 2.2, 2.3, and all of 3. Added conclusion
1.03	98/06/12	Miscellaneous changes and final formatting.

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	97/10/19	Approved	WB 525

1 Scope

1.1 Introduction

This document details the high level design of the software, including the software design methodology, and the formal software architecture of the system. Whereas the Conceptual System Design (WB 104) describes the various concepts — techniques, structures, models — which are drawn from the problem domain, and which constitute the designed solution to the problem, the High Level Design indicates how these ideas are mapped into the software domain.

The motivation for the choice of an object-oriented design methodology is discussed, and the principles of object-oriented software design are presented, including a brief discussion of the Booch notation which has been used. The benefits of object orientation in the context of the project at hand are highlighted.

The software architecture of the system is presented, from a logical as well as a physical perspective. The logical architecture focuses on the classes and objects which make up the system, their roles and responsibilities, and their interaction and collaboration in providing the desired system performance. The physical architecture details the allocation of the program source code to files and modules, the compilation dependencies which exist between the various modules, and the allocation of processes to processors.

An endeavour has been made to describe not only the objective high level software structure of the system, but also the strategic design decisions and pertinent issues which have led to the system architecture which is presented.

1.2 Purpose

The High Level Design is intended to serve as a comprehensive description of the high level structure of the software, as well as providing the reader with insight into the design considerations and architectural choices that were involved in arriving at the presented design.

1.3 Applicability

The High Level Design is crucial in understanding the structure of the software, and how the various components are integrated to provide the overall functionality of the system.

1.4 Definitions

1.4.1 Abbreviations

The following abbreviations are used in this document:

API ≡ Applications Programming Interface

EM ≡ Environment Manager

HMD ≡ Head-Mounted Display

MFC ≡ Microsoft Foundation Classes

1.4.2 Terminology

Several issues with regard to terminology are encountered in the documentation, where ambiguity could arise due to the same term being used to describe two different things or where there are alternate meanings for a term. It is hoped that this section will serve to clarify the terminology which has been used, and that the discussion provided here, together with the context in which each term is used in the text, will indicate which meaning is intended in each case.

Where possible, case and font have been used as cues in the text to indicate in which sense a word is being used. In particular, program code fragments and filenames are designated by means of the `Courier` font.

Object The term *object* (lowercase) is used firstly in the terminology of object-oriented programming to mean an instance of a class. Secondly, the term *Object* (capitalised) is used to describe the conceptual abstract notion of a "thing" or entity in the virtual environment with which the user may interact. In this sense, an Object is considered to consist of both a geometric representation, as well as some defined autonomous behaviour. Thirdly, the term `Object` (Courier font) is the name of one of the classes of the system—specifically the class which represents the abovementioned entities in the virtual environment. Therefore the second and third senses of the word can for the most part be interpreted equivalently with impunity, since their meaning is the same, if not the context of their usage. The difference will invariably be indicated by means of the font used.

Application and Process The term *application* has been used to describe the entire program which runs on a particular machine. This will sometimes also be referred to as the OOVE application. The acronym OOVE stands for Object-Oriented Virtual Environment, and refers to the entire software system that has been developed in the context of this

project. This usage of the term application corresponds to the concept that Microsoft Windows and Microsoft Visual C++ refer to by the term *process*. The reason for this change is that the term process has been used in a different sense in this project—see the following discussion for explanation.

Process and Agent The Conceptual Virtual Reality System Model which is presented in the Conceptual System Design (WB 104) partitions the system into a number of interacting components, which have been termed processes. The word *process* has been used in the text in several different ways, but the semantics of its use in each case is the same. (At the operating system level, the term process is often used to refer to an executing instance of a program; this is not the sense in which the term has been used in this project—see also the previous note)

The Process Agent Model, also presented in the Conceptual System Design, introduces the concept of *agents*. Because of their role and responsibilities in the implementation of the system, agents can also be seen as types of processes. For this reason, agents are sometimes referred to as *agent processes*. Depending on the context, the generic term *process* may refer to both processes and agents. Where a distinction needs to be drawn between a process and its respective agent, the process is referred to as the *authentic process* or *actual process*, while the agent is called an *agent process*.

The terms `Process` and `Agent` (Courier font) are used to describe the abstract base classes from which all Processes and Agents are derived. They are sometimes referred to as the `Process` class and `Agent` class.

Instantiation There are two distinct ways in which the terms instance and instantiation (or the verb instantiate) are used in the text. In the first instance, these terms refer to the relationship between an object and its class. In this sense, it is true to say that an object is an instance of a particular class—the class is instantiated in order to create the object.

The second usage of these terms refers to the relationship between a template (parameterized) class and that template class with its formal parameters specified. One may therefore refer to the instantiation of a parameterized class to obtain a class. The class so obtained is an instance of the template class.

In order to create an *object* of a parameterized class, therefore, it is necessary to first create an instance (in the second sense) of the parameterized class by specifying the formal parameters, and then to create the object as an instance (in the first sense) of the resulting class.

1.5 Notes

- The following trademarks and copyrights are hereby recognised, and are not individually acknowledged in the text:

  RenderWare is a registered trademark of Canon Inc.
  The RenderWare software library is copyright Criterion Software Limited 1995.
  Rational Rose is a trademark of Rational Software Corporation

- Throughout the document, the masculine "he" or "his" has been used for conciseness. Wherever encountered, this should be interpreted as referring to "he/she" or "his/her".

1.6 Audience

The audience for this document includes the project manager, individuals involved in further development of the project, members of the SEAL management board, the external examiner, and any other interested parties.

1.7 Applicable Documents

1.7.1 Standards

a. SEAL QMS Document Creation Template, QS 002, Revision 0.02, 3 October 1994

b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 1.00, 3 October 1994

1.8 Assumptions

It is assumed that the reader has read the Conceptual System Design (WB 104), and is familiar with the material contained therein. It is further assumed that when used as a technical reference to the software, this document will be consulted in conjunction with the Low Level Design (WB 113).

2 Object-Oriented Software Design

2.1 Overview

The object-oriented methodology for software development has been used in the design and implementation of the software. The underlying philosophy behind object-orientation is the modelling of a software system as a collection of co-operating objects, where individual objects are instances of a class within a hierarchy of classes [Booch, 1994:19].

Object-oriented software development is fundamentally different from traditional functional or procedural approaches. Procedural programming languages are usually geared towards describing abstract operations, and the division of an overall process into a number of smaller steps. The focus is on top-down structured design and algorithmic decomposition. Conversely, object-oriented development partitions the system according to key abstractions in the problem domain. The focus is on data abstraction and decomposition of the system into entities which model objects in the real world.

There are inherent advantages to the use of the object-oriented paradigm. Object-orientation brings together the concepts of abstraction, encapsulation, modularity and hierarchy within a single conceptual framework, called the Object Model. Booch [1994:20,77] has identified the following significant benefits of the use of object-oriented design and the Object Model:

- **Complexity Management** Object-orientation helps in managing the complexity which is inherent in software systems. In order to master the complexity of a system, it is essential to decompose it into smaller, more manageable parts. This division of a system's state space allows us to understand the system by examining a few parts of it at a time.

- **Stable Intermediate Forms and Risk Reduction** Object-oriented systems are more resilient to change. Because they are designed to evolve incrementally from stable smaller systems, object-oriented systems are better able to evolve over time. This reduces the risk which is normally associated with building and maintaining complex software systems.

- **Software Reuse** An object-oriented approach provides an economy of expression through the reuse of common mechanisms. This results in smaller systems than are yielded by algorithmic decomposition. Smaller systems are more maintainable—there is simply less code to maintain. Reuse can take a number of forms. The most significantly

beneficial form of reuse in object-oriented systems is the exploitation of commonality through inheritance.

- **Maintainability** The facilitation of maintainability is one of the key advantages of using object-oriented techniques. Due to their inherent ability to evolve incrementally over time from stable intermediate forms, and due to the high degree of reuse which is achievable, object-oriented systems are highly maintainable.

- **Natural Conceptualisation** Booch suggests that object-orientation appeals to the workings of human cognition. Object-oriented systems are a natural way of viewing software. They provide, as Martin [1993] puts it, a means for analysing the world in a way that seems natural to human thinking.

Some additional reasons for using object-oriented techniques, which are specific to the particular problem domain being addressed in the work at hand, have been discussed in the Literature Survey (WB 101) and the Conceptual System Design (WB 104). These are recapitulated here:

- **Decoupling** Object-orientation lends itself well to distributed processing by dividing the system into a number of functional units (objects) which are to a large extent decoupled from one another. The processes in a parallel processing system need to be decoupled, so that the volume of information exchanged between them is minimised.

- **Separation of Responsibilities** In a well-designed object-oriented system, each class has clearly defined and delineated responsibilities. Separation of responsibilities allows tasks to be completed to some extent independently of each other, and therefore allows for parallel asynchronous threads of execution, which is critical for effective distribution of processing.

- **Message Passing** The message passing mechanisms that are used for communication between objects have a convenient analogue in a distributed processing system, where message passing takes place between processes running on different processors.

- **Transparent Distributed Processing** The object oriented principles of inheritance, polymorphism and data abstraction can be exploited to achieve transparent distribution of processing through the use of the Process Agent Model (see the Conceptual System Design, WB 104).

- **Conceptual Consistency** Virtual Reality applications present the user with a synthetic world in which various 'objects' are present. These objects may have responsibilities, may exhibit autonomous behaviour, and may be interacted with by the user. Modelling the virtual world as a collection of interacting software objects is a natural and conceptually

pleasing approach to the design of such user interfaces. It is convenient for the structure of the software to reflect the structure of the application domain.

- Consistency between the domains of the real world, the virtual environment and its geometric models, and the software models which are used, contributes to making the system conceptually sound, understandable and extensible. According to Martin [1993], object orientation "reduce[s] the semantic gap between the models we use in computer programming languages ... and the conceptual models we use in thinking about the world".

2.2 Concepts

2.2.1 Objects

2.2.1.1 The Nature of an Object

An object represents a tangible entity that exhibits some well-defined behaviour [Booch, 1994]; a concrete "thing" that exists in time and space. From a software point of view, an object is a self-contained unit of software containing a collection of related procedures and data structures. According to Booch [1994:83], the essential characteristics of an object are its state, behaviour and identity.

The *state* of an object refers to the object's properties as well as the values that those properties have at a particular time. An object's properties are the characteristics that are inherent, distinctive or fundamental to the nature of that object, and are usually static. Conversely, the particular value that a property assumes may change over time.

Objects do not exist in isolation. *Behaviour* refers to the way objects act on other objects and react when acted upon. An object acts on another object by passing messages or invoking operations. The state of an object both results from and affects its behaviour.

An object's *identity* distinguishes it from all other objects. The identity of an object is determined by its existence at a unique location in memory, and is distinct from any name which may be used to reference the object. The identity of an object is unique, and is preserved over the lifetime of the object, though its name may not be. In addition, *aliases* may be used to refer to an object—more than one name may be used to reference the same object.

2.2.1.2 Relationships Among Objects

In order to be useful, objects must collaborate with one another. It is only through the *cooperation* of objects that the desired behaviour of the system is realised. Booch [1994:97] identifies two categories of collaborative relationships among objects, namely links and aggregation.

A **link** is a physical or conceptual connection or association between objects, through which a client object invokes operations or services provided by a supplier object. Therefore, links represent peer-to-peer or client/supplier relationships between objects, and provide a means through which one object may *navigate* to another. Message passing between objects is constrained to occur only across links. A key consideration with respect to links is the question of visibility. In order for a message to be passed from one object to another, the supplier object must be visible to the client object in some way.

An **aggregation** relationship denotes the ability to navigate from a whole to its parts. The parts of the aggregate object are known as attributes. Attributes are *contained by* the parent object and form part of its state. Aggregation provides a mechanism whereby, given a particular object, it is possible to navigate to its attributes. It is through this whole/part hierarchy that objects may be encapsulated as part of other objects.

2.2.2 Classes

2.2.2.1 The Nature of a Class

Classes are abstractions. A class represents a set of objects that share a common structure and a common behaviour [Booch, 1994:103]. A class encompasses all of the characteristics which are common to the group of objects which it represents. It acts as a repository of all the general information that defines that particular type of object [Dettmer, 1995:254]. Objects are *instances* of classes; according to Booch [1994:83], the terms instance and object are interchangeable. Stroustrup [1993] notes that classes represent the fundamental concepts of the application.

Pragmatically, classes provide the programmer with a means for creating new *types* that can be used with the same convenience as the built-in types provided by the programming language.

Stroustrup [1993] identifies two distinct kinds of classes in a software system—those that directly reflect concepts in the application domain, and those that are artefacts of the implementation. The former represent user-level abstractions and generalizations, while the latter may describe entities such as hardware or software resources and data structures. Booch [1994:162] concurs, noting that the identification of classes may involve both *discovery* and *invention*. The process of discovery uncovers

classes which have analogues in the terminology of the problem domain. Classes identified through invention are not necessarily abstractions which are present in the problem domain, but they are useful artefacts in the design or implementation of the system.

Representing a concept or entity in the application domain as a class in the software design of the system is a non-trivial task, and typically requires a significant amount of insight.

2.2.2.2 Relationships Among Classes

As is the case with objects, classes do not exist in isolation, and are related to one another in various ways to form a class structure.

The principle relationships which exist between classes are:

- Association
- Inheritance
- Aggregation
- Using
- Instantiation

Association is the weakest and most general form of relationship between classes. Associations are usually identified in the analysis and initial design phases of system development, and highlight semantic dependencies between classes. These weak associations are refined as the design of the system becomes more mature, and in the final implementation of the system these relationships are typically resolved into either inheritance, aggregation or using relationships [Booch, 1994:108].

Inheritance denotes a generalisation/specialisation relationship between classes, in which one class shares the structure or behaviour of another class. The former class is said to be *derived* from the latter class, and to *inherit* characteristics from it. A class from which another class is derived is termed a superclass, while the derived class is called a subclass.

Through this mechanism of structural and behavioural sharing, inheritance allows the programmer to express commonality among classes. The subclass may augment, redefine or restrict the superclass from which it is derived. Inheritance can be seen as an "is a" relationship, in the sense that if class B is derived from class A, it may be asserted that class B "is a" kind of A. Class B is a specialised kind of the more generalized class A.

The most generalised class in a class structure is called a base class. A subclass of the base class may in turn form a superclass for further derived classes, forming an inheritance hierarchy.

The concept of *polymorphism* is made possible through inheritance. Polymorphism provides a means whereby a single name may denote objects of different classes, but which are related by some common superclass [Booch 1994:72]. An object denoted by this name can therefore respond to a common set of operations in different ways depending on which actual class is being dealt with.

Multiple inheritance is also possible, through which a class may be derived simultaneously from a number of different superclasses. Such a class inherits characteristics from each of its parent classes.

Aggregation denotes a whole/part relationship between classes in which one class *contains* an instance of another class. Containment may be by value or by reference. Physical containment, or containment by value indicates that the existence of the contained class is dependent on the existence of the class of which it forms part. Instantiation of the enclosing class causes the component class to be created; destruction of the enclosing class causes it to be destroyed. Containment by reference allows each of the classes to exist independently of the other, and additionally allows parts to be shared structurally—two different class instances may contain references to the same object.

Using relationships involve one class making use of the services provided by another. The class providing a service is called the supplier, and the class which uses that service is termed the client. A class may use another class in different ways. The used class may appear as part of the *signature* of one or more of the member functions of the using class, or may be instantiated locally in the *implementation* of one of the member functions of the client class.

Instantiation is necessary when making use of genericity—parameterized or generic classes. A parameterized class is a class that serves as a template for other classes. A parameterized class must be instantiated before it can be used; upon instantiation, the template parameters are specified. Parameterized classes are often used for implementing container classes.

2.3 The Object-Oriented Design Process

The Booch method for object-oriented design has been adopted in the design of the system at hand. Booch [1994:230] has identified two features that characterize successful object-oriented software projects:

- strong architectural vision
- an iterative and incremental development life cycle

Strong architectural vision is evident in a system that has conceptual integrity and a sound internal class and object structure. Conceptual

integrity in the system's architecture makes the system understandable, extensible and maintainable, and reduces developmental risk. The key attributes which contribute to the integrity of an architecture are the existence of well defined levels of abstraction, clear separation between interface and implementation, and simplicity through the appropriate exploitation of commonality. It is felt that the project at hand to a large extent exhibits these traits.

Traditional software design approaches that are exclusively top-down or exclusively bottom-up do not lend themselves well to the object-oriented design process. More appropriate is an **iterative and incremental life cycle** in which the architecture of the system and strategic and tactical decisions are successively refined, and the experience gleaned from each iteration is carried forward to the next phase in the analysis and design of the system.

The software development life cycle needs to be structured and well managed, so that the process can be controlled and measured, but must simultaneously provide sufficient freedom for creativity and innovation.

In order to reconcile these patently disparate requirements, Booch's method for object-oriented design is comprised of two collaborative development processes—a micro development process and a macro development process. The micro process is iterative and incremental, and represents the daily analysis and design activities of the software developer. The micro process is flexible and informal, and encourages opportunism, creativity and innovation. The traditional phases of analysis and design are intentionally blurred. The macro process provides a controlling framework within which the micro process repeatedly proceeds. The macro process is formal and well-ordered, and the traditional distinct phases of analysis and design are retained.

2.3.1 The Micro Development Process

Figure WB105–1 summarises the main activities which make up the micro development process.

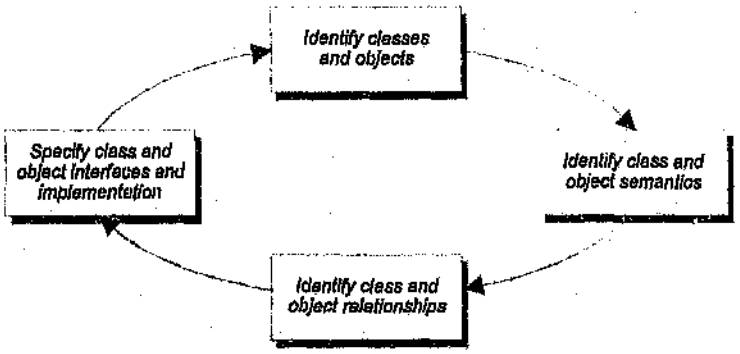


Figure WB105–1: The Micro Development Process

The phases depicted in the figure are as follows:

Identify Classes and Objects Decomposition of the system through the identification of classes and objects serves to define the boundaries of the problem. Through this process, the key abstractions which form the vocabulary of the problem domain are discovered, and new implementation-specific abstractions are invented (see section 2.2.2.1). It is also during this phase that commonality among abstractions is identified, so that it can be utilised advantageously.

Identify Class and Object Semantics Responsibilities are allocated by defining the behaviour and attributes of the classes and objects which have been identified in the previous phase. Apportionment of responsibilities defines and separates the roles of each abstraction, and eventually feeds down into the specification of the interface of each class.

Identify Class and Object Relationships The identification of semantic dependencies between abstractions serves to specify the coaction of objects and collaboration between classes to achieve the desired system functionality. It is during this phase that associations between classes are identified. These associations are eventually resolved into either inheritance, aggregation or using relationships (see section 2.2.2.2).

Specify Class and Object Interfaces and Implementation At this stage, structures and algorithms are devised, which provide the desired properties and behaviours of the abstractions which have been identified. In this way, tangible representations of the abstractions are created, and the abstractions are further refined as a result of the pragmatic consequences of implementation. This activity typically also uncovers new classes and objects at a different level of abstraction, and thus feeds back into the process of identifying classes and objects.

2.3.2 The Macro Development Process

The macro development process serves as the controlling framework for the micro process which was outlined above. Figure WB105–2 summarises the main activities which make up the macro development process as the project proceeds from conceptualisation, through analysis and design, and finally to evolution and post-delivery maintenance.

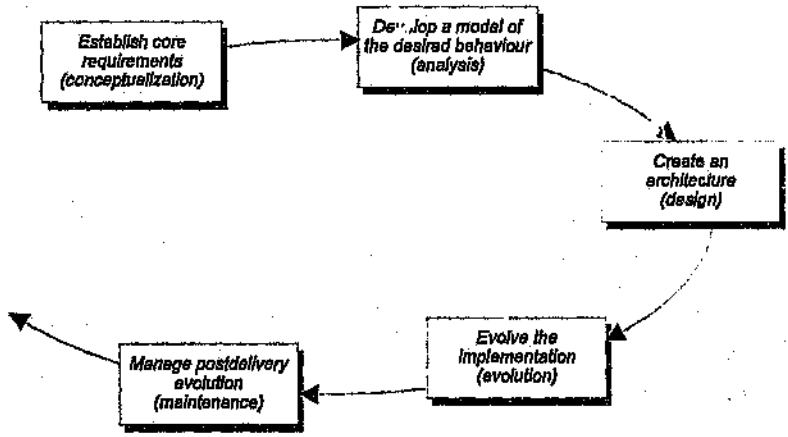


Figure WB105–2: The Macro Development Process

Conceptualisation The first phase in the macro process concerns itself with establishing the core requirements of the system which is to be developed. The main purposes of this phase are to validate the initial assumptions which have been made, and to provide proof of concept for the ideas behind the project. The deliverables of this phase are typically one or more prototypes.

Analysis The purpose of analysis is to provide a model of the system's behaviour. The result of this phase is a comprehensive description of the problem and the functional requirements of the solution. The focus is on the behaviour of the system (what the system does), rather than on its precise form. It is during the analysis phase that the classes and objects that form the vocabulary of the problem domain are identified.

Design It is during design that the architecture of the system and the strategic organisation of the software are formulated. Additionally, the common tactical policies that will be used are decided upon (see section 3 of the Low Level Design, WB 113). The architecture of the system is described diagrammatically—the logical architecture is expressed

primarily through the use of class diagrams and object diagrams, while the physical architecture is conveyed by means of module diagrams. The system architecture is presented in section 3 of this document.

Evolution The evolution phase is essentially the process of product development. The evolutionary development of the system is characterised by successive refinement of the system implementation. The evolving architecture endeavours to simultaneously satisfy various constraints: functionality, performance and available human and physical resources. The products of this phase are incremental executable releases of the of the software.

Maintenance The last phase in the macro development process involves continued evolution of the system after it has been delivered. Catalysts for such activity include augmentation of the system requirements and discovery of latent defects in the system. The former situation calls for the addition of new functionality or modification of existing system behaviour; the latter requires the formulation of bug fixes.

2.4 Context of the Project Within the Object-Oriented Design Process

Booch [1994:249] notes that the macro process invariably repeats itself after major product releases. One of the goals of the project at hand has been to deliver the first "major product release", and as such, the project can be viewed as having gone through one complete cycle of Booch's macro development process.

It is expected that the macro development process cycle will, as Booch suggests, go through subsequent iterations during the course of further development of the project in the future by other researchers within the Software Engineering Applications Laboratory.

While it *is* meaningful to regard the project as having been through the entire macro development process, it is important to note that Booch's design process is by its nature fairly recursive. The project at hand is a fully fledged object-oriented system in its own right, and *has* therefore proceeded through the various phases of conceptualisation, analysis, design and evolution. On the other hand, the software is essentially a prototype system. Prototypes provide proof of concept, and are in Booch's process, the outputs of the conceptualisation phase. Therefore the entire project can equivalently be regarded as forming part of the conceptualisation phase of a much larger "meta" macro development process, the endpoint of which will be a final commissioned version of the system.

Booch [1994:250] asserts that "prototypes are by their very nature incomplete and marginally engineered". Given its status as a prototype,

therefore, it should be realised that certain aspects of the system have not been implemented beyond a rudimentary level and that simplifying assumptions have been made, in order to achieve the overriding objective of providing proof of concept.

2.5 Booch Notation

The Booch notation has been used for this project. Booch [1994:171] notes that a well-defined and expressive notation is important to the process of software development. The use of a standard notation allows the designer of a software system to unambiguously communicate to others the design decisions which have been taken in formulating the architecture of the system. A good notation relieves the mind of the unnecessary work of expressing conceptual design structures, and allows the designer to concentrate more fully on the task at hand. In addition, an expressive notation allows for the use of automated tools for consistency and correctness checking, such as the *Rational Rose®* tool by Rational Software which has been used in this project.

In order to capture all the salient details of the software system being described, a number of different views of the system are required. Booch's notation provides facilities for describing both the logical and the physical architecture of the system. The logical view of the system is provided by means of class diagrams and object diagrams; the physical view of the system is described through the use of module diagrams and process diagrams. In addition, the notation provides mechanisms for describing the time-varying behaviour and dynamic semantics of the system, by means of interaction diagrams and state transition diagrams. The graphical notation which is used in each of these diagrams is described in the following sections.

2.5.1 Class Diagrams

Class diagrams show the existence of, and the relationships between the classes which make up the system. The notation used for class diagrams is summarised by Figure WB105–3.

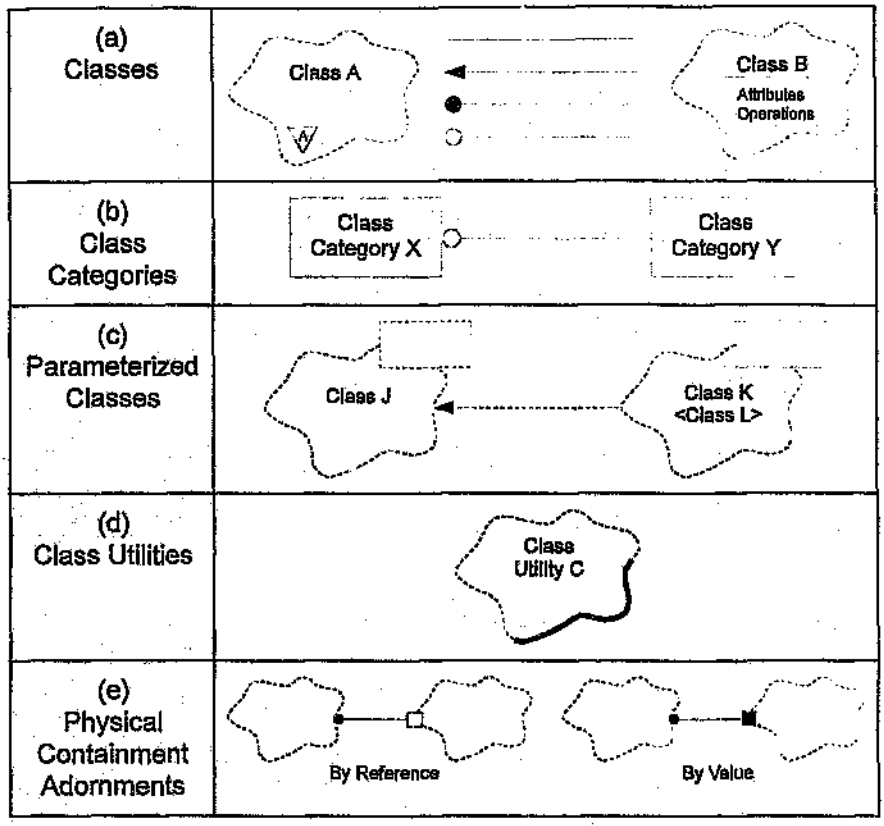


Figure WB105–3: Class Diagram Notation

As shown in Figure WB105–3(a), classes are represented in the notation by means of an amorphous blob or cloud icon—indicating that while the boundaries of the abstraction represented by a class are crisp and clearly defined, they may not always be neat or simple. The class icon has a broken outline, indicating that the class must be instantiated before it can be used meaningfully. The name of the class is written inside the icon. Selected attributes and operations may be shown below the class name, beneath a separating line. Abstract classes are indicated by means of an "▽" adornment.

The first four types of relationships between classes that were discussed in section 2.2.2.2 are indicated in a class diagram by means of the four variations of connecting lines between classes that are indicated in Figure WB105–3(a). From top to bottom, these are: association, inheritance, aggregation and using relationships—class B is respectively associated with, derived from, part of, and used by class A. Adornments to the containment (aggregation) relationship icon may be used to show the

nature of physical containment which is used—containment by value or by reference (Figure WB105–3(e)).

Class categories (see section 3.2.1.1) are depicted as rectangles, as shown in Figure WB105–3(b). The only relationship which is shown between class categories is the *using* relationship. The presence of this relationship implies that classes which are part of the client class category may inherit from, contain and use the classes which are part of the supplier class category. Using relationships between class categories are indicated by the same connecting line as is used between classes.

Parameterized classes are shown with a dashed rectangle adornment, denoting the class' formal parameters. An instantiated parameterized class is depicted using a *solid* rectangle adornment, denoting the instantiated class' actual parameters, which are positionally matched to the formal parameters during instantiation. The instantiation relationship is indicated by a dashed arrow from the instantiated class to the parameterized class from which it was instantiated.

Class utilities are indicated by a cloud icon with a grey shadow at the lower right edge.

2.5.2 Object Diagrams

Object diagrams show the existence of, and the relationships between objects (class instances). The notation used for object diagrams is shown in Figure WB105–4.

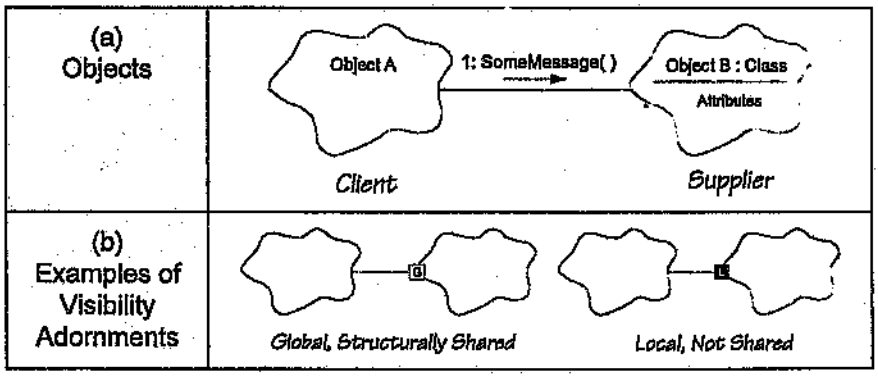


Figure WB105–4: Object Diagram Notation

To represent objects, the amorphous blob, or cloud icon is again used. However, the boundary of the icon is in this case solid, indicating that the

object is a concrete entity. The object name is written inside the icon. The object's class may be specified, prepended by a colon. Selected attributes may be shown beneath a separating line.

Links which exist between objects are indicated by a solid connecting line. Operation invocation or message passing is represented by a directed arrow, with the operation name and a sequence number for the message alongside it. Links may be adorned to show the visibility of one object to another. Examples of this are shown in Figure WB105–4(b).

2.5.3 Interaction Diagrams

Interaction diagrams are essentially simply an alternative way of representing the information contained in object diagrams [Booch, 1994]. Nonetheless, object diagrams and interaction diagrams are both useful in their own right. An interaction diagram to a large extent duplicates the semantics of the corresponding object diagram, but omits some types of information such as attribute values and visibility. Interaction diagrams, however, provide a convenient mechanism for tracing the execution of a scenario, making it easier to follow the sequence of message passing in relative order.

The graphical notation used for interaction diagrams is depicted in Figure WB105–5.

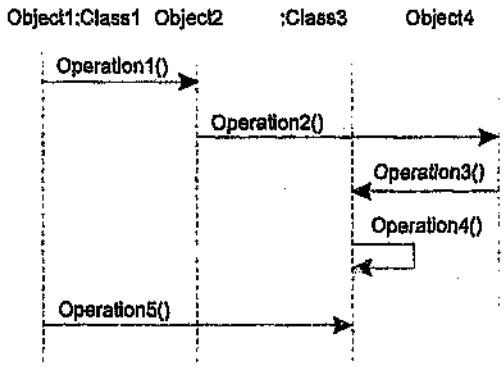


Figure WB105–5: Interaction Diagram Notation

The objects of interest appear at the top of the diagram, and may be indicated either by object name, object class or both. Objects for which the object name is not specified represent anonymous instances of the specified class. Below each object, a vertical broken line extends down the diagram. Message passing between objects is indicated by means of arrows drawn between these dotted lines, labelled with the name of the

operation being performed. Message sequence is indicated by vertical position, with the first message appearing at the top of the diagram.

2.5.4 Module Diagrams

Module diagrams show the allocation of classes to modules in the physical implementation of the system. A module represents a file which contains a portion of the implementation code of the system. The notation used for module diagrams is depicted in Figure WB105–6.

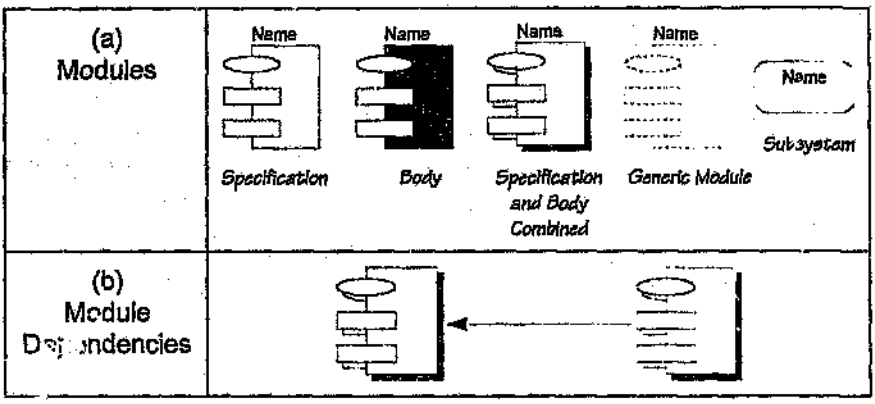


Figure WB105–6: Module Diagram Notation

Figure WB105–6(a) shows the icons which are used to depict various types of modules. A specification module contains class declarations, while a body module contains class definitions. In C++, these two module types correspond to `.h` and `.cpp` files, respectively. Since the two parts are very closely related, modules are often depicted with the specification and body grouped together to form a *package*. The icon which is used for packages is shown in the third diagram in Figure WB105–6(a). Parameterized classes are allocated to *generic packages*.

The icon for a subsystem is shown on the far right of Figure WB105–6(a).

The only relationship between modules which is depicted is a compilation dependency, as shown in Figure WB105–6(b). Such a dependency may exist between any of the module types in Figure WB105–6(a). For example, a package may depend on a subsystem.

2.5.5 Process Diagrams

Process diagrams are used to show the allocation of processes to processors and to indicate device utilisation in the physical implementation of the system. The notation used is depicted in Figure WB105–7.

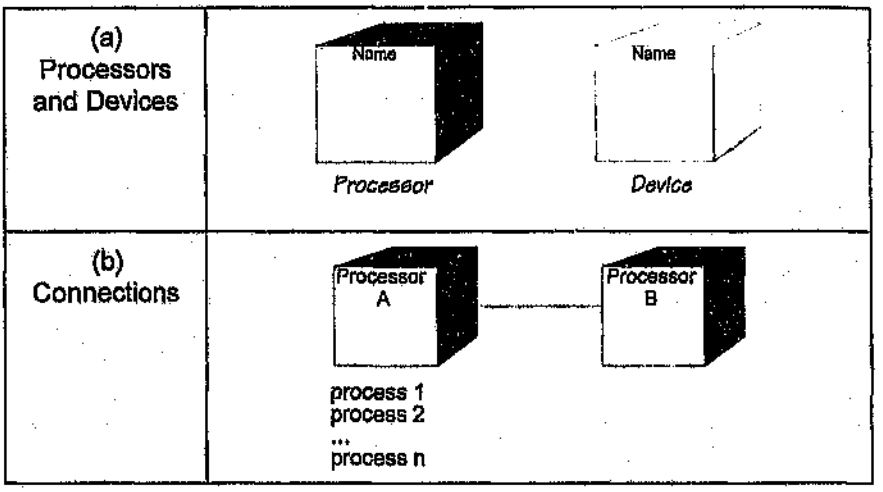


Figure WB105–7: Process Diagram Notation

Figure WB105–7(a) shows the icons that are used to represent processors and devices, respectively. The distinction between the two lies in Booch's definition of a device [1994:224] as "a piece of hardware incapable of executing programs".

Interconnections between processors and devices are indicated by a line connecting their respective icons, as shown in Figure WB105–7(b). The processes assigned to a particular processor are listed beneath the icon representing that process.

3 System Architecture

This section describes the architecture of the system, from both a logical and a physical point of view. The classes and objects which make up the system and the interaction between them to produce the desired behaviour forms the logical architecture. The physical implementation of the system in terms of where particular classes and objects are declared in practice and the dependencies which exist between them, forms the physical architecture. The Booch notation, detailed in Section 2.5, is used throughout. `This typeface` is used to denote source code fragments, class names, module names and filenames.

3.1 Contributory Source Code

Some of the source code which forms part of the software product which is described in this document, was not written entirely by the author. These contributions are discussed in this section. Apart from those components mentioned here, all of the remaining system code has been developed by the author.

3.1.1 Network Classes

The classes which provide the network communications infrastructure for the program were developed by George Spanellis, as part of the 1995_11 DPVR project within the Software Engineering Applications Laboratory. The version of the Network Classes source code which has been incorporated into the project at hand consists of:

- the ReadStruct structure (user-defined composite type)
- the CMessage class
- the CNetwork class

Care has been taken in the design of the system to ensure that there is a clear interface between the network classes and the rest of the system. The development of the remainder of the system was therefore able to proceed entirely independently of that of the network classes. The network is viewed as a black box which provides a particular range of functionality to the rest of the system. The implementation details of the network, including the queueing mechanism and protocol, are entirely encapsulated in the network classes, and are transparent to and thus independent of the rest of the system.

3.1.2 RenderWare Library

The RenderWare graphics library from Criterion Software has been used for the three dimensional modelling and rendering aspects of the project. RenderWare is a commercial, cross-platform, three dimensional graphics rendering library. The RenderWare library provides a third party Applications Programming Interface (API) which facilitates the rapid rendering of complex three dimensional graphics. The product has support for multiple platforms, providing a measure of platform independence and portability to the system implementation.

Source code which refers to the RenderWare library can be identified as follows: All RenderWare API function calls and data type declarations are prefixed with "`Rw`"; for example, `RwRenderScene()` or `RwClump*`, respectively. All RenderWare enumerated types are prefixed with "`rw`"; for example, `rwPRECONCAT`.

In this document, the RenderWare library has been identified as a class utility—a collection of free subprograms or nonmember functions. The RenderWare API takes the form of a number of *functions* which are used to control the various scene maintenance and rendering tasks which the library is able to perform. It should be noted that RenderWare *does* internally make use of hierarchical modelling and application specific types—ideas which are object-oriented in nature. The interface to the supported functionality, however, has been provided by means of standard nonmember function calls.

3.1.3 Linked List Template Class (`GenList`)

The `GenList` linked list class which has been used, was developed by Stephen Levitt, as part of the 1995_10 MSR project within the Software Engineering Applications Laboratory. This parameterized class is a very generic container class, which provides the functionality of a doubly linked list. It has been used entirely without modification. The incorporation of this class into the design of the system is a good example of software reuse, one of the key benefits of object-orientation (see Section 2.1).

By virtue of its parameterized implementation, the `GenList` class is able to contain elements of any specified class or type. In the project at hand, instantiations of the linked list have been used to house strings (objects of the MFC CString class), as well as pointers to high level agent processes—for example, a linked list of pointers to `InputHandlerAgent` objects (see Section 3.2.1.3.1). In this latter case, the parameter which is used to instantiate the linked list class is "`InputHandlerAgent*`".

The `GenList` class provides familiar linked list functionality—the ability to obtain the current list size; add elements to the list; remove elements from

the list; navigate to the previous or next element, or to the beginning of the list.

3.1.4 Debug Classes

The debug classes (refer to Section 3.2.1.5) were jointly developed by the author and George Spanellis. The classes in question are the CDebugger and CDebugDlg classes. These classes together allow the program to output debug messages during program execution to a separate debugging window. This is useful during development for tracking various aspects of the program's execution.

The ability to display debug messages is of particular importance in an application such as this, which is event-driven, multi-threaded and distributed across a group of networked machines. The program code does not by any means execute sequentially, and events may be triggered by messages which arrive from another machine. A common debug window to which the various parts of the software can log messages is extremely useful in tracing the execution of the program.

3.2 Logical Architecture

The logical architecture encompasses the key abstractions and mechanisms which are present in the system. The key abstractions are the classes and objects that form part of the vocabulary of the problem domain. Mechanisms are the means whereby objects collaborate to provide the behaviour that satisfies the requirements of the problem [Booch, 1994:162,164].

3.2.1 Class Structure

In this section, the classes that make up the system are identified, the roles and responsibilities of each class are documented, and the relationships between classes are detailed. The design decisions which were taken in formulating the class structure that is presented are conveyed.

3.2.1.1 Class Categories

At the highest level, the system can be partitioned into class categories—each class category represents a collection of classes which can be grouped together and viewed as a logical unit. The classes in a particular class category are typically fairly closely coupled; the coupling between class categories is loose.

Figure WB105–8, the top level class diagram of the system, shows the class categories which have been identified. This diagram represents the high level architecture of the system and highlights the main layers into which the software has been partitioned. The three layers which may be distinguished are the *High Level Process Classes*, the *Environment Manager Classes* and the *Network Classes*. In addition, the *RenderWare* class utility, the *Debug Classes* and the *Timer Classes* have been identified as top-level class categories.

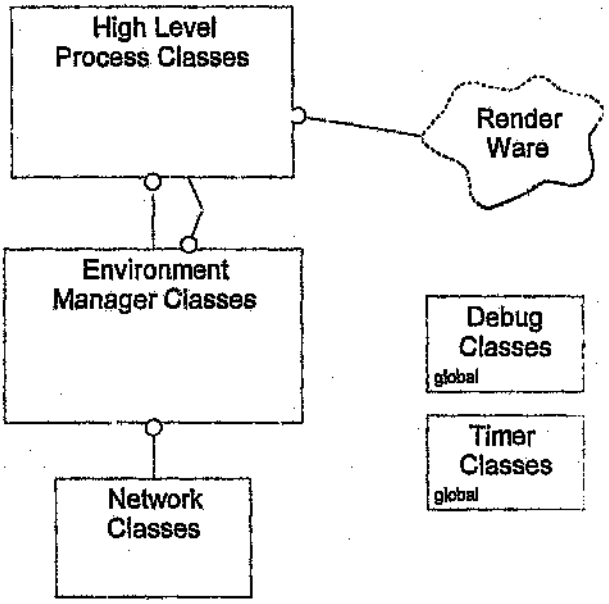


Figure WB105–8: Top Level Class Diagram of the System

The layered architecture which has been used serves to insulate the higher level layers from the intricacies of the lower layers. Specifically, the details of the underlying communications are hidden from the classes in the layers above. Communication between the High Level Processes is facilitated by the Environment Manager, which in turn utilises the Network layer to transfer messages between machines over the network. This layered architecture, not coincidentally, mirrors the layered architecture of the Process Agent Model, which forms an integral part of the conceptual design of the system (refer to WB 104, the Conceptual System Design).

The High Level Process Classes both use and are used by the Environment Manager. As will be seen below, on the one hand, it is the responsibility of the Environment Manager to manage the high level

processes that exist on a particular machine, including the instantiation of those processes; on the other hand, the high level processes use the Environment Manager to obtain the identities of other processes with which they wish to communicate. Hence there is a bidirectional using relationship between these two class categories.

The Environment Manager Classes in turn use the network for the transfer of messages over the network. Because the receipt of messages is initiated by the Environment Manager through a polling mechanism, the usage relationship is unidirectional. The polling mechanism and message retrieval details are fully discussed in the Low Level Software Design (WB 113, Section 6.12.2).

The high level process classes (specifically the Presentation Generator and Object classes) make use of the RenderWare class utility, to perform the necessary graphical model manipulation and rendering tasks.

The Timer and Debug class categories provide the functionality necessary for temporal performance logging and debugging, respectively. These services are extensively used by all of the other class categories, and are therefore designated as global.

The following sections discuss each class category in some detail. In each case, the class structure is elucidated by means of one or more class diagrams, together with an objective discussion of the relationships between the depicted classes, and how the classes collaborate to produce the desired system behaviour. The responsibilities of each class are discussed, from a high level point of view. The low level implementation details of each class may be found in the Low Level Design (WB 113), and are intentionally omitted here. Finally, for each class category, consideration is given to the important strategic design considerations that were involved in formulating the class structure that is presented.

3.2.1.2 High Level Process Classes

The various processes which comprise the Conceptual Virtual Reality System Model that is presented in the Conceptual System Design (WB 104), are realised in the software architecture of the system by means of the high level process classes. Therefore, this class category encompasses the principal functional units of the Virtual Reality system from a high level point of view.

Furthermore, the agent processes necessary for the implementation of the Process Agent Model (also discussed in WB 104) are also represented in this class category, since there is necessarily a close coupling between a particular process and its corresponding agent.

3.2.1.2.1 Class Structure

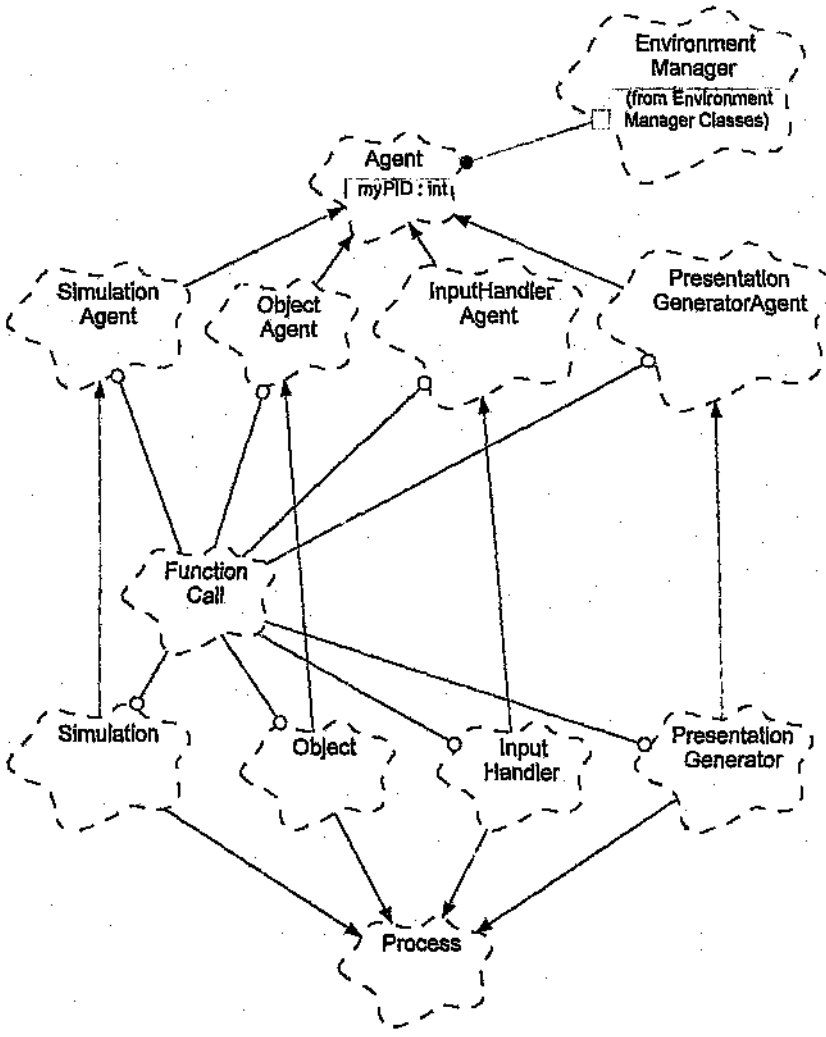


Figure WB105–9: High Level Processes Class Diagram

Figure WB105–9 depicts, inter alia, the main inheritance lattice of the high level process classes. The four main process classes that have been identified are the `Simulation`, `Object`, `InputHandler` and `PresentationGenerator` classes. Each process class is *multiply derived* from its corresponding agent class: `SimulationAgent`, `ObjectAgent`, `InputHandlerAgent` or `PresentationGenerator-Agent`, and from the base class `Process`. Each of the agent classes are in turn derived from the base class `Agent`.

There is a single instance of the `EnvironmentManager` class on each machine. Each agent and process class inherits a reference to the local `EnvironmentManager` object from the `Agent` base class.

All of the agent classes as well as the process classes use the class `FunctionCall` in order, respectively, to encode and decode function calls which are transferred between processes which reside on different machines. Note though, that as discussed in the Conceptual System Design (WB 104, Section 3.6.2), it is *only* the remote message handler of each process class that interacts with the `FunctionCall` class. This is key to preserving the notion of insulating the process class from the details of network communications and message decoding. (The remote message handler of each process class is realised by means of the `RecvMessage()` member function, as discussed in the Low Level Software Design, WB 113).

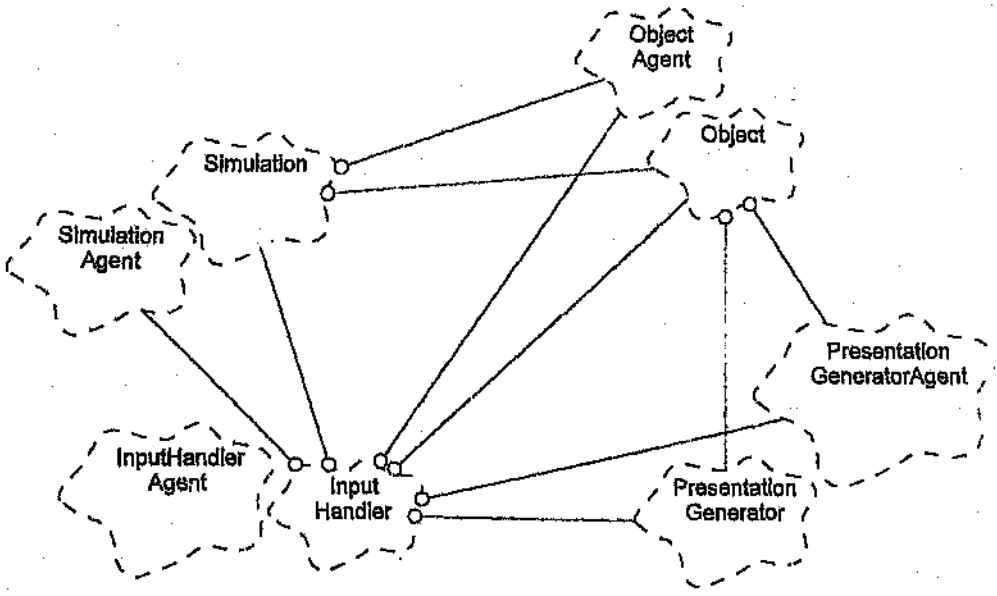


Figure WB105–10: Process Class Using Relationships

Figure WB105–10 depicts the using relationships which exist between the high level process classes. In order to place this class diagram in context, the Conceptual Virtual Reality System Model presented in WB 104 should be consulted, since these using relationships correspond to the paths of information flow which are present in the conceptual system model. A process class may use another process class either directly, or indirectly,

by means of the latter process's agent class. Interaction between processes is direct in the case of both processes being resident on the same machine. Where the supplier process is remote, the client process interacts instead with the local agent process.

As indicated by the diagram, the `InputHandler` class uses the `Simulation`, `Object` and `PresentationGenerator` classes and their respective agents; the `Simulation` class uses the `Object` class and its agent; and the `Object` class uses the `PresentationGenerator` class and its agent.

Although using relationships between the `InputHandler` class and the `Simulation` and `Object` classes are shown in the diagram, it should be noted that in the prototype implementation which has been developed, the `InputHandler` uses only the `PresentationGenerator`. Direct manipulation of objects by the user and manipulation of the simulation through user input have not been implemented in the scope of the project at hand.

3.2.1.2.2 Class Responsibilities

`Agent`

This abstract class is the base class from which each of the *agent* classes is derived. In the current implementation, four such agent classes are defined, namely `InputHandlerAgent`, `ObjectAgent`, `PresentationGeneratorAgent` and `SimulationAgent`. Each of these agent classes in turn forms one of the base classes from which the corresponding process class (`InputHandler`, `Object`, `PresentationGenerator` and `Simulation`) is derived. The `Agent` class gives the classes derived from it a unique identity in the form of a process ID, which is common between a process and its corresponding agents. In addition, the `Agent` class keeps track of the identity of the local Environment Manager, so that all of the agent and process classes contain this information. A process uses the Environment Manager to ascertain the identity of other processes with which it wishes to communicate; an agent uses the EM to send messages over the network to a remote process.

`Process`

This abstract class is one of the base classes from which each of the process classes is derived. In the current implementation, four such process classes are defined, namely the `InputHandler`, `Object`, `PresentationGenerator` and `Simulation` classes. Each process class is multiply derived from its corresponding agent class and the `Process` class. The main function of the `Process` class is to define the

interface whereby a process handles messages relayed to it by the Environment Manager (which originate on a remote machine).

`FunctionCall`

The `FunctionCall` class encapsulates the concept of a function call. Typically this takes the form of a member function of one object being called by another object. This class is responsible for maintaining all the necessary details which define a particular function call, including which function is being called, as well as the parameters and their types. Additionally, the class must perform the necessary encoding of the function call into a form suitable for transmission over the network, and decoding of this information at the other end. Coulouris and Dollimore [1991:86] refer to this process as *marshalling of the function call arguments.*

The intended usage of the `FunctionCall` class is as follows:

1. An agent process intercepting a function call destined for a remote process constructs an empty `FunctionCall` object (which we will call `fn`) to represent that function call, and assigns it a function identifier which is unique within the scope of the agent/process pair.
2. The agent then instructs `fn` to add each of the parameters in turn to its internal representation of the function call.
3. The agent then requests that `fn` marshal the complete function call, and the marshalled version which is returned is then sent to the EM by the agent.
4. At the other end, the received message is relayed by the EM to the remote message handler of the destination authentic process.
5. The remote message handler constructs a new `FunctionCall` object using the received string. (The `FunctionCall` object automatically decodes the marshalled information from the string)
6. The remote message handler invokes the appropriate function call of the authentic process using the decoded parameters which it queries from the `FunctionCall`.

`Simulation`

The `Simulation` class is the first of the four main process classes of the system. It is responsible for managing all non-graphical computation. That is to say, the `Simulation` class houses any underlying application-level processing tasks that are required to be performed by the program.

In the prototype system that has been developed, the `Simulation` class incorporates a continuously running simulation of what the output from a simple SCADA system might be. It thus contains a simple computational model, the output of which is a series of sampled parameter values at discrete time intervals. In a full scale implementation of the system, one or

more Simulation processes would be responsible for delivering a live external data feed. (For a discussion of SCADA systems, please refer to WB 103)

Much of the functionality of the class comprises the main simulation thread; the remaining functionality of the class plays a role only in supporting the running of the simulation.

`Object`

The `Object` class forms the second of the four main process classes of the system. It represents the concept of a geometric *object* or *entity* in the virtual environment with which the user may interact. The class contains a geometric representation of the object, and each object is responsible for handling updates to the part of the overall geometric model that it represents. These updates may be initiated by either the `InputHandler` (discussed below) or by the `Simulation`. Examples of possible updates to a particular object are scaling, translation, or rotation.

An `Object` also performs any autonomous behaviour which has been defined for it. Where an object acts autonomously, updates to the state of the object are initiated by the object itself.

Each `Object` is additionally responsible for furnishing the `PresentationGenerator` with updates regarding its state, so that the images that are rendered accurately reflect the latest state of the model.

`InputHandler`

The `InputHandler` class forms the third of the four main process classes of the system. The `InputHandler` is responsible for dealing with all user inputs appropriately. In the current implementation, this includes keyboard as well as mouse input. In a future full scale incarnation of the system, an `InputHandler` would also be used to handle input originating from other devices, such as a dataglove or the head tracker of a head mounted display. In the case of a dataglove, the additional inputs needing to be dealt with would include position and orientation information, as well as information regarding the shape of the hand. Input from a head tracker would comprise position and orientation information.

Additional device-specific Input Handlers would be implemented as classes derived from the base `InputHandler` class. It is in this way that the system caters for the use of multiple input devices—all of the functionality required to handle a particular device is contained in the corresponding `InputHandler` class which has been developed to support that device.

Shaw et al. [1995] distinguish between two levels of user interface design, namely semantic design and syntactic design. In the case of user input, semantic design refers to the *meaning* behind the user's actions—i.e. what he intended when he interacted with the system. Conversely, syntactic design refers to the particular *form* that the actual command took. The `InputHandler`'s primary task, therefore, is to perform the necessary syntactic pre-processing of the raw user-input stream, extracting the semantics of the user's interaction with the system. The appropriate operations are then invoked accordingly.

An example is illustrative. Suppose the user moves the mouse horizontally while depressing the left mouse button. In this case the syntactic input comprises a particular mouse movement—specified in terms of the number of pixels between the startpoint and the endpoint—as well as the time at which the mouse button was depressed, and the time at which it was released. The semantics of the user's interaction with the system, however, are that he wished to pan the camera, or adjust his viewpoint in a horizontal direction by a certain amount. The `InputHandler` must convert the former description into the latter. In this case, the fact that the mouse button was depressed is interpreted as a command to pan the camera, and the distance the mouse was moved while the button was down represents the degree of pan. Therefore the `InputHandler` will instruct the `PresentationGenerator` to pan the view by the specified amount.

Apart from semantic interpretation, the Input Handler may perform additional processing on the inputs that it receives. This may include filtering to remove noise, aggregation of inputs to extract the gestalt of the intended action, thereby reducing the frequency of discrete events having to be dealt with by the system, or predictive filtering [Liang, Shaw and Green, 1991] to reduce the lag perceived by the user.

`PresentationGenerator`

The `PresentationGenerator` class forms the last of the four main process classes of the system. The `PresentationGenerator`'s main responsibility is the rendering of the output which is presented to the user.

The Presentation Generator performs the necessary viewing transformations based on the position of the user's viewpoint relative to the scene, and then renders the scene from that point of view.

In the current implementation, the `PresentationGenerator` renders to a window on a standard display monitor. In a future incarnation of the system, the `PresentationGenerator` will also be capable of rendering to other devices, such as a head-mounted display (HMD). The possibility also exists for more than one Presentation Generator to be used

simultaneously—for example where a stereoscopic HMD is used, separate instances of this class would render slightly different images to each of the user's eyes to create the illusion of depth. Furthermore, this class has been called `PresentationGenerator`, as opposed to `GraphicsGenerator`, indicating the further possibility in some future version of the system for devices which provide output which is not necessarily visual—for example three dimensional audio or tactile feedback devices.

Additional Presentation Generators will be implemented as classes derived from the base `PresentationGenerator` class. It is in this way that the system caters for the use of multiple output devices—all of the functionality required to handle a particular device is contained in the corresponding `PresentationGenerator` class which has been developed to support that device.

`SimulationAgent`

`ObjectAgent`

`InputHandlerAgent`

`PresentationGeneratorAgent`

These four classes are the corresponding agent classes for each of the four main process classes described above. An instance of one of these agent classes is the software realisation of an *agent process*, which was the term used in WB 104, the Conceptual System Design.

The responsibilities of each agent process are similar. The agent process serves as a representative for the authentic process with which it is associated, and which resides on a remote machine. The agent in each case presents the same interface as the actual process, and is thus able to intercept messages (function calls) destined for the authentic process. The agent then sends these messages, via the Environment Manager, across the network to the actual process.

3.2.1.2.3 Design Considerations

There are two important design decisions with regard to the High Level Process Classes class category that need to be explained. These are:

- the inheritance relationship which exists between a process class and its corresponding agent class, and
- the fact that each process class is in fact *multiply* derived from the corresponding agent and from the base class `Process`.

The decision to derive a process class from its corresponding agent class was not taken lightly, and the motivation for this choice will now be explained. The decision was primarily based on:

- the structural commonality which exists between the process class and its agent
- the ability to make advantageous use of polymorphism, and
- the responsibilities of an agent class, and how these responsibilities relate to the responsibilities of the process class for which it acts as an agent.

By definition, the interface presented by an agent class must to a large extent be identical to that of the process class which it represents, in order for the fact of the existence of the agent to be concealed from the calling process. The function headers which are provided by the agent class must therefore mirror those of the process class. This leads to a high degree of structural commonality between a process class and its agent.

It is desirable to use polymorphism (see section 2.2.2.2) in referencing processes and agents, so that the calling process need not be aware of whether it is talking to the authentic process or to an agent. For this reason, it is appropriate for the relationship between a process and its agent to be one of inheritance. Therefore, given a pointer to the agent superclass, it may be determined dynamically at runtime if the actual object being pointed to is an agent or a process.

Here is evidenced an example of Coggins's Law of Software Engineering, which is quoted by Booch as follows: "pragmatics must take precedence over elegance". A corollary to this law is that the features of the programming language being used often influence architectural decisions [Booch, 1994:333]. The convenience in this case of the use of polymorphism to make the distribution of processing transparent to the high level processes, outweighs the fact that a process is not, strictly speaking, a specialised form of agent. In a particular sense, however, a process *can* be regarded as a kind of agent, as discussed below.

The main responsibility of an agent class is to receive messages destined for the authentic process, package them into a form suitable for transmission across the network, and then despatch them to the remote process. It is felt that, in a sense, a process can be regarded as a *type of* agent, since it acts as its own agent (receives its own messages) if the calling process and the target process are on the same machine. Therefore the process can be seen as a specialised form of agent, which adds the functionality of performing the actual "meat" behind the various methods which are provided.

In light of the above factors, is was decided to derive each process class from its corresponding agent class.

Each process class is furthermore derived not only from its respective agent class, but also (through multiple inheritance) from the base class `Process`. The reasons for this are that:

- there is commonality amongst all *processes*, in that they are all able to receive non-local messages, and
- it is convenient for the Environment Manager to refer to all local processes polymorphically when delivering remotely originating messages.

In addition to augmenting the behaviour of the agent class, a process is able to receive remotely originating messages from the Environment Manager (by means of the `RecvMessage` function; see WB 113 Low Level Design). This behaviour is common to all processes, but is not exhibited by any of the agent processes. As well as being seen as specialised agents, therefore, the process classes can also be logically grouped together as *processes* in their own right. Aside from all other behaviour, all processes are able to accept remotely originating messages. There is thus commonality between all the process classes, which may be efficiently expressed through inheritance.

From the point of view of the Environment Manager, it is useful to be able to refer to local processes polymorphically. The EM should not need to know explicitly what type of process is being dealt with in order to deliver a message to that process. Upon receipt of a message from the network, the EM need only perform a lookup of the destination process ID, and can immediately despatch the message to that process, regardless of which particular type of process it is.

3.2.1.3 Environment Manager Classes

The Environment Manager (EM) class category includes the `EnvironmentManager` class itself, as well as a number of classes which are contained or used by the `EnvironmentManager` class. These classes may be regarded as collectively comprising the *Environment Manager* which is referred to in the Conceptual Sy. .em Design (WB 104).

3.2.1.3.1 Class Structure

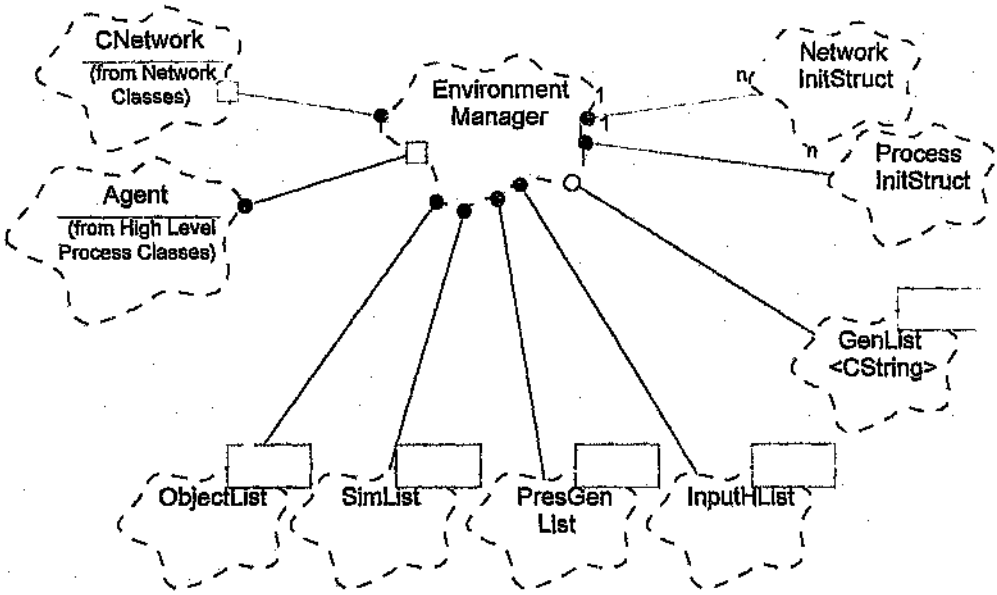


Figure WB105–11: Environment Manager Class Diagram

Figure WB105–11 shows the (predominantly aggregation) relationships which exist between the `EnvironmentManager` and various other classes. As was seen in Figure WB105–9, the `Agent` class maintains a reference to the local Environment Manager. This association is displayed again here for reference.

The EM contains each of the classes `ObjectList`, `SimList`, `PresGenList` and `InputHList`, which are respectively lists of pointers to each different type of agent process. These four classes are in fact each instantiations of the template class `GenList`, with the appropriate type of pointer specified as the formal parameter. For example,

```
InputHList ≡ GenList<InputHandlerAgent*>
```

An `EnvironmentManager` instantiates all of the agents and processes which fall under its jurisdiction, and is responsible for destroying them upon termination of the program. These linked lists are used to maintain handles to these objects for the duration of their existence.

In addition to the abovementioned instantiations of the `GenList` class, the EM also uses `GenList` in the implementation of particular member

functions (typically to store a list of strings) and therefore the `GenList` class itself has been identified as being *used* by the `EnvironmentManager` class.

Figure WB105–12 depicts graphically the instantiation of the `GenList` template class with the various parameters mentioned above. It may also be observed from this figure that each `GenList` contains a number of `DLListElem` structures. These structures constitute the elements in the linked list, and are a consequence of the implementation of the `GenList` class (refer to section 3.1.3).

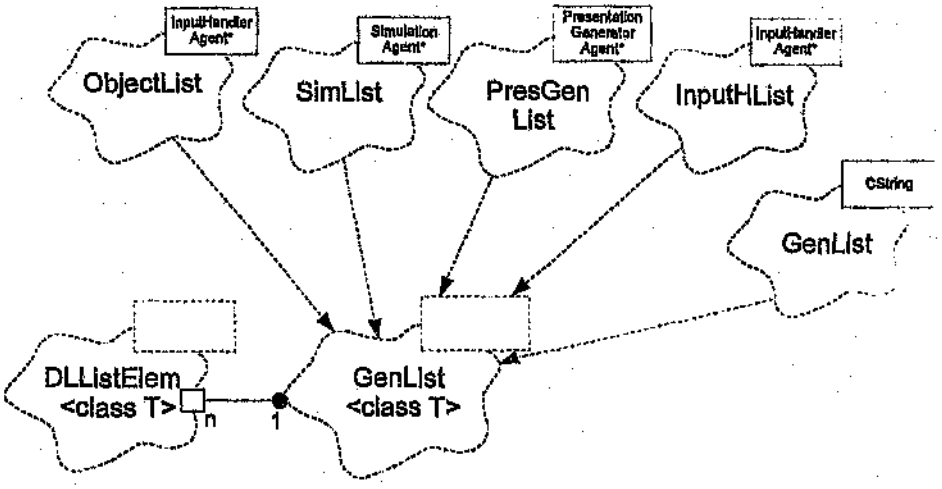


Figure WB105–12: Linked List Class Diagram

An array of `NetworkInitStruct` structures and an array of `Process-InitStruct` structures are also contained by the EM, both of which are used during initialisation. The array of `ProcessInitStruct`'s is also later used by the EM to obtain the identity of the target process for which a message has arrived, in order to deliver it. (These details are discussed in the Low Level Design, WB 113).

Finally, the `EnvironmentManager` contains by reference the class `CNetwork`, which provides the low-level network communications infrastructure of the system (see Section 3.2.1.4 below).

3.2.1.3.2 Class Responsibilities

EnvironmentManager

The EnvironmentManager class is responsible for creating and managing the execution environment of the program, as well as facilitating the transfer of messages between system processes. It forms an intermediate layer between the high level process classes and the network, shielding the high level processes and agents from the intricacies of the underlying network communications, while at the same time concealing the details of process-to-machine allocation from the network. In order to achieve this, an instance of the EnvironmentManager class exists on every machine.

The creation of the execution environment refers to the initial allocation of processes to processors, and the subsequent instantiation of all the necessary processes and agents on each machine. Instances of the EnvironmentManager running on each machine must communicate with one another to ensure that all the necessary processes have been instantiated, and that a consistent process-to-processor mapping is agreed upon.

Upon receipt of a message from an agent process, the Environment Manager adds header information which identifies the process for which the message is destined, determines the destination machine based on the process ID and the current process-to-processor mapping, and sends the message to the network for delivery to the target machine.

When a message is received from the network, the EM strips off the header, and uses it to determine the destination process. It then delivers the message to that process.

A high level process obtains from the Environment Manager the identity of another process with which it wishes to communicate.

CNetwork

The CNetwork class provides the network communications functionality required by the system. This class is part of the Network Classes class category, which is discussed in Section 3.2.1.4 below.

`ObjectList`
`SimList`
`PresGenList`
`InputHList`

These classes merely provide a repository for lists of pointers to each different type of agent process. In each case, this takes the form of a linked list of pointers to that particular agent class.

`NetworkInitStruct`

The Environment Manager uses an array of `NetworkInitStruct` structures to keep track of the existence status of each of the remote nodes during system initialisation. The precise startup mechanism that is used is discussed fully in the Low Level Design (WB 113).

`ProcessInitStruct`

The Environment Manager maintains an array of `ProcessInitStruct` structures, which forms a "process responsibility list". Each entry in the list contains the details of one of the processes in the system, and specifies the process ID of that process, the process type and the ID of the machine which is responsible for that process. Where the process is local, a pointer to the process is also maintained.

This list is populated by the Environment Manager during initialisation, and reflects the current process-to-processor allocation. Thereafter, this information is used for instantiating the necessary local processes, and agents for any remote processes which may exist.

The Environment Manager also subsequently uses the process responsibility list in order to deliver any remotely originating messages which it receives to the correct process. The EM performs a lookup based on the process ID contained in the message, and delivers the message to the process pointed to by the relevant `ProcessInitStruct`.

3.2.1.3.3 Design Considerations

The decision to have to EM maintain lists of pointers to processes was based on the following factors:

- The need for a convenient way for a process to determine the identity of another process with which it wishes to communicate
- The likelihood, in a future implementation of the system, of dynamic process to processor allocation and migration of processes based on bandwidth or processing load

- The need for a synchronisation mechanism to ensure mutually exclusive access to a process by different threads of execution

The architecture that has been chosen satisfies the first consideration by providing a convenient repository for the names of processes, which can be accessed by any of these processes, by querying the EM.

It is important to note that when process A obtains from the EM the identity of process B in the form of a pointer to a B_{agent}, there is no need for process A to know whether the pointer points to the actual process B or an agent for process B. As far as process A is concerned, it has obtained a pointer to a B_{agent}, which provides a particular range of functionality through a defined interface; the details of how that functionality is provided are not relevant. The EM makes the details of whether a process is local or remote transparent to the calling process.

This is a key issue if process to processor allocation is to be performed dynamically, or if the possibility exists for a process to migrate to a different node. If there were not this transparency, each process would have to keep track of which machine housed each other process, and each time a process migrated, all other processes would have to be informed of the change.

Synchronisation between threads is critical, since unpredictable behaviour can result if more than one thread is able to access the same object simultaneously. Querying the EM each time access to a process is required, enables the EM to monitor whether the process in question is being used by another thread, and only provide access to it once it is released by that thread.

Since the Network is used exclusively by the Environment Manager, it makes sense for the lifetimes of instances of these two classes to be connected. The Environment Manager constructs the Network and maintains a pointer to it. When the EM destructor is called, the network is also destructed.

3.2.1.4 Network Classes

This class category comprises those classes which provide the network communications infrastructure for message transfer between processes residing on different machines. The classes in this category were developed by George Spanellis (see section 3.1.1); for the purposes of the project at hand, the entire Network Classes class category has been treated as a "black box". Therefore, the design of this class category will not be expostulated in detail. For this information, the reader is referred to the 1995_11 DPVR project documentation. The class structure is, however, presented here for reference, and the responsibilities of the class category as a whole are discussed.

3.2.1.4.1 Class Structure

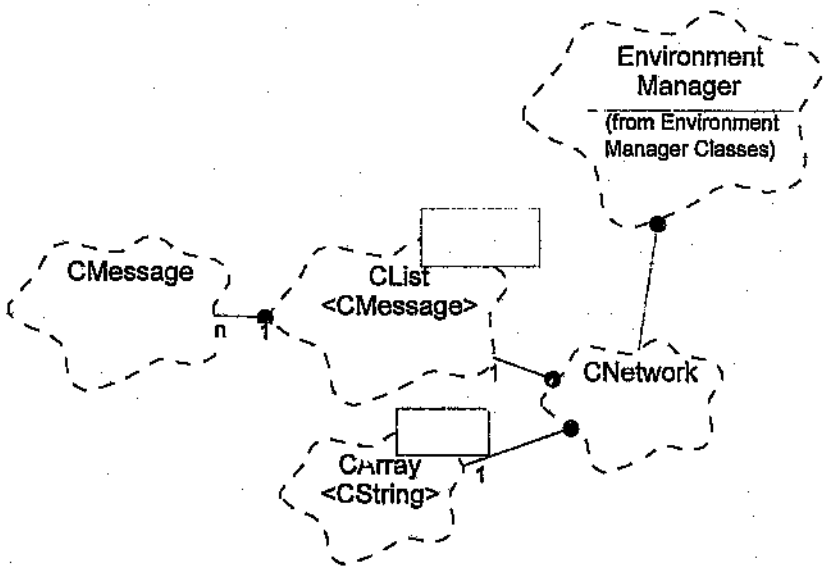


Figure WB105–13: Network Class Diagram

The `CNetwork` class contains two instances of the `CList` class, each of which is a doubly-linked list of `CMessage`'s which respectively form the incoming and outgoing message queues. In addition, CNetwork contains a CArray of CStrings, which acts as an address table.

3.2.1.4.2 Class Category Responsibilities

The Network Classes class category is in essence responsible for transferring messages from one machine to another over the network. The interface through which this functionality is provided consists of a "send message" function, a "receive message" function, and a means for checking if there are any messages in the incoming message queue.

The send message function is used to send a message from the local machine to a remote machine. The receive message function and querying of the message queue are used to retrieve messages which have arrived from a remote machine. A polling system is used, whereby the message queue is checked, and if a message has arrived, then the receive message function is used to obtain the message.

The network provides for the transfer of two different message types. These are referred to respectively as guaranteed and non-guaranteed messages. Guaranteed messages

3.2.1.5 Debug and Timer Classes

These class categories are unique in that they do not provide services which contribute to the user-observable functionality of the system. Rather, they are intended to be used during development and system testing, for debugging and for performance evaluation. They provide, respectively, the facilities necessary for debugging of the program source code through the display of relevant messages during program execution; and logging the time at which particular events take place.

3.2.1.5.1 Class Structure

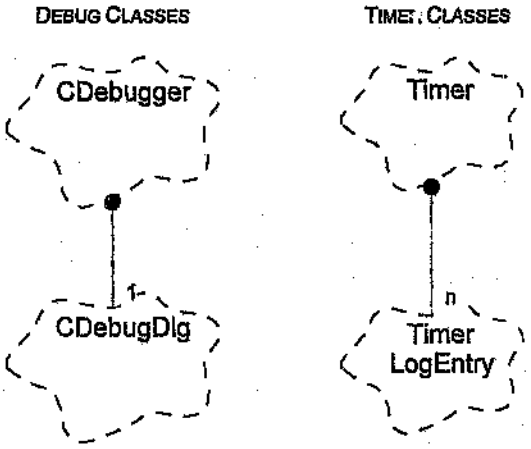


Figure WB105–14: Debug and Timer Class Diagrams

Figure WB105–14 presents class diagrams for the Debug and Timer classes. The class structure of these class categories is very straightforward. The `CDebugger` class is the main Debug class, and contains a single instance of the class `CDebugDlg`, the debug output dialog box class. The `Timer` class is the main class in the Timer class category, and it contains an array of `TimerLogEntry` structures.

3.2.1.5.2 Class Responsibilities

`CDebugger`

The `CDebugger` class is responsible for receiving debug messages, and outputting them to the debug dialog box.

CDebugDlg

The CDebugDlg class is simply the dialog box by means of which debug messages are displayed. It is responsible for displaying messages which are directed to it.

Timer

The Timer class is responsible for maintaining, in memory, a log of messages it receives, along with the time that each message was logged. When the Timer is destroyed, it dumps the log to a text file, the name of which was specified upon instantiation.

TimerLogEntry

The TimerLogEntry structure represents a single log entry. It consists of a description of the logged event, together with a time stamp for that event.

3.2.1.5.3 Design Considerations

The CDebugger class provides a generic interface to the debugging functionality which is provided. It might prove useful at some later stage to send debug messages not to a dialog box but to a text file or other destination. The use of the intermediary CDebugger class would mean that such a change could be easily implemented without necessitating changes everywhere in the code where debugging information is logged.

The decision to have the timer class log its messages to memory and only write the log out to disk upon program termination (refer to the Timer class's responsibilities above) was based on the fact that disk access is slow compared to processor/RAM-based operations. Therefore, in order for the act of timing events not to influence the measured times considerably, it is desirable to log to memory first, and later output the results to a file which can be subsequently inspected.

3.2.2 Object Structure

In this section, object diagrams and interaction diagrams are used to identify various objects that exist in the system, and to illustrate the relationships between them and the mechanisms whereby they interact. In this way, the collaboration of objects to provide the desired behaviour of the system is described.

Three different scenarios are presented:

- Translate Camera (Local)
- Translate Camera (Remote), and
- Perform Simulation Step

These three scenarios have been selected from a wide variety of possible sequences of interaction among the objects in the system. Specifically, the three mechanisms which are described are prototypical examples which provide insight into some common modes of operation of the system.

The first two scenarios serve to illustrate how the same high level operation may take place differently depending on the hardware configuration on which the system runs. The operation in question involves a change in the user's viewpoint. The sequence of events is in each case triggered by a particular mouse movement by the user, indicating his desire to change the point of view. The result is that the camera location is changed, and the updated scene is rendered. The first case is where all of the processes involved are on the same physical machine. In the second case, the necessary processing has been distributed—in this case across two machines.

The third scenario shows the interaction between the Simulation, an Object, and the Presentation Generator, when one step of the simulation is performed. The processing is again distributed across more than one machine, and in this case, bidirectional communication between machines is necessary in order to achieve the required behaviour.

3.2.2.1 Object Diagrams

3.2.2.1.1 Translate Camera (Local)

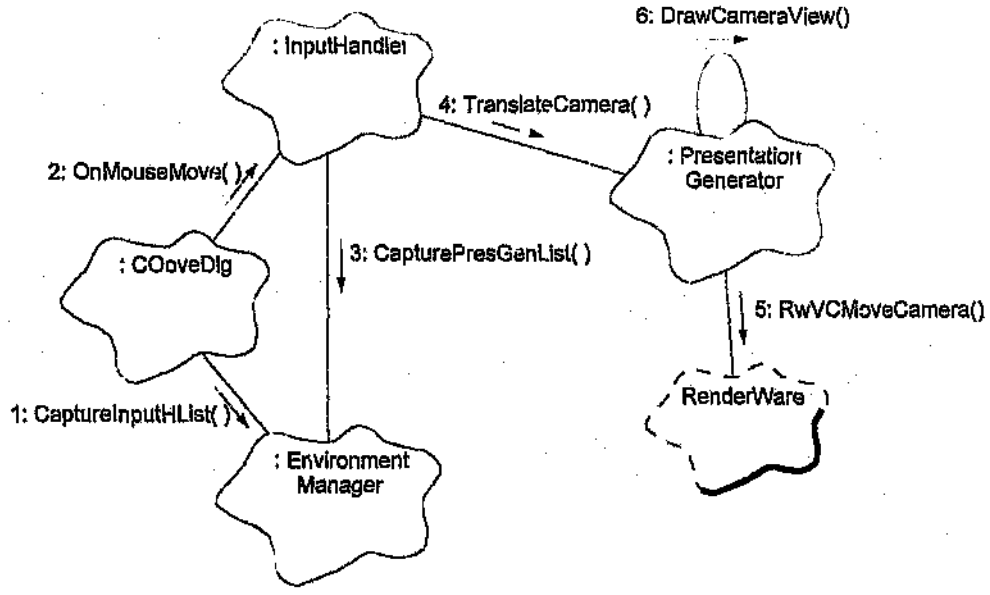


Figure WB105–15: Object Diagram—Translate Camera (Local)

The first scenario is depicted in Figure WB105–15. The mouse movement is initially detected by Windows, and the window in which the movement took place is notified. In this case, the window in question is the main application dialog box, which is of the class `COoveDlg`. The dialog box simply redirects the `OnMouseMove()` message to any Input Handlers which exist. This involves firstly obtaining handles to the Input Handlers from the Environment Manager, by means of the `CaptureInputHList()` function. In this scenario, there is only one Input Handler, and the `COoveDlg` object sends the `OnMouseMove()` message to it. The Input Handler processes the received information, and determines that it indicates that the camera must be translated. A handle to the Presentation Generator is obtained from the EM, and the Presentation Generator's `TranslateCamera()` method is then invoked. In order to do the translation, the Presentation Generator calls the RenderWare function `RwVCMoveCamera()`. Lastly, the Presentation Generator invokes it's own `DrawCameraView()` member function, which renders the scene based on the new camera viewpoint.

3.2.2.1.2 Translate Camera (Remote)

Figure WB105–16 depicts the second scenario. The mechanics of the situation are precisely the same as in the previous case. However, in this case, the Presentation Generator is on a different machine to the Input Handler, and therefore the operation involves communication over the network.

The `COoveDlg` object, as before, obtains a handle to the Input Handler from the EM, and redirects the `OnMouseMove()` message to it. The Input Handler again determines that the camera must be translated. A handle to the Presentation Generator is obtained from the EM, and the Presentation Generator's `TranslateCamera()` method is then invoked. In this case, however, the handle which the Input Handler obtains is actually a pointer to an *agent* which represents the Presentation Generator. The agent constructs a `FunctionCall` object, and adds to it the function ID and all of the parameters which were passed. The agent then obtains from the FunctionCall a string representation which fully specifies the function call, by means of the `Get()` method. The agent sends this message along with it's own process ID to the EM for transmission across the network to the corresponding authentic process on the remote machine. The EM adds the process ID to the message in the form of a header, determines the destination machine based on the process ID and its process-to-processor allocation table, and then sends the message by means of the network to that machine.

On the receiving end, the Environment Manager polls the network message queue (the `GetCount()` function returns the number of messages in the queue). When the message arrives, it is retrieved by the EM from the network, and passed to the EM's `RecvMessage()` function. The EM determines the identity of the destination process from the message header, and transfers the message to the `RecvMessage()` function of that process. In this case, the target process is, not coincidentally, a `PresentationGenerator` object.

The `RecvMessage()` function constructs a `FunctionCall` object, using the received string, and then retrieves the function ID and the parameters for the call from it (these details are not shown in the figure). It then invokes the `PresentationGenerator::TranslateCamera()` method with the decoded parameters.

Thereafter, as before, the Presentation Generator calls `RwVCMove-Camera()` and `DrawCameraView()`, to translate the camera and then render the scene.

Note that from the point of view of both the Input Handler and the Presentation Generator, the operation appears exactly the same as the previous case, where all of the processes were local. This illustrates the main advantage of the Process Agent Model (Conceptual System Design, WB 104).

3.2.2.1.3 Perform Simulation Step

The third scenario is depicted in Figure WB105–17. The Simulation has performed an iteration, and wishes to update the height of one of the objects in the scene to reflect the calculated value. In order to do this, the Simulation issues a 'scale' command to the appropriate Object. Thereafter the Object instructs the Presentation Generator to scale its representation of that object, and finally, the Presentation Generator re-renders the scene.

The Simulation first obtains from the `EnvironmentManager` a handle to the Object, and then calls that Object's `Scale()` function. The handle which the `Simulation` receives is in this case a pointer to an ObjectAgent. The Agent constructs a `FunctionCall` object, as described in Section 3.2.2.1.2 above, and sends the marshalled function call to the EM. The EM determines the identity of the destination machine, and sends the message to the Network for delivery to that machine.

The Environment Manager on the destination machine continuously polls for messages from the Network. When the message arrives, the EM retrieves the message, determines the identity of the destination process, and calls the `RecvMessage()` function of that process (in this case the Object that is to be scaled). The `RecvMessage()` function in turn calls the actual `Scale()` function, and the object scales itself accordingly.

The Object now obtains a handle to the Presentation Generator from the Environment Manager and calls the `ScaleObjRepresentation()` function. The handle obtained is again a pointer to an Agent process, and the message is again sent via the EM across the network to the machine on which the actual Presentation Generator resides. (In this case the Presentation Generator resides on the same machine as the Simulation). The actual `ScaleObjRepresentation()` function is called, which performs the necessary scaling of the Object's representation by means of a RenderWare call, and then finally re-draws the camera view.

3.2.2.2 Interaction Diagrams

Interaction diagrams corresponding to each of the three scenarios which were discussed in the previous section are presented in this section. As discussed in Section 2.5.3, an interaction diagram is to a large extent simply another way of representing an object diagram. Furthermore, Booch [1994:217] observes that object diagrams and interaction diagrams are sufficiently close in terms of their semantics that it is possible for tools to generate one diagram from the other. This has indeed been the case here—each of these interaction diagrams was generated from the corresponding object diagram by means of Rational Software's Rational Rose tool.

These diagrams are presented in addition to the object diagrams, since it is easier to follow the order of message passing in each scenario. The diagrams are presented without explanation, since the explanations which were provided in the previous section carry over without modification.

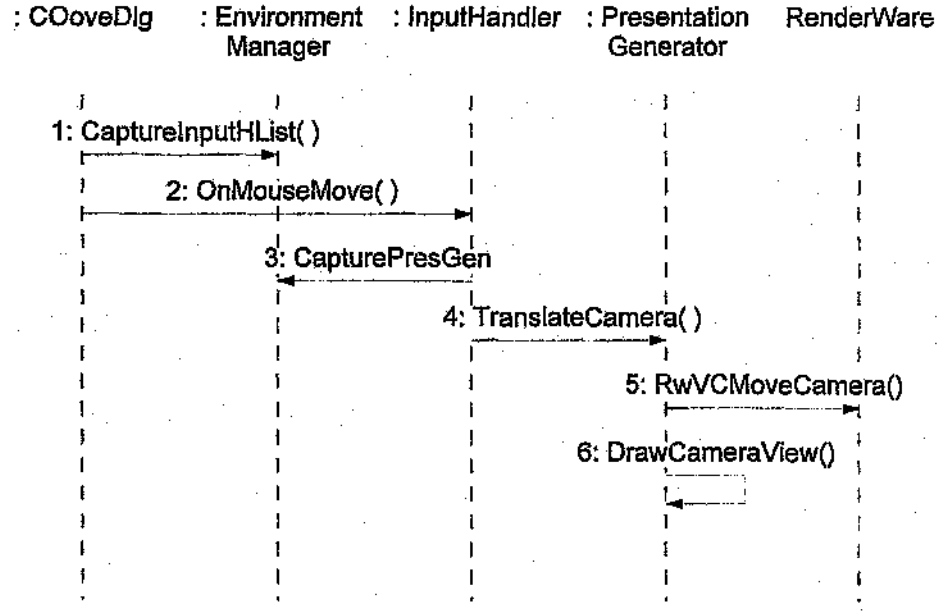


Figure 18: Interaction Diagram—Translate Camera (Local)

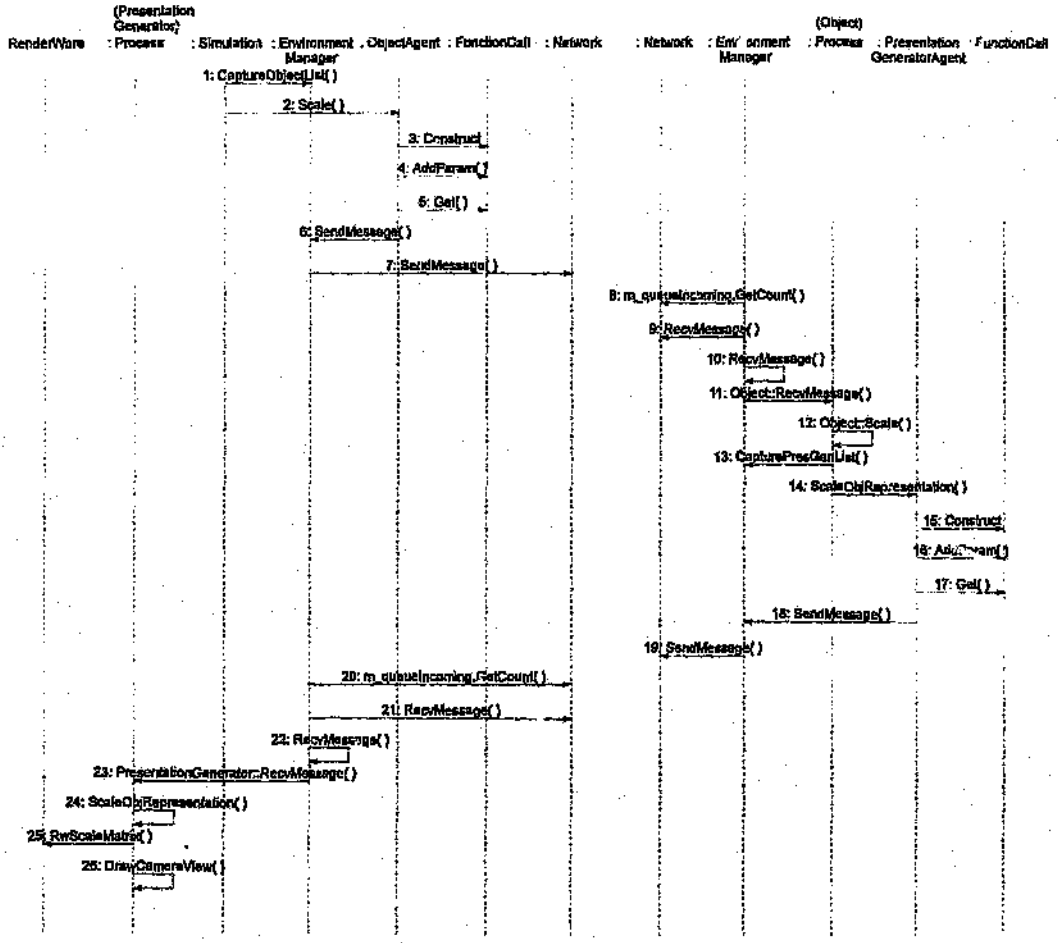


Figure WB105–20: Interaction Diagram—Perform Simulation Step

3.3 Physical Architecture

This section describes the concrete software composition of the system implementation: the allocation of classes and objects to modules, as well as the allocation of processes to processors.

3.3.1 Module Structure

The source code which comprises the system is divided into a number of files which make up modules. The software has been organised such that a particular file contains either the declaration or the definition of *one* of the classes which comprise the system. Therefore, for each class which has been defined, there are two source files, one which contains the declaration or specification of the class, and another which contains the implementation detail of the class– the body or definition. In accordance with Booch's convention, these two files have the same name, except for a distinguishing suffix [Booch, 1994:220]. As a consequence of the system having been implemented in C++, the declaration files all have the suffix or extension ".H", while the definition files have the extension ".CPP". Together, these two files comprise a single module in the physical implementation of the system.

There are some exceptions to this convention. In some cases, related classes have been grouped into a single module. In particular:

- each of the four main process classes (`PresentationGenerator`, `InputHandler`, `Object` and `Simulation`) have each been grouped together with their respective agent classes
- a number of miscellaneous small classes, structures and enumerated types have been grouped into the "`MISC`" module
- the Network classes all reside in a single module

3.3.1.1 Top Level Module Structure

Figure WB105–21 depicts the top level module diagram of the system.

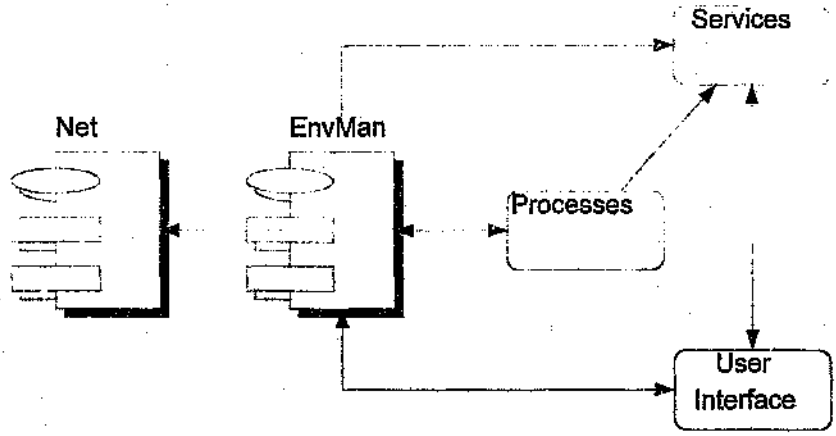


Figure WB105–21: Top level Module Diagram of the System

Five main subsystems have been identified. Two of these, the Environment Manager subsystem and the Network subsystem, each consist of a single module. The remaining three subsystems are the Processes, User Interface and Services subsystems. The modules which comprise the Processes subsystem contain the declaration and definition of each of the four main process classes and their respective agents. The User Interface subsystem contains those modules responsible for any interaction that the program has with the user via the Windows interface. The services subsystem houses various services that are commonly used by other parts of the system. Each of these subsystems are discussed in more detail in sections 3.3.1.2, 3.3.1.3 and 3.3.1.4, respectively.

The main dependencies which exist between the top level subsystems are depicted in Figure WB105–21. The EnvMan module (which contains the `EnvironmentManager` class specification) is dependent on the Net module (which contains the `Network` class specification), since the Environment Manager uses the network class to transfer messages over the network to a remote machine. The EnvMan module is dependent on the process modules, since the EM instantiates all the high level processes of the system, and maintains a list of pointers to each type of process. The processes subsystem is dependent on EnvMan to allow a particular process to maintain a pointer to its parent Environment Manager. The user interface depends on EnvMan because the Environment Manager itself is instantiated as part of the initialisation of the main dialog box of the program. EnvMan depends on the user interface, so that the EM is able to initialise the RenderWare graphics library, which is part of the user interface subsystem. One or more services are used by most of the other system components. The details of these dependencies are given in section 3.3.1.4. The reason for the dependency of the

services module on the user interface is due to the fact that the debug module makes use of user interface functionality in the form of the debug dialog box.

3.3.1.2 Processes Subsystem

The processes subsystem consists of four modules—one module for each of the four main process types. Each of these four modules contains the code for one of the process classes and its associated agent class. Dependencies exist between the process modules and other subsystems, as well as internally, among the process modules themselves.

The module diagram for the processes subsystem is shown in Figure WB105–22.

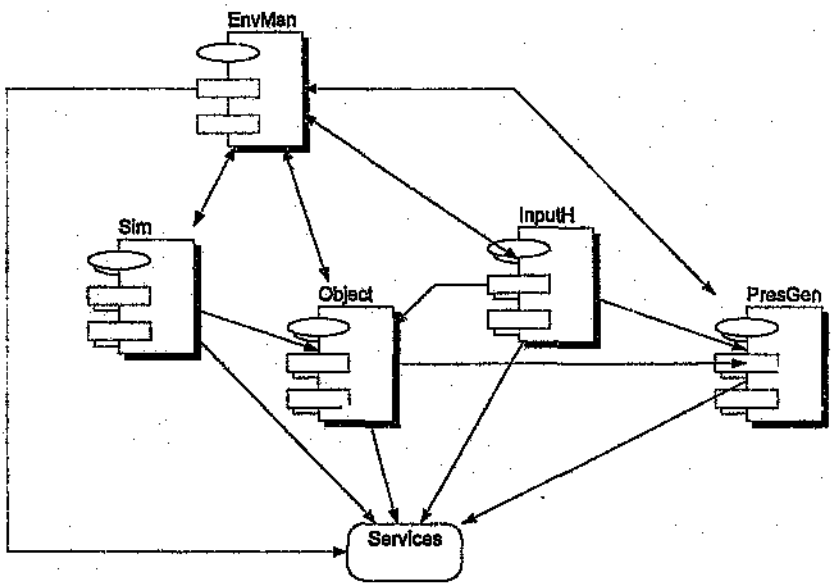


Figure WB105–22: Processes Subsystem Module Diagram

The diagram indicates the dependencies which relate to the process modules. All of the process modules depend on EnvMan; in order to communicate with one another, processes retrieve a pointer from the Environment Manager to the process they wish to access. Conversely, in order for these pointers to be maintained, the EnvMan module depends on each of the process modules. All of the process modules as well as the EnvMan module are dependent on various services, which are once again dealt with in section 3.3.1.4. Internal dependencies between the various

processes are as a result of messages passed between processes as defined by the Conceptual Virtual Reality System Model which is presented in the Conceptual System Design (WB 104). For example, the Input Handler sends messages to the Presentation Generator concerning changes in the current viewpoint, and thus the InputH module depends on the PresGen module.

3.3.1.3 User Interface Subsystem

The user interface subsystem contains the main OOVE application modules, as well as modules for the dialog boxes which make up the Windows interface of the program.

The user interface subsystem also houses the RenderWare subsystem, which is the third party graphics rendering library that has been used in implementing parts of the system. The RenderWare subsystem encompasses the RenderWare header files (including rwlib.h, rwtypes.h, rwmacros.h, and rwwin.h), as well as the RenderWare library file (rwl20.lib).

The user interface module diagram is shown in Figure WB105–23, and shows the dependencies which exist within this subsystem.

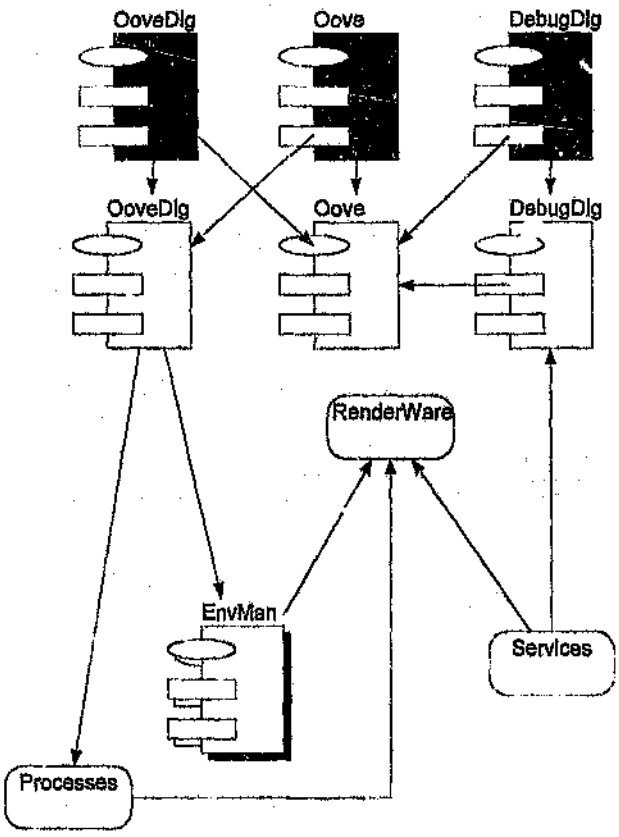


Figure WB105--23: User Interface Subsystem Module Diagram

The OoveDlg module is dependent on the EnvMan module, to facilitate instantiation of the Environment Manager during initialisation of the main dialog box of the program. OoveDlg is also dependent on the processes subsystem, specifically the InputH module, in order for mouse and keyboard input captured by the main dialog box to be forwarded to the Input Handler for processing.

The Environment Manager, process modules and service modules all depend on the RenderWare subsystem: the Environment Manager is responsible for initialising the RenderWare library, the Presentation Generator (PresGen module in the processes subsystem) is responsible for rendering the frames that are presented to the user; and the services modules use the RenderWare data type definitions, such as RwBool and RwReal.

The Debugr module (in the services subsystem) uses the DebugDlg module, to implement the debugging dialog box.

Several dependencies exist between the modules in the user interface subsystem itself. In order to reveal these dependencies, the module diagram depicts the declaration and definition of these modules separately. The definition of the OOVE application module and each of the dialog modules (OoveDlg and DebugDlg), is in each case dependent on the corresponding declaration. The other dependencies that are depicted are generated automatically by the Microsoft Visual C++ compiler to ensure that the relevant message maps and response tables are set up correctly, and to provide visibility with respect to the Microsoft Foundation Classes (MFC).

3.3.1.4 Services Subsystem

The services subsystem houses a number of commonly used components of the system. These components include the parameterized linked list class, the function call class, the base process and agent classes, and several structures and enumerated types. The module diagram for the services subsystem appears in Figure WB105–24.

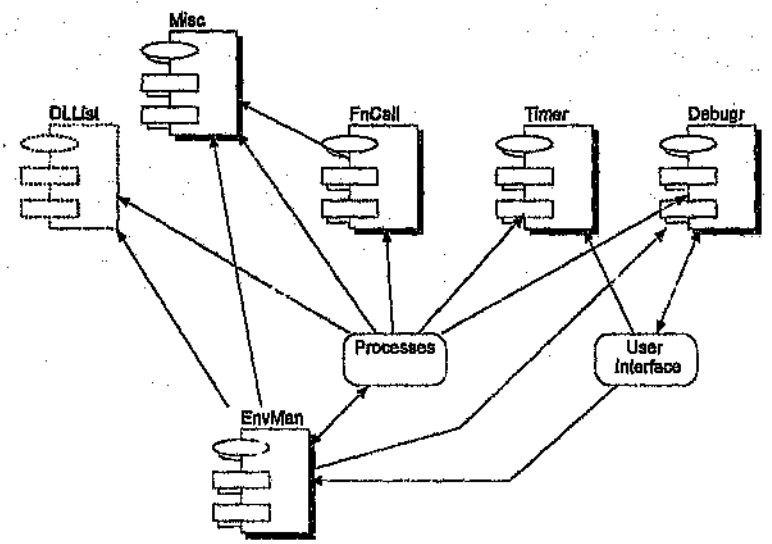


Figure WB105–24: Services Subsystem Module Diagram

Each of the other components of the system use a subset of the services housed by the services subsystem. The details of these dependencies are made apparent by the figure.

For completeness, the dependencies of the user interface subsystem on the EnvMan module, and the bi-directional dependency between the Environment Manager and the processes subsystem, which have already been discussed, have once again been shown.

3.3.1.5 Allocation of Classes to Modules

Module Name	Classes Allocated to Module
EnvMan	`EnvironmentManager`
Net	`struct ReadStruct`[Ψ] `Cmessage`[Ψ] `Cnetwork`[Ψ]
Sim	`SimulationAgent` `Simulation`
InputH	`InputHandlerAgent` `InputHandler`
Object	`ObjectAgent` `Object`
PresGen	`PresentationGeneratorAgent` `PresentationGenerator`
FnCall	`FunctionCall`

Misc	`Process`
	`Agent`
	`struct NetworkInitStruct`
	`struct ProcessInitStruct`

DLList	`struct DLListElem`[Ψ] `GenList`[Ψ]
Oove	`CooveApp`

[Ψ] Classes marked with this symbol were not developed by the author - see section 3

Module Name	Classes Allocated to Module
OoveDlg	CAboutDlg COoveDlg
DebugDlg	CDebugDlg
Debugr	CDebugger
Timer	struct TimerLogEntry Timer

3.3.2 Process Structure

The system has been designed to take advantage of distributed processing in order to improve performance. Specifically, the system runs across a number of machines which are interconnected by means of an Ethernet network. Consequently, the platform for execution of the program consists of a number of different processors, and when the program runs, each of the system processes is allocated to a particular processor.

The allocation of processes to processors is not static—the assignment may be specified by the user. Additionally, the hardware which is used as well as its configuration need not be the same each time the program is run. In the current implementation of the system, the allocation of processes to processors is performed by the Environment Manager, by means of lookup tables which are specified in configuration files. The system has been designed in such a way as to make the choice of process-to-processor configuration as flexible as possible. The only constraints that are imposed are brought about by hardware considerations: the process responsible for a particular device must reside on the machine to which that particular hardware device is physically attached. For example, the Presentation Generator must be on the machine which houses the display device which will be used.

In a future implementation of the system, it will be the responsibility of the Environment Manager to dynamically manage process-to-processor allocation, and to intelligently allocate processes to processors based on machine capability evaluation. The Environment Manager will also perform dynamic load balancing based on processor and network utilisation and effect reallocation of processes to under-utilised resources.

Because the process structure of the system is variable, and is subject to change each time the system is run, the process structure of the system may only be discussed prototypically.

Figure WB105–25 presents a sample process diagram. The situation depicted involves the system running on three machines. There are thus three processors. Two hardware devices are also shown—a mouse and display monitor.

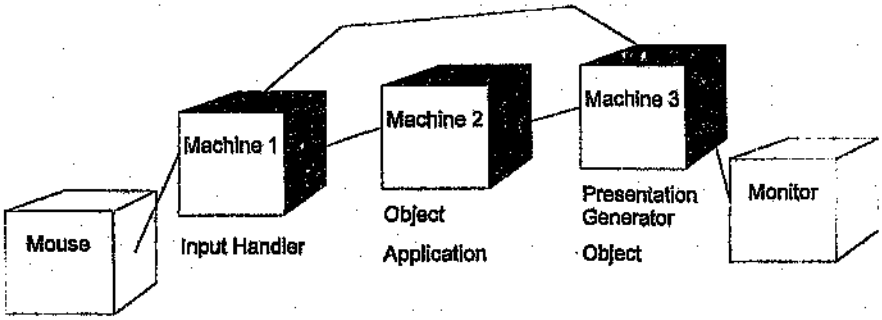


Figure WB105–25: Sample Process Diagram

As is usual in the case of an Ethernet network, each of the three machines have connections to each of the other machines, since they all share a common bus. The mouse is attached to Machine 1, and hence Machine 1 houses the Input Handler Process. Likewise, Machine 3 is the machine to which the monitor is attached, and therefore houses the Presentation Generator process. There are two Object processes and one Simulation process in this case. By means of the configuration files, the Object processes have been allocated to Machine 2 and Machine 3 respectively, and the Simulation process to Machine 2.

4 Conclusion

This document has described the software design of the system implementation from a high level point of view. The dual objectives of the document have been to provide a comprehensive description of the structure of the software from both a logical and a physical perspective, and to highlight the important design decisions which have been taken in arriving at the software structure which is presented.

The system has been implemented using the object-oriented software design methodology, a background to the principles of object-oriented design has been presented. It is clear that the use of an object oriented approach is of significant benefit in the design and implementation of complex software systems in general. Among the benefits are complexity management, resilience to change, reuse and maintainability. Several key benefits that are specific to the project at hand have also been identified. In particular, object orientation facilitates distributed processing through decoupling, separation of responsibilities and intrinsic message passing.

The architecture of the system has been presented primarily through the use of Booch diagrams. The logical architecture is moulded by the classes and objects which make up the system, their responsibilities and behaviour, and the interaction between them. The physical architecture is determined by the allocation of source code to modules and the allocation of processes to processors in the physical implementation of the system.

The overall goal of the software design process has been to realise in software, as elegantly and economically as possible, the conceptual models of the system behaviour which have been formulated, and which have been discussed in the Conceptual System Design (WB 104). It is felt that the software architecture which has been presented achieves this objective.

OOVE

Low Level Software Design

Technical Product

Version 1.03

Document Status: Approved

Table of Contents

Change History

Configuration Control

Project:	OOVE
Title:	Low Level Software Design
Doc. Reference:	WB 113
Created by:	Warren Blumenow
Creation Date:	26 March 1997

Document History

Version	Date	Status	Who	Saved as:
0.01	96/03/26	Draft	WB	\1995_07\TP\LLD\WB113WW.001
0.02	97/05/22	Draft	WB	\1995_07\TP\LLD\WB113WW.002
0.03	97/11/10	Draft	WB	\1995_07\TP\LLD\WB113WW.003
1.00	98/03/07	Approved	WB	\1995_07\TP\LLD\WB113WW.100
1.01	98/05/29	Approved	WB	\1995_07\TP\LLD\WB113WW.101
1.02	98/06/23	Approved	WB	\1995_07\TP\LLD\WB113WW.102
1.03	98/09/02	Approved	WB	\1995_07\TP\LLD\WB113WW.103

Revision History

Version	Date	Changes
0.01	96/03/26	New document created using WB 002
0.02	97/05/22	Created structure for documenting class data members and member functions.
0.03	97/11/10	Added section on naming conventions.
1.00	98/03/07	Document status changed to Approved. Updated Sections 4, 5 and 6.
1.01	98/05/29	Mainly updates to Section 6.
1.02	98/06/23	Documented remaining classes and updated Common Tactical Policies section.
1.03	98/09/02	Final revisions and formatting.

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	98/03/07	Approved	WB 528

1 Scope

1.1 Introduction

This document provides a detailed low level description of the implementation of the software. It supplies the technical details of the realisation of the high level system design in actual program source code, and highlights various tactical decisions that have been made in implementing the system architecture.

After specifying the naming conventions and common tactical policies that have been adopted, the document describes in detail the implementation of each of the classes which make up the system. For each class, an explanation of each of the constituent data members and member functions is provided. Where appropriate, the precise algorithms that have been used in implementing particular non-trivial member functions are described fully.

The system has been implemented using the Microsoft Visual C++ compiler and the Microsoft Foundation Classes (MFC) Version 4.2, as well as Criterion's RenderWare Version 2.0. The software runs under Windows 95 and Windows NT version 4.0. The code has been successfully compiled under both operating systems. When the system is distributed over a group of networked PCs, the use of a heterogeneous mix of the two operating systems is acceptable.

`This typeface has been used throughout the document to denote source code fragments, class names, module names and filenames.`

1.2 Purpose

The Low Level Software Design is intended to serve as a reference document for anyone wishing to understand or modify the program source code which has been developed. It provides insight into the algorithms that have been employed, and the logic behind the practical realisation of each functional component of the system.

1.3 Audience

This document is intended to be used primarily by individuals involved in further development of the project, who will need to understand the low level implementation details thoroughly in order to be able to extend or modify the functionality of the system. The wider audience for the

document includes the project manager, the external examiner, members of the SEAL management board, and any other interested parties.

1.4 Applicable Documents

1.4.1 Standards

a. SEAL QMS Document Creation Template, QS 002, Revision 0.02, 3 October 1994

b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 1.00, 3 October 1994

1.4.2 References

a. References and Bibliography, WB 114

1.5 Assumptions

It is assumed that the reader has consulted the Conceptual System Design (WB 104) as well as the High Level Software Design (WB 105), and consequently is well aquainted with the high level logical and physical structure of the software. This knowledge is a key requirement to the understanding of the material presented here.

It is further assumed that the reader has a thorough knowledge of the C++ programming language and syntax, as well as object oriented programming concepts and terminology.

2 Source Code Naming Conventions

The conventions that have been adopted in the choice of names for, *inter alia*, the classes, objects, types, variables and class methods used in the implementation of the software, are discussed in this section. The use of an appropriate and consistent naming convention assists the reader in understanding the code. According to Booch [1994:163], the most important characteristics of any naming convention that is used, are consistency and clarity. The naming of an entity should capture its semantics—the essence of the abstraction it represents. It is felt that the conventions which have been chosen, and which are described below, exhibit these features.

2.1 Classes, Structures, Enumerated Types and Type Definitions

Mixed case identifiers have been used for classes, structures, enumerated types and type definitions (`typedefs`). Descriptive names are used for each entity—unabbreviated except where necessary for conciseness. Each of the words in the name of the entity begins with a capital letter, with no spaces or underscores between the words. For example:

class:	`EnvironmentManager`
structure:	`ProcessInitStruct`
enumerated type:	`ProcessType`
type definition:	`ObjectList`

The base functionality of certain of the user interface classes that form part of the system was initially generated by the Microsoft Visual C++ MFC *AppWizard*. In addition, the system interfaces with some classes that were not developed or not developed *entirely* by the author (refer to Section 3.1 of the High Level Software Design WB 105). In the case of both the compiler-generated classes and these ancillary classes, there is a slight variation on the above naming convention—each identifier is prefixed by a 'C' (identifying it as a "class"). For example:

```
COoveDlg
CNetwork
```

2.2 Objects, Variables and Class Data Members

The names of objects, variables and the data members of a class all begin with a lowercase letter. Where more than one word is used, the second and subsequent words are capitalised. No spaces or underscores are used in these names. For example:

object: `FunctionCall fn`

variable: `int result; FILE* addressFile`

class data member: `PresentationGenerator::inUse`

One notable special case is the use within a name of a reserved word or acronym which is in uppercase, such as `UINT`. In this case, the word remains uppercase, unless the object or variable name begins with this reserved word, in which case the first letter becomes lowercase. For example:

```
paramUINT
uINTCount
```

2.3 Class Methods

Class method names have the initial letter of each word capitalised, in much the same way as the names of classes and structures. Class *methods* can be distinguished from classes by the context of their usage. Invariably, a class method name will be followed by a list of parameters, whereas a class name will not. For example:

```
RecvMessage(char* message)
DrawCameraView()
```

It should be noted that during construction of an object, the class name and the name of the constructor method are identical.

3 Common Tactical Policies

Common tactical policies are localised mechanisms which appear throughout the system, and represent standard approaches to the handling of commonly encountered tactical issues during implementation of the system.

3.1 Compilation and Management of Executables

The system has been designed to be distributed across a number of workstations, and from a conceptual point of view is divided into a number of processes, one or more of which execute on each machine.

While conceptually there are multiple processes, from a physical point of view, only a single executable program runs on each machine. For practical reasons, the approach that has been taken is for the *same* executable to run on every machine, configured at run time by means of configuration files. In this way, the source code does not require recompilation when the system configuration changes, and multiple binaries do not need to be managed. A different subset of the functionality of the system will be invoked on each machine, depending on the configuration that has been specified.

Accordingly, a single set of source code files provides all of the functionality of the entire distributed system. This facilitates maintenance of the software as well as simplifying configuration management.

3.2 Threads of Execution

Microsoft Visual C++ and the Microsoft Foundation Class Library (MFC) provide support for creating multithreaded 32-bit Windows 95 and Windows NT applications. The software that has been developed has been written to take advantage of this. Multiple threads of execution are useful in an application such as this where multiple activities need to be managed simultaneously and asynchronously [Microsoft Visual C++, 1996]. Separate threads of execution have been used for the following tasks:

- The main application thread, which handles input from the user, generates the output that is displayed, and provides the majority of the functionality of the system
- Polling and retrieval of messages from the network hardware by the `Network` class

- Retrieval of messages from the `Network` class by the `EnvironmentManager` class
- Simulation-related computation performed by the `Simulation` class

There is exactly one instance of the `Network` class and one instance of the `EnvironmentManager` class on each machine, each of which contributes a separate thread of execution. There is typically a single instance of the Simulation class, and hence a single simulation thread, across the entire system. On the machine that houses the Simulation process, therefore, there will be a total of four separate threads of execution, including the main application thread. On every other machine there will be three threads. In principle, it is possible for there to be more than one Simulation process—a fourth thread of execution will exist on any machine that runs an additional Simulation process.

A distinction is drawn in MFC between worker threads and user-interface threads. User-interface threads are able to handle user input and respond to events and messages generated by the user. Worker threads on the other hand, are used to perform tasks that do not involve user input. The main application thread is a user-interface thread (being derived from the `CWinApp` class); all of the other threads that have been described are worker threads.

In a multi-threaded application such as this, attention needs to be given to thread synchronisation. Allowing two or more threads of execution to have simultaneous access the same data can lead to undesirable, unpredictable and often disastrous results. The details of the thread synchronisation mechanisms that have been used are described in Sections 3.4 (Inter-Process Communication) and 6.12.2 below.

3.3 Memory Management

Most of the large scale memory allocation performed by the program is carried out by the Environment Manager. The EM is responsible for instantiating the high level processes, as well as agents for processes which reside on remote machines. Memory for these processes and agents is allocated by means of the `new` operator, and released upon destruction of the EM by means of the `delete` operator.

Care has been taken to ensure that any memory that is allocated by an individual object is released when that object is destroyed. Therefore any data structures which are created dynamically during the lifetime of an object are deleted in that object's destructor. In this way, memory leaks are avoided. A conscious effort has also been made to avoid orphaned memory caused by the resetting of pointers before the allocated memory

to which ' ey point has been released. This is a further means whereby the potential for memory leaks can be minimised.

Memory not allocated dynamically as described above, is allocated statically, either in the form of class data members, or as locally defined variables. In these cases, measures do not need to be taken to free up the memory which has been allocated, since this is done automatically when the data structures in question go out of scope—either when the block of code in which they are defined ends or when the object of which they are part is destructed.

3.4 Inter-Process Communication

The methodology which has been adopted with respect to inter-process communication, is embodied in the Process Agent Model (PAM) which has been developed. The PAM is fully documented in the Conceptual System Design (WB 104). The goal of the PAM is to make remote communication between processes transparent to the software objects which constitute those processes. There are a few key issues which need to be noted in this regard as far as common tactical policies are concerned.

Firstly, in order to facilitate the (future) dynamic allocation and reallocation of processes by the Environment Manager, processes cannot maintain static naming references to other processes with which they wish to communicate. A process object wishing to call a function of another object must first obtain or "capture" a pointer to that object from the Environment Manager. The object then makes the required calls before "releasing" the pointer that it has been given. It should be realised that the pointer that is obtained may be either point to the actual object in question, or to an agent for that process, depending on whether the target object resides on the local machine or not. In keeping with the goals of the PAM, the calling process is unaware of this distinction. The EM guarantees that a pointer provided in this way remains valid for the duration that it is captured; that is until it is released.

Furthermore, only one process may capture a pointer to another process at a time. This consideration is necessary due to the multi-threaded nature of the program, and acts as a resource access control and synchronisation mechanism. The concept of capturing and releasing pointers to objects provides a simple mechanism whereby simultaneous access to an object by more than one thread can be prevented.

The PAM imposes two limitations with regard to message passing by means of function calls between distributable processes. Firstly, such function calls may not have a return value, and secondly, parameters may not be passed by reference. Only function calls that have remote calling capability are affected by these constraints (i.e. function calls that are able

to be invoked by a process which resides on a remote machine). These limitations are due, respectively, to the undesirability of suspending the calling process while it waits for a return value to be sent over the network; and the absence of shared memory. The former allows for asynchronous message passing between concurrently executing processes, in line with the objectives of distributed processing. The latter implies that all data must be passed by value.

Though not implemented at present, it is possible for passing of parameters by reference to be catered for by the system architecture. This would involve the target process asynchronously sending back a message to its agent process on the calling machine, which would then update the relevant data. It is important to realise that although the semantics are the same, this mechanism would not be exactly equivalent to standard reference parameter passing, since it does not occur synchronously. The calling process would not see updates to the data until some indeterminate time after the function call had returned. The fact that the data had been updated could, however, be *signalled* to the calling process by some means—for example by setting a flag.

3.5 Error Detection and Handling

The detection and handling of errors in a distributed system such as this is a nontrivial task, and is an issue which undoubtedly warrants significant further investigation.

In the case of a standalone system, a common approach to error detection is to use a function's return code as a flag representing the error status of that particular operation. If an integer return code is used, then typically a zero indicates success, while a non-zero return value denotes an error. The particular code that is returned identifies what type of error has occurred. If a function returns a pointer, then an error condition may be signalled by returning a null pointer. In either case, it is the responsibility of the client that calls the function to check the error status and act accordingly. In the project at hand, this approach has been used wherever local function calls are involved. For example, many of the RenderWare library functions return a null pointer in the event of failure; the Presentation Generator verifies the return value of any RenderWare function that it calls, and accordingly handles any errors that occur.

Unfortunately, this strategy does not transfer well to a distributed system, particularly where asynchronous remote message passing is used. As discussed above, in order not to suspend the calling process while messages are transferred over the network, function calls with remote calling capability do not provide a return value. The responsibility for dealing with errors that occur must in these cases be transferred to the supplier object.

The manner in which errors are dealt with depends on the severity as well as the type of the error. In the case of critical errors, an error message is displayed and the program terminates gracefully. Non-critical errors are reported, but do not cause the program to shut down. Network errors (name server lookup failure, timeout due to bandwidth saturation, etc.) are dealt with either by allowing the operation to be retried a number of times, or by discarding the message if it is regarded as non-critical.

Displaying informative messages in the event of an error assists the user or maintainer of the system in tracing the problem and effecting appropriate changes so that the error can be eliminated.

3.6 Code Maintenance

The use of distributed processing and the Process Agent Model introduces some specific requirements with regard to the development and maintenance of the program source code.

In accordance with the PAM, for every distributable process class, a corresponding agent class as well as a remote message handler function (see Section 3.6.2 of the Conceptual System Design WB 104) must be defined. The public interface of the agent class must be identical to that of the associated process. Measures must be taken to ensure consistency between the member functions of the process class, the member functions of the agent class and the remote message handler function lookup tables, each time the interface of the class changes.

A further consideration is that only certain parameter types may be used when defining function calls with remote calling capability. This is due to the fact that the parameter encoding and decoding functionality built into the `FunctionCall` class is only able to interpret a limited subset of datatypes. If support for additional types is required, the functionality of the `FunctionCall` class must be extended.

Issues such as these arise largely due to the fact that a distributed application is being developed using a standalone compiler—a compiler designed to write code to be executed on a single machine. Many of the tasks that are required to be performed are well suited to being entirely automated. A compiler providing specific support for the PAM would, for example, automatically generate the necessary function tables, and perform consistency checking between a process class and its associated agent at compile time.

4 Structures, Enumerated Types and Type Definitions

This section explains the user-defined composite types (`structures`), `enumerated` types and type definitions (`typedefs`) that have been defined in the source code and used in the implementation of the software. These types will be referred to in the subsequent discussion of class data members and methods.

4.1 struct `NetworkInitStruct`

4.1.1 Overview

The `NetworkInitStruct` structure is used by the Environment Manager during initialisation of the network to store the initialisation status of each of the remote nodes on the network. The status of one node is stored in each instance of `NetworkInitStruct`. Two boolean values are stored for each node, indicating, respectively, whether the node in question is up and running (an initialisation message has been received from that node) and whether the node in question knows that the local node is up and running (an acknowledgement of receipt of an initialisation message from the local node has been received from that node).

4.1.2 Fields

`[ int nodeID ]`

During initialisation, the Environment Manager assigns each node a unique zero-indexed node identifier by means of a numeric index into a list of hostnames specified in the address configuration file (refer to Section 3.2.1 of the User Reference Manual, WB 107).

[Note: When an array of `NetworkInitStruct` structures is used to store the existence status for a number of nodes, the node identifier of each node is implicitly given by the array index. Consequently, this field has been consciously omitted from the structure definition]

`RwBool exists`

A boolean value indicating whether the node in question is up and running. A value of TRUE or 1 indicates that the node exists; a value of FALSE or 0 indicates that the node's existence has not yet been established.

`RwBool knowsWeExist`

A boolean value indicating whether the node in question has acknowledged the existence of the local node. TRUE or 1 indicates that acknowledgement has been received; FALSE or 0 indicates that an acknowledgement has not yet been received.

4.2 struct `ProcessInitStruct`

4.2.1 Overview

The `ProcessInitStruct` structure is used by the Environment Manager to keep track of relevant information concerning each of the processes which comprise the system. For each process, the details recorded are the process ID, process type, responsible node, and a pointer to the process if the process resides on the local machine. An array of these structures enables the EM to maintain a process to processor allocation table, by means of which it can manage the local processes for which it is responsible, route messages destined for a remote machine to the correct node, and deliver remotely originating messages to the appropriate local process.

4.2.2 Fields

`[ int PID ]`

A unique process identifier for the process in question.

[Note: When an array of `ProcessInitStruct` structures is used to store the responsible node, process type and process pointer for each of a number of processes, the PID of a process is given implicitly by the array index. Consequently, this field has been consciously omitted from the structure definition]

`ProcessType processtype`

The process type. This field is assigned a value of the enumerated type `ProcessType` (see section 4.6). In the current implementation, a process may be one of the following: Input Handler, Presentation Generator, Object or Simulation.

`int responsibleNode`

Indicates which node is responsible for the process in question. The responsible node will house the authentic process, and all other nodes will house an agent for the process.

Process* processptr

A pointer to the process (an instance of the appropriate process class) if the responsible node is the local machine; otherwise a pointer to NULL.

4.3 struct DLListElem

4.3.1 Overview

This parameterized structure is used internally by the GenList linked list class, and represents a single element of the doubly linked list.

4.3.2 Fields

T element

An instance of the canonical element of the linked list. The type of this field is defined as T, being the type with which the linked list class is instantiated. A GenList instantiated with type T constitutes a linked list of elements of type T. For example, GenList<int> creates a linked list of integers.

DLListElem<T> *next
DLListElem<T> *prev

Pointers to the next and previous elements, respectively. These pointers are both of type DLListElem<T>, which is *this* structure instantiated with type T. The object pointed to in each case will itself contain an element of type T, and its own set of pointers. These pointers are used to navigate through the elements in the linked list.

4.4 struct TimerLogEntry

4.4.1 Overview

This structure is used internally by the Timer class to represent a single entry in the log of events created by the Timer.

Fields

SYSTEMTIME time

A variable of type SYSTEMTIME which serves as a timestamp for the entry in question. The SYSTEMTIME structure is part of the Win32 Application Programmers Interface, and represents a date and time

using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

`CString event`

A text string holding a description of the event that has been logged.

4.5 enum `Parameter`

This enumerated type represents the allowable *types* of parameters which may be passed when an inter-process function call is made. It is primarily used when encoding the parameters of a function call which is to be executed on a remote machine.

The following table details the possible values of a variable of type `Parameter`. Six data types are currently supported.

VALUE	LABEL	TYPE OF REPRESENTED PARAMETER
0	paramInteger	`int`—standard C++ integer type
1	paramString	`char*`—pointer to the start of an array of characters (standard C++ string type)
2	paramChar	`char`—standard C++ character type
3	paramDouble	`double`—standard C++ floating point type
4	paramRwReal	`RwReal`—the RenderWare floating point type, which is equivalent to the `float` type if the floating point RenderWare libraries are used, but allows for the representation of floating point numbers as 32-bit fixed point numbers if the fixed point libraries are used.
5	paramUINT	`UINT`—standard C++ 32-bit unsigned integer type (synonym for `unsigned int`).

4.6 enum `ProcessType`

This type enumerates the different kinds of processes which may exist in the system. Each of the processes running on a particular machine will be of one of these types. The Environment Manager uses this enumerated type to maintain a record of the process type of each running process. In

the case of an agent process, the value refers to the type of the process which that agent *represents*.

The type is defined as follows:

VALUE	LABEL	PROCESS
0	`processInputH`	Input Handler / Input Handler Agent
1	`processPresGen`	Presentation Generator / Presentation Generator Agent
2	`processObject`	Object / Object Agent
3	`processSim`	Simulation / Simulation Agent

4.7 `ObjectList`, `AppList`, `PresGenList` and `InputHList` typedefs

These type definitions simply provide synonyms for the parameterized class GenList, instantiated with pointers to each different type of agent process. They are defined as follows:

```
typedef GenList<InputHandlerAgent*> InputhList
typedef GenList<PresentationGeneratorAgent*> PresgenList
typedef GenList<ObjectAgent*> ObjectList
typedef GenList<SimulationAgent*> SimList
```

An object of the type `InputhList`, for example, is a `GenList` of pointers to `InputHandlerAgent` objects.

5 List of Classes

A list of the classes which have been developed is presented here for reference. The high level responsibilities of each class and the relationships between them are fully documented in the High Level Software Design (WB 105). The implementation details of each class are provided in the Class Reference which forms Section 6 of this document.

The low level internal structure of classes denoted by a "*" will not be discussed in detail, since these classes were not developed by the author (Refer to Section 3.1 of the High Level Software Design, WB 105). However, the *interface* that these classes provide to the rest of the system *will* be discussed, in order to reveal their functionality.

The symbol "ψ" denotes classes derived from the standard Microsoft Foundation Class Library. Much of the functionality of these classes is inherited from their respective MFC parent classes. For this reason, the internal structure of these classes will also not be dealt with in detail. A brief summary of the functionality provided by each class will be given; for more detail, the reader is referred to the Microsoft Visual C++ documentation [Microsoft Visual C++, 1996].

Process / Agent Classes	
Agent	
Process	
FunctionCall	
InputHandler	InputHandlerAgent
Object	ObjectAgent
PresentationGenerator	PresentationGeneratorAgent
Simulation	SimulationAgent
Environment Manager Classes	
EnvironmentManager	

Debugging / Logging Classes
Cdebugger
ψ CdebugDlg
Timer
Network Classes
* Cmessage
* Cnetwork
Parameterized Classes
* GenList
Windows / User Interface Classes
ψ CaboutDlg
ψ CooveApp
ψ CooveDlg

6 Class Reference

In this section, a member by member and method by method account of the implementation of each class is presented. This section serves as a guide to the programmer attempting to understand the source code, as well as providing a detailed description of some of the important and non-trivial data structures and algorithms which have been used in the implementation of the software. It is intended that the discussions given below be read in close conjunction with the actual source code, since there is a direct correlation between the two. In addition, annotations in the form of comments are liberally scattered throughout the source code, and these complement the material presented here. In order to determine the physical location of the relevant source code for a particular class, the reader is referred to the module diagrams presented in the High Level Software Design (WB 105, Section 3.3.1), which illustrate the physical architecture of the system, and the Program Code Fragments List contained in the Master Document List (WB 001, Section 3).

6.1 class `Agent`

6.1.1 Protected Data Members

`int myPID`

The process identifier (or process ID, or PID) of the agent or of a process derived from the agent. This attribute is inherited by any agent class that is derived from the `Agent` class. Furthermore, any process class subsequently derived from such an agent class in turn also inherits this data member. Each instantiated system process is assigned a unique process ID. Any agents for a particular process that exist will be assigned the same process ID as the process itself. In the case of an agent, therefore, this value should be interpreted as the ID of that process with which the agent is associated.

`EnvironmentManager* parentEM`

A pointer to the local Environment Manager (EM). There is a single instance of the `EnvironmentManager` class on each machine. An agent uses this pointer to call on the Environment Manager to forward messages that it receives to the appropriate remote authentic process. Each process class also inherits this data member, and uses it to obtain from the EM the identity of another process to which it wishes to send a message. (As will be discussed in Section 6.4, the Environment Manager maintains pointers to all processes and agents residing on a particular machine).

6.1.2 Public Member Functions

`Agent(int PID, EnvironmentManager* parent)`

Parameters Taken: The process ID to assign to the agent (or process) being constructed, and a pointer to the local Environment Manager.

Return Value: None. This is a constructor.

Discussion: The constructor simply initialises the `myPID` and `parentEM` data members (see section 6.1.1) to the values specified.

6.2 class `Process`

6.2.1 Data Members

None.

6.2.2 Public Member Functions

`virtual void RecvMessage(CString* message)`

Parameters Taken: The message to be received. The message is passed in the form of a string which will be decoded to recreate the original function call.

Return Value: None.

Discussion: This pure virtual function provides a template for the Remote Message Handler (see Section 3.6.2 of the Conceptual System Design WB 104) of any process class derived from the `Process` class. The `Process` class is an abstract base class, and therefore contains no implementation code of its own for this function. The function is inherited and redefined by any derived process class.

6.3 class FunctionCall

6.3.1 Private Data Members

`int functionID`

The function identifier (or function ID, or FID) of the function call. Each member function of a class is assigned a function identifier, which is unique within that class. This function ID is used to unambiguously determine which member function is being called when a function call is made. The function ID is assigned by the

process which creates the function call object, and is initialised upon instantiation.

`Parameter paramOrder[ ]`

An array containing the parameter type for each of the parameters of the function call, in sequence. Specifically, the value in position 0 in the array represents the type of the first parameter passed, the value in position 1 represents the type of the second parameter, and so on. The values in this array are of the enumerated type `Parameter` (refer to Section 4.1)

`int paramMapping[ ]`

An array which maps the parameters as listed in the `paramOrder` array into the individual parameter arrays, differentiated by type (see the `<yyy>Count` and `<yyy>ParamList` data members below). For example the parameter sequence:

int, int, char, string, int, char

is mapped as:

1, 2, 1, 1, 3, 2

which means that:

the first parameter is the first parameter of type integer,
the second parameter is the second parameter of type integer,
the third parameter is the first parameter of type char
the fourth parameter is the first parameter of type string
the fifth parameter is the third parameter of type integer
the sixth parameter is the first parameter of type char

```
int intCount
int stringCount
int charCount
int doubleCount
int rwRealCount
int uINTCount
```

The total number of each type of parameter that is present in the function definition.

`int paramCount`

The total number of parameters present in the function definition.

ParamCount = intcount+ stringCount+ ...+ uINTCount

```
int intParamList[ ]
CString* stringParamList[ ]
char charParamList[ ]
double doubleParamList[ ]
RwReal rwRealParamList[ ]
UINT uINTParamList[ ]
```

The individual parameter arrays, which contain the actual function call parameters, differentiated by type. The six data types that are currently supported by the software are integers, strings, characters, double precision floating point numbers, the *RenderWare* floating point data type, and unsigned integers (see also the definition of the `Parameter` enumerated type in section 4.1). All of the parameters of each particular type are stored in the respective corresponding array.

```
CString* fnCallStr
```

The string into which the function call is encoded. When retrieved by means of the `Get()` function, this string is sufficient to fully specify the function call, and it is this string that is eventually transferred across the network. At the receiving end, when the `FunctionCall` is constructed using its string representation, the supplied string is stored in this variable while it is being decoded.

6.3.2 Public Member Functions

```
~FunctionCall()
```

Parameters Taken: None. This is the destructor.

Return Value: None. This is the destructor.

Discussion: The destructor frees up the memory that has been allocated for the string representation of the function call, as well as for those parameters that are strings. All other types of parameters besides strings do not need to specifically be destroyed, since memory for these other data types is allocated statically.

[Note: The following member functions are used in creating a function call on the sending side. The functions in question are: the constructor which takes only the function ID as parameter; the `AddParam<Yyy>()` functions; and the `Get()` function.]

```
FunctionCall(int fid)
```

Parameters Taken: The function identifier (function ID or FID) of the function call being constructed.

Return Value: None. This is a constructor.

Discussion: This constructor assigns the function call the function ID which has been specified, and initialises the parameter count variables to zero. (When a function call is constructed in this manner, initially none of its parameters are specified. The parameters must be added subsequently by means of the `AddParam<Yyy>()` functions discussed below). In addition, the constructor initialises the `fnCallStr` string, into which the function call will be encoded during a *get*.

```
void AddParamInt(int param)
void AddParamString(char* param)
void AddParamChar(char param)
void AddParamDouble(double param)
void AddParamRwReal(RwReal param)
void AddParamUINT(UINT param)
```

Parameters Taken: Exactly one parameter of the specified type.

Return Value: None.

Discussion: Each of these functions adds the relevant parameter to the internal representation of the function call. The overall parameter count, as well as the individual parameter count of the type in question are incremented, and the parameter is added to the relevant parameter list array. The paramOrder and paramMapping arrays are updated for later use by the `Get()` function, in the manner which has already been described. Note that it is important that these functions be called in the order in which the parameters appear in the function call definition.

```
CString* Get()
```

Parameters Taken: None.

Return Value: A string representation (in the form of a pointer to a `CString` object) which fully specifies the function call.

Discussion: This function is used to "get" a string representation of the function call once it has been built using the above constructor and `AddParamYyy()` functions. This function *marshals* or encodes the arguments which have been specified, into the form of a character string (refer to `FunctionCall` class responsibilities in the High Level Design, WB 105, section 3.2.1.2.2). The string which is returned forms the body of the message representing the function call that will be sent across the network.

The process of marshalling of the function call into its string representation proceeds as follows:

The number of parameters (`paramCount`) is written to the `fnCallStr` string, followed by a block which *describes* the parameters: For each parameter, the parameter type and the start of that (encoded) parameter as an index into the string are specified. Following this block, the actual encoded parameters are added to the string. Finally, the function ID is *prepended* to the string. The final format of the returned string is as follows:

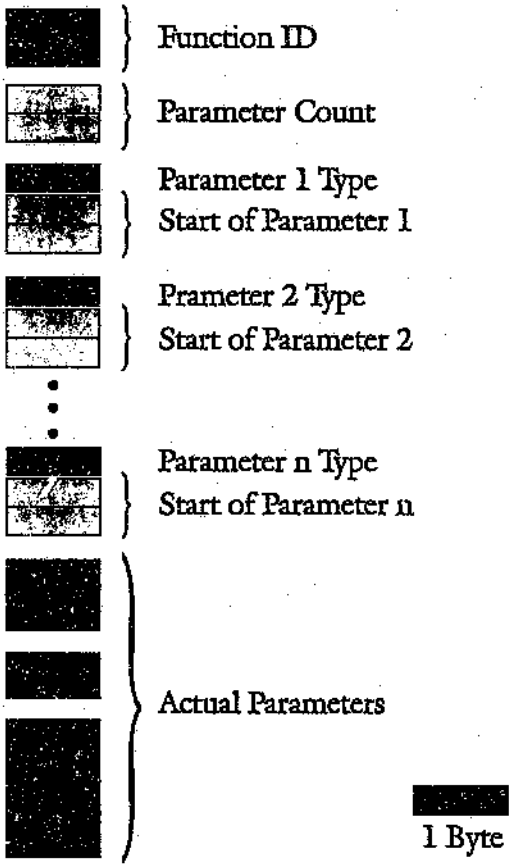


Figure WB113–1: Format of the String Returned by the `Get()` Function

[Note: The following member functions are used in decoding a function call on the receiving side. The functions in question are: the constructor which takes a string representation of the function call as a parameter; the `GetFID()` function; and the `GetParam<Yyy>()` functions.]

```
FunctionCall(CString* fncallstring)
```

Parameters Taken: A string representation (in the form of a pointer to a `CString` object) which fully specifies the function call to be constructed, in the format specified in Figure WB113–1.

Return Value: None. This is a constructor.

Discussion: This constructor creates a `FunctionCall` object based on the information contained in the supplied string. Once constructed, the `FunctionCall` is in the same form as it was before it was marshalled on the sending side by the `Get()` function. This constructor therefore performs exactly the reverse operation to the `Get()` function.

```
int GetFID()
```

Parameters Taken: None.

Return Value: The function ID which identifies the function call.

Discussion: This function simply returns the function ID, and is used by the `RecvMessage()` function of a process class in order to determine which function is to be called.

```
void GetParamInt(int paramnum, int& param)
void GetParamString(int paramnum, char* param)
void GetParamDouble(int paramnum, double& param)
void GetParamChar(int paramnum, char& param)
void GetParamRwReal(int paramnum, RwReal& param)
void GetParamUINT(int paramnum, UINT& param)
```

Parameters Taken: The sequence number of the parameter to be retrieved, and a reference to the variable of the designated type which is to hold the retrieved parameter.

Return Value: Each of these functions returns (by means of the reference parameter) the function call parameter specified by `paramnum`. The one exception is in the case of a string, where the contents of a *pointer* to the string are updated by the function.

Discussion: These functions are used to retrieve the parameters for the function call from the function call object. In each case, the appropriate parameter is simply returned by means of the reference parameter (or pointer, in the case of strings) `param`.

6.4 class InputHandlerAgent

6.4.1 Data Members

None.

6.4.2 Public Member Functions

```
InputHandlerAgent(int PID, EnvironmentManager* parent)
```

Parameters Taken: The Process ID to assign to the Input Handler agent being constructed and a pointer to the local Environment Manager

Return Value: None, since this is a constructor.

Discussion: The constructor of the parent class `Agent` is called, passing through the supplied parameters unmodified. (All of the data members that are to be initialised are inherited from the `Agent` class).

```
virtual void OnKeyUp(UINT nChar, UINT nRepCnt,
                     UINT nFlags)
virtual void OnKeyDown(UINT nChar, UINT nRepCnt,
                       UINT nFlags)
virtual void OnRButtonUp(UINT nFlags, CPoint point)
virtual void OnRButtonDown(UINT nFlags, CPoint point)
virtual void OnLButtonUp(UINT nFlags, CPoint point)
virtual void OnLButtonDown(UINT nFlags, CPoint point)
virtual void OnMouseMove(UINT nFlags, CPoint point)
```

These `InputHandlerAgent` functions contain no code at present, since the Input Handler is not required to receive messages from any remote process. If support for feedback to the Input Handler becomes a requirement, then these functions will need to be populated with code that marshals the function arguments and sends the functioncall over the network via the Environment Manager to the associated remote `InputHandler` process.

6.5 class InputHandler

6.5.1 Protected Data Members

```
CWnd* inputDlg
```

A pointer to the window from which mouse input is to be received. This is the window over which the user must move the mouse with one of the mouse buttons depressed in order to navigate. In the current implementation of the system this refers to the main application dialog

box. The `inputDlg` pointer is primarily used for *capturing* the mouse movement, so that when a mouse button is pressed within the dialog box, the mouse is tracked until the button is released, regardless of whether the mouse has moved outside the window.

```
RwBool lookAround
RwBool moveAround
```

Boolean flags indicating the navigation *mode* that is currently being used. These two flags are mutually exclusive, since only one navigation mode can be active at any particular moment. The *look around* mode is activated by the right mouse button, and allows for control of viewpoint orientation and lateral translation. The *move around* mode allows for control of lateral and forward-backward translation. Refer to WB 107, the User Reference Manual, for a more complete discussion of navigation modes. Navigation mode 1 referred to in WB 107 corresponds to the *move around* mode, while navigation mode 2 is synonymous with the *look around* mode.

```
int previousMousex
int previousMousey
```

These variables are used to maintain the previous x and y values of the mouse. The increment by which the mouse has been moved is able to be calculated by comparison of the current mouse position with these previously stored values. The values are updated when either of the mouse buttons is depressed, and after each detected mouse movement that occurs while a mouse button is down.

6.5.2 Public Member Functions

```
InputHandler(int PID, EnvironmentManager* parent,
             CWnd* dlg)
```

Parameters Taken: The Process ID to assign to the Input Handler process being constructed, a pointer to the local Environment Manager, and a pointer to the dialog box from which mouse input will be received.

Return Value: None, since this is a constructor.

Discussion: The constructor of the parent class `InputHandler-Agent` is called, supplying the first two parameters unmodified. (The `InputHandlerAgent` constructor in turn calls the constructor of the base class, `Agent`, in order to initialise the inherited data members). Thereafter, the `inputDlg` data member is initialised to point to the window specified by `dlg`.

```
void RecvMessage(CString* message)
```

This function contains no code at present, since in the current implementation, the Input Handler does not receive messages from other system processes. If support for feedback to the Input Handler becomes a requirement, then this function will need to be populated with code that decodes remotely originating function calls received via the Environment Manager, and invokes the appropriate `InputHandler` function.

```
virtual void OnKeyDown(UINT nChar, UINT nRepCnt,
                       UINT nFlags)
```

Parameters Taken: The virtual-key code, repeat count and flags which characterise the keyboard key that has been pressed, as supplied to the `CWnd::OnKeyDown()` function, which calls this function. The latter two parameters are not used in the current implementation.

Return Value: None.

Discussion: This function reacts to various keys on the keyboard being depressed. It is through this function that navigation within the application by means of the keyboard is facilitated.

The keys which are monitored and the interpreted meaning in each case are as follows:

Key Code	Key pressed	Interpretation (based on current navigation mode)	
		Navigation Mode 1	Navigation Mode 2
80	`P` Key	Move observer forward	increase view orientation pitch angle
186	`Semicolon (;)` Key	Move observer backward	decrease view orientation pitch angle
76	`L` Key	Move observer left	Move observer left
222	`Apostrophe (')` Key	Move observer right	Move observer right
17	Ctrl Key	navigation mode change from Mode 1 to Mode 2	N/A (Being in this mode implies that the Ctrl key is already being pressed)

The status of the Ctrl key is used to determine the currently selected mode of navigation. The two navigation modes which are provided are described in the User Reference Manual (WB 107). The default navigation mode, corresponding to the Ctrl key not being depressed, is Mode 1. When the Ctrl key is depressed, the navigation mode is changed to Mode 2.

```
virtual void OnKeyUp(UINT nChar, UINT nRepCnt,
                     UINT nFlags)
```

Parameters Taken: The virtual-key code, repeat count and flags which characterise the keyboard key that has been released, as supplied to the `CWnd::OnKeyUp()` function, which calls this function. The latter two parameters are ignored in the current implementation.

Return Value: None.

Discussion: This function simply checks to see whether the "`Ctrl`" key has been released (virtual key code=17), and changes the navigation mode from Mode 2 to Mode 1 accordingly. (For an explanation of the two navigation modes which are used in the program, refer to the User Reference Manual, WB 107).

```
virtual void OnRButtonDown(UINT nFlags, CPoint point)
```

Parameters Taken: A "flags" value indicating which keyboard function keys were depressed, and the x and y coordinates of the mouse cursor, at the moment when the right mouse button was depressed. The coordinates are relative to the upper-left corner of the application window. The first parameter is ignored in the current implementation, since keyboard input coupled with mouse movement is not used.

Return Value: None.

Discussion: This function simply changes the navigation mode to Mode 2, or 'look around' mode in response to the right mouse button being depressed. The mouse is *captured*, so that mouse movement will be tracked until the right button is released, regardless of whether the mouse has moved outside the application window. Also, the coordinates of the mouse pointer are stored, for use in determining incremental mouse movements, as described in Section 6.5.1 above.

```
virtual void OnRButtonUp(UINT nFlags, CPoint point)
```

Parameters Taken: A "flags" value (ignored) and the x and y coordinates of the mouse cursor (see `OnRButtonDown` above), at the moment when the right mouse button was released.

Return Value: None.

Discussion: When the right mouse button is released, Mode 2 or 'look around' navigation as well as the capture of the mouse are cancelled.

```
virtual void OnLButtonDown(UINT nFlags, CPoint point)
```

Parameters Taken: A "flags" value and the x and y coordinates of the mouse cursor (see `OnRButtonDown` above), at the moment when the left mouse button was depressed. The first parameter is ignored in the current implementation, since keyboard input coupled with mouse movement is not used.

Return Value: None.

Discussion: This function simply changes the navigation mode to Mode 1, or 'move around' mode in response to the left mouse button being depressed. The mouse is once again captured and the coordinates of the mouse pointer stored, as described for the `OnRButtonDown` function above.

```
virtual void OnLButtonUp(UINT nFlags, CPoint point)
```

Parameters Taken: A "flags" value (ignored) and the x and y coordinates of the mouse cursor (see `OnRButtonDown` above), at the moment when the left mouse button was released.

Return Value: None.

Discussion: When the left mouse button is released, Mode 1 or 'move around' navigation as well as the capture of the mouse are cancelled.

```
virtual void OnMouseMove(UINT nFlags, CPoint point)
```

Parameters Taken: A "flags" value (ignored) and the new x and y coordinates of the mouse cursor.

Return Value: None.

Discussion: It is via this function that the program responds to movement of the mouse. The semantics of the user's interaction with the system are determined by which of the mouse buttons are currently depressed, as indicated by which navigation mode is currently active. This function simply returns if neither mouse button is depressed.

The function firstly determines the incremental movement of the mouse since the last time the position of the mouse was recorded in the `previousMousex` and `previousMousey` variables. Based on this information and the currently active navigation mode, the amount of

pan, tilt and translation of the virtual camera that should result from the mouse movement that has occurred are calculated. Two constants, `ROTATEINCREMENT` and `MOUSENSITIVITY` influence the proportional relationship between mouse movement and the resultant effect on the camera. A pointer to the Presentation Generator (or Presentation Generator Agent) is obtained from the Environment Manager, and the `PanCamera()`, `TiltCamera()` and `TranslateCamera()` functions are invoked. The new mouse coordinates are then once again stored using the `previousMousex` and `previousMousey` variables.

6.6 class `ObjectAgent`

6.6.1 Data Members

None.

6.6.2 Public Member Functions

```
ObjectAgent(int PID, EnvironmentManager* parent)
```

Parameters Taken: The Process ID to assign to the Object agent being constructed and a pointer to the local Environment Manager

Return Value: None, since this is a constructor.

Discussion: The constructor of the parent class `Agent` is called, passing through the supplied parameters unmodified. (All of the data members that are to be initialised are inherited from the `Agent` class).

```
virtual void Scale(RwReal sx, RwReal sy, RwReal sz)
virtual void Rotate(RwReal rx, RwReal ry, RwReal rz,
                    RwReal angle)
```

Parameters Taken: The parameters taken by these functions are identical to those taken by the respective functions in the Object class.

Return Values: None.

Discussion: The semantics of the implementation of both of these functions, as well as most of the functions of the other agent classes, are very much the same. Therefore a prototypical example is provided here, and referred to elsewhere in this document.

Each function simply marshals the supplied parameters into a string representation of the function call using the `FunctionCall` class, and then sends the message to the Environment Manager for delivery to the counterpart remote process. A sample of the code used to do this is provided below for the purposes of discussion

```
void ObjectAgent::Scale(RwReal sx, RwReal sy, RwReal sz)
{
   FunctionCall fn(2);  // This is function 2
   fn.AddParamRwReal(sx);
   fn.AddParamRwReal(sy);
   fn.AddParamRwReal(sz);
   parentEM->SendMessage(myPID, fn.Get(), FALSE);
};
```

In this example, the function call in question, `Scale()`, has been assigned a function ID of 2, being the second function defined for the class `Object/ObjectAgent`. An "empty" `FunctionCall` object is constructed, using the function ID as a parameter. Each of the supplied parameters are then added to the FunctionCall using the *`AddParam`* functions. In this particular case, all three of the supplied parameters are of the type `RwReal`, and therefore the `AddParamRwReal` function is called three times consecutively. Finally, the Environment Manager function `SendMessage()` is called, supplying it with the following parameters: the process ID associated with this agent process; a string representation of the function call obtained from the FunctionCall object by means of the `Get()` function; and a boolean flag indicating whether this is an event message (requiring guaranteed delivery). The intended usage of the `FunctionCall` class is described in more detail in Section 6.3.

6.7 class `Object`

6.7.1 Protected Data Members

`RwBool inUse`

A boolean variable used to control synchronisation and mutually exclusive access to the object. This is required because more than one thread may simultaneously attempt to access the object. To prevent corruption of the object's data, only one thread at a time should be allowed access. If the object is in use, the second thread must wait until it becomes available before proceeding. This flag indicates whether the object is in use or not.

6.7.2 Public Member Functions

```
void Scale(RwReal sx, RwReal sy, RwReal sz)
```

Parameters Taken: Scale factors to be used for scaling of the object in the x, y and z directions respectively. Each scale factor is a number between 0 and 1, and indicates the fraction of the *original* size to which the object is to be scaled. Note that for computational accuracy and to avoid cumulative approximation errors, subsequent scaling operations are performed with reference to the *original* object, and not a previously scaled version.

Return Value: None.

Discussion: This function performs any local scaling operation on the object that is required and then sends a message to the Presentation Generator to scale its representation of the object, through a call to the `ScaleObjRepresentation()` function. (In the current implementation, the local scaling operation is has been provided for but omitted). In order to access the Presentation Generator, the object obtains a pointer to it (or to its agent) from the Environment Manager.

In order to control mutually exclusive access by more than one thread, this function initially waits until the `inUse` flag has a value of `FALSE`, forcing the calling thread to wait until the object is exclusively available. It then sets the flag to `TRUE` indicating that the object is now in use (other threads wishing to access the object must now wait until it becomes available again). Finally, just before the function ends, the flag is again set to `FALSE`, freeing up the object once more.

```
void Rotate(RwReal rx, RwReal ry, RwReal rz,
            RwReal angle)
```

Parameters Taken: The x, y and z components of the axis of rotation, and the angle of rotation. The first three parameters give the cartesian components of a vector representing the axis around which the rotation is to occur. The units used are not significant, only the relative size of each component. The angle of rotation must be specified in degrees.

Return Value: None.

Discussion: This operation is very similar to the scaling operation described in the immediately preceding section. As might be expected though, the `RotateObjRepresentation()` function of the `PresentationGenerator` class is called in this case. The approach adopted with regard to thread safety and mutually exclusive access to the object is identical to that used in the former case.

`Object(int PID, EnvironmentManager* parent)`

Parameters Taken: The Process ID to assign to the Object process being constructed, and a pointer to the local Environment Manager.

Return Value: None, since this is a constructor.

Discussion: The constructor of the parent class `ObjectAgent` is called with the parameters unmodified. (The `ObjectAgent` constructor in turn calls the constructor of the base class, `Agent`, in order to initialise the inherited data members). The `inUse` flag is set to false, indicating that the object process is initially not in use.

`void RecvMessage(CString* message)`

Parameters Taken: A string representation of a functioncall invoked by a remote process, as delivered by the Environment Manager.

Return Value: None.

Discussion: The `RecvMessage()` function serves as the remote message handler of the Object process. It does not contribute to the application-oriented functionality of the class. Its function is to decode any remotely originating messages that are received from the Environment Manager, and invoke the appropriate function accordingly.

In order to decode the received message, this function constructs a FunctionCall object, supplying the message as a parameter. The FunctionCall object then decodes the function ID and the parameters, and makes them available via the `GetFID()` and `GetParamyyy()` functions, as described in Section 6.3 above. After identifying the target function that is to be called and obtaining the parameters, the `RecvMessage()` function simply invokes the function.

6.8 class `PresentationGeneratorAgent`

The semantics of the implementation of this class are the same as for the class ObjectAgent. The particulars are slightly different, but the approach adopted is identical. Therefore the reader is referred to the discussion provided in Section 6.6.

6.9 class PresentationGenerator

6.9.1 Protected Data Members

RwBool inUse

A boolean variable used to control synchronisation and mutually exclusive access to the PresentationGenerator. This is required because more than one thread may simultaneously have access to an instance of this class. To prevent corruption of data, access by only one thread at a time should be permitted. If the PresentationGenerator is in use, the second thread must wait until it becomes available before proceeding. This flag therefore indicates whether the object is in use or not.

CWnd* outputDlg

A pointer to the window to which rendered output will be sent. In the current implementation this points to the main application dialog box. This data member is initialised upon construction of the PresentationGenerator object by the Environment Manager.

RwCamera* camera

This data member is a pointer to the RenderWare data type RwCamera, and represents the virtual "camera" which is used to render the current frame (see the function PresentationGenerator::DrawCamera-View() below).

RwScene* scene

This data member is a pointer to the RenderWare data type RwScene, and represents the "scene" which is to be rendered. There is exactly one scene associated with each PresentationGenerator object. It is this scene which constitutes the geometric space in which the user interacts with the program. The scene contains a number of lights and a number of clumps (geometric "renderable" objects) both of which are discussed below.

RwLight* lights[2]

This data member is an array of pointers to the RenderWare data type RwLight. The elements of this array point to the light source objects which RenderWare uses to calculate the illumination of the scene. In the current implementation, there are two light sources in the scene. The first light is a point light source, which provides ambient illumination and the second source is a directional light source which specifically illuminates the objects of interest. These light objects are instantiated

within the scope of the `PresentationGenerator` constructor, discussed in Section 6.9.2.

[Note: the following three data members are of the RenderWare datatype `RwClump`, which is defined in the RenderWare API Reference [Renderware API, 1995] as "a collection of polygons and vertices". Each software object of this type represents a geometric object in the rendered scene. A scene rendered by RenderWare consists of a number of such clumps, illuminated by one or more light sources such as those described above.]

`RwClump* map`

This RenderWare clump represents the "map" on which the other objects reside. In the current implementation, the map is not an active object—it does not participate in the simulation, nor are any of its parameters modified during runtime. It may therefore be regarded as merely providing a "backdrop" for displaying the other objects of interest. It should, however, be noted that the map does not exist purely for aesthetic reasons. Pausch [1991] indicates the importance in VR systems of displaying a ground plane which provides a frame of reference for the other objects which are being displayed. The presence of a ground plane helps the user to orient his world view appropriately, contributing to the meaningful representation of the remainder of the scene.

`RwClump* clumps[]`

The clumps in this array represent the objects of interest that are displayed on the map. Each clump corresponds to one of the objects on the map. These are active objects in that they respond to input from the user and from the simulation.

`RwClump* origclumps[]`

This array initially mirrors the above array. Its purpose is to maintain a record of the original state of each of the clumps in the scene. For reasons of preserving accuracy, each subsequent state of a clump is calculated using the original state as a reference point. This avoids the cumulative approximation error associated with successively computing new states from slightly inaccurate previous states.

6.9.2 Public Member Functions

```
PresentationGenerator(int PID,
                      EnvironmentManager* parent,
                      CWnd* dlg)
```

Parameters Taken: The process ID to assign to the `PresentationGenerator` process being constructed, a pointer to the local Environment Manager, and a pointer to the dialog box to which the rendered output will be directed.

Return Value: None. This is a constructor.

Discussion: The constructor of the parent class `PresentationGeneratorAgent` is called, supplying the first two parameters unmodified. (The `PresentationGeneratorAgent` constructor in turn calls the constructor of the base class, `Agent`, in order to initialise the inherited data members). The `outputDlg` data member is initialised to point to the window specified by `dlg`.

Thereafter, the constructor performs a number of rendering-related initialisation tasks. The RenderWare virtual camera is created, and its initial location, as well as other aspects such as lens characteristics are specified by means of the appropriate RenderWare API calls. The scene is created, and the light sources are defined. Finally, the necessary clump objects are created and placed in their appropriate initial locations in the scene.

```
~PresentationGenerator()
```

Parameters Taken: None. This is a destructor.

Return Value: None. This is a destructor.

Discussion: The `PresentationGenerator` destructor deallocates the memory that was allocated by the constructor during instantiation. In particular, this involves deleting the clump, light and scene objects that were created.

```
void RecvMessage(CString* message)
```

The remote message handlers of each of the process classes are all implemented in essentially the same manner. Therefore, for these details the reader is referred to the discussion of the `Object::RecvMessage()` function in Section 6.7.2.

`void DrawCameraView()`

Parameters Taken: None.

Return Value: None.

Discussion: This function causes the Presentation Generator to re-render the view. The rendered frame rate is thus directly determined by how often this function is executed. The function firstly obtains the window handle of the output window and creates a device context for that window. RenderWare API calls are then used to initiate an update of the virtual camera, and then render the scene to the output window. This function is called after transformations have been made to the objects in the scene and the camera by calls to the `TranslateCamera()`, `PanCamera()`, `TiltCamera()`, `RotateObjRepresentation()` and `ScaleObjRepresentation()` functions.

```
virtual void TranslateCamera(RwReal x, RwReal y,
                             RwReal z)
```

Parameters Taken: The x, y and z values by which the camera is to be translated.

Return Value: None.

Discussion: This function simply translates the camera by the specified amounts by means of the RenderWare API function `RwVCMoveCamera()`, and then calls the `DrawCameraView()` function to update the display.

```
virtual void PanCamera(RwReal angle)
```

Parameters Taken: The angle of pan in degrees.

Return Value: None.

Discussion: This function simply pans the camera through the specified angle by means of the RenderWare API function `RwPanCamera()`, and then calls the `DrawCameraView()` function to update the display.

```
virtual void TiltCamera(RwReal angle)
```

Parameters Taken: The angle of tilt in degrees.

Return Value: None.

Discussion: This function simply tilts the camera by the specified angle. A rotation matrix is created based on the supplied angle, and is

then applied to the camera by means of the RenderWare API function `RwTransformCamera()`. The `DrawCameraView()` function is then called to update the display.

```
virtual void OnPaint()
```

Parameters Taken: None.

Return Value: None.

Discussion: The Presentation Generator is responsible for dealing with the `OnPaint()` function of the output dialog box. As a result, the `CWnd::OnPaint()` function is set up to simply redirect any calls that it receives to this function. The current implementation of this function involves simply re-rendering the scene through a call to `PresentationGenerator::DrawCameraView()`.

```
virtual void OnSize(UINT nType, int cx, int cy)
```

Parameters Taken: The type of resize operation that has been performed and the new client area of the output window.

Return Value: None.

Discussion: The Presentation Generator is responsible for dealing with the `OnSize()` function of the output dialog box. As a result, the `CWnd::OnSize()` function is set up to simply redirect any calls that it receives to this function. The implementation of this function involves resetting the RenderWare camera viewport and lens characteristics to reflect the new output window parameters, and then re-rendering the scene through a call to `PresentationGenerator::DrawCameraView()`

```
virtual void RotateObjRepresentation(int objID,
                                     RwReal rx,
                                     RwReal ry,
                                     RwReal rz,
                                     RwReal angle)
```

Parameters Taken: A unique ID identifying which object representation is to be rotated, the x, y and z components of the axis of rotation, and the angle of rotation in degrees. In the current implementation, the `objID` parameter is the process ID of the object process that calls this function.

Return Value: None.

Discussion: A rotation matrix is created representing the rotation operation that is to be performed. This matrix is then applied to the

clump representing the specified object. Finally, the view is re-rendered by calling `PresentationGenerator::DrawCameraView()`.

```
virtual void ScaleObjRepresentation(int objID,
                                    RwReal sx,
                                    RwReal sy,
                                    RwReal sz)
```

Parameters Taken: A unique ID identifying which object representation is to be scaled, and the x, y and z direction scale factors to be used. In the current implementation, the `objID` parameter is the process ID of the object process that calls this function.

Return Value: None.

Discussion: The clump representing the specified object is firstly reverted back to its original dimensions, as recorded in the `origclumps` array. A transformation matrix is built representing the scaling operation that is to be performed. This matrix is then applied to the clump. Finally, the view is re-rendered by calling `PresentationGenerator::DrawCameraView()`.

6.10 class `SimulationAgent`

The semantics of the implementation of this class are the same as for the class ObjectAgent. The particulars are slightly different, but the approach adopted is identical. Therefore the reader is referred to the discussion provided in Section 6.6.

6.11 class `Simulation`

6.11.1 Private Data Members

`int valuesCount`

The number of simulation values contained in the `simulate.cfg` configuration file. This value appears as the first entry in the `simulate.cfg` file, and indicates how many simulation steps are represented by the data contained in the rest of the file. This value is read in from the file and stored in this data member.

`RwReal* values`

An array of floating point numbers which is used to hold the simulation values read in from the `simulate.cfg` configuration file. The size of

this array is determined dynamically; memory is allocated according to the value of the `valuesCount` data member discussed above.

`RwBool threadflag`

A boolean value indicating whether the simulation thread is currently running. When the simulation thread is started, `threadflag` is set to `TRUE`. The thread continues to run as long as the value of `threadflag` remains true. This variable can therefore be used to initiate a shutdown of the thread by setting it to `FALSE`. (This is done by the `Terminate` function as discussed below). This variable is set back to `TRUE` by the thread when termination of the thread has been completed.

6.11.2 Public Member Functions

`Simulation(int PID, EnvironmentManager* parent)`

Parameters Taken: the process ID to assign to the Simulation process being constructed, and a pointer to the local Environment Manager.

Return Value: None, since this is a constructor.

Discussion: The constructor of the parent class `SimulationAgent` is called with the parameters unmodified. (The `SimulationAgent` constructor in turn calls the constructor of the base class, `Agent`, in order to initialise the inherited data members). Thereafter, the constructor sets the `threadflag` flag, and starts up the simulation thread, by invoking the `Simulation::Simulate()` member function (documented below), which is the controlling function for the simulation thread.

`void Simulate()`

Parameters Taken: None.

Return Value: None.

Discussion: This member function is the controlling function for the simulation thread.

The thread initially opens the `simulate.cfg` configuration file, from which the `valuesCount` data member is read. The `values` array is initialised to the correct size to house the simulation values, and these values are read in from the `simulate.cfg` file.

The thread then waits for an initial time period before beginning the simulation, to ensure that the other local processes, and agents for

remote processes have been instantiated by the Environment Manager. It is important that all other *local* processes (including agent processes) exist when the simulation begins—messages sent to other local processes by the thread will cause unpredictable behaviour if those processes do not yet exist. The simulation need not wait for all *remote* processes to be instantiated, though. As long as all local *agent* processes have been instantiated, and the local and remote *Network* processes are running[1], messages sent to a remote application process will be queued by the remote network process until the target process is ready to receive them.

The actual simulation takes the form of an "infinite" loop, which proceeds until the Environment Manager initiates shutdown of the thread by calling the `Terminate` function. The loop repetitively sends messages to the objects in the geometric model, instructing them to scale to values commensurate with those contained in the `values` array, which represents the sampled simulation output variation over time. After each cycle of the loop, the simulation introduces a delay which determines the sampling interval between successive simulation steps. In a full scale system, part of this delay would be attributable to the computation necessary to calculate the next simulation values.

`void Terminate()`

Parameters Taken: None.

Return Value: None.

Discussion: This member function is called to initiate shutdown of the simulation thread. The `threadflag` variable (discussed above) is used to signal to the thread that it should terminate. The thread controlling function continuously monitors this flag to determine whether it should shut down. To end the thread, therefore, this function simply sets the flag to `FALSE`. It then waits for the thread to set the flag back to `TRUE` (indicating that the thread has been successfully shut down) before returning.

`void RecvMessage(CString* message)`

The remote message handlers of each of the process classes are all implemented in essentially the same manner. Therefore, for these

[1] This is always the case upon instantiation of the Simulation process—refer to the discussions regarding the startup mechanism of the Environment Manager in Section 6.12.2.

details the reader is referred to the discussion of the `Object::RecvMessage()` function in Section 6.7.2.

6.12 class `EnvironmentManager`

6.12.1 Private Data Members

`CWnd* parentDlg`

This data member is used to maintain a pointer to the window in which the application runs. This pointer is needed by both the Input Handler and the Presentation Generator: it specifies, respectively, the window from which input is to be received, and the window to which output should be directed. The Environment Manager maintains this pointer since it is the EM's responsibility to instantiate the `InputHandler` and `PresentationGenerator` objects. As depicted in Figure WB113–2, this data member is supplied by the `COoveDlg` object upon instantiation of the EM, and the respective pointers in the Input Handler and Presentation Generator are in turn initialised when these processes are instantiated by the EM.

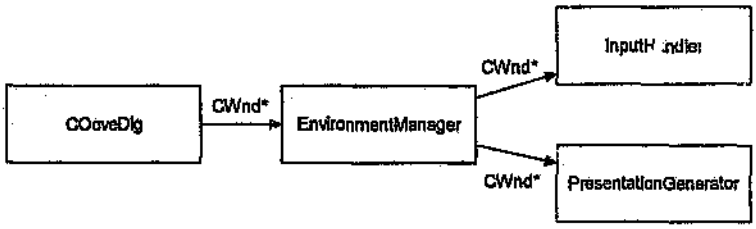


Figure WB113–2: Initialisation of `CWnd*` Data Members Upon Instantiation

`CNetwork* network`

A pointer to the local network communications layer object. There exists exactly one `CNetwork` object on each machine, and the Environment Manager maintains a record of its identity by means of this data member. It is the responsibility of the Environment Manager to instantiate the network, and destroy it upon termination of the program.

`int machineCount`

The number of machines in the current hardware configuration of the system. In the current implementation, this value is obtained from the `address.cfg` configuration file. For details of the format of this file, refer to the User Reference Manual (WB 107).

`int myMachineNo`

An index into the list of machine addresses indicating which machine is the local machine. In the current implementation of the system, this value is obtained from the `address.cfg` configuration file, which reflects the hardware configuration that is being used. It is by means of this data member and the information contained in the `process.cfg` configuration file that process responsibilities are resolved by the Environment Manager.

`int processCount`

The total number of application processes comprising the system. This value represents the number of processes and agents that are to be instantiated by the Environment Manager on each machine, and includes Input Handler, Presentation Generator, Object and Simulation processes/agents. In the current implementation of the system, this value is specified in the `process.cfg` configuration file, which specifies the high level process-to-processor allocation that is being used.

`RwBool localNwExists`

A boolean value indicating whether the local network object has been instantiated and the machine address table initialised. This flag is set by the Environment Manager constructor only once the necessary local network setup has been performed. This includes reading and interpreting the address configuration file, instantiating the network, updating the network object's address table and creating an "I'm alive" message that will be sent out to all other nodes. (Refer to the discussion of the Environment Manager constructor in Section 6.12.2 below).

`RwBool networkExists`
`RwBool networkKnowsWeExist`

Boolean flags used to indicate respectively whether all remote nodes are up and running, and whether all remote nodes are aware that the local node is up and running.

Both flags are initially set to `FALSE`. The `networkExists` flag is set to `TRUE` once initialisation messages have been received from all other machines. The `networkKnowsWeExist` flag is set to `TRUE` once acknowledgements of the initialisation messages sent from the local node have been received from all other machines.

RwBool localProcessesExist

A boolean flag used by the EM constructor to signal to the HandleMessages thread that all local processes have been constructed, and that incoming messages destined for local processes can now be delivered.

RwBool threadflag

A boolean flag used to initiate graceful shutdown of the HandleMessages thread. The HandleMessages thread has been designed in such a way that it runs only while the value of this flag is True. When the flag is set to False, the thread terminates its processing and resets the flag to True just before it shuts down. In order to initiate shutdown of the thread, therefore, the Environment Manager simply sets this flag to False, and then waits for it to be set back to True by the thread, indicating that the thread has shut down properly.

NetworkInitStruct* nwExistsResponses

A pointer to the beginning of a dynamically allocated array of NetworkInitStruct structures that is used to record the existence status of each of the remote nodes during system initialisation. Two pieces of information are recorded for each node: whether an initialisation message has been received from that node, and whether an acknowledgement of a locally generated initialisation message has been received. For more details, refer to Section 4.1.

ProcessInitStruct* processResponsibilityList

A pointer to the beginning of a dynamically allocated array of ProcessInitStruct structures which forms a process-to-processor allocation table in which the Environment Manager stores information about each of the processes that make up the system. The information that is recorded for each process includes the process ID, process type, responsible node, and a pointer to the process if the process resides on the local machine (which is used for delivering remotely originating messages). For more details, refer to Section 4.2.

InputhList inputhList
PresgenList presgenList
ObjectList objectList
SimList simList

Linked lists of pointers to each different type of application process. The Environment Manager constructor populates these linked lists when it instantiates the various processes and agents. Each entry contains either a pointer to an agent process or a pointer to the process

itself. This polymorphic behaviour is made possible by the fact that each process class is derived from its respective agent class. The EM subsequently uses these lists to provide processes with a means for obtaining access to other processes with which they wish to communicate, by means of the `CaptureYyyList()` and `ReleaseYyyList()` functions that are described below.

6.12.2 Public Member Functions

```
InputhList* CaptureInputhList()
PresgenList* CapturePresgenList()
ObjectList* CaptureObjectList()
SimList* CaptureSimList()

void ReleaseInputhList()
void ReleasePresgenList()
void ReleaseObjectList()
void ReleaseSimList()
```

as well as

```
RwBool inputhInUse
RwBool presgenInUse
RwBool objectInUse
RwBool simInUse
```

data members

These eight functions and four data members are used to control mutually exclusive access to the various processes that are managed by the EM. At any given moment in time, the Environment Manager grants access to a particular list of processes to one and only one client process. In order to gain access to another process, the requesting process must *capture* the list of (pointers to) target processes, and *release* this list once it is finished. While captured, the list will not be made available to any other requesting process; subsequent processes must wait until the list is released before they will be able to capture it.

Guaranteeing mutually exclusive access is very important due to the multi-threaded nature of the program. As was discussed in Section 3.2 above, simultaneous access to the same data by more than one thread can lead to data corruption and unpredictable behaviour.

The list of pointers returned by the *capture* function should in each case only be assumed valid for the period during which the list remains captured. As soon as the *release* function is called, these pointers should be regarded as insecure and should not be used. These pointers should not be stored for later use. Whenever subsequent

access to a process is required, a new pointer must be obtained by calling the appropriate capture function. This will ensure mutually exclusive access, as well as allowing for the possibility of dynamic process-to-processor reallocation by the EM.

`void HandleMessages()`

The `HandleMessages()` function serves as the Environment Manager's main message-handling thread. The primary responsibilities of this thread are to poll for messages from the network, and react to received messages in the appropriate manner. Messages received by the `HandleMessages` thread are dealt with differently depending on their type. The possible message types that exist in the current implementation of the system are as follows:

Message Type	Description
0	State Update Message
1	Event Message
2	Initialisation Message

Only the first two message types (state updates and events) are relevant to the high level application processes. Messages of the third type (initialisation messages, also referred to as EM messages) are passed between the Environment Managers on each machine during initialisation of the system, as described below. Additional message types may be added at a later stage to handle other forms of communication between peer Environment Managers or messages that originate from the network.

The `HandleMessages` thread operates in one of three modes. In order to determine which mode it is currently running in, the thread monitors two flags—`localNwExists` and `localProcessesExist`—both of which are set by the Environment Manager constructor.

The first mode of operation is before initialisation of the local network has been completed. In this state, the thread simply waits for the local network setup to be completed. The thread cannot poll for messages from the network until it is sure that the network object is ready. The EM constructor signals to the thread when this is the case by setting the `localNwExists` flag. (The *handle messages* thread is intentionally started up by the constructor *prior* to the local network being set up, so that it can immediately begin receiving messages the moment it is given the signal to do so).

The second mode of operation is when initialisation messages (message type 2) are being transferred across the network. In this mode, only initialisation messages are handled. Any other messages which arrive while the thread is in this mode are discarded, since the local target processes for which they are destined have not yet been instantiated. Initialisation messages are either existence announcement messages or acknowledgement messages. The EM uses the `nwExistsResponses` response table to keep track of initialisation messages that have been received. Only when a full complement of responses has been received (all other machines have announced their presence and acknowledged) will the constructor instantiate the local processes and signal the thread to move into the third mode of operation.

The third mode of operation constitutes the system's normal running mode, and comes into effect after system-wide initialisation has been completed. In this mode, the thread simply handles incoming state update and event messages (message types 0 and 1) by passing them to the EM's `RecvMessage()` function.

A flowchart of the workings of the HandleMessages thread is provided in Figure WB113–3. The combination of the outcomes of the two decision boxes titled "Local Network Exists?" and "Still Performing Network Initialisation?" determines which of the three modes the thread is running in.

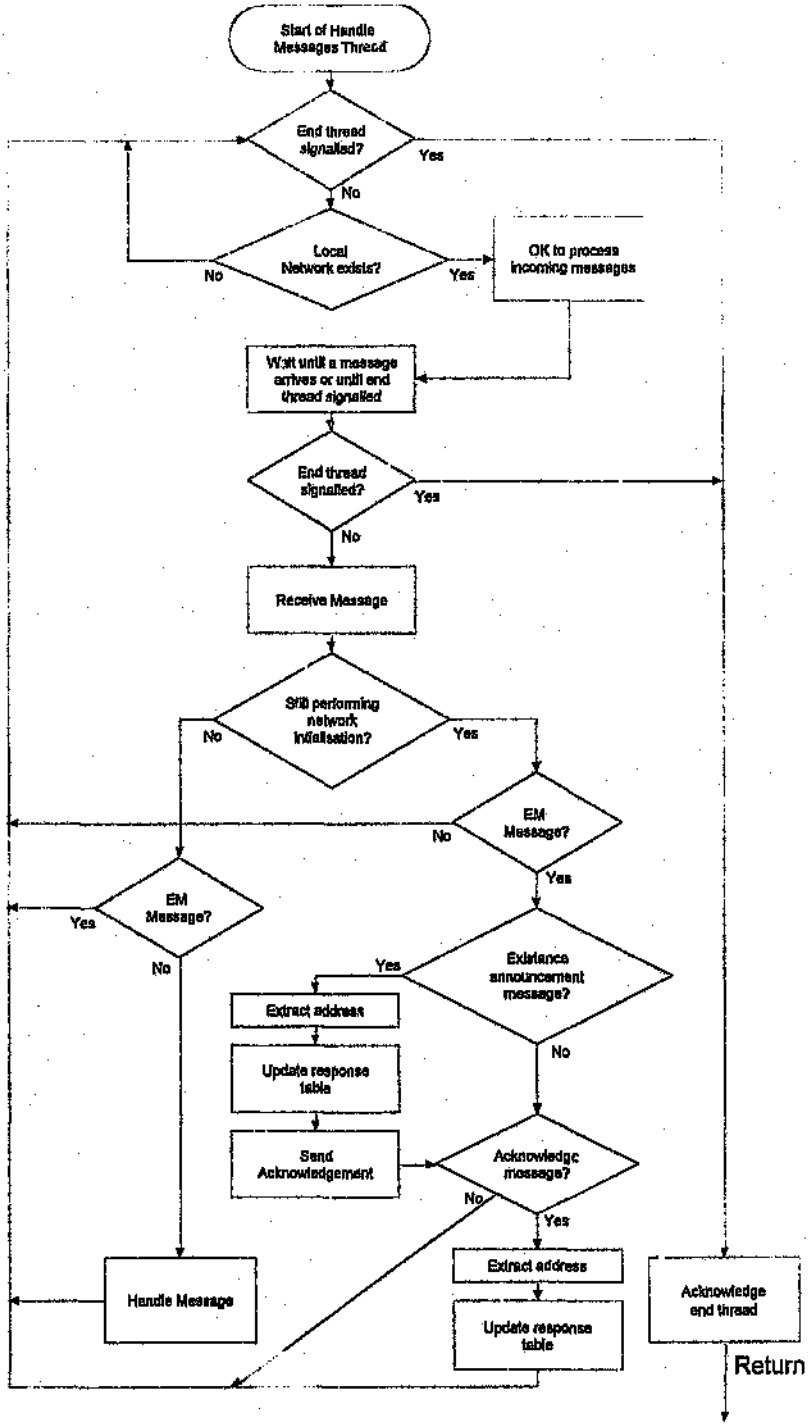


Figure WB113–3: `HandleMessages` Thread Flowchart

`EnvironmentManager(CWnd* dlg)`

The Environment Manager constructor performs all of the necessary initialisation upon startup of the system, including synchronisation between the various components of the system residing on different machines.

The constructor's initial tasks are to open the RenderWare Library, start up the HandleMessages thread, read the address configuration file, and instantiate the network object on the local machine.

Once the network layer has been set up, the EM begins broadcasting "I'm alive" messages, and waits for acknowledgements of these messages from each remote machine. At the same time, the EM waits for "I'm alive" messages from each of the remote machines and acknowledges them as they are received. Each acknowledgement or "I'm alive" that is received is recorded in the `nwExistsResponses` array.

Only when the network layer on each of the remote machines is known to exist and the network layer on the local machine is known to be *known* by all other machines to exist, does the EM cease broadcasting "I'm alive" messages, and continue with the instantiation of processes. Further broadcasts are unnecessary once acknowledgement has been received from each remote machine; and once it is known that all remote network layers are in place, the local processes can be started with impunity, since any messages sent to a remote machine will be queued on that remote machine until the destination process has been instantiated.

In order to instantiate the necessary processes, the EM reads the process configuration file, and from it populates the `processResponsibilityList` array. It then determines for each process whether the process itself or an agent for that process should be instantiated on the local machine, and performs the appropriate instantiation, recording a pointer to the newly instantiated object in the linked list for that type of process. (For example, if the instantiated object was a Simulation process, a pointer to it would be added to the `simList` linked list). If the process itself resides on the local machine, then the `processResponsibilityList` array is also updated to contain a pointer to the process. This pointer allows the `RecvMessage()` function to deliver messages that have been received from a remote machine to the correct process.

A flowchart of the activities of the Environment Manager Constructor is provided in Figure WB113–4.

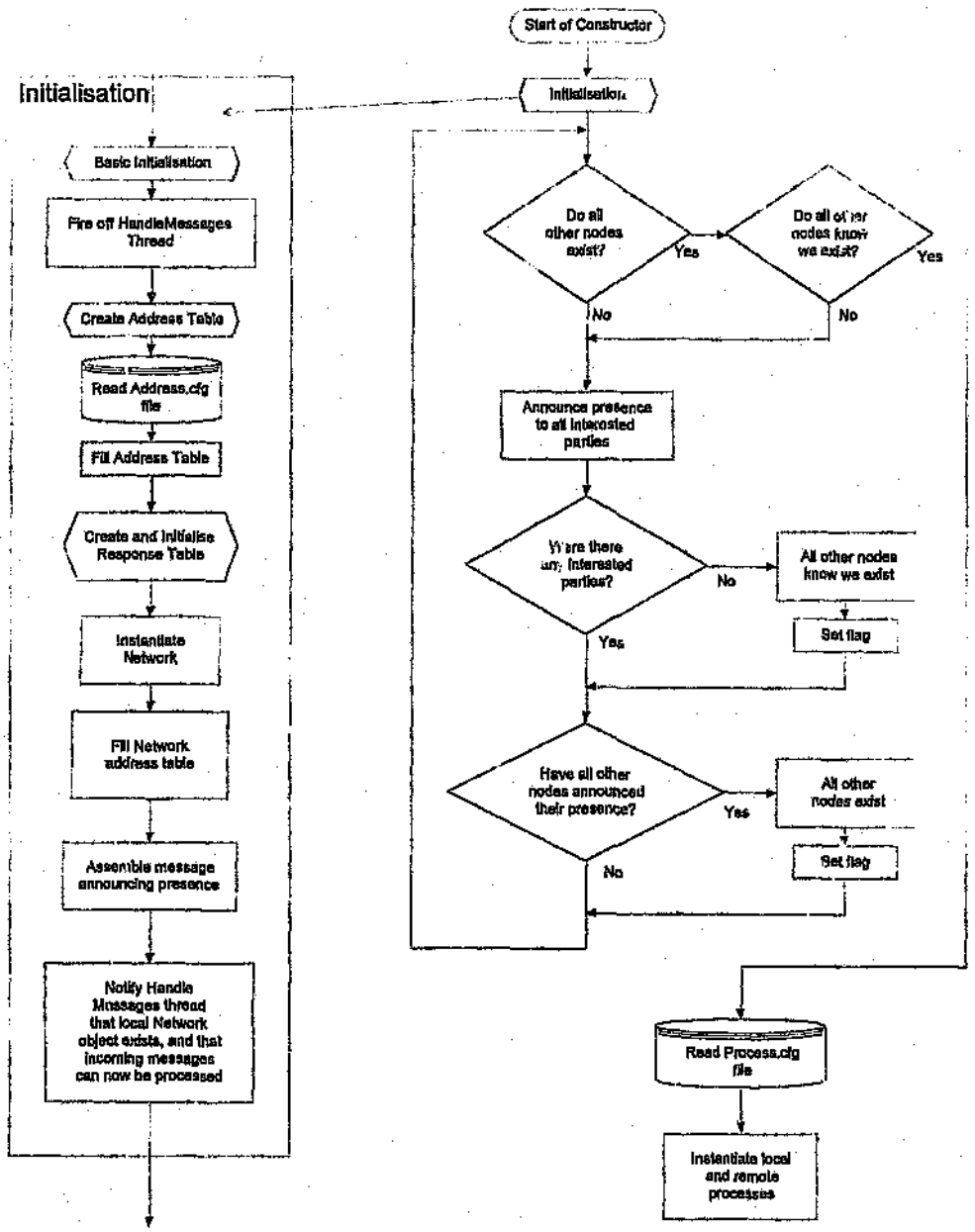


Figure WB113–4: Environment Manager Constructor Flowchart

```
void SendMessage(int PID,
                 CString* message,
                 BOOL isevent)
```

Parameters Taken: The process ID of the target process that the message is being sent to, a string representation of the message that is to be sent, and a boolean flag indicating whether this is a state update or an event message (as these message types are handled differently by the network).

Return Value: None.

Discussion: This function is called by any agent process that wishes to send a message to its corresponding authentic process on a remote machine. The calling agent process supplies its own process ID, which corresponds to that of the remote authentic process. The `SendMessage()` function prepends the process ID to the message, performs a lookup of the process ID in its process-to-processor allocation table to determine on which node the target process resides, and then sends the message by calling the `Network::SendMessage()` function, specifying the destination node and the message type.

```
void RecvMessage(CString* message, int messagetype)
```

Parameters Taken: A string representation of the remotely originating message that has been received, and the message type. The message type indicates whether this is an event message or a state update message.

Return Value: None.

Discussion: This function is invoked each time a message destined for one of the application processes is received by the Environment Manager's `HandleMessages` thread. It is the responsibility of this function to deliver the message to the remote message handler of the appropriate process. The function strips the process ID (first two characters) off the message and uses it to perform a lookup in the `processResponsibilityList` to obtain a pointer to the destination process. It then simply calls the `RecvMessage()` function of that process.

~EnvironmentManager()

Parameters Taken: None. This is a destructor.

Return Value: None. This is a destructor.

Discussion: The `EnvironmentManager` destructor performs the necessary shutdown of all of the system processes for which the EM is responsible. It firstly ends the HandleMessages thread, and destroys the network layer object (this also terminates the network polling thread). It then instructs any Simulation processes that are running to end their simulation threads. Once all of the worker threads have been shut down in this way, the destructor destroys all of the process and agent objects that were created during system startup. Its final tasks are to close the RenderWare library and cleanup memory allocated to local Environment Manager data members.

6.13 class CDebugger

6.13.1 Private Data Members

CDebugDlg* debugdlg

A pointer to the dialog box to which debug output will be directed. The output dialog box is an instance of the class `CDebugDlg`.

inUse

A flag used to ensure mutually exclusive access to the CDebugger object by different threads.

6.13.2 Public Member Functions

Cdebugger()

The constructor initialises the `inUse` data member to `FALSE` indicating that the debugger is initially not in use, and instantiates and initialises the debug dialog box.

~Cdebugger()

The destructor simply initiates destruction of the debug dialog box.

void Write()

This function provides the functionality which allows a message to be written to the output dialog box. It places the specified message in the

message queue of the `CDebugDlg` dialog box, using the `CWnd::PostMessage()` function.

6.14 class `CDebugDlg`

6.14.1 Summary of Functionality

The `CDebugDlg` class is a dialog box class derived from the MFC `CDialog` class, and represents the dialog box to which debug information is written by the CDebugger class.

6.15 class `Timer`

6.15.1 Private Data Members

`TimerLogEntry log[]`

This data member is used to store the log of events that is generated by calls to the Log() member function of the Timer class. It comprises an array of `TimerLogEntry` structures, each of which holds a timestamp and an associated event description. (The `TimerLogEntry` structure was described in Section 4.4).

`int logCount`

The `logCount` variable keeps track of the number of messages that have been logged. It is primarily used as an index into the `log` array.

`CString logfilename`

A string containing the name of the file to which the log will be written when the `Timer` object is destructed.

6.15.2 Public Member Functions

`Timer(char* filename)`

The constructor simply initialises the log count to zero and sets the logfile name to the value specified as a parameter to the constructor.

`void Log(char* eventDescription)`

The `Log()` function is used to log an event. The parameter taken is a null terminated text string which describes the event being logged. The current time, along with the supplied description, are simply added to the `log` array, and the log count is incremented. There is an upper limit

on the number of messages that may be logged by each individual instance of the Timer class. After this limit is reached, subsequent messages are discarded.

`~Timer()`

In order to limit the impact that the use of the Timer class has on performance during program execution (which could influence the measured results), the log of events is stored in memory (by means of the `log` array), and only written to the specified logfile upon destruction of the timer object. The destructor simply writes to file the timestamp and event description for each item that has been logged, preceded by some header information, including the date and time that the logfile was created.

6.16 class `CMessage`

6.16.1 Interface Member Functions

None. This class is used internally by the CNetwork class, and therefore does not have any direct interface with the rest of the system.

6.17 class `CNetwork`

6.17.1 Interface Member Functions

`int m_AddressTable.Add(CString hostname)`

This is really a call to the `CArray::Add()` member function, and is used to add a node to the network object's address table. The hostname that is to be added is supplied as a string. The return value is the index of the added element.

`CNetwork(int nPort)`

This is the constructor for the `CNetwork` class. It takes as its only parameter the network port number that is to be used. The current version of the software communicates exclusively on port 6409, and this is the value specified by the Environment Manager upon instantiation of the network.

```
void SendMessage(CString *message, int messagetype,
                 int destnode)
```

This function is used to send a message to a remote node via the network. The parameters taken are a string representation of the

message, the message type as an integer, and the destination node as an index into the network address table. If the message type is set to 1, the network treats the message as critical and will provide guaranteed delivery. Other message types are delivered using a "best effort" approach.

`int m_queueIncoming.GetCount()`

This is really a call to the `CList::GetCount()` member function, and is used to determine the number of messages in the incoming network queue. The return value is simply the number of messages that are currently available in the queue. A return value of 0 indicates that the queue is empty.

`void RecvMessage(CString *message, int *messagetype)`

This function is used to retrieve a message from the network. It should only be called once it has been established that there is in fact a message in the queue (by means of a call to the `m_queue-Incoming.GetCount()` function). The reference parameters that are required to be passed are a pointer to the string object that is to hold the retrieved message, and a pointer to an integer that will hold the message type.

`~CNetwork()`

The destructor is called when the network object is to be destroyed. It shuts down the network polling thread and performs the necessary cleanup of any memory that has been allocated.

6.18 class `GenList`

6.18.1 Interface Member Functions

`GenList()`

The GenList constructor simply constructs an empty linked list.

`int AppendElement(T& new_element)`

The `AppendElement()` function is used to add an element to the linked list. The `newelement` parameter that is supplied must be of the type (`T`) that was specified when the parameterized class was instantiated. In addition, if the element is not of scalar type or a structure, then it must provide a copy constructor.

`void GoToStart()`

This function resets the *current* pointer in the linked list to the start of the list, so that the next element returned by the `GetElement()` function will be the first element in the list.

`unsigned GetListSize()`

This function returns the number of elements that are currently in the linked list as an unsigned integer.

`T& GetElemer`

This function returns a reference to the element pointed to by the *current* pointer.

`int NextElement()`

This function moves the *current* pointer to the next element in the list. If there are no more elements in the linked list, then the function returns an error code of `-1`.

`~GenList()`

The destructor simply deletes all of the elements in the linked list.

6.19 class `CAboutDlg`

6.19.1 Summary of Functionality

The `CAboutDlg` class is a dialog box class derived from the MFC `CDialog` class, and represents the dialog box that is displayed when the `Help->About` menu option is chosen.

6.20 class `COoveApp`

6.20.1 Summary of Functionality

This class provides the initial entry point for the program. It performs a number of general initialisation tasks and then instantiates the main `COoveDlg` dialog box object.

6.21 class COoveDlg

6.21.1 Summary of Functionality

The COoveDlg class is a dialog box class derived from the MFC CDialog class, and represents the main program dialog box. This class handles all windows messages that are generated pertaining to the main dialog box. This includes menu option selection, mouse events, and keyboard input, as well as the OnPaint and OnSize functions. Mouse and keyboard input that is received is simply redirected to the appropriate member functions of the InputHandler class that have been described in Section 6.5 above. The OnPaint and OnSize functions are likewise redirected to the respective member functions of the PresentationGenerator class, as described in Section 6.9.

OOVE

User Reference Manual

Technical Product

Version 1.01

Document Status: Approved

Table of Contents

Change History

Configuration Control

Project:	OOVE
Title:	User Reference Manual
Doc. Reference:	WB 107
Created by:	Warren Blumenow
Creation Date:	23 October 1996

Document History

Version	Date	Status	Who	Saved as:
0.01	96/10/23	Draft	WB	\1995_07\TP\URM\WB107WW.001
0.02	97/05/06	Draft	WB	\1995_07\TP\URM\WB107WW.002
0.03	97/09/15	Draft	WB	\1995_07\TP\URM\WB107WW.003
1.00	97/10/26	Approved	WB	\1995_07\TP\URM\WB107WW.100
1.01	98/08/26	Approved	WB	\1995_07\TP\URM\WB107WW.101

Revision History

Version	Date	Changes
0.01	96/10/23	New document created using WB 002
0.01	96/11/09	Updated sections 2,3,4.
0.01	97/04/04	Added sections 5,6; Various small changes.
0.02	97/05/06	Changes to Sections 1,4,5,6; Diagrams included.
0.03	97/09/15	Added Example of System Usage Section.
1.00	97/10/26	Document status changed to Approved. Miscellaneous corrections made throughout.
1.01	98/08/26	Final corrections and formatting.

Management Authorisation

Version	Date	Status	Management Board Minute Reference
1.00	97/10/19	Approved	WB 525

1 Scope

1.1 Introduction

The OOVE Visualisation Tool is a prototype application that has been developed to demonstrate the concepts that are relevant to the project. The acronym OOVE stands for Object Oriented Virtual Environment. The envisioned full scale application would provide the user with an intuitive means for visualising an electric power network through the use of an interactive Virtual Reality style user interface. The user would be able to navigate freely through a three dimensional representation of a model of the network, inspect its parameters and interact with the network in various ways.

The prototype tool provides much of this functionality, albeit in a simplified form. The system has been designed to run distributed across a group of low-cost PCs connected to an Ethernet network, making use of distributed processing to improve performance.

The prototype is intended to give the user a feel for the potential capabilities of a full scale implementation, and provide proof of concept by practically demonstrating the key techno es and software models that underlie the system's implementation.

1.2 Purpose

The purpose of this documen is to provide the user with insight into the functionality provided by the OOVE Visualisation Tool. It provides a user-level view of the system's operation, facilitating the successful setup and running of the program and the user's interaction with it. It should be seen as an operational reference manual for the prototype application.

1.3 Terminology

1.3.1 Abbreviations

DNS = Domain Name Server

1.4 Audience

The intended audience for this document includes anyone who wishes to run the OOVE application—specifically, the product developer, the project manager, individuals involved with subsequent development of the system, the external examiner, and other interested parties.

1.5 Applicable Documents

1.5.1 Standards

a. SEAL QMS Document Creation Template, QS 002, Revision 0.02, 3 October 1994

b. SEAL QMS Document Layout, Presentation and Typesetting Guide, QS 003, Revision 1.00, 3 October 1994

1.5.2 References

a. References and Bibliography, WB 114

1.6 Assumptions

It is assumed that the user has a reasonable knowledge of the Windows operating system, and therefore the exact mechanics of editing files, running programs, and so on will not be described.

It is assumed that from a networking perspective, the user is familiar with such concepts as host names, IP addresses and Domain Name Server (DNS) name resolution. It is further regarded as outside the scope of this document to describe the process of configuring the operating system for running the program. This includes, for example, the installation of the TCP/IP protocol stack. For information on these topics, the user is referred to the relevant literature provided by the vendor.

2 Hardware and Software Requirements

The following hardware and software is required to run the OOVE Visualisation Tool:

2.1 Hardware Requirements

The hardware requirements for running the system consist of the following:

1. A number of workstations, each with the following minimum configuration:

   - Intel or equivalent 80486 DX2-66 PC minimum (Pentium class PCs are recommended)
   - At least 16 MB RAM
   - At least VGA resolution display adapter and monitor with at least 256 colours.
   - Mouse
   - Ethernet Network Adapter

   The number of PCs used can be anywhere from *one* to the number of processes that will be running. A dedicated machine can be used for each of the processes, or several processes can be housed on a single machine.

2. An ethernet network

2.2 Software Requirements

The system requires the following software to be installed

1. Microsoft Windows 95 or Windows NT Workstation 4.0
2. A Winsock compliant TCP-IP network stack
3. The OOVE program, resource and configuration files, as follows:

`oove.exe`	`address.cfg`
`map.rwx`	`process.cfg`
`jhb.rwx`	`simulate.cfg`

4. RenderWare Version 2.0 or higher

   or

   The following Dynamic Linked Library (DLL) files installed in the `Windows\System` directory:

rwdl16a20.dll	rwdl18a20.dll	rwl3ds20.dll
rwdl16b20.dll	rwdl18b20.dll	rwldxf20.dll
rwdl16c20.dll	rwdl18c20.dll	rwl20.dll
rwdl16d20.dll	rwdl18d20.dll	

3 System Setup and Configuration

3.1 Software Installation

Installation of the software simply involves the following:

- install the necessary software (TCP/IP, RenderWare, etc.), or (as appropriate) copy the necessary DLL files to the `Window \System` directory
- ensure that the program, resource and configuration files all reside in the same directory
- edit the configuration files to reflect the current hardware configuration and the process to processor allocation that has been decided upon.

3.2 Configuration Files

OOVE makes use of three configuration files, namely:

ADDRESS.CFG
PROCESS.CFG
SIMULATE.CFG

The format of these files and how they are used is described below:

3.2.1 ADDRESS.CFG

The ADDRESS.CFG file is used to specify the network configuration which is to be used to run the program. Information about the total number of workstations or nodes, as well as address information in the form of a host name or IP address for each of the networked computers is extracted from this file. In addition, by means of this file, each workstation is able to determine its own identity.

Each node maintains a slightly different version of the ADDRESS.CFG file.

The format of the ADDRESS.CFG file will be explained by means of the example shown in Figure WB107–1.

```
3
0
machine1.ee.wits.ac.za
machine2.ee.wits.ac.za
machine3.ee.wits.ac.za
```

Figure WB107–1: Sample ADDRESS.CFG Configuration File

The file consists of two numbers, followed by a list of host names, each on a separate line in the file. The first number represents the total number of nodes that are being used. The minimum is *1*. The second number is a zero-based index into the list of host names, identifying the local machine. A workstation uses this number to determine its identity. Finally, the host names for each of the machines in the configuration are listed.

In the example given in Figure WB107–1, the system will be run on three machines. This particular configuration file is housed on the first machine (with an index of 0). The host names of the three nodes are `machine1...`, `machine2...` and `machine3.ee.wits.ac.za`, respectively. The host name of this machine is therefore `machine1.ee.wits.ac.za`.

The host names specified in the ADDRESS.CFG file may be replaced with IP addresses (for example, `146.141.16.221`). This obviates the need for domain name resolution by means of a DNS server.

3.2.2 PROCESS.CFG

The PROCESS.CFG file is used to specify the processes that comprise the application, as well as their distribution among the available computers. For each process, the process ID, process type and the node responsible for the process are specified.

Since each machine's view of the system must be consistent with that of all other machines, each machine uses an exact copy of the same PROCESS.CFG file.

The PROCESS.CFG file contains firstly a single integer value representing the total number of processes which exist, and then a number of records of the form:

Process ID *Responsible Node* *Process Type*

where the three numbers are all integers. The process IDs must be unique, and are typically listed in numerical order, starting with 1. The responsible node field is a zero-based index into the list of host names contained in the ADDRESS.CFG file, and specifies which node that process will be housed on. If the responsible node matches the *local machine index* specified in the ADDRESS.CFG file, then that process will be housed on the local machine.

The process type field is interpreted as follows:

0	Input Handler
1	Presentation Generator
2	Object
3	Simulation

It should be noted that due to the way the system has been implemented, in the current version of the software the object processes should be specified first. A sample PROCESS.CFG file is shown in Figure WB107–2.

```
5

1 0 2

2 1 2

3 1 0

4 0 1

5 2 3

// Five processes

// process 1, responsible=0, object

// process 2, responsible=1, object

// process 3, responsible=1, input handler

// process 4, responsible=0, presentation generator

// process 5, responsible=2, simulation
```

Figure WB107–2: Sample PROCESS.CFG Configuration File

In this case, there are five processes, distributed over three machines. The first two processes are Object Processes, and run on machine 0 and machine 1 respectively. The third process is an Input Handler, and runs on machine 1, and so on.

As indicated by this example, comments may be included in the file after the process specifications and will be ignored. Although for clarity the C++–style "//" prefix has been included in this example to designate these lines as comments, the lines are in fact ignored altogether. (The file parser reads the number of processes, and then reads exactly that number of subsequent lines of the file).

3.2.3 SIMULATE.CFG

The SIMULATE.CFG configuration file is used to specify the time-varying simulation values that will be used by the program. This file serves as an input data stream to the simulation component of the program. In the eventual final incarnation of the visualisation tool, this data file could be replaced by a live stream of data from external sources, such as transducers or monitoring equipment.

This file consists of a sequence of numbers. The first of these is an integer representing the number of data points contained in the file, and hence in the time-varying sequence that the simulation uses. The sequence of floating point numbers thereafter are the actual data points.

The SIMULATE.CFG file which is used in the prototype application was generated by means of MATLAB™.

3.3 Running the Program

Before running the OOVE software, the hardware configuration and process allocation which will be used must be decided upon. Thereafter the configuration files described above must be edited to reflect these choices.

Although the system is designed to be distributed across a number of machines and different parts of the system run on each machine, the same executable program, `oove.exe`, is run on every machine. The exact roles of each instance of the program are determined at run time.

All that needs to be done in order to start up the system, therefore, is to start the `oove.exe` executable on each machine. The main program window on each machine may take some time to appear, since the machines initially communicate with one another, and only when all of the machines are ready does the system actually start up. Figure WB107–3 shows a sample of the main program window which appears on the machine which houses the Presentation Generator; Figure WB107–4 shows the "blank" main program window which appears on the other machines.

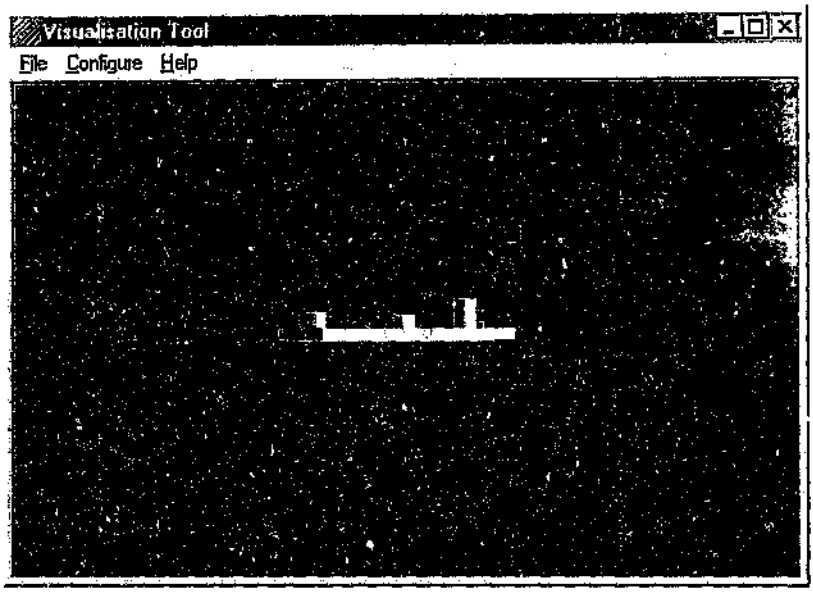


Figure WB107–3: Sample Main Program Window (Presentation Generator Machine)

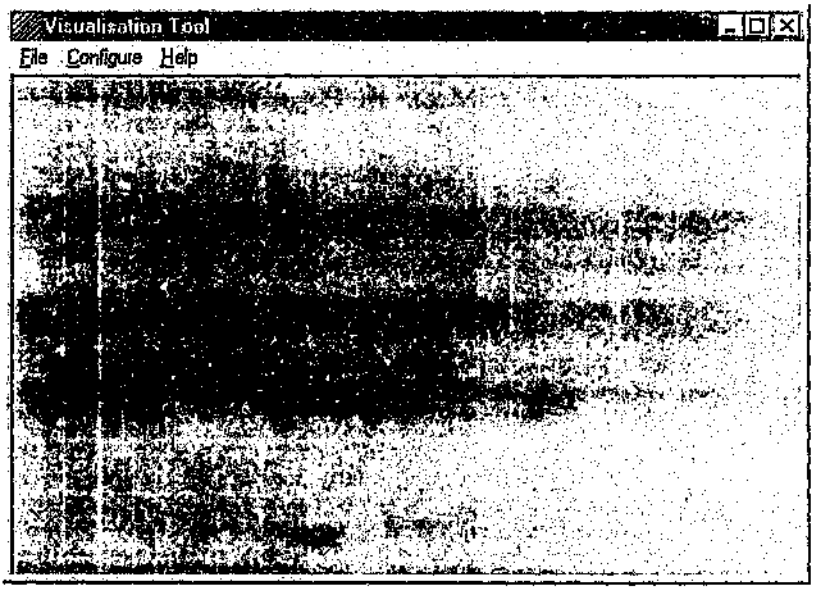


Figure WB107–4: Sample Main Program Window (Non-Presentation Generator Machines)

4 Menu System

This section details the menu system of the OOVE Visualisation Tool. Since most of the interaction with the program is done by direct manipulation of the user's viewpoint by means of the navigation controls (see section 5), the menu system provides a limited amount of functionality. The functions provided are listed below:

- File - Exit

This function is used to exit the program. It is identical to double clicking the icon in the top left corner of the main window or clicking the icon in the top right corner.

- Configure - Options

This function brings up the *Options* popup submenu, which in turn consists of the menu items:

- Edit Address.cfg
- Edit Process.cfg
- Edit Simulate.cfg

These functions respectively allow the address, process and simulation configuration files to be edited. These menu options are provided for convenience only — the configuration files can equivalently be modified by means of any standard text editor, such as the Windows *notepad*.

- Configure - Refresh View

The *refresh view* function causes the main window to be redrawn, and is useful if the display has become corrupted. If the system is running distributed over a number of machines, then this function can be invoked on any of the machines, in order to refresh the display of the machine which houses the graphics generator process.

Help - About

This menu option brings up the *about* dialog box shown in Figure WB107–5 below, which gives brief information about the program.

Figure WB107–5: *About* Dialog Box

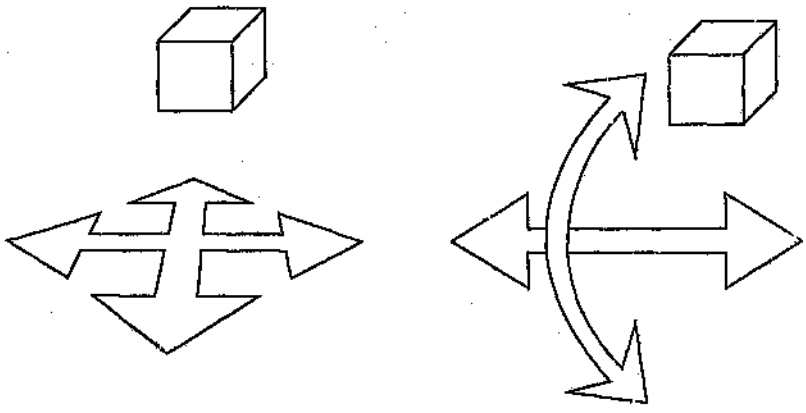
5 Navigation

The navigation controls provide the main means for interaction with the program. By means of these controls, the user is able to change his viewpoint, and navigate through the virtual environment.

5.1 Navigation Using the Mouse

Navigation through the virtual environment by means of the mouse is very intuitive and natural.

Two modes of navigation are provided. Mode 1 navigation allows the user to change his position in the X-Z plane. This allows forward and backward movement, as well as movement from side to side. Mode 2 navigation allows for a change in orientation of the viewpoint of the user with respect to the area of interest, coupled with side to side translation, as before. By providing a means for changing orientation, the angle at which the objects of interest are viewed can be modified to reveal hidden detail. These modes of navigation are explained by Figure WB107–6.



Mode 1 Navigation Mode 2 Navigation

Figure WB107–6: Navigation Modes

Mode 1 navigation is achieved by holding down the left mouse button and moving the mouse in the desired direction. Moving the mouse forward moves the observer forward; moving the mouse backward moves the observer backward. Similarly, the observer's side to side motion follows the left and right movements of the mouse. A sample of the output display during Mode 1 navigation is provided in Figure WB107–7.

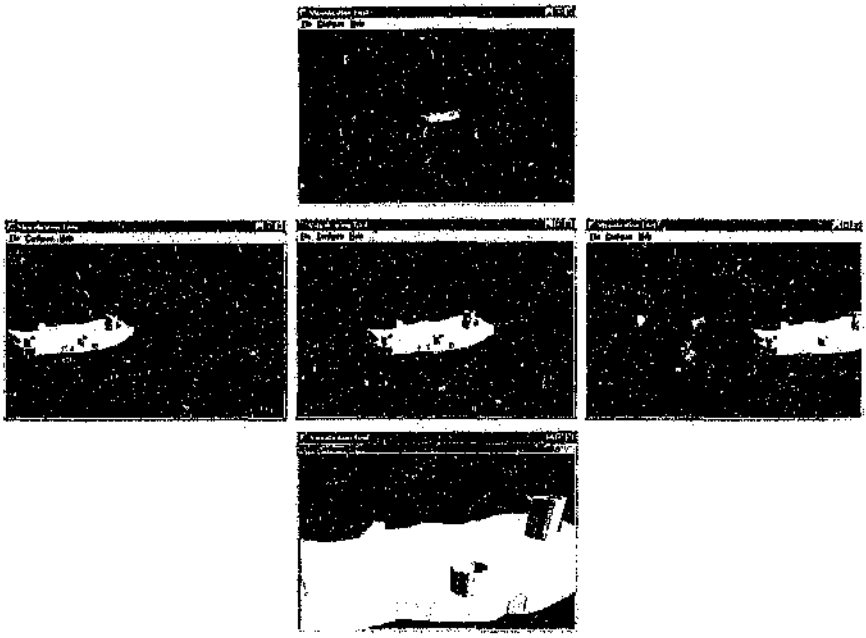


Figure WB107–7: Mode 1 Navigation

For Mode 2 navigation, the right mouse button is used. With the right mouse button depressed, left and right movements of the mouse result in movement of the observer to the left and right, as before. Forward and backward movements of the mouse cause the pitch angle of the view orientation to increase and decrease, respectively. A typical output display during Mode 2 navigation is shown in Figure WB107–8.

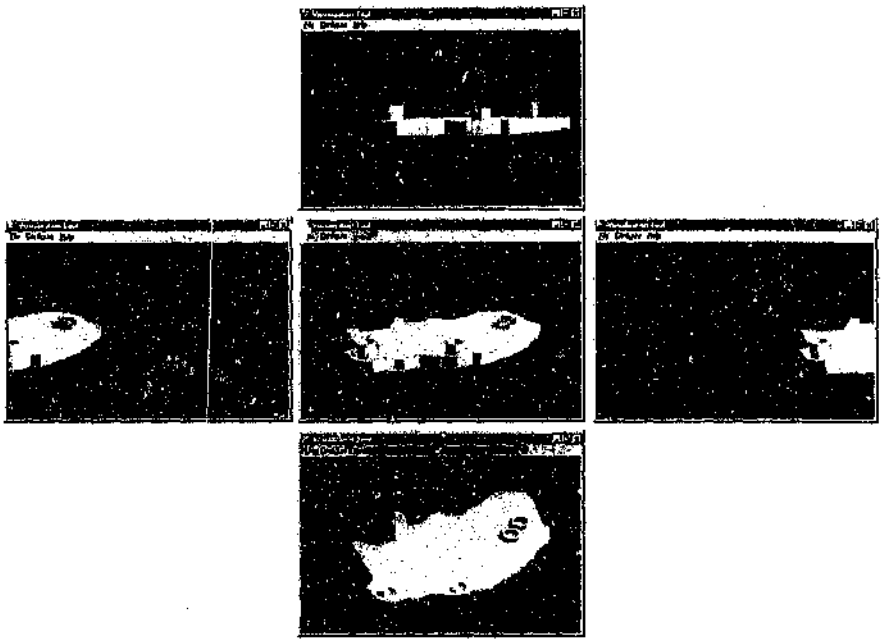


Figure WB107–8: Mode 2 Navigation

5.2 Navigation Using the Keyboard

The keyboard controls mimic the mouse controls. Both modes of navigation are again provided. Mode 1 navigation is the default mode, and corresponds to holding down the left mouse button. The "P", "semicolon", "L" and "apostrophe" keys are used to move forward, backward, left and right respectively.

Mode 2 navigation is achieved by holding down the "Ctrl" key while pressing the above keys, to increase and decrease the view orientation pitch angle, and move the observer left and right respectively.

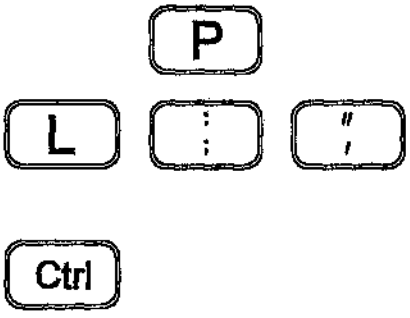


Figure WB107–9: Keyboard Navigation Controls

6 Example of System Usage

This section presents an example of the configuration, running, interaction with and termination of the program. It provides a sample run through of the sequence of tasks that typically need to be performed by the user.

In the example provided, the system is to be run distributed across three machines.

6.1 Specification of Process to Processor Allocation

1. Decide on the process to processor allocation that will be used. For the purposes of this example, the following allocation has been selected:

Machine	Allocated Processes
Machine 1	Object Process Presentation Generator
Machine 2	Object Process Input Handler
Machine 3	Simulation

2. Confirm that there is a suitable input device attached to Machine 2 and that there is a suitable display device attached to Machine 1.

6.2 Software Installation and Configuration

3. The system is to run under Windows NT Workstation 4.0 on machines 1 and 2, and under Windows 95 on Machine 3. Verify that TCP/IP connectivity has been installed and tested on all of these machines.

4. None of the machines on which the program is to run have RenderWare installed. Therefore copy the appropriate DLL files from the distribution disk to the `Windows\System` directory on each machine, by typing:

```
copy a:\oove\rwdll\*.dll c:\windows\system
```

at the Windows NT Command Prompt.

5. Copy the program, configuration and support files to the directory from which the program will be run on each machine:

```
copy a:\oove\*.* c:\oove
```

6. Edit the three configuration files using the Windows *notepad*. For this example, the configuration files should be edited to look exactly as shown in Figure WB107–1 and Figure WB107–2 (The second line of Figure WB107–1 is changed to `1` and `2` for Machine 2 and Machine 3, respectively).

6.3 Running the Program

7. Execute the `oove.exe` binary on each machine. If the system runs correctly, a blank window should appear on Machine 2 and on Machine 3. The output display should appear on Machine 1. If the program does not start up properly, then shut down all running instances of the program on each of the machines as described in Section 7.1, and check that the configuration files are correct and that all the necessary program files are in the correct locations, before attempting to run the program again.

6.4 Interaction With the System

8. Once the system has been started up successfully, the user should be presented with a map on which various objects of interest appear. By means of the navigation controls described in Section 5, the user is able to move freely around the environment which is presented, and inspect the objects that are represented. The user interface is very intuitive, and it is left to the user to investigate the different means of interaction with the system that have been provided.

9. Note that the behaviour of the objects is influenced by the simulation which is running. To observe the effect that the Simulation Process is having on the system, shut down the OOVE program on Machine 3 only, by selecting *Exit* from the *File* Menu. The animation of those objects that are influenced by the Simulation should immediately cease.

10. The user interface provides the ability to edit the configuration files so that the parameters can be changed. Changes that are made in this way will take effect the next time the program is run. To illustrate, change the PROCESS.CFG file to place the Simulation process on Machine 2 instead of Machine 3. Select *Options→Edit Process.cfg* from the *Configure* menu. Make the required change to line 6, so that it reads "5 1 3". It is left to the user to shut down and re-run the system to observe the effect of this change. (In this configuration, shutting down the program on machine 3 should no longer yield the same result).

11. Finally, in order to check the version number of the program, select the *About* option from the *Help* menu. The dialog box shown should report that this is Version 1.0 of the system, as shown in Figure WB107–5.

6.5 Shutting Down

12. Click the ■ icon in the top right hand corner of the window on each machine to shut down the program.

7 Troubleshooting

7.1 Multiple instances

Only one copy of the program should be running on any particular machine at a time. Network communications will be adversely affected if more than one instance of the program is running simultaneously, since it is not clear which instance of the program should handle incoming network messages, or which instance has generated an outgoing message. Under these conditions, the program will behave unpredictably.

If the program does not startup properly or terminate gracefully, the user should check that the program has been shut down completely, and that traces of the program are not left running on the system. Under Windows 95, this is done by means of the *Close Program* dialog box which is accessed by pressing the *Ctrl-Alt-Delete* key combination. Under Windows NT, the Task Manager can be used for this purpose. In either case, any tasks called "Visualisation Tool" or "Oove" should be terminated, by selecting "End Task".