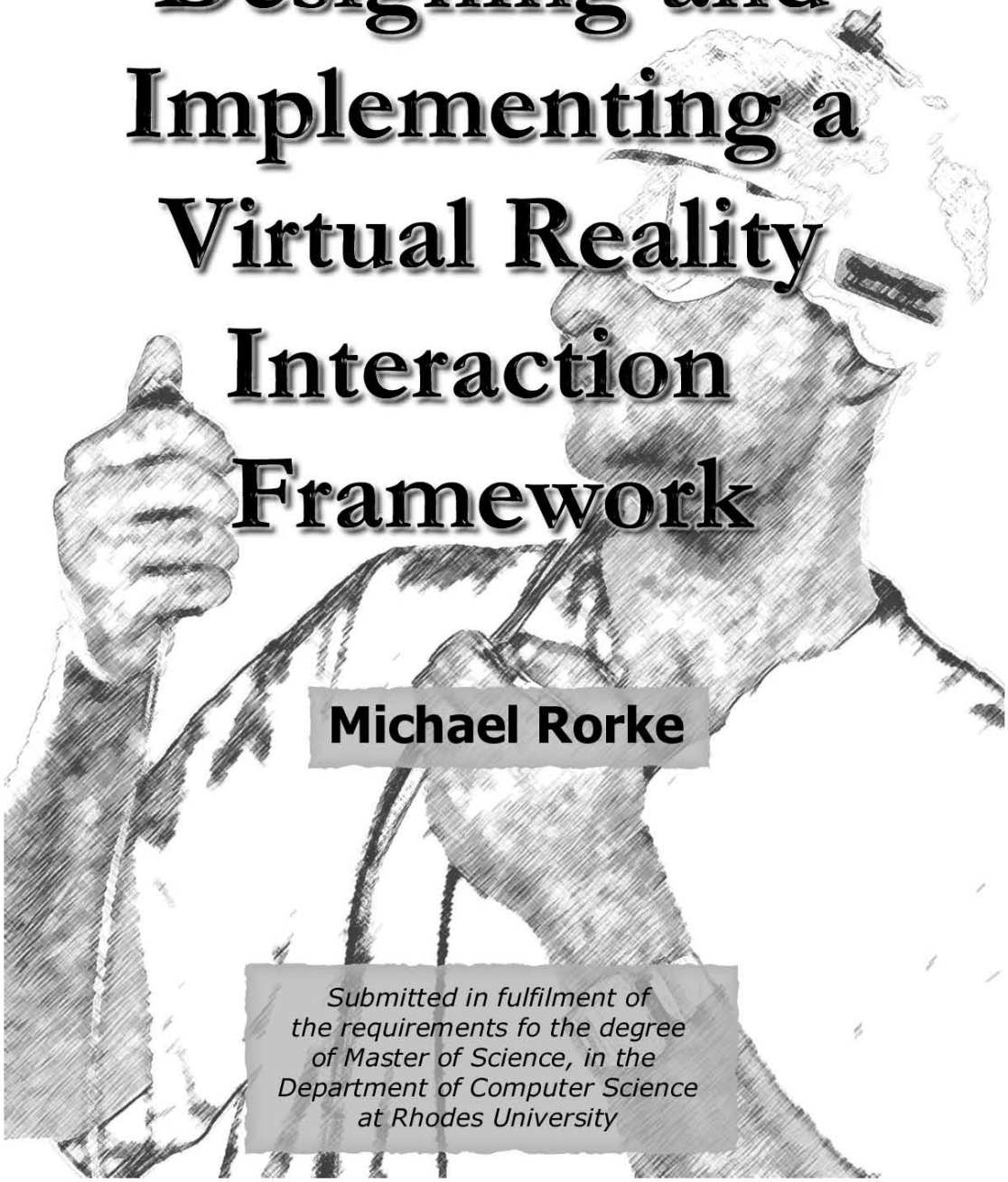
Designing and Implementing a Virtual Reality Interaction Framework



Michael Rorke

Submitted in fulfilment of the requirements fo the degree of Master of Science, in the Department of Computer Science at Rhodes University

Abstract

Virtual Reality offers the possibility for humans to interact in a more natural way with the computer and its applications. Currently, Virtual Reality is used mainly in the field of visualisation where 3D-graphics allow users to more easily view complex sets of data or structures. The field of interaction in Virtual Reality has been largely neglected due mainly to problems with input devices and equipment costs. Recent research has aimed to overcome these interaction problems, thereby creating a usable interaction platform for Virtual Reality.

This thesis presents a background into the field of interaction in Virtual Reality. It goes on to propose a generic framework for the implementation of common interaction techniques into a homogeneous application development environment. This framework adds a new layer to the standard Virtual Reality toolkit – the interaction abstraction layer, or *interactor* layer. This separation is in line with current HCI practices. The interactor layer is further divided into specific sections – input component, interaction component, system component, intermediaries, entities and widgets. Each of these performs a specific function, with clearly defined interfaces between the different components to promote easy object-oriented implementation of the framework. The validity of the framework is shown in comparison with accepted taxonomies in the area of Virtual Reality interaction. Thus demonstrating that the framework covers all the relevant factors involved in the field.

Furthermore, the thesis describes an implementation of this framework. The implementation was completed using the Rhodes University CoRgi Virtual Reality toolkit. Several postgraduate students in the Rhodes University Computer Science Department utilised the framework implementation to develop a set of case studies. These case studies demonstrate the practical use of the framework to create useful Virtual Reality applications, as well as demonstrating the generic nature of the framework and its extensibility to be able to handle new interaction techniques.

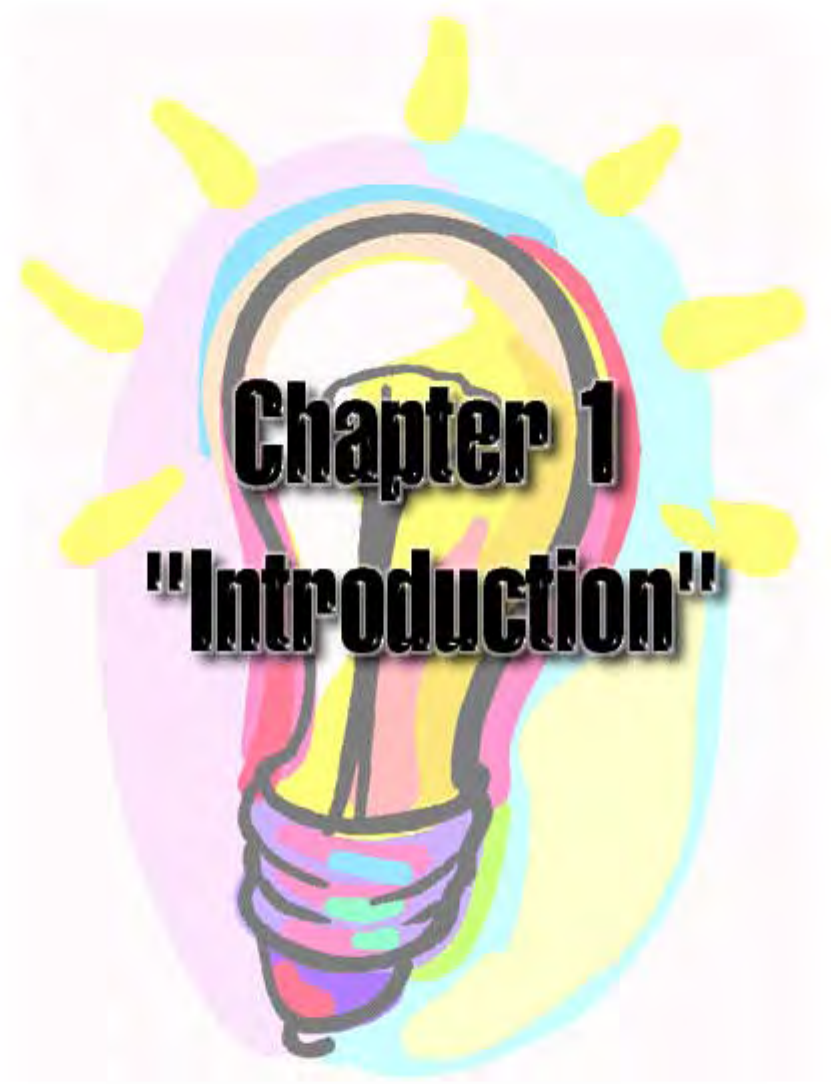
Finally, the generic nature of the framework is further demonstrated by moving it from the standard CoRgi Virtual Reality toolkit, to a distributed version of this toolkit. The distributed implementation of the framework utilises the Common Object Request Broker Architecture (CORBA) to implement the distribution of the objects in the system. Using this distributed implementation, we are able to ascertain that CORBA is useful in the field of distributed real-time Virtual Reality, even taking into account the extra overhead introduced by the additional abstraction layer.

We conclude from this thesis that it is important to abstract the interaction layer from the other layers of a Virtual Reality toolkit in order to provide a consistent interface to developers. We have shown that our framework is implementable and useful in the field, making it easier for developers to include interaction in their Virtual Reality applications. Our framework is able to handle all the current aspects of interaction in Virtual Reality, as well as being general enough to implement future interaction techniques. The framework is also applicable to different Virtual Reality toolkits and development platforms, making it ideal for developing general, cross-platform interactive Virtual Reality applications.

Acknowledgements

First and foremost, I would like to thank my supervisors Prof. Shaun Bangay and Prof. Peter Wentworth for their guidance and help with this project. I would also like to thank all the people whose work I referenced, especially Douglas Bowman and the people on the 3D User Interfaces (3DUI) mailing list for the great ideas and helpful discussion. Finally, I would also like to thank my fellow students. Matt, Colin and Holger for their help with the case studies section and all the other Masters students (you know who you are!) for many hours of procrastination as well as their (sometimes!) useful comments and criticisms.

Table of Contents


Chapter 1
"Introduction"

With the current growth in the power of the desktop computer and the growing availability of dedicated graphics rendering hardware, virtual reality is becoming more and more mainstream in its application. PC Magazine [Ozer, 98] predicts that standard desktop machines will soon be equipped with 3D accelerator cards, with performance levels on these machines reaching four times the level of performance on current, high-end workstations. In fact, the current generation of personal computer processors (e.g. the Intel™ PIII and AMD K6) have had their instruction sets increased to incorporate dedicated 3D graphics rendering instructions. The specialised input devices necessary to implement virtual reality are also becoming more commonplace. For example, many companies now sell head-mounted displays (HMDs). While the hardware now exists to support virtual reality applications, the software tools available in this field are still lacking in usability.

1.1. Towards a Better Interaction Framework

Interaction in immersive virtual reality is an important step towards making virtual reality a useful computing tool and for the evolution of the next generation of computer interfaces. There are various problems associated with interaction in virtual reality. Most of these pertain to our inability to identify all the features, movements, etc. of the human being that are used for interaction in the real world, in order to reproduce this interaction style inside a virtual environment. Compromises are made and values that are measurable (with current technology) are combined with various *interaction techniques* (the result of many years of research) to approximate natural interaction inside a virtual environment. These interaction techniques have mostly been developed independently of one another and do not conform to any common framework. The goal of this project is the design and implementation of a generic interaction framework, which will allow developers to quickly integrate existing interaction techniques with their applications. The framework also provides a generic method for the implementation of new techniques in such a way that they may easily integrate with new and existing applications.

1.2. Background

1.2.1. What Exactly is Virtual Reality?

The term *Virtual Reality (VR)* has been around for many years, and has been used to describe different systems, ranging from the original mechanical flight simulators, through 3-D games through to motion capture applications. A very broad definition of the term would be:

> *" The use of advanced technology to visualise large and complicated sets of data more easily"*

VR usually involves a computer generating multi-sensory output i.e. vision, sound, etc. and specialised input and output devices (e.g. HMDs and magnetic trackers). Isdale [Isdale, 98] divides the field of VR up by considering the different interfaces presented to a user. Using this idea, there are five main branches of VR:

- *Window on World (WoW) systems*: These systems use standard desktop monitors to display a 2D image of a 3D system. Most desktop 3D applications are examples of this.
- *Video Mapping*: Video mapping involves mapping video onto a 3D object and changing the users' views of the video depending on their relative positions and orientations to the object.
- *Immersive systems*: Immersive systems attempt to use non-standard input and output devices to make users feel that they are part of the system, not observing the system from outside (e.g. through a window.)
- *Telepresence*: Telepresence involves users performing some action at a given location, and having their actions electronically reproduced at some remote location. An example of this may be a doctor in one country, remotely performing surgery on a patient in another country.
- *Mixed (or Augmented) Reality systems*: Augmented reality systems combine the real world with a computer-generated environment (e.g. using a head mounted display that allows the user to see through the screens to the outside world).

This project deals mainly with immersive systems and how they are designed and implemented to make it easier for a user to interact with the system.

1.2.2. What are Immersive Systems?

Immersive VR places a user inside the application environment. This usually involves a head mounted display (HMD) and some form of tracking which enables the application to pin-point the position and orientation of various important reference points on the user's body (e.g. the hands and head). The information from the trackers is used to generate a Virtual Environment (VE) where users are able to interact in an intuitive way with the application they are trying to use. For example, when the user moves their head, the picture displayed on the HMD updates to give the impression that they are looking around inside a room.

1.2.3. Why Focus on Interaction?

Currently successful virtual reality systems make good use of immersion techniques to enable users simply to explore virtual worlds, with little or no interaction. Applications like building walkthroughs, psychotherapy applications and some games fall into this category. On the other hand, applications that rely on interactions between users and their environments have not been as successful. The reasons for this lack of success when attempting to allow a user to interact with a virtual environment are covered in the following chapter. These factors tend to lower the sense of realism that a user feels when using interactive VR systems, as well as making the systems difficult and frustrating to use. Study in the field of immersive, interactive VR has been carried out since the early 1980's and various ideas for overcoming specific problems associated with this field have been proposed [Bowman, 97; Mine, 95; Mine, 97]. This research has produced various interaction techniques which, coupled with current advances in hardware, have paved the way for a usable VR interaction system.

1.2.4. Why Not Stick With 2D Interfaces?

The current desktop interface which is prevalent on most machines (known as the Windows, Icons, Mouse and Pointer (WIMP) interface) is the third generation of interface to be used on computers [van Dam, 97]. The first generation of user interface (1950s and 60s) was characterised by the *batch mode* use of computers where programs were written (usually on punched cards) and run with no possibility of interactive use by the user. The second generation (1960s to early 1980s) was characterised by the *timesharing* use of mainframe computers by many users, all operating through small 'dumb' text terminals. This command line based interface was the first interface for the desktop machine. The WIMP interface was developed at Xerox PARC labs in the early 1980s and first appeared on the Apple Macintosh in 1984. Since then, the interface has spread to all forms of desktop machine, making such machines readily accessible to users from a wide range of backgrounds.

There are various problems with the WIMP interface [van Dam, 97]. Firstly, the more complex the interface, the nonlinearly harder the interface becomes to learn due to the profusion of widgets and features. Secondly, users spend too much time manipulating the interface as opposed to the application itself. Thirdly, WIMP interfaces are designed for 2D applications and do not scale well to the 3D realm. Fourthly, the mouse and keyboard interfaces are not natural to users. WIMP interfaces take no advantage of speech, hearing and touch.

" WIMP GUIs based on the keyboard and the mouse are the perfect interface only for creatures with a single eye, one or more single jointed fingers, and no other sensory organs" – Bill Buxton (of Alias/Wavefront)

[van Dam, 97]

The fourth generation of user interface (called the Post-WIMP) interface is currently being developed. Post-WIMP interfaces attempt to involve all the senses in parallel, utilise natural language interaction and involve multiple users. The immersive virtual reality interface is the eventual goal of Post-WIMP research.

1.3. Document Overview

This thesis is arranged as follows:

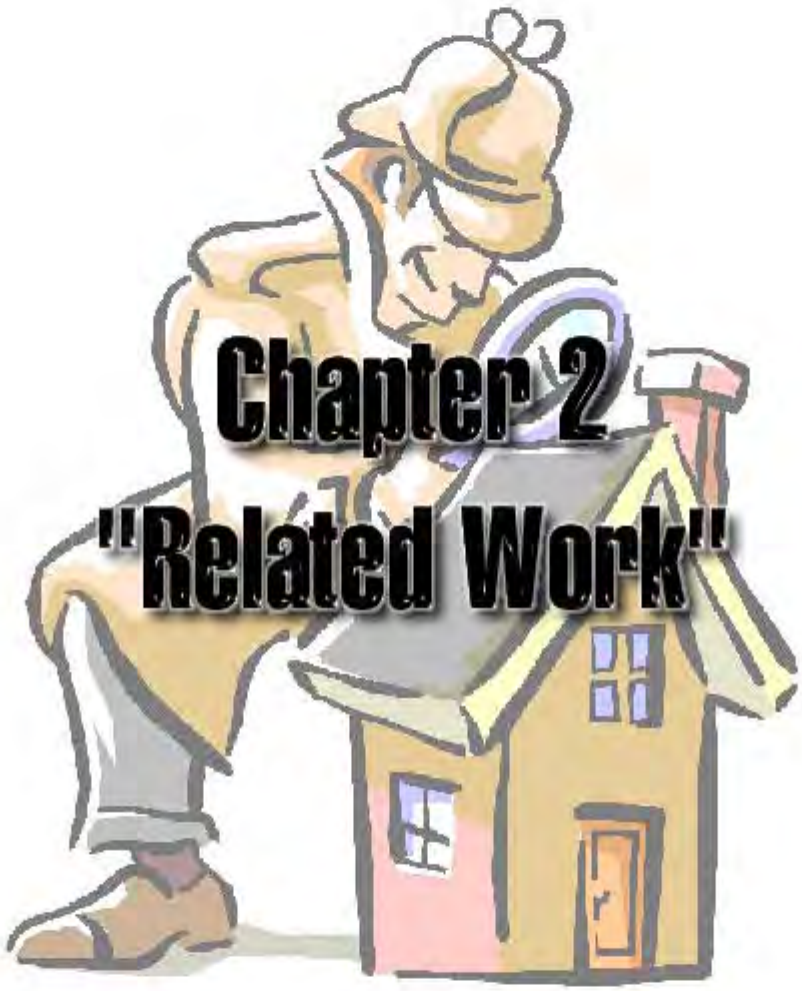
- Chapter 2 deals with related work in the field of VR. We detail the specific problems with immersive VR interaction, as well as their solutions. We examine the various interaction techniques and show how they can be combined into a generic interaction system. We then examine the different proposed methods for implementing such a system, and examine various systems which have done this.

- Chapter 3 details our implementation of an immersive VR interaction system. The abstract framework is discussed first. This framework is then justified by considering how it would implement the various interaction techniques. Details of the implementation are then discussed.

- Chapter 4 lists various case studies on the use of the system. The system was used to implement various projects in the Rhodes University Computer Science Postgraduate school. These projects are discussed, as well as various other small applications designed to shown the usability of the system.

- Chapter 5 details how the system was moved from a single user system to a multi-user distributed system, using the Common Object Request Broker Architecture (CORBA). The chapter begins by giving details about CORBA and its usage. We then proceed with details about how CORBA was used to distribute the system.

- Chapter 6 details the conclusions we gained from the project as well as detailing what future work can be performed on the system.


Chapter 2
"Related Work"

2.1. The Problems with Interaction in VR

Virtual reality aims to allow users to interact with the systems they are using on a more intuitive level (e.g. via gestures and movement as opposed to typing commands on a keyboard). These added interaction possibilities are what make virtual reality so exciting in its application. Unfortunately, they also make it difficult to learn and use efficiently. The most notably successful applications of virtual reality all fall into the realm of spatial visualisation [Mine, 97; Mine, 97-2], with little or no attempt made to allow direct object manipulation. The main reasons for this lack of interactive usability can be summarised as follows:

2.1.1. Object Manipulation

The precise manipulation of objects in the virtual environment is difficult. While one is able to accurately track the positions of objects and represent this visually, the lack of haptic feedback makes it difficult for the user to precisely manipulate objects. Haptic feedback is the term given to 'feel' of an object. This 'feel' is a result of the weight of the object pulling the hand downwards, and pressure exerted by the fingers on the object, all of which the brain registers and uses to help accurately position or manipulate the object. At present, virtual reality is able to reproduce only the visual information about the object. Thus, users may see their 'hands' holding an object, but at no point do they actually believe that they are holding something real.

There is no easy way to simulate the weight of an object. It is possible to simulate, to a certain degree, the 'pressure' felt by the hand as a result of holding the object. The methods for simulating the feel of an object range from electrical stimulation of the nerves of the fingers, to the usage of air sacks to put pressure on the fingers. None of these methods satisfactorily reproduces the sense of touch that a user has when holding a real object. An alternative solution to this problem is to provide real world equivalents of the objects in the virtual world, which the user is physically able to pick up, thus utilising the full range of haptic feedback. The *Virtual Tricorder* [Wloka, 95] is an example of this idea, whereby a 3D mouse is used as the principal input device, and given a representation in the virtual world, corresponding to its physical size, shape, etc. Another example of this type of input device is the *Virtual Remote Control* [Rorke, 99] developed as part of the CoRgi interaction system, and described in detail in the Case Studies section.

Allowing for the constrained movement of objects [Bowman, 95] is another method that helps solve the problem of accurately placing objects inside a virtual environment. Constrained movement means allowing the object only to move in a certain direction at any given time. For example, the user may choose to constrain the object to move only along the x-axis, in which case the y- and z-axis changes that come from the input device are simply disregarded. This idea can be further extended to the case

where numerical input of data (from the keyboard or some virtual representation thereof [Mine, 1997; Mine 97-2]) is allowed for the precise placing of objects.

2.1.2. Equipment Cost

The equipment used to capture data about the user for the virtual environment (e.g. magnetic trackers, which provide a 3D value for their position and orientation) is often prohibitively expensive. Thus one is forced to use a limited number of them (usually 2 or 3). This, in turn, restricts the amount of input one is able to receive from the user, and thus, restricts the usability of the system. The problems associated with having only a limited number of trackers present can often be alleviated using mathematical methods like *inverse kinematics*. In inverse kinematics the positions of limbs, joints, etc. of the user that are not directly tracked can be estimated, based on the known positions of their other limbs, joints, etc. Another possible solution to this problem comes from the fact that, as the technology behind these devices becomes more established, their price will drop.

2.1.3. Measurement Precision Limitations

Another problem associated with many, if not all, virtual reality input devices is that of limited precision. There is a limit to the accuracy of the readings that these devices produce, and they are also prone to an effect known as ‘drifting’ whereby the value from the device changes when it should remain steady (e.g. if the tracker has not moved). This lack of precision means that applications are only able to utilise gross movement on the part of the user (e.g. the overall position of the hand) with the fine movements (e.g. the position of the individual fingers) not being measured. Initially, it was believed that such gross movements were sufficient to create a believable experience. Unfortunately, this lack of measurable fine movements does impact on the usability of the system. The use of gross movement often results in users having to exaggerate their movements in order to use the system, resulting in user fatigue.

The problems associated with the limited precision of the input devices are often a matter of the technology behind the device. As with the problem of cost, as the devices are used, more and more research goes into their manufacture, so they will become more accurate. Other problems, like drifting, can also only be solved with better hardware. New devices are always being developed (e.g. inertial trackers) which solve many of the problems associated with the current hardware.

2.1.4. Physical Work Surfaces

The lack of physical work surfaces in the virtual environment is also a major interaction problem. People depend on naturally occurring physical constraints to give them some idea of the behaviour of objects (e.g. a book pushed over the edge of a desk will fall to the ground [Mine, 97; Mine, 97-2]). These physical workspaces also often provide some form of support for the user, alleviating fatigue and allowing more precise manipulation of limbs. The addition of 'workbenches' and touch-sensitive tablets

[Mine, 97; Mine, 97-2; Rorke 99] in both the virtual and real worlds of the user, look to go a long way towards alleviating this interaction problem. But such devices/objects often restrict the movements of the user in one way or another, and thus their introduction into a general interaction system may have detrimental effects. They have proved very useful in solving specific interaction problems, but as yet, no single device exists which solves general interaction problems.

2.1.5. Interface Standards

Virtual environments also lack a common interface standard. The standard Windows, Icons, Mouse and Pointer (WIMP) interface is now common on all desktop computer platforms, and the user is easily able to identify common interface elements and begin productive work with little or no learning curve. Unfortunately, no such common interface exists in virtual reality, so the user is forced to start from a very basic level whenever a new piece of software is encountered.

The reason for this lack of unity in the field of virtual reality interaction stems mostly from the fact that there is no standard set of input devices. Even with assumptions made as to what input devices are to be used, the range of interaction possibilities makes it difficult to settle on a 'common group' of actions which will be able to service the whole of the virtual reality field. A further problem arises from the fact that the WIMP metaphor is no longer sufficient. Yet, the 'real-world' metaphor where how to use an object may be gleamed from its physical constraints (for example) is also not completely applicable. The lack of information an application is able to convey to the user about the objects in the virtual environment makes it impossible to simulate all the different nuances of the real world accurately.

There are also other very fundamental differences between the desktop and virtual reality interaction metaphors. For example, in virtual reality, the user can be considered to be *inside* the interface [Mine, 97; Mine, 97-2]. As users move around the world, the interface elements that they use to interact with it must move around as well, in order to be easy to locate and reach. These elements also take up valuable space on the display, so they must also be kept out of the 'field of vision' of the user when not needed. Proprioception [Mine, 97; Mine, 97-2] is the term used to describe one’s sense of the position and orientation of one’s body. This idea has been used, with some success, to solve the problem of where to place interface elements so that they are always at hand when the user needs them, yet not constantly obscuring the display. The interface elements are attached to different parts of the user's bodies e.g. behind their heads, out of the field of view, yet they can always be easily reached when needed.

While immersive interfaces are definitely more complex than their desktop counterparts and most desktop specific HCI research is not directly applicable, certain high level concepts do apply. Specifically, the guidelines given by Norman [Norman, 90] apply even more to the immersive interface than they do to the desktop interface. Norman introduces a set of guidelines, which are summarised as follows: Interface elements should have *affordances*. These affordances are elements of the objects that explain their operation to the user. *Mappings* must exist between user actions and their effects on the system i.e. any input action by the user should produce a proportional output action in the system.

Feedback is also a major factor contributing to a usable interface. The user should never be in doubt as to whether or not an action has been accomplished or not. Bowman [Bowman, 95] adds that good feedback should naturally follow from good mappings.

2.1.5.1. 3D Widgets

The success of the WIMP interface can be attributed mainly to the fact that it provides users with a familiar set of tools, no matter what particular application they may be using. Virtual reality can offer this same familiarity, but not just widgets that look and act like widgets that the user is familiar with from other computer programs. Rather, virtual reality can take the idea a step further, presenting the user with widgets which look and act like objects or tools with which the user is familiar from the real world.

The interface elements in the virtual world should all be modelled, as closely as possible, after real world equivalents. Thus, users should be able to immediately identify the operation of the various elements from their knowledge of the real world [Norman, 90]. For instance, the CoRgi system (and most other interactive VR systems) uses a model of a human hand as the basic interaction element, with gestures being the basic operations. This method has proved very intuitive, with even novice users being immediately able to identify the interface element and use it to manipulate objects in the world, based on their intuitive knowledge of the working of the human hand.

The creation of widgets that conform to a user's intuitive knowledge of the real world is a subject based largely in the field of psychology. Norman [Norman, 90] describes some of the basic requirements of an interactive widget, based on an understanding of human psychology. These guidelines are covered in more detail later in this chapter.

2.2. Requirements of a VR Interaction Toolkit

2.2.1. 2D vs. 3D Interaction

Research into human computer interfaces (HCI) has been ongoing ever since the first computers were built. Many complex and usable 2D interfaces have been developed for the standard WIMP interaction metaphor that is prevalent on today's desktop computer systems. Unfortunately, much of the 2D-interface research is not directly applicable to interaction in an immersive system.

Bowman [Bowman, 99] explains that the main difference between the 2D and 3D interaction system is that the desktop (2D) interface is inherently more constrained than its immersive (3D) counterpart. He goes on to point out that most desktop applications use only two dimensions of input (or 2 degrees of freedom (DOF)), which map directly onto the standard 2D controller, the mouse. In an immersive system, on the other hand, the user often has to deal with input that has 6 DOF (3 positional and 3 rotational) input, which immediately places higher cognitive load on the user.

Several researchers are currently working on incorporating these types of constraints into immersive interaction systems [Lindeman, 99; Rorke, 99; Wloka, 95]. Most of this research involves giving users some form of flat surface (e.g. a touch-pad) on which they can select/manipulate objects, as they would use a mouse or a touch-sensitive screen. These types of interaction technique have advantages in that they offer a sense of touch to the user (supplied by the tablet) which would be impossible to generate otherwise and which adds a feeling or reality to the system. Unfortunately, they are rather restrictive in that they are forcing a 3D immersive medium to act in the same way as a 2D desktop medium. Our experience with this type of interface is that it is popular with users, but limited in what it is able to add to the system.

The desktop interaction system has also had many years in which to mature, resulting in a standard set of interaction metaphors and widgets with which users have become familiar. Interactive, immersive systems have only been around for about 10 years. They do not have a standard set of input devices or interaction metaphors (like the 2D widget). Most immersive VR systems do utilise certain common devices, like the head-mounted display (HMD) and magnetic trackers, but how these are used is not standardised. For example, one developer may decide to track the position of the user's hand, while another decides to track the position of one of the user's fingers. Both methods attempt to utilise the user's intuitive abilities to select objects using their hand, but the resultant effect in each case is quite different. These devices are also very costly, so users lack exposure to the technology.

Jacob, Deligiannidis and Morrison [Jacob, 99] explain the difference between the traditional desktop interface and immersive interfaces, by considering the different flows of data that each operate with. The standard desktop interface operates in a serial manner with *tokens* or commands being placed into a single command stream and processed by the system. Immersive interfaces are typically characterised by the use of multiple, parallel command streams. They summarise the particular differences as follows. Considering the differences of desktop *versus* immersive interfaces:

- Single-threaded input/output *versus* parallel, asynchronous, but interrelated dialogues.
- Discrete tokens *versus* continuous and discrete inputs and responses.
- Precise tokens *versus* probabilistic input, which may be difficult to tokenise.
- Sequence, not time, is meaningful *versus* real-time requirements and dead-line based computations.
- Explicit user commands *versus* passive ("non command-based") monitoring of the user.

2.2.2. Design Philosophies

The eventual design of a VR interaction system depends largely on the particular application being developed. The designer is always required to make trade-offs between various different aspects of the system and their choice of what aspects to focus on should be driven by some eventual goal application. In the field of interactive VR, there are also several basic philosophies that a designer needs to be aware of when designing systems. These philosophies include ideas like the *naturalist* versus the *magical* interaction approach, as well as more fundamental ideas like *direct* versus *indirect*

interaction with objects. Researchers are still divided as to which philosophy is better and the end result usually depends on the final application and, to some degree, on the developer's preference.

2.2.2.1. Naturalist vs. Magical

Many interactive VR applications attempt mimic the way that we interact in the real world (the *naturalistic* approach) using the assumption that we will be able to use our intuitive knowledge of real world interaction to better utilise an interactive VR system. Unfortunately, due to various factors these systems are often not able to recreate real world interaction accurately [Rorke, 98]. Bowman, Mine and many other researchers [Bowman, 99; Mine, 97] argue against the totally naturalistic approach to interactive VR, opting rather for a mixture of natural and *magic* (not based on real world interaction) interaction metaphors. VR enables the user to move beyond what is possible in the real world, so that utilising a purely natural interaction metaphor may preclude many useful VR applications which would not be possible in the real world.

2.2.2.2. Direct vs. Indirect

Another major design choice for interactive VR systems is the *direct* versus the *indirect* approach. In the direct approach, the user selects and manipulates objects in the VE using simple mappings to some movement (e.g. hand movement). This is analogous to the way in which these actions are accomplished in the real world and again, the assumption is that intuitive knowledge of the real world will allow the user to interact with greater efficiency. The indirect approach forces the user to interact with the system using some intermediary (e.g. widgets or tools). This technique is analogous to the desktop interaction metaphor, where all user input is mediated through the use of simple widgets. Mine [Mine, 95] believes that interacting with an environment indirectly is a natural interaction metaphor based on the fact that many actions in the real world are carried out through tools and not directly. Norman, [Norman, 90] on the other hand, states that interaction should be as direct as possible and the addition of intermediate tools often serves to complicate the system. Whereas direct interaction does have a very strong intuitive basis, the lack of accuracy when tracking the user's hand, for example, leads to problems with purely direct interaction techniques. Often, indirect techniques offer the user finer control at the expense of intuitive use.

2.2.2.3. Formal Interface Design and Evaluation Methods

Many of the interaction methods currently used in interactive VR were designed solely by intuition on the part of the developer. Most researchers agree that intuition is not sufficient to produce usable systems and that some form of formal evaluation is necessary [Bowman, 97; Bowman, 99; Hix, 99; Poupyrev, 97]. There is a growing trend in the field to apply various system design ideas (taken from standard HCI research) to the creating of immersive interfaces [Bowman, 99; Hix, 99; Poupyrev, 97] that will hopefully produce some standardisation in the field in the near future. There are many

different formal evaluation techniques available, most involving the creation of progressively refined versions of the system, with user testing of each design to identify weaknesses.

2.2.3. The Basics of Interaction

Bowman [Bowman, 99] suggests that there are several universal tasks that are common to all interactive VR applications and that implementing these tasks correctly can lead to a usable application. These universal tasks are *travel*, *selection*, *release* and *manipulation*. Each of the separate universal tasks will now be examined in more detail.

2.2.3.1. Travel

Travel or *viewpoint control* involves changing the user's view of the VE. The most common techniques for choosing where to move in VR environments are the *gaze-oriented*, *pointing* or *discrete target specification*.

2.2.3.1.1. Continuous Specification

Both gaze-oriented and pointing travel are termed to be *continuous specification* travel techniques, because the user is able to change the direction and speed of travel continuously throughout the movement. These techniques require constant cognitive thought on the part of the users (i.e. they must constantly choose where they wish to go next). While this has the advantage that it offers great flexibility of movement, the amount of thought that goes into the movement is considerably larger than when using the discrete target specification technique. The term *cognitive thought* is used to describe the measure of thought that the user has to apply in order to achieve a given task. The more cognitive thought required by the user to effect travel, the less they have available to perform useful actions in the system.

In gaze-oriented travel, users are moved in the direction in which they are looking. In pointing travel, the orientation of a user's hand is used to select in what direction the user wishes to travel (i.e. userS point in the direction in which they want to travel.)

Bowman shows that the pointing technique only becomes useful when relative motion travel (i.e. moving relative to some object in the VE) is required by the application [Bowman, 97-2; Bowman, 99]. Users of the pointing technique often approximate a gaze technique by placing a 'hand' in front of them and simply travelling where they are looking.

2.2.3.1.2. Discrete Target Specification

The discrete target specification technique requires a number of set viewpoints in the VR environment, which the user is able to move between at will. The lower flexibility of this technique is

counterbalanced by the fact that very little cognitive thought is required on the part of users - they simply select their destination and the application takes them there.

Discrete target specification allows the user to travel only between a set of pre-defined destinations. These destinations are usually pre-programmed into the application, but methods for defining them dynamically while the application is running have been demonstrated.

When using the discrete target specification technique, it is still advisable to move the user between the two points in a smooth manner and to avoid simply changing the viewpoint to that position/orientation of the required destination – a method known as *teleporting*. Instantaneously moving the viewpoint to a new destination causes a dramatic drop in spatial awareness and can cause user discomfort. The speed with which users are moved between the pre-set points has been shown to have little effect on their spatial awareness, contrary to intuition [Bowman, 99; Bowman, 97-2].

2.2.3.1.3. Controlling Speed, Acceleration, etc.

As well as deciding where to go, travel techniques require methods for allowing the user to start and stop travel, as well as select velocity and acceleration for the movement. Starting and stopping travel is usually linked to some sort of command from the user e.g. using some hand gesture. The orientation of the viewpoint is usually linked to a tracker on the HMD, allowing users to *look around* in the environment as they would in real life. The speed at which users travels is also an important factor, but in most cases, it is sufficient to simply have a set speed, as opposed to giving the user specific speed control. Mine [Mine, 95] details various speed control techniques.

The addition of constraints to movement has been shown to relieve much of the cognitive load placed on the user. The most common form of constraint is one, which appears in the real world i.e. the constraint to travel at a fixed height from the 'ground'. Bowman [Bowman, 99; Mine, 95] demonstrated a marked improvement in spatial awareness, speed and other metrics when constraining users to move at a set height from the ground (i.e. constraining their movement to 2 dimensions).

2.2.3.2. Selection and Release

The two universal tasks of selection and release are often considered together, as they tend to complement each other. Selection involves the user communicating with the application about which object in the VE they wish to manipulate. Release is the message from the user to the application that they do not wish to perform any further operations on the currently selected object. Selection is often the precursor operation to *manipulation*.

The selection and manipulation operations are often at the heart of the naturalist interaction implementation. Developers often implement selection to mimic the way we pick up objects in the real world. In many cases, this purely naturalistic approach to selection is not appropriate.

Bowman states:

"When careful consideration is taken, it should be obvious that a real-world technique would be inadequate for selection and manipulation tasks in VEs, since the tasks we wish to perform go beyond real-world capabilities. In the same way, a travel technique based on physical walking will be completely inadequate if the application requires travel on a global scale. The power of VEs is not to duplicate the physical world, but to extend the abilities of the user to allow him to perform tasks not possible in the physical world."

[Bowman, 99]

The majority of applications employ a simple naturalist selection technique whereby the user selects an object by moving their 'hand' to within the boundaries of the object. This interaction technique is very intuitive and works well for small VEs where all the interesting objects are within the reach of the user. In the case of larger systems, more complicated techniques have to be employed. Most of the more complicated selection techniques can be categorised as being based on one of the following ideas, *arm-extension*, *ray-casting* or *image plane*.

2.2.3.2.1. Arm-Extension Techniques

Arm-extension techniques allow users to extend their reach in the VE. This addresses the problem of not being able to reach object outside of the physical reach of a user's arm. There are many different methods for deciding how a user's reach might be extended. The most common of these is *scaling* [Bowman, 97; Bowman, 99; Mine, 97] and a technique called *Go-Go* [Bowman, 97; Bowman, 99; Poupyrev, 96].

The scaling technique involves scaling the size of users so that their arm reach is extended to a length sufficient to reach a particular object [Mine, 97]. Technically, the scaling technique uses *occlusion* (an *image plane* technique) or *ray-casting* to choose what object the user wishes to select, then scales the user appropriately, so that their reach now extends to the object. Scaling techniques can cause user disorientation and discomfort, especially if not implemented correctly.

The Go-go [Poupyrev, 96] technique maps the length of the user's reach in the VE onto the function

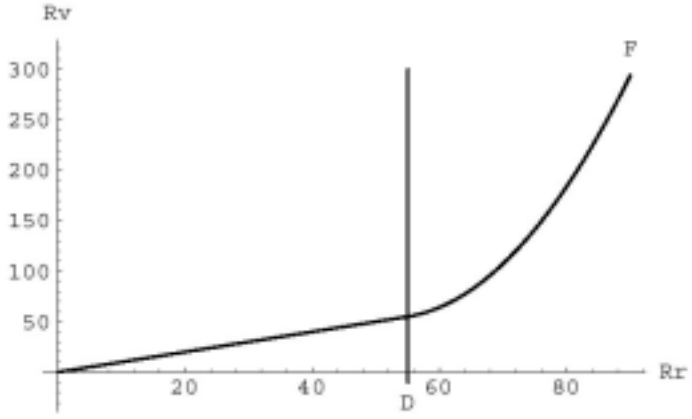


Figure 2 – 6 – The Go-Go Mapping Function [Poupyrev, 96]

shown in Figure 2 – 6. Up to a certain distance (**D**) from the user, the distance reached is a linear function of the distance from the user's viewpoint. After point **D**, the mapping function becomes non-linear. The reach of users with this method is still finite, but it extends far beyond their physical reach. There are several variations to this method [Bowman, 97; Bowman, 99] involving mainly changes to the mapping function and additions like extending the reach over time.

There is an additional subset of arm extension techniques that use some indirect means to extend and retract the users' reach. Examples of this are the *reeling* techniques used by Bowman [Bowman, 99]. With the reeling techniques, a user is able to manually control the reach of their virtual hand using buttons to extend or reduce the reach. Bowman found that, while these techniques had no basis in real world interaction, they proved popular with users.

2.2.3.2.2. Ray-Casting Techniques

Ray-casting techniques also allow users to extend their reach in the VE. Ray casting in a VE is similar to using a laser-pointer in the real world. The system defines a 'ray', emanating from the user's hand and extending into infinity. Users control the direction of this ray by altering the position and orientation of their hands. When an object intersects the ray, it may be selected. Ray-casting techniques extend the real world idea of pointing at objects. Bowman [Bowman, 99] showed that ray-casting was the most efficient way of selecting objects that were out of the user's reach, provided that they were not too far away to be easily seen and that there were no other objects obscuring the user's view of them. Where ray-casting fails is in the manipulation that is possible on the selected object.

2.2.3.2.3. Image Plane Techniques

Image plane techniques are a combination of both 2D and 3D interaction. Selection is carried out in the view-plane by disregarding the depth dimension of the scene. An example of an image plane technique is *occlusion*, where users select objects by occluding the object in their view of the VE. Most image plane techniques may be considered to be ray-casting techniques, with the ray emanating from the user's eye.

2.2.3.2.4. Other Selection Techniques

There is also a further group of selection techniques, which do not fall into any of the above three categories. Many of these types of techniques are based on a naturalist approach with the additions of extensions that make the technique more suitable to an immersive environment. An example of this type of method is the World In Miniature (WIM) technique [Stoakley]. In the WIM technique users are presented with a miniature (or *dolls-house*) version of the VE which they can use to select an object in the VE and manipulate it. Any changes made to the WIM are also made to the full size VE.

2.2.3.2.5. Controlling Selection

One other aspect of selection that needs to be considered is that there needs to be some specific command from the user to the application to select a particular object. The reason for this is that there will often be a number of objects that can possibly be selected at any given time. Users need to be provided with some sort of feedback as to which objects they are currently able to select, and some method to inform the application that the currently selected object is, indeed, the one required for selection. This command is usually in the form of some sort of gesture. The release of a selected object is often accomplished by the same method that was used to perform the final selection. Again, a command needs to be given to the application that the currently selected object needs to be released.

2.2.3.3. Manipulation

The manipulation of objects in a VE is the eventual goal of a large number of interactive VR applications. The most common method for manipulating the position and orientation of objects in the VE is simply to match their position and orientation to that of the user's hand. This is a very naturalist form of interaction technique that works well when fine manipulations of objects are not required. The other common manipulation technique is the use of *tools*. The user is given tools in the VE, each of which causes some manipulation effect on the selected object. This form of interaction is less intuitive than the previous method (being an indirect interaction method), but is possibly more powerful in that there is a wider range of possible manipulations available to the user.

The addition of constraints to the system has been shown to greatly improve the accuracy with which users are able to manipulate objects [Bowman, 95]. Even simple constraints like allowing the user to change only the position of the object have been shown to greatly improve accuracy. Limiting the number of states that an object can be in has also been shown to improve accuracy. *Snapping* of position and orientation limits the flexibility that the user has when manipulating objects, but increases the accuracy of the final placement [Bowman, 95; Mine, 95].

The naturalist interaction technique is often closely tied in to the selection technique employed initially. Ray-casting is a very intuitive method of object selection, but does not combine well with naturalistic manipulation, since the user's 'hand' is located at a different position and orientation from the object being manipulated. Changing the position of the object is still possible, but accurately orienting the object becomes impossible. Variations of the ray-casting selection technique have been developed to overcome this problem. The HOMER technique [Bowman, 97] does this by relocating the user's 'hand' to the position of the selected object for the duration of the manipulation.

2.2.3.4. System Commands

System commands may be considered to be a specific selection/manipulation exercise but, as far as the user is concerned, the actions are sufficiently different to warrant a new universal task. Systems

commands are often issued through a menu/button type interface, similar to that in the desktop interaction systems i.e. the user is required to activate some form of control in the VE in order to issue a particular command. This is an indirect interaction metaphor and presents several problems in the VE. Firstly, there is no obvious region in which to place these menus. Often, they are implemented as free floating objects in the VE and must be interacted with through the standard selection/manipulation metaphors provided by the system. This creates several problems. For example, it is now possible for users to 'lose' their control system (menu) somewhere in the VE. Many researchers have proposed various methods for attaching these menus to the user's viewpoint. Thus, as users move around the VE, so their interfaces move to follow them. In order to reduce screen clutter, these menus are often placed out of sight of the user and are accessed only when necessary. Mine [Mine, 95] proposes using the users' own sense of their bodies to place menus where they are always easily within reach, yet do not obscure a user's view of the system when not needed.

The preferred method for issuing system commands is definitely some form of direct method. Speech recognition is the most intuitive way to do this, mirroring the natural way in which commands are issued in the real world. Recent advances in speech recognition software have brought this interaction metaphor within reach of most computer systems. The other direct interaction metaphor for issuing system commands is that of gestures. Most VR systems track the orientation, position and finger bends of a user's hands. Since gestures are the primary method for implementing many of the subtasks in the other universal tasks, there are not many easy/intuitive gestures left for system control. System control functions can also be many and varied, making gesture control less appealing than speech input.

2.2.4. Bowman's Taxonomies

Bowman [Bowman, 99] organises these universal interaction tasks into various taxonomies, breaking them up into subtasks and detailing the different methods used to accomplish these subtasks. These taxonomies are presented in Figures 3 – 1, 3 – 2, 3 – 3 and 3 – 4. Any generic interactive VR system needs to be able to handle all the tasks listed in these taxonomies. Additionally, the subtasks should all be implemented in some form of generic framework, allowing particular subtasks to be easily interchanged to suit the needs of different applications. In addition to Bowman's list of universal tasks, we propose a further universal task, *System Command*. The system command taxonomy is detailed in Figure 2 – 5. Bowman's taxonomies and universal tasks were based on what a user needs to do in an immersive VR environment.

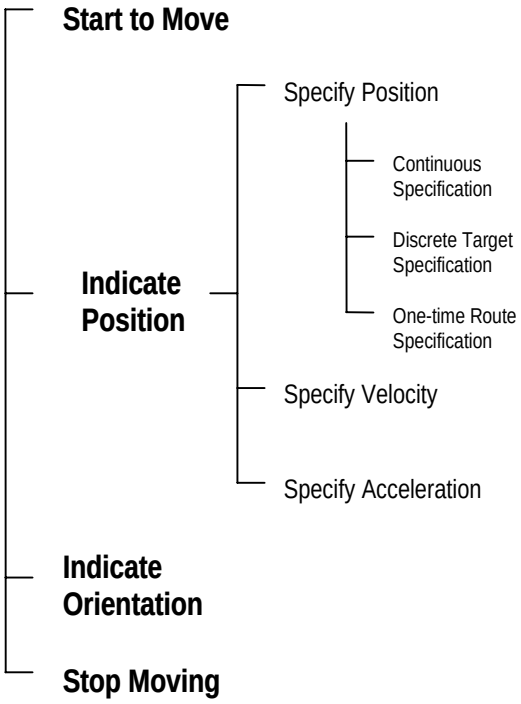


Figure 2-1 – Travel Taxonomy [Bowman, 99]

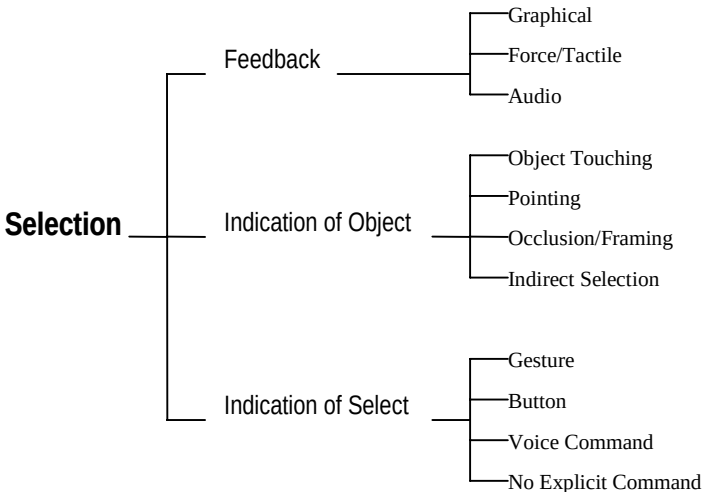


Figure 2-2 – Selection Taxonomy [Bowman, 99]

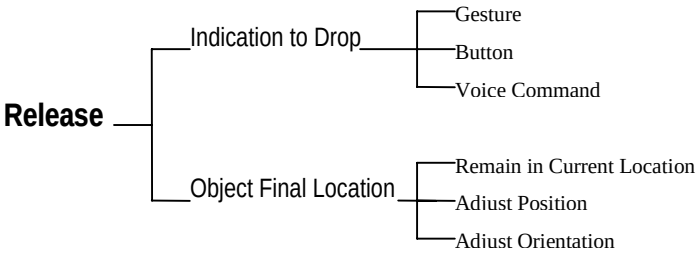


Figure 2-3 – Release Taxonomy [Bowman, 99]

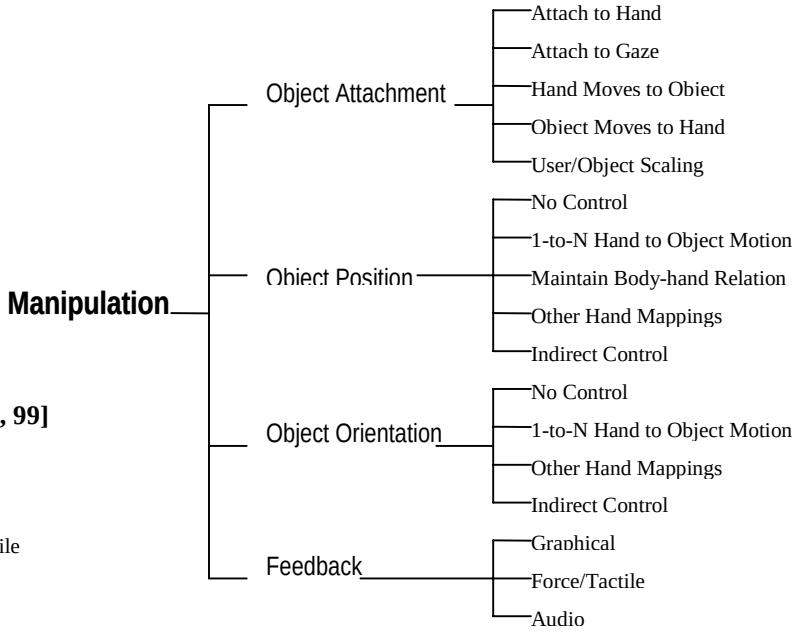


Figure 2-4 – Manipulation Taxonomy [Bowman, 99]

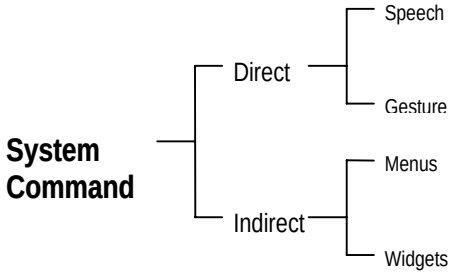


Figure 2-5 – System Commands Taxonomy

2.3. Interaction Frameworks in VR Toolkits

The idea of a set of application libraries and abstractions that ease the creation of VR applications is not a new one. Several companies and research institutions produce VR toolkits for use in the fields of visualisation, simulation, etc. Bowman's taxonomies give a good overview of what the user wishes to accomplish in an immersive, interactive VR system, but they do not give any idea of how this should be accomplished in a generic way.

Taking standard desktop interface toolkits as a starting point, most interface toolkits are comprised of three layers [Kessler, 99]: the user input and graphical output management layer, the interactor framework layer and the interactor set layer. The most important layer, from the point of view of interaction, is the interactor layer and the support it provides to implement the interactor set. The following is an overview of some currently available VR application toolkits concentrating on how they implement user interaction in immersive systems.

2.3.1. The 'Leave it Up to the Developer' Approach

The majority of currently available VR application toolkits concentrate mostly on the user input and graphical output layer of the interface toolkit taxonomy. Products like the Sense8 WorldToolKit™ [Rahn, 98] (WTK), the Swedish Institute of Computer Science (SICS) DIVE (Distributed, Interactive Virtual Environment) [Stahl, 97] and the University of Alberta's MR Toolkit [White, 99; Shaw, 93] all fall into this category. The toolkits abstract the input and output device specifics away from the developer, allowing the developer to focus on the semantics of the application itself, rather than the hardware specifics of a particular platform. All these toolkits provide the basic functions necessary to implement an interaction system (e.g. collision detection, access to objects in the system, reading input devices, etc.). Unfortunately, they do not provide any framework to ease the implementation of the interaction system or to support the reuse of interaction code in the system. Kessler explains this problem as follows:

> *"… many VE application development systems treat interaction as a process of translating user input into a change in the environment model. Many interaction techniques, however, involve complex relationships between user input, the state of the environment, and changes to the model, both geometric and abstract. In order for a complex interaction technique to be easily incorporated into a VE application, it must be encapsulated into an interactor that can be easily instantiated"*
>
> [Kessler, 99]

Some of these toolkits (e.g. DIVE) do go a step further and implement actual interaction techniques i.e. interactor sets. Yet the range of implemented techniques is limited and there is no unifying framework i.e. interactor layer, upon which all these sets are built. This is not to say that these toolkits cannot implement interaction in a generic way, only that this level of abstraction has not yet been implemented in these particular cases.

2.3.2. The Interactor Layer

Various models have been proposed for the problem of splitting up a generic VR application into separate parts that can execute independently of one another. Shaw [Shaw, 92; Shaw, 93] separates the system up into 4 parts, *Interaction*, *Presentation*, *Computation* and *Geometric Model*. Robertson [Robertson, 89] separates the system into 3 parts, *User*, *User Discourse Machine* and *Task Machine*. Both of these systems propose the complete separation of the interaction system from the other parts of

the application, an idea mirroring that found in the desktop interface toolkits approach. This separation of interaction layer from the underlying layers for user input and display means we may safely assume that a generic form of interface abstraction can be adapted to suit any form of VR toolkit. The interaction layer behaves (to the underlying toolkit) as a simple application, while providing high level routines to the developer. Thus, we may develop our interaction layer independently of the VR toolkit with which it will interface. The implementation of particular instances of this layer will have a certain amount of dependence on the underlying VR toolkit.

The differentiating factor between current immersive VR toolkits is the way in which they handle communication between the user, objects in the system and interface elements. The two main methods for doing this are the standard desktop methodology, using an *event queue* and the idea of *data flow*.

2.3.2.1. Event-Based Systems

Event-based systems operate by placing *tokens* or events into a central event queue, to signal operations in the system. The event system is used in current desktop interfaces and is characterised by the inherently serial nature of the interactions i.e. each interface item takes turns in communicating with the rest of the system by issuing simple commands. Even when there are several hundred interface elements, the input stream is treated as a single, multiplexed stream operating in half-duplex between the system and the interface. Event-based systems are characterised by the use of *callbacks*, which are executed in response to the various events.

The event-based system has many years of study and optimisation behind it, but the inherent assumptions made about the use of a 2D interface do cause several bottlenecks in the system. Additionally, the system is not ideally suited to the immersive environment, as mentioned in Section 2 [Jacob, 99]. These problems aside, it is possible to implement immersive versions of the interactor layer using an event-based system.

2.3.2.1.1. SVE (Simple Virtual Environment) & SVIFT (Simple Virtual Interactor Framework and Toolkit)

The SVE toolkit is a research toolkit developed by the Virtual Environments Group of the Graphics, Visualisation and Usability Centre at the Georgia Institute of Technology in the USA. The toolkit provides the developer with an abstraction level to handle the basic functions of VR applications e.g. rendering to various output devices, interfacing the input devices, simple geometric transformations, etc. [Kessler, 97].

As with the toolkits mentioned earlier, the SVE itself provides no abstraction to support interaction. However, Kessler [Kessler, 99] developed an extension to SVE, the Simple Virtual Interactor Framework and Toolkit. SVIFT was designed primarily to assist in the creation of immersive interfaces for architectural applications. The system aims to reproduce the design of interaction toolkits used for standard desktop interfaces, with provision made for the special requirements of immersive interfaces.

SVIFT is based on the idea of *interactors*. These interactors are defined to be software encapsulations of the interactive behaviour between immersed users and the environment on which they wish to act [Kessler, 99]. The interactors usually comprise some geometric form, with details of their reactions to certain events (e.g. selection, activation). Interactors may respond to events generated by the user, or through changes in the environment. SVIFT events are handled by the standard event queue that forms part of the SVE toolkit.

Using the SVIFT it is possible to implement all the major interaction techniques as well as interactive elements such as 3D widgets, etc. The toolkit provides a set of classes that the developer can use to create these elements with a minimum of difficulty. Unfortunately, as SVIFT and SVE are based on C (and not C++), code reuse (e.g. through polymorphism) is reduced. In addition, as discussed earlier [Jacob, 99] the event-based system of handling interface events is not well suited to generic immersive VR applications. Relating the system back to the hierarchy introduced earlier, SVE provides the user input and graphical output layer, while SVIFT provides the interactor framework and interactor set layers.

2.3.2.2. Data Flow Systems

Steed describes a data flow system as follows:

> *"A data flow model describes a system in terms of the data being passed between functions that transform its state. The tracing out of all such data flows through the system forms a directed graph, with nodes corresponding to functions and arcs indicating the possible routes for data to take."*
>
> [Steed, 97]

Data flow starts at trigger nodes, which form the external interfaces to the input devices for the system. Once a trigger is activated, data flows from it, to any other nodes that connect to it. These nodes, in turn, propagate the data into other nodes, and so on, until the data is finally used to affect the system. Each of the nodes in the system is able to receive multiple streams of data. The nodes perform some processing on the streams, producing a single resultant output stream that is passed on to the rest of the system. The resultant output data may be passed to multiple recipients, but all recipients receive the same output. A data flow system can be directly represented by a directed graph.

According to Jacob [Jacob, 99], data flow systems describe immersive interfaces more accurately than event-based systems. This can be attributed to the inherently parallel nature of the input into an immersive system (see Section 2). With a standard desktop system, only one input device is in use at a given time (e.g. a user will usually not use both keyboard and mouse simultaneously). On the other hand, in an immersive system, many of the input devices (e.g. the trackers) provide continuously updating streams of data about the user, all of which must be processed and used in the system. The operations of immersive systems (e.g. moving the user's viewpoint) are often not easily tokenised (i.e. it is difficult to represent the actions required using only a single event token). More often a continuous stream of data is required to implement a single operation. Immersive systems often have real-time

requirements for their inputs. For example, when users move their hands, the visual representation of a hand in the VE must move immediately or the immersive effect of the system is lost. Event-based systems do not have explicit real-time constraints and rely more on the sequence of events as opposed to their exact time. Probably the most important difference is in the type of commands received from the user. In the desktop event-based system, there is a finite list of explicit commands that can be used to operate the application. Immersive systems, on the other hand, rely on passive, continuous monitoring of the user in order to ascertain what changes need to be made to the system.

The ideal case is a hybrid system that uses both event based and data flow ideas, such as that designed by Jacob [Jacob, 99]. Currently though, most immersive interactor layers implement a data flow idea. The following are examples of interactor layers implemented using data flow systems.

2.3.2.2.1. VEDA (Virtual Environment Dialogue Architecture)

VEDA (Virtual Environment Dialogue Architecture) is an immersive VR system developed at University College, London in England. The system defines the user interface and interaction system in terms of a *dialogue structure* [Steed, 94; Steed, 96; Steed, 97]. This dialogue structure is a particular implementation of a data flow network. The system was designed with the intention of allowing users to build up an interface immersively, from within the application.

Each input device in the system returns one or more streams of data. These streams are send to *virtual tools*, which manipulate the properties of the environment and its objects. Between the tools and the input devices, the stream of data passes through various *nodes*, which perform simple operations (e.g. logical conjunction) on multiple streams of data, to produce a resultant output. This output is then passed on to the next node in the system, or to the tool.

The VEDA system implements an immersive interface building system. The interface is built up as a 3D graph, with pipes representing the flow of data. The system also uses a *heaven and earth* style interface, where a distinction is made between the application being developed and the development of its interface [Steed, 94].

2.3.2.2.2. VB2 (Virtuality Builder II)

Virtuality Builder II (VB2) is an experimental VR toolkit developed at the Swiss Federal Institute of Technology with the aim of experimenting with 3D interaction techniques and to provide a basis for the construction of interactive applications. The goal of the system is to place users in the loop of a real-time simulation, immersed in a world both autonomous of and responding to their actions [Gobbetti, 93; Gobbetti, 94]. A VB2 application comprises of various different processes, each executing separately, but communicating via *inter-process communication* or IPC calls. A central application process manages the model of the virtual world, and simulates its evolution in response to events coming from the processes that are responsible for reading the input devices.

During interaction, the user is the source of a flow of information, propagating from the input device sensors, and manipulating the environment. Multiple *mediators* can be interposed between sensors and models in order to transform the information according to interaction metaphors. The application can be viewed as a network of interrelated objects (a data flow network) whose behaviour is specified by the actions taken in response to changes in the objects on which they depend [Gobbetti, 93]. The dynamic components in the system are modelled by *active variables* (which store the state of various properties of the system), while relations are modelled using *hierarchical constraints* (a method of declaring long-lived, multi-way relations between active variables). *Daemons* are used to sequence between system states in response to changes in variable values and an incremental constraint solver (based on *SkyBlue*) efficiently evaluates the constraint network. The constraints are not limited to simple algebraic expressions, but can be general side-effect free procedures that ensure the satisfaction of the constraint after their execution by computing some of the constrained variables as a function of the others [Gobbetti, 93].

The VB2 system implements a very detailed data flow model where the links between objects can be either on or off (depending on the constraints) and must be evaluated whenever new data needs to be propagated. Using this system, techniques such as direct manipulation, gestural input and virtual tools can all be implemented in a generic way. Additionally, since all the tools are simply nodes in the data flow network, they can be combined to create powerful composite tools [Gobbetti, 94].

2.3.2.2.3. VRML ’97 (Virtual Reality Modelling Language ’97)

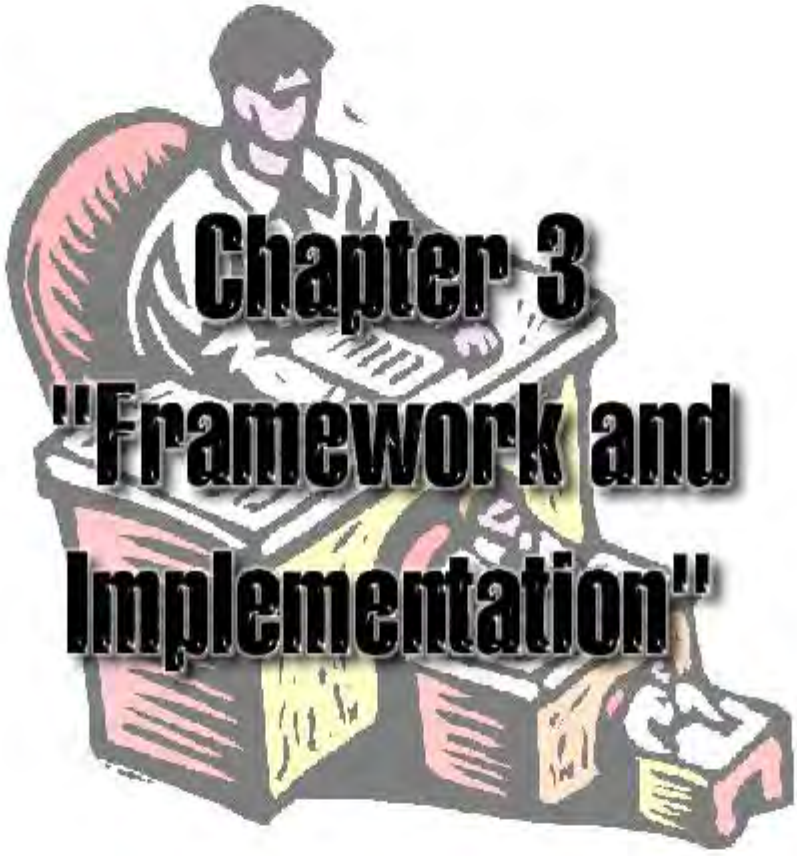
The Virtual Reality Modelling Language (VRML ’97) is a standard for publishing 3D content on the World Wide Web [Carey, 97]. While this is not an immersive environment, the VRML standard is used widely for specifying VR environments and may be used to construct immersive VR environments. VRML is also not a VR toolkit as such, rather it can be considered as being a transmission standard for specifying VR systems. VRML does include support for limited interaction system development using a data flow type system.

The VRML system is based on a scene graph hierarchy for specifying the properties of the objects in the system. Each node in the system may have one or more inputs. The arrival of data in these input can trigger the execution of an internal function (or *script*), which may propagate data to any other connected nodes (a standard data flow network). VRML allows for data such as colour and position. to be exchanged between nodes. User interaction is achieved through the use of various *sensors*, which can be activated by the user in various ways. The data from the sensors is transformed in the *interpolator* nodes and eventually affects the environment.

VRML allows for the specification of a data flow system that can be rendered using various toolkits. Steed [Steed, 97] gives a good comparison of VRML with formal data flow systems (e.g. VEDA).

2.4. Summary

Past research into the workings of immersive VR interfaces provides important insight into what is required to achieve a usable immersive, interactive VE. The eventual goal of any VE interface should be to allow the user to interact in an effective way, while at the same time, keeping the cognitive load introduced by the interface to a minimum. Reducing the cognitive load introduced by the interface allows users to concentrate on the task they are trying to achieve, rather than concentrating on using the interface. Many VR systems attempt to mirror the real world in as many ways as possible. While this naturalistic approach is important and effective, we believe that unnatural (or *magic*) interaction techniques can go a long way to making an interface more effective and usable. The set of taxonomies for travel, selection, release, manipulation and system commands give an abstract view to application developers of what their interfaces need to be capable of doing. Various VR toolkits have been studied and the different methods for implementing them (e.g. data flow *vs.* event based) have been compared. A brief overview of various systems has also been provided for the purposes of later comparison with the CoRgi interaction system.


Chapter 3
"Framework and
Implementation"

3.1. Overview

The interaction system implementation was built on top of the CoRgi VR toolkit (developed at Rhodes University) using a simplified data flow model and the concept of *listeners* to implement interaction between the user and the system. A data flow model (covered in the previous chapter) describes the operation of an application in terms of input data, which is transformed via various *nodes* in the system, until it is finally used to update the application, producing some form of output. The sources and destinations of this data are not important to the data flow model (i.e. a piece of data from an input device will be treated in the same way as a piece of data of the same type generated by the application itself). The data flow model used in the CoRgi interaction system is loosely based on the Virtual Environment Dialogue Architecture (VEDA), which is similar to the data flow model used in the Virtual Environment Modelling Language (VRML) 2.0 standard [Steed, 96; Steed, 97]. The use of a data flow system to describe the interaction between various objects is justified by considering the differences between the standard desktop interface (commonly an event-based system with a single command stream) and an immersive interface [Jacob, 99].

Additionally, the CoRgi interaction system uses ideas from the MR Toolkit [Shaw, 92; Shaw, 93] and the Cognitive Coprocessor Architecture [Robertson, 89] to define the interfaces between the various parts of the system. Shaw [Shaw, 92] proposes that VR applications be divided into four separate parts, *presentation*, *interaction*, *computation* and *geometric model*. Robertson [Robertson, 89] separates his system into three parts, a *user*, a *user discourse manager* (interaction) and a *task machine*. Both support the idea of separating the application program from the interaction part of the system, as was done with the CoRgi system. This separation, along with a thorough definition of the interfaces between the separate parts, allows the system to be developed as several separate entities, each optimised for its particular task.

3.2. Abstract Implementation of the Model

As an aid to understanding the system it is presented first in an abstract form. The abstract system will then be expanded with more implementation specifics followed by an example of the use of the system by a developer. The example consists of a simple Table Tennis game where the user controls a bat, which must be used to hit a ball. In a distributed version of this system (explained in the CORBA section) one user could compete against another user.

The overall interaction system can be broken up into six main parts. These six parts are the *system component*, *input component*, *interaction component*, *intermediaries*, *entities* and *widgets*. The recommended data flow relationships between these parts are detailed in Figure 3 – 1.

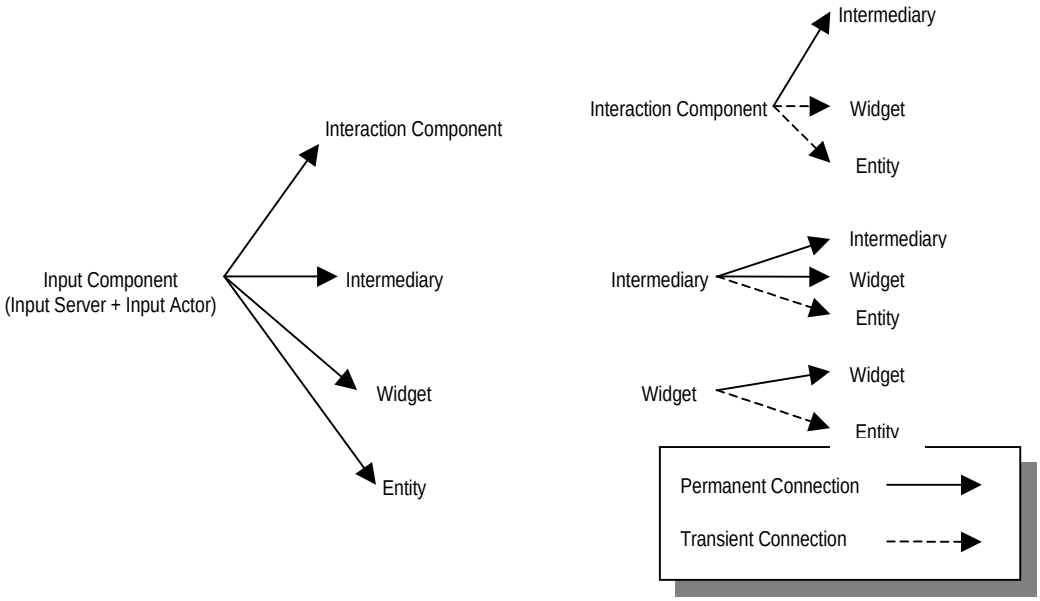


Figure 3 – 1 – The Recommended Data Flow Hierarchy for the Components of the Interaction System.

3.2.1. System Component

The system component will be different for each particular application and the only thing that we describe about it is how it must interact with the other components. The purpose of the system component is to implement the semantics that are unique to the particular application it supports. The system component is often implemented as the overall application, encapsulating the other components and bringing them all together to form some useful whole. The system component is also responsible for producing the output that the user sees.

3.2.2. Input Component

The input component is responsible for getting values from all the different input devices and passing them on to the relevant *interaction component*. This component holds the implementations of all the device level drivers required for each particular input device. These drivers are written to conform to a certain standard, which allows them to plug into the *input server*.

The input component is implemented as a pair of processes (*input server* and *input actor*) that communicate with each other via a network connection (Figure 3-1). The *input server* usually runs on a dedicated machine and communicates with a number of *input actors* (one for each particular device present on the server machine). These in turn communicate with the application they service. This form of implementation is necessary, since there is a wide range of different input devices available and they are not all able to run on a common architecture. The network layer separation provides the basic flexibility to be able to use all these different devices in a single application.

3.2.3. Interaction Component

The interaction component receives data from the input component and must interpret this data, passing on any relevant parts to the other components. Each application may use several different interaction components (e.g. one to implement a virtual hand and another to implement a head-tracked viewpoint display). Each interaction component may, in turn, use several different input devices (e.g. the virtual hand uses a Polhemus magnetic tracker [Polhemus, 96] and a data glove/stick). The CoRgi interaction system currently uses several different interaction components, each implementing some form of recognised immersive VR interaction technique [Bowman, 97; Mine, 95; Mine, 97]. For example, the hand actors use a simple form of gesture recognition to determine what action the user wishes to perform. By checking which of the fingers are *closed* and which are *open* on the user's hand, different gestures (like a point or a fist) can be recognised.

3.2.4. Intermediaries

Intermediaries make up the data flow part of the interaction system implementation. An intermediary corresponds to a node on a data flow diagram that performs some processing. Since objects are only willing to process types of data that correspond to some attribute they possess, intermediaries are required to convert data from one form to another. For example, a *Menu* generates integer data. This data does not correspond to any of the attributes of a *Entity* object (for example), but using an intermediary, the integer data may be transformed into 3D positional data, which can then be passed on to the *Entity* object. Thus, intermediaries receive data from some source (or multiple sources), process the data in some way (usually producing some other data type) and pass it on to its destination. The sources of data in the system are usually the interaction components, *widgets* or input actors. The sinks (or destinations for the data) are usually *entities* but, as in the Table Tennis example, the data may also be passed on to interaction components.

3.2.5. Entities

The coupling between the interaction component and the objects in the world is based on the idea that each object should know how to react to a finite set of commands issued by the interaction component. Such objects in the system are called *entities*. Each entity knows details about itself, so these details do not have to be stored centrally (in the system for example) but can be distributed amongst the entities themselves. Thus, the system does not need to know how to deal with each particular type of object - rather it has a finite set of commands that it is able to issue to any entity, and the entity itself must decide what action to take. The entity also provides feedback to the system as to whether the command was executed or not. This is useful in the case where we have several possible entities that could receive a certain command. If the first on the list does not execute the command, the system may send the command on the second, and so on through the hierarchy.

The choice of what commands the entities must be able to understand is a difficult one. Rather than try to create a complete list of commands that satisfy all possible VR applications, the system implements a group of commands which satisfy the basic needs of a large number of VR applications. Additionally, the entities and interaction components are constructed in such a way as to allow extra commands to be added easily. The current set of commands is:

- **Grab:** Select an object within the 'reach' of the user.
- **Drop:** Unselect a 'grabbed' object.
- **Point:** Select an object outside of the 'reach' of the user.
- **UnPoint:** Unselect a 'pointed' object
- **Press:** Activate an object.

The method by which the application decides what command to send to what entity is totally dependent on the programmer and usually implemented as part of the interaction component. The various interaction components, already implemented in the system, use methods such as collision detection and ray-casting to identify the entity, and simple gesture recognition to decide what command to send.

In addition to the standard set of commands that an entity is able to respond to, each entity has a unique set of attributes. These attributes include information like size, shape, position, etc. The attributes are openly available to the system, and can be changed to reflect changes in the application. Entity attributes are usually changed via intermediaries, activated by widgets through a data flow network. As discussed earlier, each attribute usually has an associated `Set` method that can be used by the application to include the entity in a data flow network.

3.2.6. Widgets

Widgets in the CoRgi system are simply specialised forms of the basic entity. They take the form of controls, which users are able to manipulate in order to send commands to the application. Unlike standard entities (which simply respond to commands received from an interaction component), widgets generate data depending on the commands they receive from the various interaction components (e.g. a button widget generates a unique integer when it receives a *press* command) as well as storing data about their current state. This data is then sent along a data flow network (usually containing at least one intermediary) to its destination.

3.2.7. System Summary

Component Name	Purpose	Example	Purpose of Example
Input Component (Input Device/Input Actor)	Connect the user input device into the application in a generic way.	`VRPolhemusInputDevice`	Provide position and orientation data read from a Polhemus InsideTrak magnetic tracker.
Interaction Component	Act as the extension of the user in the virtual environment.	`VRHandInteractionActor`	Provide the user with a *virtual hand* with which to interact inside the application.
Intermediaries	Perform functions to transform the data in the data flow network.	`VRIntScaleIntermediary`	Transform integer data into scale data using some pre defined method.
Entities	Container class for the objects in the system allows easy integration with the data flow system.	`VRGenericEntity`	Enable to object to be interacted with using a interaction component i.e. selected, manipulated, etc.
Widget	Container class for implementing user control objects in the system.	`VRButtonWidget`	Provide a button control with which the user can perform some action in the system.

3.3. The CoRgi Interaction Model

The CoRgi data flow model is based on data types defined by the system. This data is passed between the different objects that make up the system, operating by changing the values of the attributes of these objects. Each different category of object in the system has several different attributes, depending on its purpose. An object can only be included as part of a data flow network, if it supports an attribute of a given data type. Additionally, the setting of attributes in objects need not be a passive operation i.e. having a given attribute set may cause an object to propagate some data through another data flow network of which it is the root, or to execute some code.

Since, in a data flow system, only the data itself (not its source or destination) are important, the system does not need to concern itself with the source or destination of the data. If two objects are linked in a data flow network, then the system may presume that both are able to handle whatever type of data the network carries. This idea is enforced through the use of attribute setting methods. If an object is to be included in a particular type of data flow network, then it will contain a method (linked to the type of data that the particular network propagates) that is used to propagate the data. Thus, at compile time, the compiler type-checking makes sure that no object can be linked in a data flow network for which it has no handler method. It is also possible to build up the data flow networks during run time. Here again, the compiler type-checking ensures that no object can be connected into a network for which it has no handler method.

We call this idea of commonly named handler methods the *listener* model. In this model, objects contain methods based on the types of data they are able to handle. The type of data handled is, in turn, linked to the attributes of the object type. For example, the `VRWidget` object type (described in more

detail later) has a position attribute, which stores the current position of the object in the system in the form of a `Point3D` (a data type defined in the base CoRgi system). Related to this attribute, the `VRWidget` contains a method named `SetPoint3D`, which takes a single `Point3D` data value as its parameter. The invocation of this method on a particular instance of a `VRWidget` object causes that instance to change its position as specified in the parameter passed to the `SetPoint3D` method.

Certain types of objects (called propagators) pass data on to other objects when certain methods are invoked to build up the data flow network. These propagators have attributes (called *listeners*) which consist of pointers to objects of a specific type, to which the propagator can be linked in a data flow network. Each different type of object a propagator wishes to be linked to requires a separate listener attribute of that type. These listener attributes are assigned to particular instances of an object by the `LinkTo method`. Each propagator has a `LinkTo` method (defined with the appropriate parameters) for each specific listener attribute. Thus, a propagator may have multiple listeners, each of a different type. For example, the `VRButtonWidget` object type has two listeners, one of type `VRMenuWidget` and one of type `VREntity`. Thus, the `VRButtonWidget` can act as the root for two separate data flow networks. The activation of the data flow network and the data it carries can come from any source within the application.

The usage of the data listener idea greatly reduces overhead on the system by removing the need for any sort of centralised messaging/event system. The event model works well when many recipients need to receive a given event. The listener system works well where there is a specific (usually single) destination for a given message. We decided that most of our events would be directed toward specific objects and those that were not could be simulated by the listener system. Thus we chose the lower overhead listener system over the central event based system used by most current (2D) user interface systems.

3.4. Integrating the Interaction System With a VR Toolkit

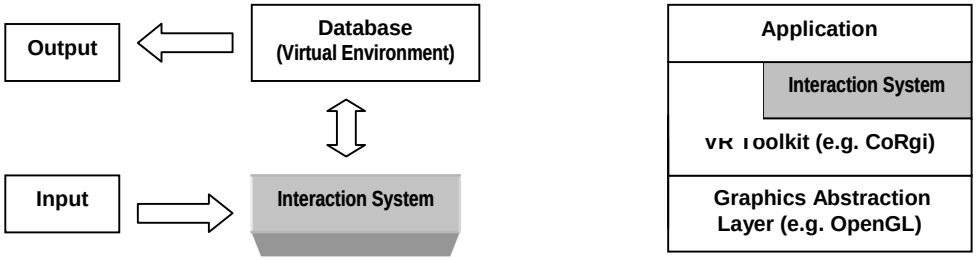


Figure 3 – 2 – The Interaction System Integration with a VR Toolkit.

The interaction system is designed to integrate with an existing VR toolkit. Figure 3 –2 details the relationship between the interaction system, the VR toolkit and the hardware. The interaction system resides between the user and the VR application (built on the VR toolkit). The interaction system must

be able to get data from the input devices and use this data to make changes in the virtual environment (usually implemented as a database of objects and their properties). As long as a VR toolkit is able to provide these services, the interaction system can be altered to co-exist with it.

3.5. Justification of the Framework

In order to prove the effectiveness of our system as a VR interaction toolkit we describe a set of case studies of how the system has been used to create interactive VR applications (Chapter 4). In addition, we relate the toolkit back to the guidelines described in the previous chapter in order to ascertain how the system fits into the abstract interaction taxonomies. Each of these taxonomies is considered separately, in the order in which they were presented in the previous chapter.

3.5.1. Travel

The position and orientation of the user's viewpoint in the system is controlled via a *camera* object. The camera determines what the view of the world should be, based on its position and orientation attributes. The *Indicate Orientation* branch of the travel taxonomy is implemented by linking an input actor to the orientation attribute of the camera object and having the data from the input device (e.g. a Polhemus tracker, which is attached to the user's head) constantly update the orientation for the display. The input actor may also provide positional information, which is used to make small movements in the environment corresponding to movements made by the user's head.

The *Indicate Position* branch of the travel taxonomy is implemented by setting a *vector movement* attribute in the camera object. The vector movement method stores a 3D-vector value, which is used to update the position of the camera object at constant intervals. The camera object has a thread method, which is called, at regular intervals by the system component, thus allowing the camera to update its position, giving the feeling of movement.

All three of the sub-branches of the *Specify Position* branch can be implemented by setting the vector movement attribute as follows:

3.5.1.1. Continuous Specification

Continuous specification can be implemented by continuously updating the vector movement attribute. For example, *gaze-directed travel* could be implemented by linking the input actor for the camera object (which gives a value for where the user is looking) to an intermediary, which takes in a Quaternion and produces a 3D-vector value based on it. The intermediary is then linked back to the movement vector attribute of the camera object. Thus, whenever users change the orientation of their heads, this new orientation is translated into a 3D-vector value and used to update the movement of the camera object. An example data flow diagram for this is shown in Figure 3 – 3.

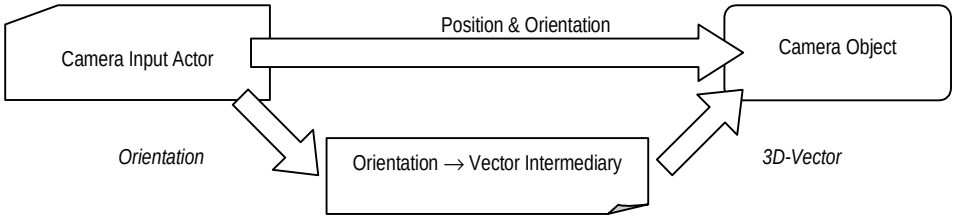


Figure 3 – 3 – Simple data flow diagram for gaze directed movement.

3.5.1.2. Discrete Target Specification

These sub-branches can be implemented by setting the relevant values into the movement vector attribute of the camera object. The position attribute of the camera object could also be used simply to place the camera in a new position, but as explained in the previous chapter, it is better to change the position of the user by small steps to avoid disorientation. Figure 3 – 4 illustrates a system whereby the user chooses a destination using a virtual menu; this destination is then translated into a corresponding 3D-vector value and used to move the camera object.

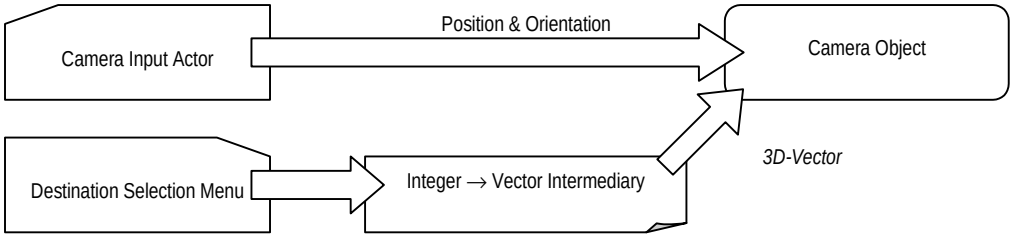


Figure 3 – 4 – Simple data flow diagram for a discrete destination selection system.

3.5.1.3. Controlling Speed, Acceleration, etc.

The *Specify Velocity* and *Specify Acceleration* branches can also be implemented using the vector movement attribute. The length of the vector sets how far the user is moved for each discrete step, and this value can be changed over time to simulate acceleration. The *Stop Moving* and *Start Moving* branches are implemented using a movement attribute, which can be set or cleared and is checked by the thread routine before updating the position.

3.5.2. Selection

The selection taxonomy is a product of the interaction between the interaction components and the entities in the system. *Feedback* can be supplied to the user through either graphical or audio means in the CoRgi system. The method used the most in the current system involves displaying a semi-transparent bounding box around the selected object, making it obvious to a user what has been

selected. The sub-branches of the *Indication of Object* branch vary widely in their implementation and each will be considered separately.

3.5.2.1. Object Touching

Collisions detection in a VR system is a detailed field of research in itself. We simply use the collision detection routines implemented as part of the CoRgi VR system. Checking for collisions can be done in one of two ways. The system component can do a global check of all the objects in the system, at regular intervals. When two objects are identified as colliding, the system component informs each of them and they are left to react appropriately. The other method involves a particular object (usually an interaction component) checking for collisions between itself and the other objects in the system. When an object is identified as colliding with the interaction component, the appropriate command (e.g. `Grab`) is dispatched for the object to execute.

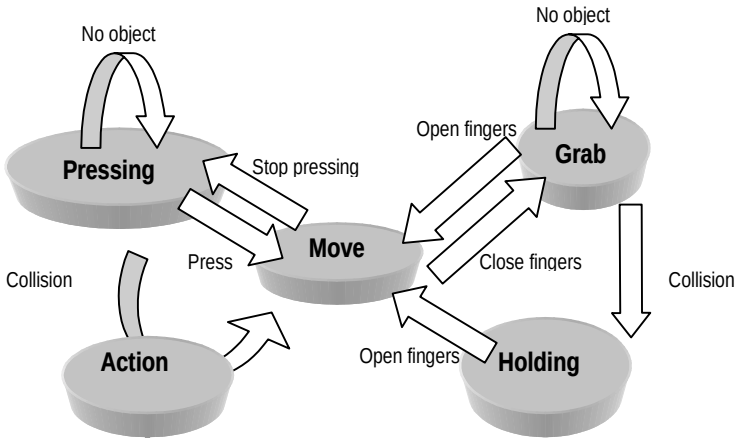


Figure 3 – 5 – Automated Transition Network (ATN) detailing the operation of the close hand interaction component.

Figure 3 –5 details the operation of an object touching selection system as an automated transition network (ATN). The *close hand interaction component* begins execution in the **Move** phase. In the **Move** phase, the position and orientation of the representation of the hand is updated from the corresponding input actor. The values being read from the stick input device (measuring the *gesture* of the user's hand i.e. what fingers are open, and what fingers are closed) are used to change the system into one of either the **Pressing** or **Grab** phases (depending on the gesture of the user's hand). Once the system is in **Grab** mode, it remains there until the user's hand gesture changes (back to **Move** phase) or there is a collision with an object in the system (**Holding** phase). In the **Holding** phase, the object is manipulated with the hand, until the user's hand gesture changes and the system is placed back in the original **Move** phase. The **Pressing** phase is similar to the **Grab** phase. Once an object collision is detected in the **Pressing** phase, that object is sent a *press* command and the system automatically returns to the **Move** phase, with no further intervention from the user.

3.5.2.2. Pointing and Occlusion

Pointing implies a ray-casting technique with the ray oriented according to the orientation of the interaction component. Occlusion can be implemented using the same ray-casting technique, but with the ray oriented according to the orientation of the user's head i.e. the camera object. The ray-casting technique is implemented by taking each of the objects in the scene and checking whether there is any intersection between that object and the ray. This is implemented in the CoRgi system by first approximating the shape of the object to a sphere enclosing the object and then using geometry to check for an intersection between this sphere and the ray. This is done by computing:

$$H = S - I$$
$$\mu = R \bullet H$$
$$\lambda = H \bullet H$$

$$\eta = \mu^2 - \lambda + r^2$$
$$= \begin{cases} >= 0 & \Rightarrow yes \\ < 0 & \Rightarrow no \end{cases}$$

where ***S*** gives the 3D-co-ordinates for the origin of the object, ***I*** gives the co-ordinates of the origin of the ray, ***R*** is the 3D-vector describing the direction of the ray and ***r*** is the radius of the sphere approximating the shape of the object. Each of the objects in the scene is checked in turn, until an intersection is found or all the objects have been checked.

3.5.2.3. Indirect Selection

Indirect selection can be implemented simply by storing pointers to the objects of interest and presenting these to the user is some useful way. For example, all the objects in the system can be used to generate a menu, from which the user can select the required object.

The *Indication of Select* branch is implemented as part of the Interaction Component. For example, the interaction component may do simple gesture recognition on the user's hand, deciding what command

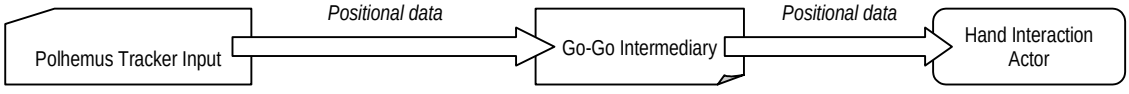


Figure 3 – 6 – Simple data flow diagram for Go-Go arm extension technique

to issue to the selected object based on this.

The arm extension techniques used for selecting distant objects can be implemented using an intermediary between the input actor and the interaction component. For example, Figure 3 – 6 shown a simplified data flow diagram for the implementation of the go-go extension technique. Instead of the

Polhemus tracker input actor connecting directly to the hand interaction actor, it passes through a Go-go intermediary, which implements the function detailed in Figure 2 – 6. Thus, if the positional data coming from the Polhemus tracker input actor is closer to the user than some threshold **D** it is simply passed on and updates the position of the interaction component. If the data is greater than **D** then the data is updated (using the non-linear function **F**) and this new value used to update the position of the interaction component.

3.5.3. Release

The *Release* taxonomy is also implemented as part of the interaction component. The same method used for the *Indication of Select* can be employed for *Indication of Drop*, but using a different activator (e.g. different gestures for select and release). The *Object Final Location* is also implemented as part of the interaction component. The final position of the object is controlled by the interaction component using the pointer to the object received from the selection phase. When the object is released, it remains in the position specified by the interaction component just before release.

3.5.4. Manipulation

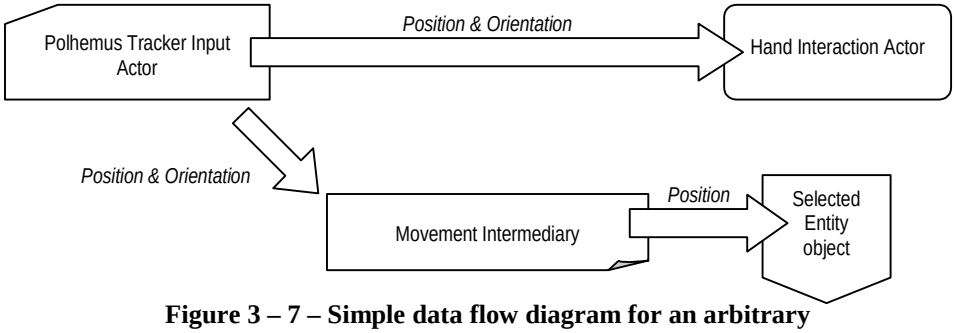


Figure 3 – 7 – Simple data flow diagram for an arbitrary manipulation technique.

The *Manipulation* taxonomy lists several different groups of methods that allow the user to map some movement into the environment and use this to manipulate the position/orientation of an object. The interaction component is usually responsible for this mapping. All of these mappings require some way of recording the movements made by the user and applying these to an object in the system. The movements made by the user are accessible thought the various input actors, and the object to be manipulated is identified in the selection phase. Thus, any manipulation technique of this kind can be implemented using an intermediary to do the mapping between the user's movements and the corresponding movements of the object. For example, consider an arbitrary manipulation technique whereby the position on the interaction component was mapped directly to the selected object, but the orientation of the object was not changed. Figure 3 – 7 shows the simple data flow diagram for this system. The data from the Polhemus tracker input actor is sent to the interaction component (as usual) and it is also sent to the movement intermediary. The movement intermediary then passes on the positional data from the Polhemus tracker input actor to the object, updating its position.

3.5.5. System Commands

All of the branches of the *System Commands* taxonomy are supported in the CoRgi system. The *Direct Input* sub-tree is supported through the use of the IBM ViaVoice speech recognition package and a speech input actor object. Gesture recognition is also being developed [Winnemoeller, H, 99]. The menu and button functions supplied in the widget objects provide the *indirect* mechanisms.

3.6. Comparison with Other VR Interaction Frameworks

As mentioned earlier in this chapter, the CoRgi Interaction system and its associated generic framework are built on the assumptions of the MR Toolkit [Shaw, 92; Shaw, 93] and the Cognitive Coprocessor Architecture [Robertson, 89]. All three systems operate on the principle that the interaction system may be totally separated from the remainder of the VR toolkit. Communication between the interaction system and the toolkit takes place through rigidly defined interfaces only. This assumption is well in line with modern object oriented programming practices and allows the system to be developed and extended in a generic way i.e. not dependent on a single VR toolkit.

The CoRgi Interaction system uses a data flow model similar to that defined in the VRML ’97 standard [Carey, 97] and that used in the VEDA architecture [Steed, 96; Steed, 97]. The data flow system used is simpler than that used in the VB2 system actions [Gobbetti, 93; Gobbetti, 94] where the network uses a sophisticated constraints-based solver to continuously evaluate the connection in the network. In the CoRgi system, the making and breaking of links in the data flow network is left completely up to the application developer. The constraint solving system was identified as the major bottleneck in the VB2 system and thus, we opted for a simpler, more efficient data flow system.

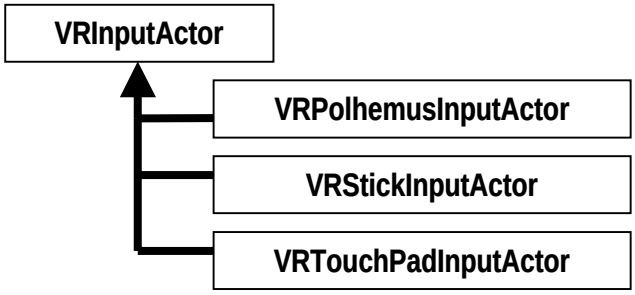
Jacob’s system [Jacob, 99] of combining the data flow model with an event-based model was also considered. The CoRgi interaction system does not implement specific support for any particular event based system, but this does not preclude the developer from using event-based models. Jacob illustrates how a data flow system can be used to simulate an event-based system. The CoRgi interaction system does this by setting up a standard data flow network between all the parties interested in a particular event, and propagating the event as data through this network. The interactions between the interaction component and the entities/widgets can also be thought of as an event-based system since only single events (e.g. grab, drop, etc.) are dispatched to the recipient. In addition, most VR toolkits, while not providing specific interaction layer support, do provide a generic event system for communicating between the objects in the system. This lower level event handling system can easily be integrated into the interaction layer.

3.7. Implementation Specifics

3.7.1. System Component

The standard system component will create a `VREnvironment`, a `VRSink` and any input, interaction, entity and widget components that the application needs. Once all the pieces are in place, the system component enters into a processing loop which continually executes the `RunComponents` method which activates the *run* loop for the system to update the display, get data from input devices and so on.

3.7.2. Input Component

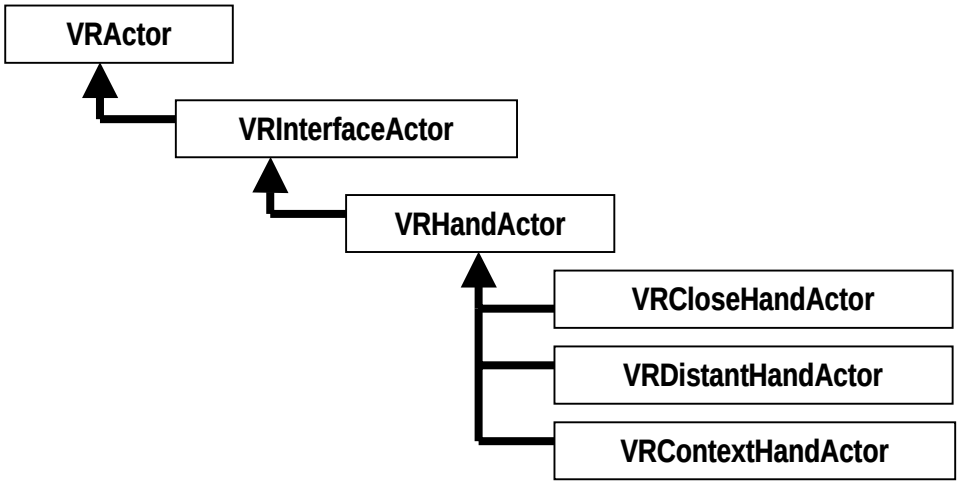


The input actors communicate with the application using the data flow and listener methods. Each input actor holds a listener attribute of the `VRActor` type. `VRActor` is the base class for any object that is controlled directly by the user (i.e. needs to communicate with an input device). Depending on what input device the input actor is connected to, it calls a method in the listener, passing it data received from the input server. For example, the `VRPolhemusInputActor` connects (via the input server) to a Polhemus FastTrak magnetic tracker [Polhemus, 96]. The magnetic tracker provides values related to its position (a `Point3D`) and its orientation (a `Quaternion`). These two values are contained in a composite class called `VRInputCoordinates`. Thus, when the `VRPolhemusInputActor` receives new data from the input server, it calls the `SetVRInputCoordinates` method in the instance of the `VRActor` class pointed to by the listener attribute, passing the new position and orientation as parameters.

All input actors inherit from the `VRInputActor` base class. They provide individual functionality by overriding the `HandleData` method and implementing semantics for their specific types of data. At the input actor level, we are no longer concerned with hardware specifics, rather we are already focusing on data provided. Thus, two separate pieces of hardware that provide the same data (e.g. two different makes of magnetic tracker) would use the same input actor. The input server on the other hand is concerned with hardware specifics. The input server creates instances of specific input devices, one for each device connected to the machine on which it is running. All of these input devices inherit from the base class `VRInputDevice`, and override the method `GetData` to implement the semantics of a particular input device. The input server then calls the `GetData` methods in all the

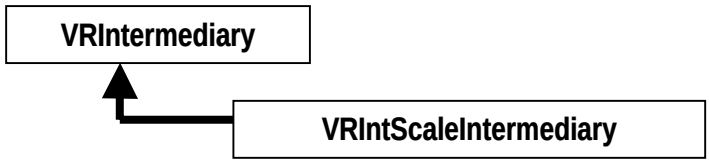
`VRInputDevice` instances it has. It is responsible for abstracting the device from a hardware level to a data level and providing the input actor with this data. Further details of the `VRInputActor` classes are shown in Appendix A, Figure 1.

3.7.3. Interaction Component



The base class for interaction components is the `VRInterfaceActor` (which inherits from `VRActor`). The methods for connecting to input actors (e.g. `SetVRInputCoordinates`) can be used as inherited from `VRActor` with no changes required. Each interaction actor is allowed to implement its specific semantics in its own way. For example, the `VRHandActor` implements its particular semantics in the `CheckGesture` method. This method is defined as `virtual` and is overridden in the child classes of the `VRHandActor` (e.g. `VRCloseHandActor` and `VRDistantHandActor`) to provide different types of functionality. For example, the `VRCloseHandActor` uses collision detection to detect what object the user wishes to interact with, while the `VRDistantHandActor` uses ray-casting techniques. Further details of the `VRInterfaceActor` classes are shown in Appendix A, Figure 2.

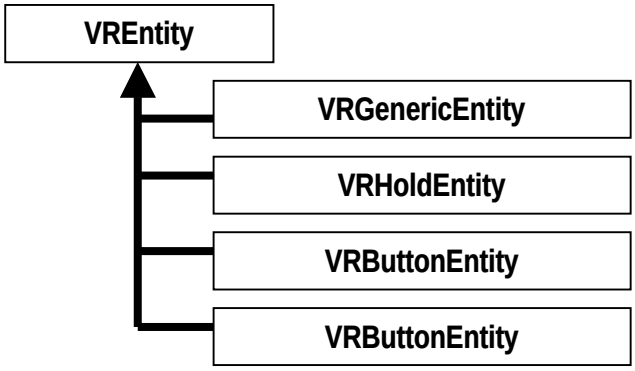
3.7.4. Intermediaries



All intermediaries inherit from the base class `VRIntermediary`. The base class defines two methods, `Register` and `Link` and any extra input methods required to handle input data types (e.g. `SetInt`). There is usually only one intermediary type for each pair or input-output data types. For example, the `VRIntToScaleIntermediary` takes an integer data type as input, producing a `Scale3D` as output. The conversion from integer to scale can be done in a number of different ways, as is the case with any generic input-output pair. Instead of having a separate intermediary type for each different conversion method, intermediaries implement many different conversion methods within

the same object. The choice of what method to use in a particular instance is set using the `Register` method. Each intermediary contains a list (in string form) of all the different conversion methods it implements. When setting up the data flow network containing the intermediary, the application first calls the `Link` method with a source and a destination. The source is where the intermediary gets its input data and the destination is where it sends its resultant output. The intermediary then becomes a listener of the source. Once the `Link` method has been invoked, the `Register` method needs to be invoked to set a conversion function for the link. The `Register` method takes in a pointer to the source object (must be the same pointer used to set up the link) and a string. The string identifies the conversion method to be used. Once the intermediary identifies the particular conversion method to use, it associates a function pointer to the conversion method with the source. When the source invokes an input method in the intermediary, the method referred to by this function pointer is used to make the conversion and pass out the output data. Further details of the `VRIntermediary` classes are shown in Appendix A, Figure 3.

3.7.5. Entities

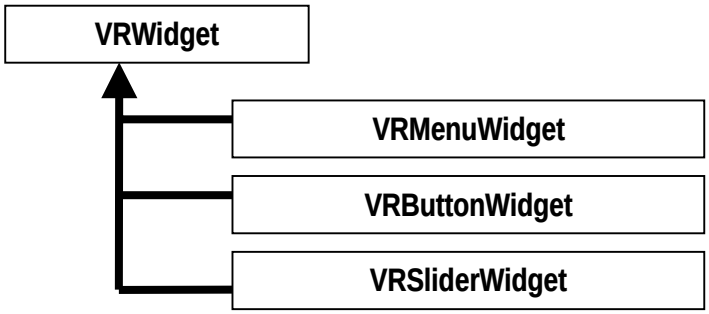


Entities are based on the `VREntity` base class. The base class contains methods to handle all the commands that may be dispatched by an interaction component. In this base class, these methods are implemented as dummy methods, simply returning 0 to the calling component to indicate that this specific entity does not implement that action. In order to do something useful, this base class must be overridden and these methods implemented. Once the method has performed some useful action, it should return 1 as an indication that the command was received an executed. For example, the `VRGenericEntity` class inherits from the `VREntity` class and implements the `Grab` and `Drop` methods. Thus an interaction actor is able to *grab* and *drop* `VRGenericEntity` objects. The `VREntity` base class also implements the attribute setting (input) methods. These can be overridden if necessary, but most objects will simply use them as they are.

Additionally, the `VREntity` base class is responsible for maintaining a list of all active entities in the system. Pointers to all the entities in the system are stored, indexed by their `objectID` (a unique ID assigned by the system) in a linked list, which forms part of the `VREntity` base class. For example, when using collision detection to find the object a user is interested in, the system returns to the interaction actor, `objectIDs` of all the objects that are within a certain distance of the interaction

actor. These are cross-referenced with the list of entities, giving a list of entities that are colliding with the interaction actor. In the distributed system (explained in a later chapter) this list of entities is moved into a totally separate object and stored in a central server. Further details of the `VREntity` classes are shown in Appendix A, Figure 4.

3.7.6. Widgets



The design and implementation of widgets for a VR system is a field of study in itself. For the purposes of the CoRgi interaction system, we implemented a set of simple widgets. To show the flexibility of the system, we also implemented a more complicated slider widget.

Currently, the system has two main types of widget, the `VRMenuWidget` and the `VRButtonWidget`, both inheriting from the base `VRWidget` class. The `VRWidget` class acts as a wrapper class to the standard `VREntity` class, encapsulating the extra code required for a widget. For example, the `VRButtonWidget` class creates an instance of the `VRButtonEntity` class in its constructor. The `VRButtonWidget` class is a propagator class, meaning that it passes data along into the data flow network. It has a `LinkTo` method and listener attribute, as described earlier in this chapter. The `VRButtonEntity` provides a visual representation of the widget in the system and allows the interaction component to interact with the widget as it would interact with any other entity in the system. The `VRButtonEntity` class includes a pointer to an instance of the parent (`VRButtonWidget`) class. It overrides the `Press` method of the standard `VREntity` class causing it to pass a unique *action ID* (set in the constructor) up to the parent widget class, through the parent's `SetInt` method. The `SetInt` method of the `VRButtonWidget` simply passes the value received from the button to the next node in the data flow network, as specified by its listener attributes. Usually, a group of buttons (each with a unique ID) is contained within a single `VRMenuWidget` object. The `VRMenuWidget` also has an associated `VREntity` object that provides a visual representation for the widget as well as enabling the user to interact with it (e.g. by overriding the `Grab` and `Drop` methods). The action ID of the last button that was pressed (passed through the data flow network) is stored in the `VRMenuWidget` as well as being passed on to any further nodes in the data flow network. The `VRMenuWidget` object also implements a `GetInt` method that can be used by the main program to check if any of the buttons on the menu has been activated and if so, which one. Further details of the `VRWidget` classes are shown in Appendix A, Figure 5.

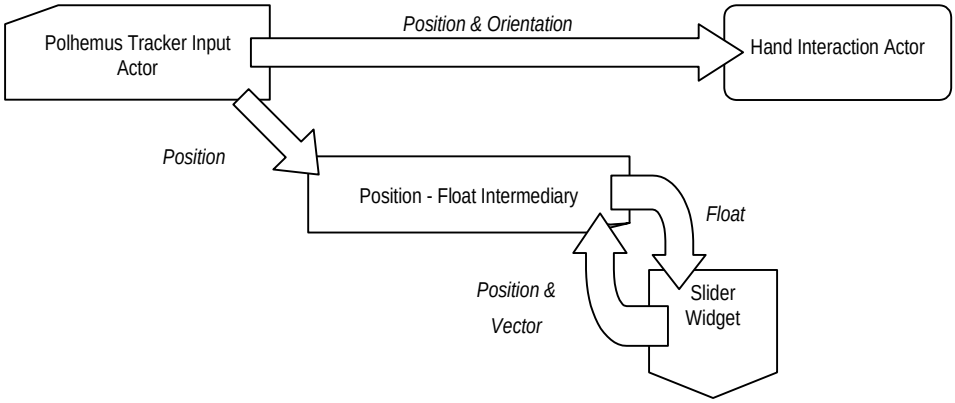


Figure 3 – 8 – Simple data flow diagram for the slider widget.

The slider widget data flow diagram is shown in Figure 3 – 8. The slider widget operates by providing a rail, along which a marker can be moved. The distance of the marker from the bottom of the rail represents some `float` value. When the hand interaction actor *grabs* the widget, a connection is activated between the Polhemus input actor (for the hand) and a Position – Float intermediary (as shown). The position of the hand, the position of the slider widget and a vector describing the orientation of the slider widget and its length are all used to calculate how far along the rail the marker is moved (i.e. what value the slider represents). This value is then passed back to the slider (with the `SetFloat` method) and can be accessed through the `GetFloat` method, or through connecting the slider to a listener.

3.8. Example Application

As an example of the use of the interaction system to produce an immersive, interactive application, we have chosen to develop a simple Table Tennis game. The user controls a bat (using a Polhemus tracker) which must be used to hit a ball. The user also interacts with the system using a second Polhemus tracker and a simple button input device, which operate a *virtual hand* inside the system. The hand can be used to pick up the ball and operate the various system controls. A simplified data flow model of the application is shown in Figure 3 – 9.

3.8.1. System Component

In the Table Tennis game example, the system component is also responsible for rendering the scene, getting the data from the various input devices and passing this on to the input actors. The system component is also used to do global collision detection on the objects in the system, informing each when a collision occurs. The gravity in the system was also implemented as part of the system component, though this could have been distributed amongst the components.

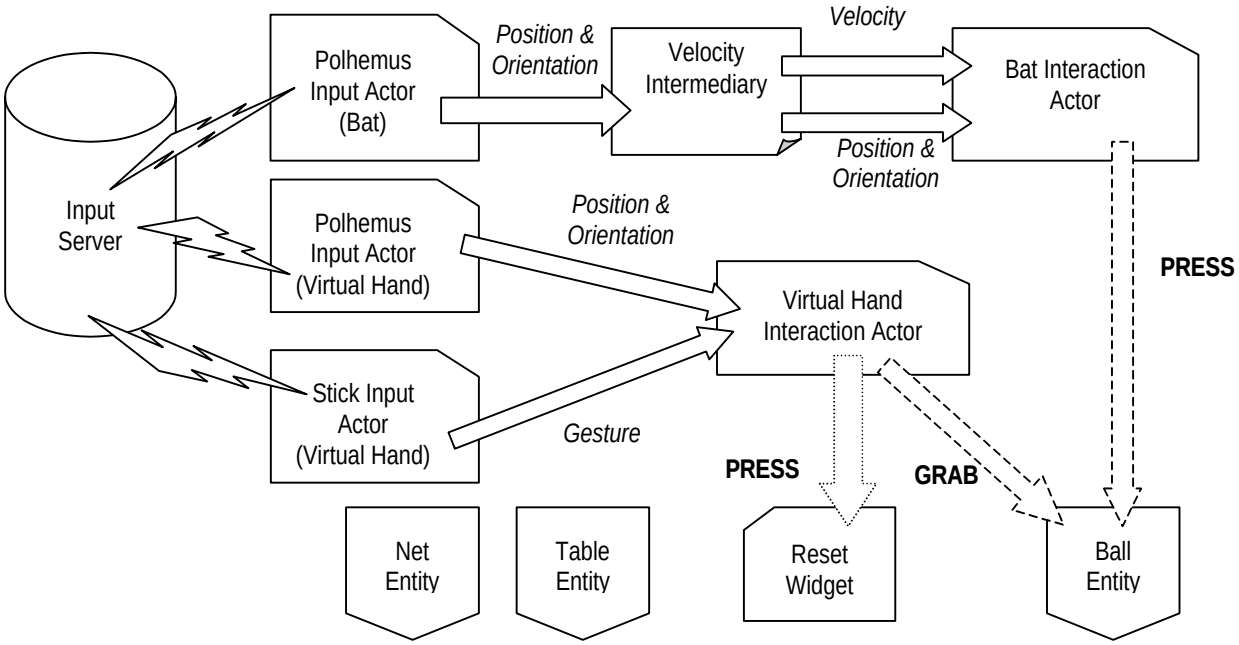


Figure 3 – 9 – Data flow diagram for example table tennis application

3.8.2. Input Component

The application requires a Polhemus tracker with which the user controls the position of the bat in the system. This is plugged into the input server machine and communicates with a `VRPolhemusInputActor` in the application. The input actor supplies data to the application relating to the position and orientation of the user's hand and thus the bat, which is used to update the display. There is an additional `VRPolhemusInputActor` for the hand interaction component and another for the head mounted display. The Polhemus tracker on the head mounted display is linked to a `VRHMDActor` which updates the display to coincide with the movements of the user's head. The hand also requires a `VRStickInputActor` to measure the gesture of the user's hand.

3.8.3. Interaction Component

The Table Tennis application has two different interaction components. One of these is the bat, which the user uses to interact with the ball during the game. When the bat collides with an object in the scene the global collision detection system works out the new trajectory of the object and sets it. The speed and position of the object are attributes of the object's `VREntity` and are updated using the `Set` methods. The other interaction component is a virtual hand (`VRCloseHandActor`), which the user can use to retrieve balls that have fallen off the table and to give various commands to the system. The `VRCloseHandActor` picks up the ball by issuing a `Grab` command to the ball entity when a collision occurs between the hand and the ball and the user is performing the correct grabbing gesture. The ball is released by issuing a `Drop` command to the ball entity when the user performs the *dropping* gesture. Commands are relayed to the system through the use of control widgets. These widgets are

activated by `Press` commands from the `VRCloseHandActor`. These `Press` commands are activated when the hand collides with the control widget and the user is performing the correct *pressing* gesture.

3.8.4. Intermediaries

The Table Tennis game example uses an intermediary to calculate the velocity at which the bat is moving. The `VRPolhemusInputActor` receives data about the position of the tracker from the input server. This constantly updated position information is then used, in conjunction with timing information, to calculate the velocity at which the bat is moving (`VRPositionVelocityIntermediary`). This velocity is then passed on to the bat interaction component (along with the standard position and orientation information) and used for calculating new trajectories when interacting with the balls in the system (Figure 3 – 9).

3.8.5. Entities

There are several entities in the system - the ball, the table, the net, the floor and the control widget representations. The attributes of the ball, include its shape, colour, elasticity (how much it reacts to collisions with other objects e.g. the bat or table), position and velocity. The ball knows how to respond to `Grab` and `Drop` commands from the interaction actors. A `Grab` command causes the ball to attach itself to the interaction component issuing the command, thereby allowing the user to manipulate its position. The `Drop` command causes the reversal of the attachment formed from the `Grab` command. Interactions between the bat and the ball, where the ball changes direction/velocity based on the movement of the bat, are handled by the global collision detection system. The table, net and floor entities do not respond to any commands from the interaction actors (i.e. the user is not able to interact with these entities). They are used to check for collisions with the ball. Collisions between the ball and each of these entities are handled by the global collision detection system. The control widget entities react to `Press` commands from the interaction actors. These `Press` commands activate the operation of that particular control. For example, *pressing* the reset widget causes the state of the game to be set to some initial value.

3.8.6. Widgets

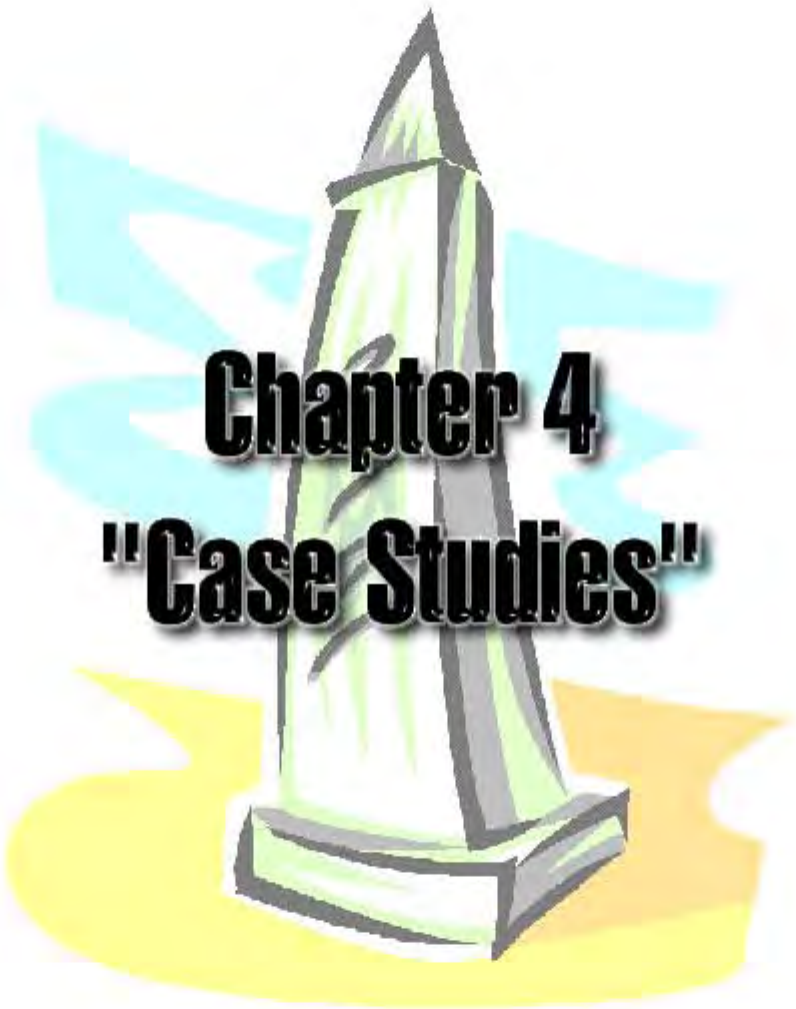
The Table Tennis example contains a single, simple button widget, which responds to a `Press` command by instructing the system component to reset that game and start again. In this case, the widget is simply polled from the system component and the system resets when the widget is activated.

3.9. Summary

Immersive VR interfaces require a different approach from that used for traditional desktop interfaces. The data flow approach allows the interface to consist of multiple data pathways all operating in parallel. The CoRgi interaction system implements this data flow idea using *listener* attributes. The listener attribute is simply a pointer to an object in the system that is *informed* when a certain operation occurs. Thus, the data flow network in the CoRgi interaction system comprises objects, linked together by their listener attributes.

The CoRgi interaction system is abstractly separated from the remainder of the CoRgi VR system according to the framework detailed in Figure 3 – 2. This separation allows the different portions of the system to be developed independently, provided the interfaces between these parts are specified and those specifications adhered to. We have detailed the specifications for these interfaces and given details on the current implementation of the system under the CoRgi VR toolkit.

We have provided an example to the operation of the system, and justification of its workings based on the interaction taxonomies detailed in Chapter 2. We have also compared the system with other similar toolkits.



Chapter 4
"Case Studies"

4.1. Introduction

Thus far, our interaction framework has been justified in terms of what have been identified as the important aspects of interaction in immersive VR. In order for the framework to be truly justified, we need to present examples of its use in the area it was designed for, namely the production of immersive, interactive VR applications. The system has been used by a number of post-graduate students in the Computer Science Department of Rhodes University. The applications they have developed are described, with the emphasis placed on their usage of the interaction framework. We also describe various smaller applications developed along with the framework itself, to illustrate the use of the various components. All of the screenshots shown in this chapter are reproduced, in colour, in Appendix C.

4.2. Sample Framework Applications

Several small applications were developed along with the interaction framework, to illustrate and test the various components.

4.2.1. Virtual Remote Control

The Virtual Remote Control (VRC) [Rorke, 99], is not a standalone application, but rather an interaction component/widget hybrid, which when used in conjunction with a specialised hardware device (a small touch-pad and Polhemus tracker) presents a unique interaction paradigm to the user.

> *"The VRC consists of a physical device (a small touch-pad tracked using a Polhemus InsideTrak magnetic tracker) which the user is able to hold and for which there is a representation in the virtual environment. The VRC is represented in the environment by a virtual menu. Users are able to make selections from the virtual menu by moving their thumb around the touch pad part of the VRC and 'tapping' on the required action. Additionally, the user is able to select an object for the action to be applied on, by 'pointing' the representation of the VRC at the object - as one would point a remote control at a Hi-fi or TV set."*
>
> [Rorke, 99]

There are two distinct functions for the VRC – that of making menu selections (a *widget* function) and that of selecting objects in the VE (an *interaction component* function). The VRC is implemented in the CoRgi Interaction system by the `VRRemoteControlActor` class. This class inherits from both the `VRInterfaceActor` class and the `VRWidget` class, making it both an interaction component and a widget. The VRC operates in a similar manner to the `VRMenu` object in that it behaves as a container class for a set of `VRButton` widgets, each of which is assigned a particular function. The user selects

what button to activate by moving a finger over the touch-pad to position a cursor over the required button. 'Tapping' a finger on the touch-pad then makes the selection.

In the case of the menu widget, a separate interaction component (usually a virtual hand) is responsible for activating the buttons by sending *Press* commands. The VRC is itself an interaction component, and activates its own buttons (with *Press* commands) in response to the user 'tapping' on certain

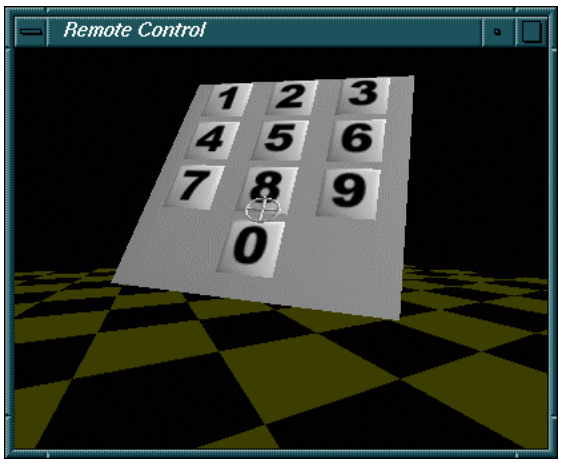


Figure 4 – 1 – The Virtual Keypad.

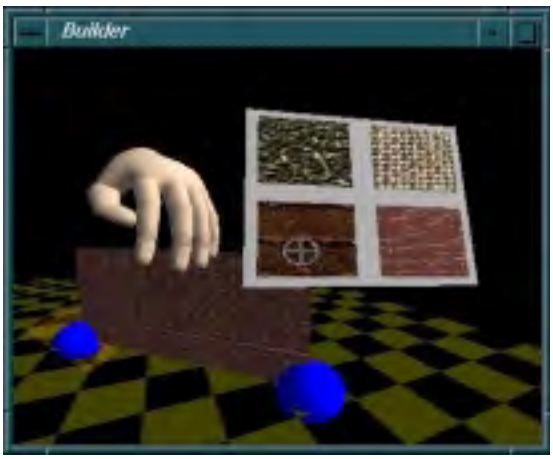


Figure 4 – 2 – The Texture Selector.

portions of the touch-pad. The selection part of the VRC implements the ray-casting technique described in Chapter 3. This is the same technique used by the `VRHandActor`.

Figures 4 – 1 and 4 – 2 show screenshots from two example applications that use the VRC. Figure 4 –1 shows the Virtual Keypad application which enables a user to input exact numerical data into an immersive VR application, using the VRC – a task which would otherwise be rather difficult since the user is unable to see a keyboard through a HMD. Figure 4 – 2 shows the Texture Selector application. The Texture Selector allows the user to interactively apply textures to entities in the system. The textures are previewed as buttons on the VRC, and applied by selecting the required object (pointing

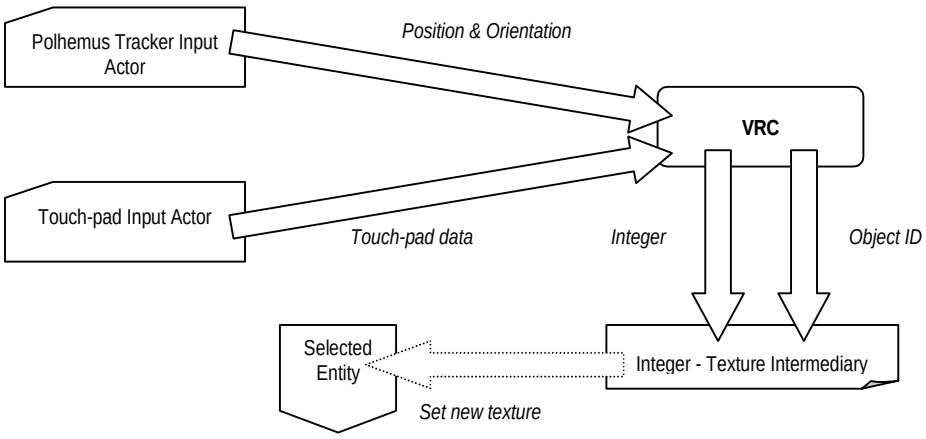


Figure 4 – 3 – Simple data flow diagram for the texture

the VRC at it) and selecting the texture by 'tapping' over the correct button on the VRC. Figure 4 – 3 shows the data-flow network for this component.

The button widgets on the VRC generate integer values when *pressed* these are sent to the 'integer to texture intermediary' which generate new textures, passing them on to the object. The 'integer to texture intermediary' also needs to know what object is currently selected in order to properly generate and apply the new texture. Also present in Figure 4 – 2 is a `VRHandActor` interaction component.

The VRC application demonstrates how multiple interaction components can exist in the same application, interacting with the system in the same way, but each offering a unique interaction paradigm to the user. Since the entities in the environment simply respond to commands, without regard as to what particular object issued the command, the addition of multiple interaction components becomes trivial. Different interaction components can be dynamically added to the system at runtime, without the need for any change to the interaction code. The implementation of the VRC did not require the implementation of any extra widgets in the system, but the integration of a widget and input component object into a single entity proved to be useful. This integration was accomplished without any need to change the design or implementation of the interaction framework.

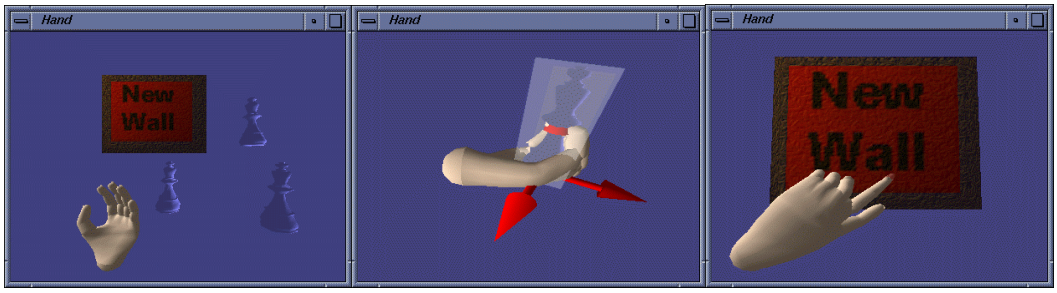


Figure 4 – 4 – Screenshots of the VRHandApp.

4.2.2. VRHandApp

The `VRHandApp` is a small application designed to test the interactions between the interaction components and the entities/widgets in the system. The application presents the user with an immersive virtual environment, which contains a number of entities. The user is presented with a virtual hand interaction component with which to interact with the entities in the system. The initial number, type, position and orientation of the objects in the system are read in from a configuration file when the application is executed. The system may also include a set of widgets, which the user can use to send commands to the system. The main purpose of this application is to test the workings of the different components of the interaction system, but it also serves as a good introduction to immersive, interactive VR systems for novice users requiring training.

The three screenshots in Figure 4 – 4 show the three of the main aspect of the VRHandApp application. The first shows the interaction component (the hand), a widget (a virtual menu with a single button) and entities (the chess pieces). The second screenshot shows the interaction component 'grabbing' one of the entities – notice the feedback to the user supplied in the form of a transparent bounding box

(selection feedback) and a set of axes (orientation feedback). The last screenshot shows the interaction component activating the 'New Wall' button on the virtual menu widget.

4.3. Virtual Reality Image Viewer

Mundell [Mundell, 99] researched the feasibility of creating a VR operating system (i.e. an application to control the operation of a computer from inside a VE).

> *"… explores the virtual interface from a design perspective. A simple virtual reality image viewer application is tested with a variety of users, producing a virtual interface design reference model, and a set of virtual interface design guidelines."*
>
> [Mundell, 99]

The image viewer is intended as a test case whereby the different interaction paradigms of VR could be tested on a variety of users in order to asses the usefulness of each. The purpose of the image viewer is to present the user with an intuitive way to store, select and display images stored on a computer, in a virtual environment. Two versions of the application were developed to test the two main paradigms of VR interaction – a *real world* version, which mimics objects from everyday life, and an *abstract version*, which moves away from real objects, toward more creative representations of objects. These two versions correspond to the *natural* and *magical* interaction paradigms (introduced in Chapter 2) respectively. The applications were both developed under the CoRgi system, using the interaction framework to implement the various VR interaction paradigms required.

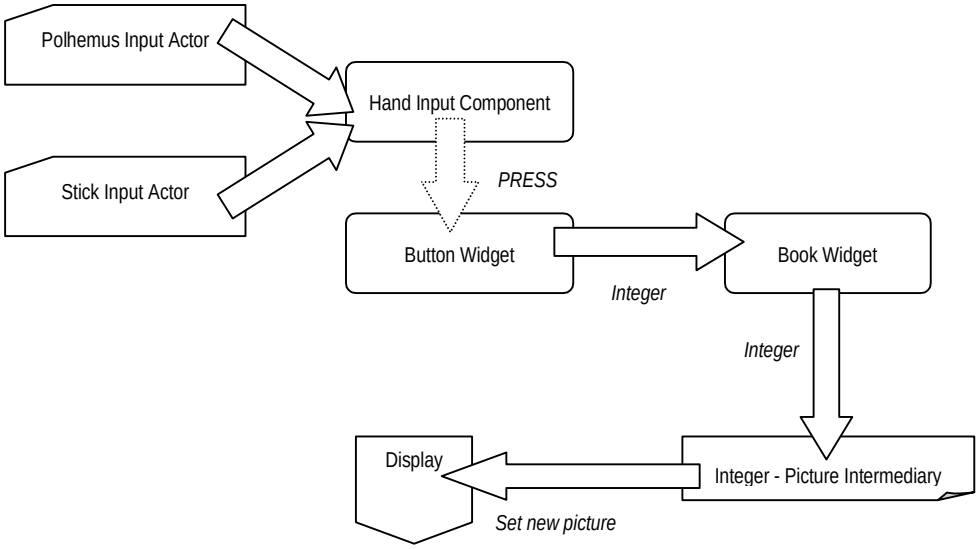


Figure 4 – 5 – Simple Data Flow Diagram for the VR Image Viewer.

Both applications use the virtual hand (`VRHandActor`) as their primary interaction component. The applications are only concerned with allowing the users to interact with objects within their reach. Thus, there is no allowance made for the user to travel inside the world, and only a collision selection technique is used to choose objects. Figure 4 – 5 details the data flow for the applications.

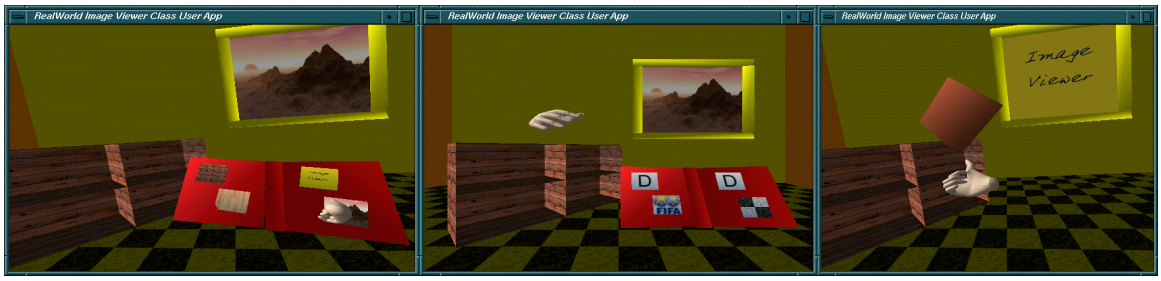


Figure 4 – 6 – The 'Real World' Image Viewer.

4.3.1. Real World Image Viewer

"This version of the image viewer is aimed at copying real life actions and objects. This favours the use of the hand interaction metaphor. The VE consists of a room with the hand and three objects: a bookcase, a picture frame, and a book of pictures. The idea is that many sets of images can be held in different books in the bookcase. A book removed from the bookcase can be placed anywhere, at which point it will open, displaying a set of images. Any of the images can then be selected for display in the picture frame; the book page can be turned to display another set of images, or the book can be returned to the bookshelf."

[Mundell, 99]

In addition to using the `VRHandActor` as the primary interaction component, various widgets from the CoRgi interaction system were also used in this application. The book is a specialized version of the `VRMenuWidget`, with additional functionality included to make it behave like a book (e.g. the ability to open and close, respectively hiding and displaying its component buttons). The images contained in the book were simply instances of a `VRButtonWidget`, linked to an integer-to-picture intermediary, which translated the unique integer ID each button produced when pressed, into a picture, which was displayed on the wall.

Figure 4 – 6 shows three screenshots from the real world image viewer. The first shows the selection of a particular image from the 'book'. The second shows the open book and virtual hand. The third shows the hand manipulating the 'closed' book.

The real world image viewer application required the implementation of several new widgets in the system. The book widget was the most interesting of these because, while based on the standard menu widget, significant extra functionality was implemented. The standard grab method used in the menu widget was overridden to implement the closing book function so that when moving the book around with the input component, the book closed and hid the buttons from the user. The bookcase is another interesting widget developed for this application. The bookcase widget implements a hierarchy of container widgets (i.e. containers for containers). In this case, the book acts as a container widget for the buttons, while the bookcase acts as a container widget for the book. This container hierarchy was

implemented using the listener (or parent) property of the widgets. The buttons have the menu as their parent and report any change in their state to the menu. The menu, in turn, records which button changed state last. Thus, the application need only query the menu (as to which button changed state last) instead of querying each separate button as to its state. The bookcase, which is essentially a container of menus, can also record which menu had a button that changed state, thereby eliminating the need to poll each menu. All of these extensions to the system were completed using the standard interaction framework – there was no need to change the basic framework to implement these applications.

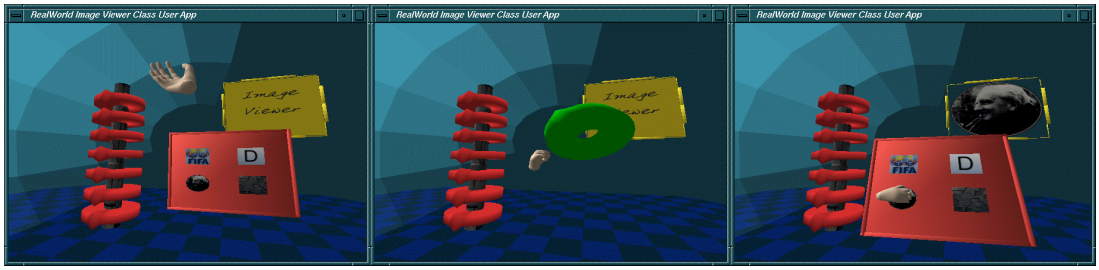


Figure 4 – 7 – The 'Abstract' Image Viewer.

4.3.2. Abstract Image Viewer

"Originally it was thought that the abstract image viewer would use surfaces with grids on them in place of the bookshelf. A shaded block in the grid selected would produce a plane holding the images. A network of these gridded surfaces was envisaged as a start towards experimenting with effective ways of accessing the system in a virtual world. However, once the real world version was developed it became clear that this is too complex and the specification was changed to a more direct replacement of the real world version."

[Mundell, 99]

The implementation of the Abstract Image Viewer mirrored that of the Real World Viewer in that all the components are the same, only their visual representations changed. The book is replaced by a ring, which gives the user better spatial cues about where to position the widget for best effect.

Figure 4 – 7 shows three screenshots from the abstract image viewer. The first shows the virtual hand, the virtual menu and buttons for each of the images and the pole to store the menu. The second screenshot shows the virtual hand manipulating the 'closed' menu. The last screenshot shows the selection of a particular image from the menu.

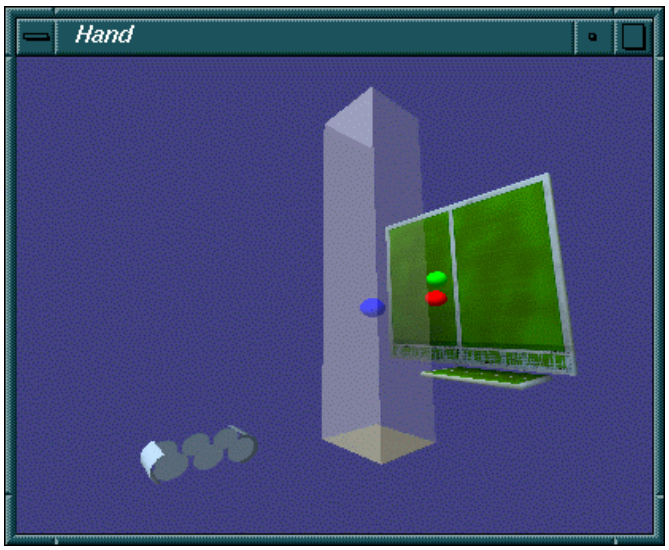


Figure 4 – 8 – The VRPhysicaApp Application.

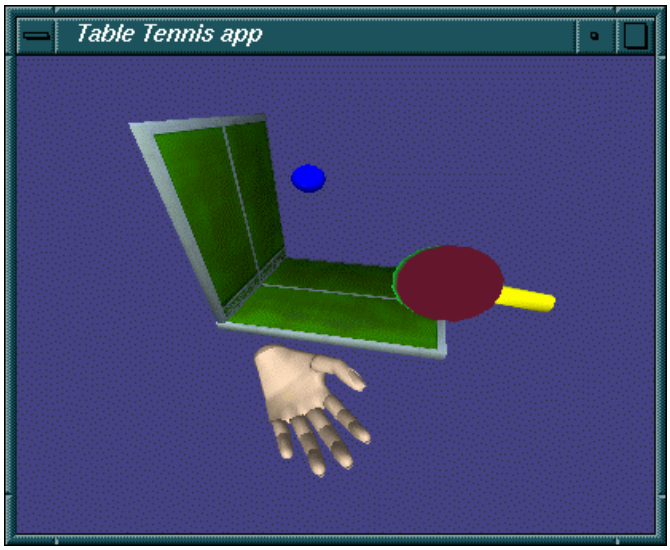


Figure 4 – 9 – The VRTTApp Application.

4.4. Physical Modelling

Dembovsky [Dembovsky, 99] developed a generic system for handling object interaction and physical modelling in a VR system.

"Physical modeling in Virtual Reality is a new and expanding field. Most such applications are either graphically rich but present little usability and have poor object interactions or are physically correct but have little visual appeal.

We design a framework, VRPhysicsEnvironment, that is at once graphically rich, has a wealth of object interactions and physical correctness."

[Dembovsky, 99]

The system implements a fast collision detection algorithm (RAPID) based on oriented bounding boxes. The objects in the system are all based on a specialized form of the `VREntity` object, called `VRPhysicsEntity`. The `VRPhysicsEntity` class adds the following attributes to the standard `VREntity` class:

- *Mass* and *velocity*: necessary for calculating momentum for use in collision equations.
- *Force*: necessary for calculating the overall effect of two objects colliding.
- *Gravity constant*: set to 9.8 m/s by default but changeable if required.
- *Physical Type*: this attribute defined how the object behaves in a collision. For example, a wall will not move, not matter how hard it is hit.
- *Collision Type*: defines which of the two collision detection types to use – either spherical or box shaped. This feature was not completely implemented in the final system.

These attributes are accessed in the same way that the standard `VREntity` attributes (e.g. position) are accessed usually via a `Set`/`Get` method pair (e.g. `SetMass` and `GetMass`). Collisions are handled centrally by a global collision detection and handling function which resides in the virtual environment. To demonstrate the use of this system, two applications were developed – the `VRPhysicsApp` and the `VRTTApp`.

4.4.1. VRPhysicsApp

Figure 4 – 8 shows a screenshot taken from the `VRPhysicsApp` application. The `VRPhysicsApp` application is designed as a physical modelling simulation application. Objects are placed in the system and the system is then 'released' allowing gravity and object interactions to take the system into its final stable, state. Figure 4 – 8 shows four different types of object in the system. The green square is a wall object, the coloured spheres are all ball objects, the semi-transparent object is an anti-gravity pad (reversing the effects of gravity on objects within its influence) and the final object is a conveyor belt. Each of these objects behaves differently in an object collision situation by having implemented different versions of the various property setting methods. For example, in a collision between a wall and a ball, each of the objects has their `SetVelocity` methods called by the collision detection system. The implementation of these two methods is different for each different type of object. The ball sets its new velocity to that given to it by the collision detection system. The wall on the other hand, simply ignores the commands to change velocity, since it cannot move and therefore has no velocity.

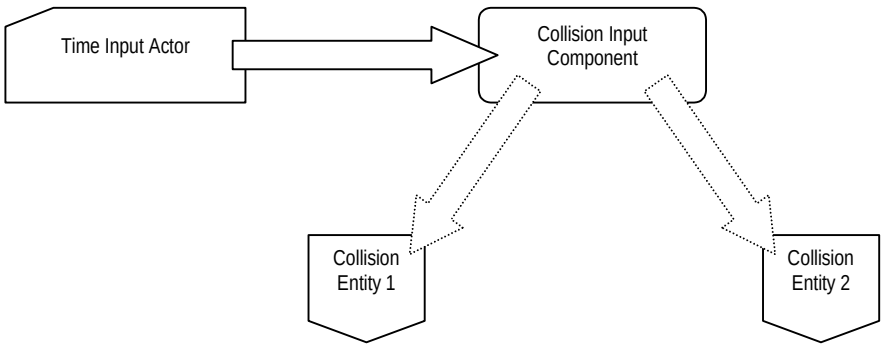


Figure 4 – 10 – Simple Data Flow Diagram using the Collision Input Component.

In the case of the `VRPhysicsApp`, the collision detection system runs as part of the VR application, not as a component of the interaction system. But, since the collision detection system operates only on `VRPhysicsEntities`, it can be considered to be another example of an interaction component. The input actor in this case would be an timer function, activating the collision input component at discrete intervals to check for collisions. Once a collision was detected, the mathematics for the new properties of each of the colliding parties could be calculated and the end results set in each of the `VRPhysicsEntities`. Figure 4 – 10 shows a data flow diagram using the collision input component.

The implementation of the `VRPhysicsEntity` class illustrates how the standard entity class (and the overall interaction framework) can be overloaded with specific operations and properties to provide extended functionality. In this case, the standard entity was given extra properties and methods. These new entities still interact with standard input components in the same way, but specialised input components, which are aware of their extra functionality are able utilise this extra functionality. Thus, the interaction framework is extensible in that overloading an existing class can add new types of interaction and/or object, which remain compatible with the general system. Figure 9 of Appendix A shows the UML diagrams for the `VRPhysicsEntity` classes and their relation to the standard `VREntity` class.

4.4.2. VRTTApp

Figure 4 – 9 shows a screenshot from the `VRTTApp` application. The `VRTTApp` application is a physical modelling system designed to simulate a table tennis game. This example was used in Chapter 3 to illustrate the use of the interaction framework to create a real application (see Figure 3 – 9). Figure 4 – 9 shows the table tennis table with one end rotated at 90^o to the other. This orientation of the table is to allow a single user to play the game by hitting the ball against the upright portion of the table. The diagram also shows the two interaction components in the system i.e. the virtual hand and the virtual bat, as well as the ball. The virtual hand in this system is a specialised form of the standard `VRHandActor`, which automatically collects the ball when the grab command is issued. The bat interaction component works simply by following the movements of the user's hand (via a Polhemus tracker) and causing object interactions with the ball.

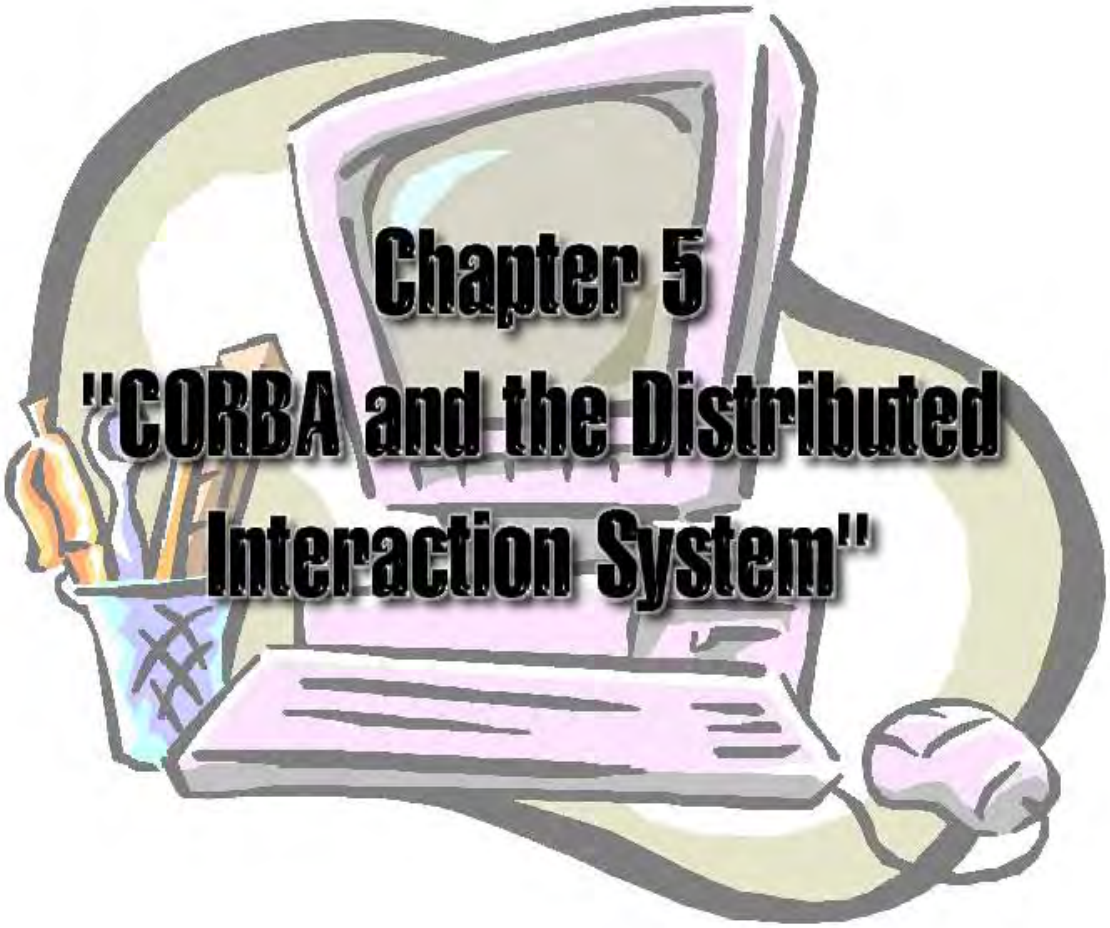
4.5. Conclusion

The choice of applications to use as case studies was one made by the users of the interaction system i.e. those people wanting to write immersive, interactive, VR applications. As such, the choice was made based on what applications were currently required as opposed to what applications will best use, test and justify the our implementation of the framework. Chapter 3 justified the framework itself, but a particular implementation of that framework can only be justified by creating real applications with it. Thus, while the framework may be considered to be complete, our implementation must be justified by actual applications such as those presented here.

Certain aspects of the framework were not covered by any of these applications. For example, none of the case studies uses wide range travel techniques. These techniques form part of the framework, and were implemented with the CoRgi interaction system. They have been tested in so far as writing a small application to check their basic functionality, but no large integrated applications have used them so far. Nonetheless, the case studies presented here integrate and use most of the functions of the framework and provide proof of the functionality of our implementation.

4.6. Summary

Several applications have been developed using the interaction framework described in the previous chapters. They have been described from the point of view of what parts of the interaction framework they utilise as well as any extensions they made to the original framework. These applications, along with the theoretical justification for the framework given in Chapter 3, serve as justification for the usefulness and completeness of the interaction framework, and its implementation – the CoRgi Interaction system.


Chapter 5
"CORBA and the Distributed
Interaction System"

5.1. The CoRgi Environment Distribution Paradigm

The basic CoRgi system supports distributed VR in the form of a client-server architecture. An application can create a central server environment, to which other client environments can connect. The server environment acts as a central repository for all the data comprising the virtual environment. Clients connect to the central server and obtain copies of this data, which they cache locally. When the data changes, the clients receive updates from the server. The distributed environment is currently implemented using standard TCP/IP and UDP. Since the interaction system is implemented separately from the rest of the system, as proposed by Shaw [Shaw, 1992], it is not necessary to distribute both in the same way.

In order to distribute the basic VR system, it is necessary to efficiently disseminate large amounts of data to all the clients. Thus, the system is distributed at the lowest possible level (the TCP network layer) in order to make it as efficient as possible. The interaction system is designed from the start to be separable into numerous distinct parts, with the interfaces between these parts designed to minimise the data flow between them. Thus, since the amount of data passing between the different parts is relatively small, we decided to distribute the interaction system at a higher level than the rest of the VR system. The definition of the interfaces between the different components was standardised early on in the implementation phase of this project. This interface definition was then used, in conjunction with the Common Object Request Broker Architecture (CORBA) middleware layer, to allow the interaction system to be spread over different platforms in a network.

5.2. The Common Object Request Broker Architecture (CORBA)

5.2.1. Introduction

"The CORBA specification, written and maintained by the Object Management Group (OMG), suppiles a balanced set of flexible abstractions and concrete services needed to realise practical solutions for the problems associated with distributed heterogeneous computing" [Henning, 1999]

Modern networks typically consist of various different types of machine, operating system, applications and transport layers, all having to talk to one another. The heterogeneous nature of these networks is due partially to legacy systems and partially to the fact that there is no *best tool* for all jobs. Any given combination of hardware, software and network will only perform well for a small subset of applications i.e. different applications place different demands on a system. The heterogeneous nature of modern data networks is also an advantage in that it increases the resilience of the overall system -

hopefully any problem (short of a loss of physical networking media) will only affect an isolated section of the network.

This heterogeneous nature of networks makes the writing (or porting to new systems) of distributed networking applications a difficult task. Not only do programmers have to deal with the different hardware and software conventions of the different systems, but they also have to deal with different network types, in order to get their product to the widest audience. [Henning, 1999] proposes two very general rules for solving the problems of distributing an application over a heterogeneous network:

- Find platform-independent models and abstractions that you can apply to help solve a wide variety of problems.
- Hide as much low-level complexity as possible without sacrificing too much performance.

These general rules are the basis of CORBA.

5.2.2. The Object Management Group (OMG)

The OMG was formed in 1989 to address the problems associated with developing a portable, heterogeneous application distribution system. The OMG produced a set of specifications, called the Object Management Architecture (OMA), which has at its core the CORBA specification. The OMA specifies how distributed objects are handled in a platform independent way and how these objects are able to interact with one another. The OMA is split into two related models – the *Object Model* and the *Reference Model*.

5.2.2.1. The Object Model

The Object Model describes how the interfaces to distributed objects may be described in a platform independent way. It describes an object as an *encapsulated entity* with an *immutable distinct identity* whose services are accessed only through well-defined *interfaces*. Clients use an object's services by issuing *requests* to the object [Henning, 1999]. The implementation of these services is not important to the calling object and, along with the location of the object, is not directly accessible.

5.2.2.2. The Reference Model

The Reference Model describes how distributed object interaction is achieved across heterogeneous networks. It provides *interface categories* that are general groupings for object interfaces. An Object Request Broker (ORB) conceptually links all of these interface categories. The ORB transparently facilitates the communications between the objects, activating them (if necessary) when they are requested. The Reference Model defines several categories of interfaces, all linked together using the ORB communications infrastructure:

- *Object Services*: these are domain independent interfaces, used by many different distributed object applications. Examples of these are the *Naming* and *Trading Services*. The naming services

acts as a central repository of object identities, which an application may search in order to get a reference (pointer) to the object that it requires. The trading service provides the same functionality, but instead of searching for a particular object, the application make a request for some particular service, and the trading service returns a reference to the best possible object.

- *Domain Interfaces*: these services play roles similar to the Object Services, except that they are domain specific i.e. less general in their application than the Object Services.
- *Application Interfaces*: these are developed specifically for a given application. These application specific services are not standardised by the OMG. However, should similar services appear in many different application domains, they may become standardised as part of one of the other interface categories.

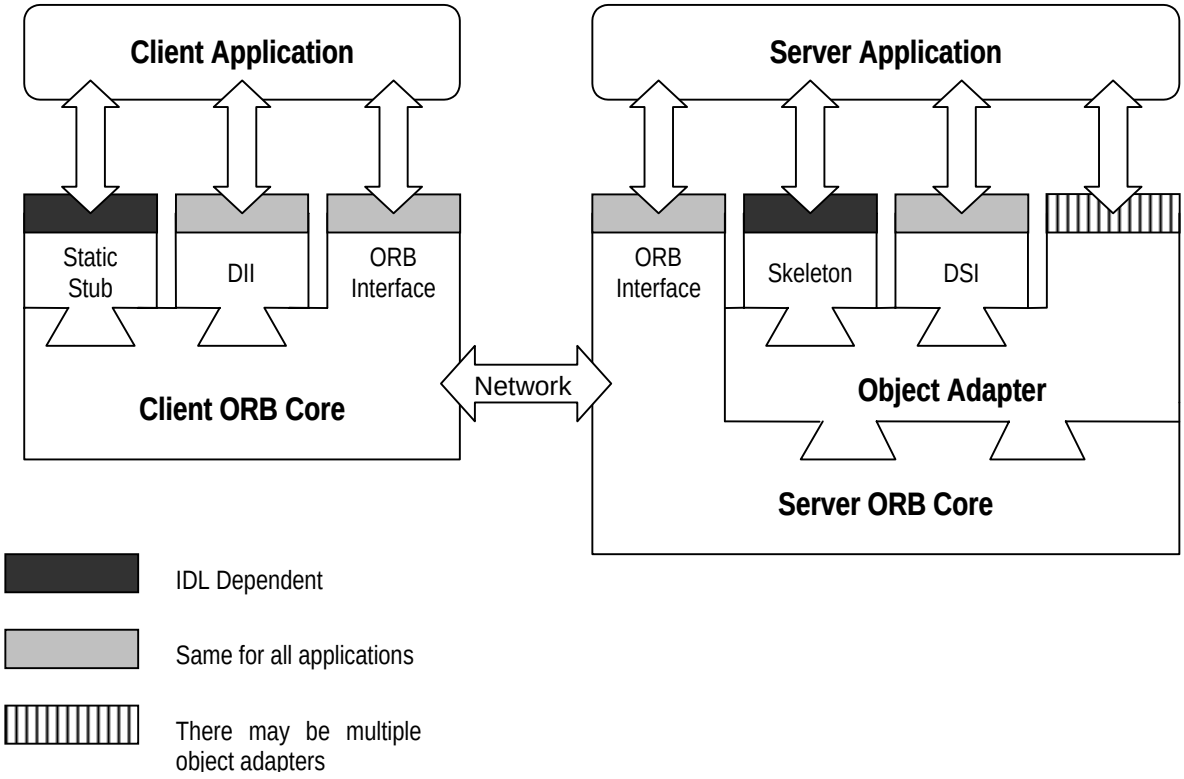


Figure 5 – 1 – Basic Data Flow in the CORBA Model.

5.2.3. CORBA Features

5.2.3.1. General Request Flow

Figure 5 – 1 shows the abstract data flow model for a client application making a request of a server application. Requests pass from client to server as follows:

1. The client has a choice of two options when making a request. The request can be passed to the ORB through either the *static stubs* or the *Dynamic Invocation Interface (DII)*. The static stubs are compiled into the object's interface implementation whereas the DII interface allows object interfaces to change during runtime. The DII also allows for the addition of new objects during

runtime. For the majority of cases, (and the CoRgi Interaction system in particular) the static stubs are sufficient, as the object interfaces are known at compile time.

2. The client ORB dispatches the request to the server ORB using the networking infrastructure.
3. The server ORB, on receiving a request, dispatches it to the object adapter responsible for creating the target object.
4. The client side object adapter then contacts the *servant*[1] on the server side, which implements the target object. Similarly to the client side, the server has the choice between a static and a dynamic invocation mechanism when contacting the servant.
5. After the servant has executed the request, the return values are passed back to the caller object in the client.

CORBA also implements several different types of request:

- *Synchronous*: dispatching a synchronous request causes the client to block until a return value is received. This form of request is identical to a *remote procedure call*.
- *Deferred Synchronous*: in this case, the client makes the request, continues processing and later polls for the response. Currently (as of CORBA 2.2) this form of request is only available through the DII interface.
- *Oneway*: this is a *best effort* type of request where the client is not assured that the request will get to the server. Oneway requests are simply sent to the server, the client continues executing and no response is allowed. These requests are most often used by ORB's for communicating network conditions e.g. congestion.

5.2.3.2. Interface Definition Language (IDL)

One of the requirements for a heterogeneous distributed system is to have some platform independent way of defining and distributing the interfaces to the objects in the system. The OMG defined the IDL to accomplish this task. The interface to an object lists the operations that it handles, as well as the data

OMG IDL	C++
short	CORBA::Short
long	CORBA::Long
long short	CORBA::LongLong
unsigned short	CORBA::Ushort
unsigned long	CORBA::Ulong
unsigned long long	CORBA::UlongLong
float	CORBA::Float
double	CORBA::Double
long double	CORBA::LongDouble
char	CORBA::Char
wchar	CORBA::Wchar
boolean	CORBA::Boolean
octet	CORBA::Octet

Table 5 – 1 – IDL to C++ Data Type Mapping

[1] CORBA terminology for the eventual implementation of an abstract object

types that those operations utilise. The IDL is not a programming language and defines the interfaces to objects only, not the implementation of the operations listed in the interface. The language independent nature of the IDL allows object implementations written in various different languages (those the IDL Mappings [2]) to inter-operate seamlessly. The IDL has integrated support for simple data types like `int` and `long`. Since these data types need to be language independent and have a fixed size, the IDL data types are mapped onto CORBA:: data types as detailed in Table 5 – 1. In addition to the simple data types listed in Table 5 – 1, the IDL also supports constructed types, such as enumerated types, structures, discriminated unions, sequences and exceptions. IDL also provides a module construct which is used for name scoping purposes.

The following is a fragment of the IDL definition of the CORBA Entity object defined as part of the CoRgi distributed interaction system:

```
struct CORBA_objectID
{
  long id;
};

// IDL definition of CORBA_Entity object
interface CORBAEntity
{
  // Attributes identifying whether to 'snap' the object
  attribute short SnapPosition;
  attribute short SnapOrientation;

  // VREntity action definitions
  long Grab(in CORBA_objectID parent);
  long Drop(in CORBA_objectID parent);
  long Point(in CORBA_objectID parent);
  long UnPoint(in CORBA_objectID parent);
  long Press(in CORBA_objectID parent);
};
```

The example shows the definition of a data structure called `CORBA_objectID` containing a single `long` value and the interface to an object called `CORBAEntity` which has four methods `Grab`, `Drop`, `Point`, `UnPoint` and `Press`, each taking a `CORBA_objectID` as a parameter and returning a `long`. The `CORBAEntity` object also includes two attributes, `SnapPosition` and `SnapOrientation`.

In IDL, method arguments need to have specified directions in order for the ORB to know what values need to be returned to the calling object. All of the parameters in the example are defined to be `in` parameters, meaning that they are passed from the client to the server only. It is also possible to define a parameter as an `out` parameter, in which case it is passed from the server back to the client or as an `inout` parameter which is passed both ways. IDL interfaces are also able to inherit from one another allowing for the definition of polymorphic objects. It should also be noted that all IDL objects inherit

[2] Currently only C, C++, Smalltalk, COBOL, Ada and Java

from the `Object` base class defined in the CORBA module. The `Object` base class defines a set of operations that are necessary for all CORBA objects.

In order to implement and use an IDL interface, it has to be translated into a suitable *language mapping*. For each IDL construct, the language mapping defines how that particular facility is implemented in a given programming language, making the objects accessible to application programmers. The C++ language mapping maps the IDL interfaces onto classes and operations of those interfaces are mapped onto methods within that class. Object references[3] map onto the `operator->` function i.e. they are either a pointer to a class, or an object of a class with an overloaded `operator->` member function).

5.2.3.3. Operation Invocation and Dispatch Facilities

CORBA applications operate by receiving/invoking requests on CORBA objects. The OMG specifies two general approaches to do this:

- *Static invocation and dispatch*: here, the IDL is translated into language specific *stubs* and *skeletons*, which are then compiled into the application programs. This gives the applications static knowledge of the programming language data types and functions mapped from the IDL definitions of the remote objects. The *stub* is the client side construct that allows the request to be made to the remote object via a normal function call. In C++, the *stub* is a member function of a class called a *proxy* because it represents the remote target object in the local application. Similarly, the *skeleton* is the server side construct that processes the request and dispatches it to the appropriate servant function.

- *Dynamic invocation and dispatch*: here, the construction and dispatch of CORBA requests is handled at run time rather than at compile time (as was done with the previous approach). Information about the interfaces and types of the remote objects is obtained either from a human operator or from an *Interface Repository (IR)*, a CORBA service that provides run time access to IDL definitions.

5.2.3.4. Object Adapters

An *object adapter* is an object that adapts the interface of one object to a different interface expected by a caller i.e. it uses delegation to allow a caller to invoke requests on an object without knowing the objects true interface [Henning, 1999]. Thus, the object adapter serves to connect the language dependent servant implementations (along with their invocation and dispatch facilities) to the CORBA ORB. CORBA object adapters have the following requirements:

[3] An *object reference* is a handle used to identify, locate and address a CORBA object

- The create the object references that allow clients to access the objects.
- They ensure that each target object has a servant implementation.
- The dispatch requests from the client-side ORB to the appropriate servant.

The object adapter was introduced as a separate layer from the ORB in order to perform these duties and thereby simplify the implementation of the ORB itself. In C++, servants are instances of C++ objects, typically derived from base classes produced by compiling the IDL interface definitions. In order to use these servants, they must be registered with an object adapter, which will dispatch any client requests to the appropriate servant. As of version 2.1 of CORBA, there are two different types of object adapter, the *Basic Object Adapter (BOA)* and the *Portable Object Adapter (POA)*.

5.2.3.4.1. Basic Object Adapter (BOA)

Initially, CORBA had only one type of object adapter, the BOA. The BOA was designed to be the only object adapter, but due to unforeseen circumstances, had the following deficiencies:

- The BOA did not account for the fact that, due to the need to support the servant implementations themselves, the BOA would end up being language dependent.
- Some necessary features were omitted from the BOA. Certain interfaces were not defined and there was no mechanism for servant registration operations. ORB vendors tended to overcome these problems with proprietary solutions, resulting in poor portability between different ORB implementations.

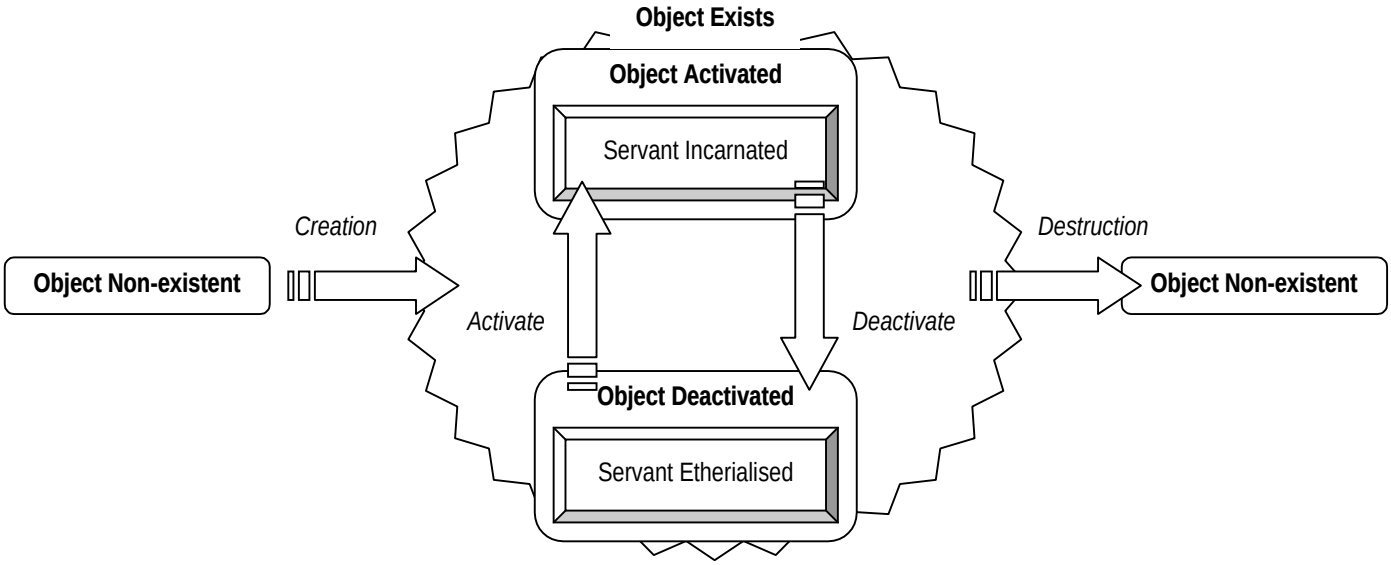


Figure 5-2 – The CORBA Object Life Cycle

5.2.3.4.2. Portable Object Adapter (POA)

The POA provides a portable (between different ORB vendors) way of interfacing language dependent servant implementations with language independent IDL interfaces. The POA is responsible from creating objects and object references. An object reference always results from the creation of a CORBA object. Once created, the object can alternate between being *activated* and being *deactivated*. When activated, the object is capable of servicing client requests. In order to receive client requests, the

object must be *incarnated* (or implemented) by a servant. It should be noted that the lifetimes of servants are completely separate from the lifetimes of CORBA objects. A single servant incarnates any given CORBA object at a given time. But the servant object incarnated by a CORBA object is able to change over time. Eventually, each servant is *etherialised* to break the bond between it and its CORBA object. Finally the CORBA object is destroyed and returns to a non-existent state (Figure 5-2). POA's maintain no persistent state i.e. they have to re-register each of their servants at each execution of the application.

A key feature of the POA specification is that applications can contain multiple POA instances. Each POA instance represents a group of objects with similar characteristics. The POA characteristics are controlled via *policies*, which have to be specified when the POA is created. All server applications must have at least one POA, the *RootPOA*, which has a standard set of policies. POA policies are, in fact, objects in the system with their interfaces defines in the `PortableServer` module.

The CoRgi distributed interaction system was implemented using the BOA initially but was later changed to support the POA for portability (between different ORB vendors) issues. The system does not require any of the advanced options supported by the POA (like *persistent* or *transient* [4] objects), so only the RootPOA was used. Additionally, a single automatic `ServantManager` was used to automate the process of binding CORBA objects to servants, taking further load off the developer. In our case, the change from BOA to POA did not introduce any additional functionality into the system, but it did improve the system's overall portability.

5.2.3.5. Inter-ORB Protocols

Before CORBA 2.0, ORB vendors used their own network protocols (or borrowed them from other vendors) and there was no standard way for different vendors ORBs to communicate. In CORBA 2.0, this communication was standardised with the *General Inter-ORB Protocol (GIOP)*. GIOP specifies transfer syntax and a standard set of message formats to allow independently developed ORBs to communicate over any connection-oriented network connection. The *Internet Inter-ORB Protocol (IIOP)* is a GIOP implementation over TCP/IP and must be supported by all ORBs that claim CORBA 2.0 (or greater) compliance. Additionally, ORB interoperability requires the use of a standard object reference format. While object references are opaque to the applications that use them, they contain information that all ORBs must be able to understand in order to communicate with the desired object. The standard reference format is called *Interoperable Object Reference (IOR)* and remains flexible enough to support any GIOP implementation. The IOR identifies one or more supported protocols and, for each protocol, encapsulates the data required to contact the server using that particular protocol. For

[4] These types of objects have their activation handled automatically by the CORBA system and are able to preserve their internal data between different program executions

example, an IIOP IOR contains a host name, TCP/IP port number and an *object key* that identifies the given object on the particular host/port combination.

5.2.3.6. Request Invocation

Clients manipulate objects by sending requests and getting results back from these requests. In order to make a request, the client must hold an object reference to the target object. This object reference acts as a unique handle for identifying the target object. In order to make a request, the following sequence of events occurs:

- The client obtains an object reference to the target object. This would usually be done using a Naming or Trading Service, but the object reference could also be passed by some other method. For example, in the CoRgi interaction system, the VR server creates an Entity Naming Object which identifies all the objects in the systems and stores their corresponding object references. In order for clients to be able to access this central object repository, they need to be supplied with an object reference for it. This object reference is passed to the client (using standard TCP/IP network methods) when it initially registers with the server.
- The client ORB then uses this object reference to locate the target object.
- The request is then passed on to the server ORB. The server ORB is responsible for activating/allocating a servant to the object if there is not already one present.
- The server ORB then calls the servant object with any arguments that were passed in the request. The ORB then waits for the servant to process the request.
- The server ORB then passes back to the client ORB any `in` and `inout` parameters that were part of the request. Should there be any problem with the execution of the request, the server ORB returns an exception (along with any additional data about the problem) to the client ORB.
- The client ORB then passes the returned request back to the client.

To the client application, the request invocation mechanism is completely transparent. The client makes the request using standard C++ method invocations on the CORBA stub (or DII interface) and the remainder of the mechanism is handled by the CORBA system. In particular, request invocation has the following characteristics:

- *Location transparency*: the client is not made aware of the actual location of the target object. The mechanism for making the request is the same whether the target object exists in the local address space, in a separate thread or process or on a separate machine altogether. In addition, server processes do not have to always execute on the same machine. There are mechanisms in CORBA to move them transparently to other machines.
- *Server transparency*: the client has no knowledge of which server implements which objects.
- *Language independence*: the implementation language of the servant is no important to the client object. For example, a C++ client can call an Java server using the same mechanism that it would use to call a C++ server. In addition, the implementation language of a servant can be changed without affecting any client requests.

- *Implementation independence*: the client needs no knowledge of the implementation details of the servant object. For example, the server may implement its servants by using non-object oriented techniques, yet the client still accesses these servants using the same, consistent object-oriented techniques that it uses for any other request.
- *Architecture independence*: the client is unaware of the server CPU architecture and is transparently shielded from details such as byte ordering, marshalling and structure padding. In addition, the client can make assumptions about the size of data types on the server.
- *Operating system independence*: the client is not made aware of the operating system of the server architecture.
- *Protocol independence*: the client has no control over the communication protocol used to contact the server. If several protocols are available, it is the job of the ORB to choose the best one for the job.
- *Transport independence*: the client is not aware of the transport and data link layer used to communicate the requests. ORBs can transparently use a variety of different networking technologies like Ethernet and ATM.

5.2.3.7. Object References

CORBA object references are analogous in use to C++ class instance pointers. Unlike C++ pointers though, they may denote objects implemented in different threads, processes or even on different machines. Aside from the distributed addressing capability, object references have many of the same attributes as C++ class instance pointers. The following is a list of the major attributes of CORBA object references:

- *An object reference supports exactly one object instance* i.e. a client holding an object reference may expect that the reference will always denote the same object, while that object continues to exist. An object reference is permitted to stop working only when its target object is permanently destroyed, in which case, any requests made using that object reference will produce an exception. Additionally, after an object is destroyed, its reference becomes permanently non-functional i.e. there is no chance that a reference to a destroyed object will accidentally refer to some other object at a later date.
- *Several different references may denote the same object* i.e. each reference *names* a single object, but any given object may have several *names*. This means that two object references, with different contents, do not necessarily refer to different objects. This is analogous to the operation of C++ class instance pointers.
- *References may be null* i.e. point nowhere. CORBA actually defines a specific pointer value to mean a NULL pointer.
- *References may* dangle (i.e. resemble C++ pointers that have had their instances deleted). Once an object adapter has supplied a client with an object reference, the adapter no longer has any control over it. This means that there is no automatic mechanism to inform a client when the object servant to the object reference that it holds, no longer exists. To find out whether a servant exists for a

given object reference, CORBA provides the `non_existent` object method to all CORBA objects.

- *References are opaque* i.e. the client may not examine or alter their contents. Parts of the object reference are standardised across all ORB implementations and parts are proprietary to a particular vendor. Thus, altering parts of the reference may render it unusable by certain ORBs.
- *References are strongly typed* i.e. each object reference contains some indication of the interface of the object that it points to. This allows ORBs to enforce type safety at run time. For statically typed languages (not using the DII interface) type safety is also enforced at compile time.
- *References support late binding* i.e. clients may treat a reference to a derived object as if it were a reference to a base class object. This is directly analogous to C++ virtual method calls. One of the major advantages of CORBA over traditional RPC implementations is that CORBA fully supports the concepts of late binding and polymorphism.
- *References can be persistent*. Clients can convert the object reference to a string and store it somewhere to be used at a later date. Provided that the servant that the reference pointed to is still operational, the reference will still operate as expected.
- *References can be interoperable* i.e. a reference supplied by one vendor's ORB will be equally valid on the ORB of another vendor, provided the standard IOR format is used.

5.2.3.7.1. Object Reference Content

Due to the issues of transport and location transparency inherent in CORBA, the object references all need to contain some minimum amount of information, encapsulated in a standard for that all ORBs are able to understand, the IOR.

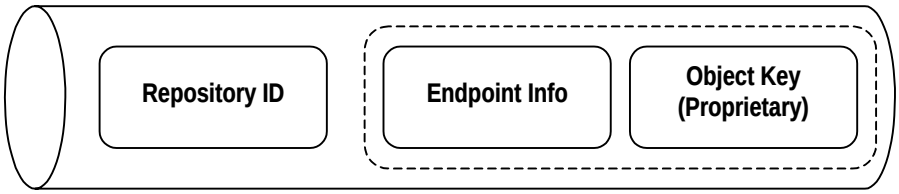


Figure 5 – 3 – Object Reference Contents.

An IOR contains the following basic information (Figure 5-3):

- *Repository ID*: a string identifying the most derived type of the IOR at the time the IOR was created. The repository ID allows you to locate a detailed description of the interface in the *Interface Repository*. The ORB is also able to use the Interface Repository to implement type-safe down casts.
- *Endpoint Info*: this field contains all the information needed by the client ORB to establish a network connection with the server ORB. The field contains information about what network protocol to use and physical addressing information appropriate to the network protocol chosen. The endpoint field may contain the actual address of the endpoint server, or it may point to some other implementation repository, which in turn contains details about the location of the server. This abstraction allows server processes to migrate from one machine to another without breaking

existing object references. CORBA also allows for the inclusion of several different types of network information in the endpoint field. This data allows the ORB to choose the best form of network connection when contacting the server.

- *Object key*: this field contains information, which is proprietary to a particular ORB vendor. The client side simply sends this field as a block of data (even the client ORB may not be able to decode it). Since the server ORB created the reference in the first place, it knows how to read it and will use it to address the correct object.

The combination of endpoint and object key fields may appear multiple times in the IOR. These multiple endpoint-key pairs (known as *multicomponent profiles*) permit an IOR to efficiently support more than one protocol and transport layer.

5.2.3.8. References and Proxies

When a client obtains an object reference, the client-side ORB instantiates a *proxy object* in the client's address space. This proxy is a C++ instance that supplies the client with an interface to the target object. The interface that the proxy presents to the client is the same as the interface on the target object. Any requests sent using the object reference go to the proxy object. The proxy object, in turn, sends a corresponding request to the remote object (through the ORB). When both objects (client and server) exist within the same address space, the proxy object is still used in order to maintain a uniform interface to all objects, remote and local.

5.3. The CoRgi Distributed Interaction System

The CoRgi distributed interaction system allows numerous different VR applications to share the same virtual environment i.e. users of one application are able to see and interact with objects created by users of other applications. Using this system, applications have the ability to create objects and (using CORBA) make them accessible to other applications.

The CoRgi interaction system consists of three main parts, the *CORBA Entities*, the *CORBA Interaction Actors* and the *Entity Naming Object*. The CORBA Entities and CORBA Interaction Actors are extensions of the standard CoRgi system Entities and Interaction Actors respectively, while the Entity Naming Object is an addition to the system. In the standard system, the `VREntity` object contains a `static` variable based linked list structure, which is used by the system to identify what entities are present in the environment. With the distributed system, this list needed to be centralised and accessible to all the clients, so it was moved into a separate object, the Entity Naming Object.

In order to retrofit CORBA into the already existing interaction system, several details had to be considered. The data passing between the objects was often in the form of system specific data types and not simple data types (e.g. `int` or `float`). In order to distribute these data types through CORBA, they first had to be defined in IDL. It was then necessary to create various translation

procedures to switch between the data types used in the base VR system and those defined for distribution by CORBA. Since the data types were very similar, the translation procedures were small, but they did introduce further overhead into the system. The other option was to redefine the base VR system data types as those created from the CORBA IDL. This idea was rejected, as it would have forced the use of CORBA even when the system was not being used to create a distributed application. This in turn would have introduced unnecessary overhead into single user applications.

5.3.1. CORBA Entities

The `VRCORBAEntity` is a class of the distributed system, corresponding to the `VREntity` base class from the standard system. Details of the `VRCORBAEntity` class are depicted in Figure 6 of Appendix A. The VRCORBAEntity class inherits from the `POA_CORBAEntity` base class which is generated automatically by the CORBA system from the IDL interface definition. These automatically generated base classes should not be edited, thus the functionality of the CORBA Entity is implemented in the inherited `VRCORBAEntity` class.

The `VRCORBAEntity` class contains the same methods as the `VREntity` class, making it seem equivalent from the application developer's point of view. Since many of the method calls pass or return data types defined by the system, the methods had to be overloaded with the CORBA data types as parameters. Making a method call on a `VRCORBAEntity` using the system data types as parameters will result in the parameters being converted into equivalent CORBA types, and the method call repeated with the new type parameters. Thus, the method calls with the system data types should not be overridden in an inherited object (as was done with the `VREntity` class), rather functionality is included by overriding the CORBA data type methods, which are defined as `virtual` in the `VRCORBAEntity` class. As an example of an inherited class that implements functionality, the `VRGenericCORBAEntity` inherits from the `VRCORBAEntity` class and overrides the `Grab`, `Drop`, `Point` and `UnPoint` methods.

The IDL used to generate the `CORBAEntity` base class contained definitions for 2 attributes for the class, *SnapPosition* and *SnapOrientation*. These attributes were also present in the `VREntity` class, where they were implemented as publicly available variables. They are used to set whether or not an entity should have its position/orientation snapped to some set of finite values. In the `CORBAEntity` class, these attributes are implemented as CORBA attributes. They have methods (`SnapPosition` and `SnapOrientation`) implemented in `VRCORBAEntity` which are used to set/get the values of the attributes. The other attributes associated with Entities (e.g. position, orientation, shape, etc.) are not implemented using the standard CORBA attribute implementation. In order to retain compatibility with the previous system (i.e. same method names), the setting/getting of the standard entity attributes was implemented as standard method calls, identical to the `VREntity` implementation.

5.3.2. CORBA Interaction Actors

The `VRCORBAHandActor` class is the distributed equivalent of the `VRHandActor` class. Both inherit from the VRInterface base class. Unlike the `VRCORBAEntity` class, the `VRCORBAHandActor` does not inherit from an CORBA generated class i.e. it does not offer any of its methods over the network. The `VRCORBAHandActor` does include the `CORBA.h` header file in order to be able to handle the object references that it uses to access objects. The major difference between the VRHandActor and the VRCORBAHand actor is the way in which they locate the correct object to perform an action on. Both use some method (collision detection by default, but ray-casting has also been implemented) to decide which object the user wishes to perform an operation on. These techniques provide the hand actor with an `objectID` (a unique identifier, used by the CoRgi system to address the objects). In the distributed system, when an Entity is created, its `objectID` is automatically stored, along with its object reference, in the central Entity Naming Object. In the non-distributed system, Entities are created inside of a linked list structure, searchable by `objectID`. Thus, the `VRCORBAHandActor` queries the central database to get the object reference, while the standard `VRHandActor` simply searches the local linked list of entities. Once the hand actor has the object reference (or pointer) to the required object, making a method call on the target object performs the relevant action. Further details of the object structure of the `VRCORBAHandActor` can be found in Figure 7 of Appendix A. As with the standard `VRHandActor`, the `VRCORBAHandActor` is usually inherited and the `CheckGesture` method overridden to provide the eventual functionality. This is demonstrated in the `VRCloseCORBAHandActor` class.

5.3.3. Entity Naming Object

The Entity Naming Object is a simplified version of the CORBA Naming Service. The exact details of the class are documented in Figure 8 of Appendix A. During normal usage of the system, it was noted that lookups for object references were the most prevalent of the CORBA requests made by the system. This was identified as being an area that required the maximum possible optimisation. The standard CORBA Naming Service is not optimised for this kind of system (it was designed for a more general case scenario), so we opted to develop our own, simple naming service which we could optimise as much as possible. The Entity Naming Object is defined in the class `EntityNamingObject,` which inherits from the class `POA_EntityNamingObject_CORBA,` which is automatically generated from the IDL interface definition of the interface. The EntityNamingObject implementation makes use of a class called `TreeNode` which, along with a static variable defining the root of the tree, implements a binary search tree structure for storing object references based on `CORBA_objectIDs`. The interface to the `EntityNamingObject` is similar to the interface provided by the standard CORBA Naming Service. There are two `public` methods, `LocateName` and `BindName` for retrieving and saving object references respectively. There are additional `private` methods that implement the binary search tree semantics.

The CORBA system is plagued by the same *chicken and egg* problem that plagues all networking systems. In order for the system to operate, it needs to know the location of a Naming Service (or the Entity Naming Object in our system), but how is this reference passed to the system? Most current CORBA systems overcome this problem in one of two ways. First, the Naming Service can be persistently running on a certain machine/port combination. Many ORBs provide a function for contacting a service running on a certain machine/port, but this is not standardised. The other method (currently being standardised by the OMG) is called Resolve Initial References. This involves the ORB itself storing details about CORBA services (usually retrieved from a system wide configuration file) and passing these on to the application as required. The CoRgi interaction system uses a slightly different method to those described above. Since we are dealing with a client server type architecture, we decided to store the object reference for the naming object in the central server. When an Entity Naming Object is created (either by a client or a server) its object reference is converted into a string and stored in the central server. When an application needs to access the naming object, it contacts the central server, retrieves the string corresponding to the object reference, and creates a CORBA object reference which is then used to address the naming object. The object reference for the naming object is stored in a string form in the central server so as not to force the central server to have any knowledge of the inner workings of CORBA object references.

The storing of object references in the Entity Naming Object is carried out transparently. When an application creates a new `VREntity` object, the `CreateObject` method in the `VREntity` class automatically retrieves the object reference of the naming object (if this has not already been done) and *binds* the object reference of the newly created object.

5.3.4. Results

The system implementation has been recently completed and is currently being tested. Several application programs are being developed, using the system, as part of the Rhodes Computer Science postgraduate degree. The main question that is currently being investigated is whether the extra overhead of using a middleware layer in a real-time application like VR is acceptable.

	IRIX - Linux CORBA CoRgi	IRIX (Local) CORBA CoRgi	Linux (Local) CORBA CoRgi
Standard Deviation (s)	0.0161900	0.0009234	0.0149617
Average time (s)	0.0187984	0.0026867	0.0170428

Table 5 – 2 – Timings for CORBA Implementation of Distributed CoRgi Interaction System.

In order to make some measurement of the performance of the system timings for function calls were made on the system, running in three different configurations. We chose to time the `Find()` method as it was found to be the method which was called most often. The `Find` method is part of the Entity Naming Object and implements a binary search tree containing CORBA object references for all the

entities in the system, indexed by their `objectID`. A call to `Find` takes an `objectID` as its only parameter, and returns a CORBA object reference. The values of these timings are listed in Table 5 – 2.

	IRIX (Local)	IRIX - Linux	Linux (Fast)
Standard Deviation (s)	0.0001987	0.0208853	0.0003160
Average Time (s)	0.0008882	0.0028763	0.0008098

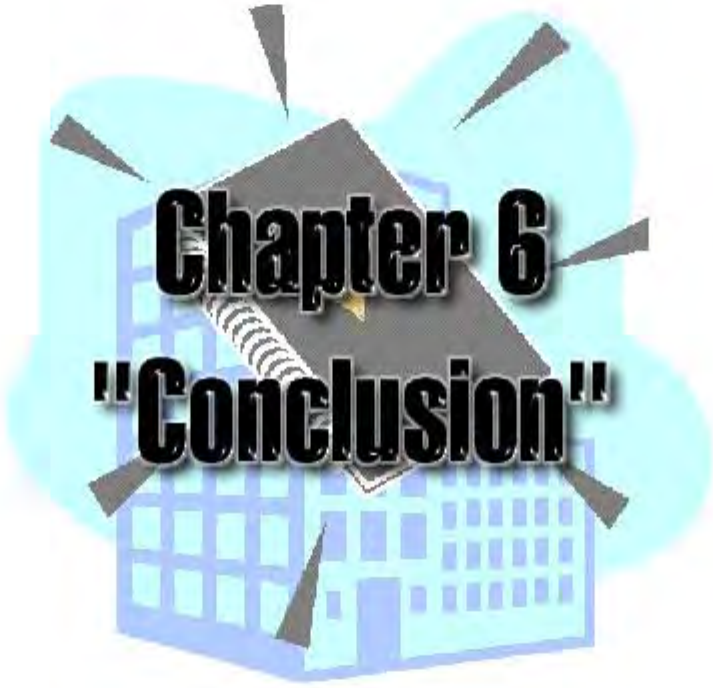
Table 5 – 3 – Timings for Simple CORBA Application.

In order to isolate the effects of the CORBA component on the system, a simple CORBA application was developed, independent of the CoRgi interaction system. The application implemented a single object which mimicked the Entity Naming Object's interface, but performed no processing i.e. a method call to the object simply resulted in the return of a NULL value. The timings for this system are detailed in Table 5 – 3. These timings, when compared with those in Table 5 – 2 show that the overhead introduced by CORBA is negligible (at least an order of magnitude less) than the overhead introduced by the system itself.

With the current CoRgi system, the bottleneck created by distributing the base system over the network is greater than any bottleneck in the interaction system. This system is being actively developed, and future versions will have more efficient and faster network distribution. With the advent of each new system, it will be necessary to measure performance metrics for the two different distribution factors and make the comparisons again.

5.4. Summary

The CoRgi interaction system was designed to be easily separable from the remainder of the CoRgi VR system. We defined the interfaces between the different parts of the interaction system and the remainder of the VR system at an early stage. CORBA is ideally suited for abstracting away network specific details in object oriented programming. Using CORBA and our predefined interfaces, we were able to extend the CoRgi interaction system to a distributed system. The main problem encountered when using CORBA for real time applications (like VR) is the extra overhead introduced by the abstraction layer. We found that with careful planning of the interfaces to minimise data flow across them, we were able to reduce the impact of the CORBA abstraction layer to the point where other factors in the system (namely the distribution of the VE itself) caused significantly more overheads.


Chapter 6
"Conclusion"

6.1. Introduction

Our framework was designed based on previous research and other interaction toolkits. The implementation was completed and used for several example applications. What follows is a synopsis of the work detailed in this thesis, along with a list of the achievements of this project and some ideas for future work in this field.

6.2. Related Work

Past research into the workings of immersive VR interfaces provides important insight into what is required to achieve a usable interactive VE. The eventual goal of any VE interface should be to allow users to interact in an effective way, while at the same time keeping the cognitive load introduced by the interface to a minimum. Reducing the cognitive load introduced by the interface allows the user to concentrate on the task they are trying to achieve, rather than concentrating on using the interface.

Many VR systems attempt to mirror the real world in as many ways as possible. While this naturalistic approach is important and effective, it is believed that unnatural (or *magic*) interaction techniques can go a long way to making an interface more effective and usable.

The set of taxonomies for travel, selection, release, manipulation and system commands give an abstract view to application developers of what their interfaces need to be capable of doing. Various VR toolkits have been studied and the different methods for implementing them (e.g. data flow *vs.* event based) have been compared. A brief overview of various systems has also been provided for the purposes of later comparison with the CoRgi interaction system.

Many years of research have produced a good understanding of the techniques required to implement interaction in current VR systems. All these different techniques were considered when designing and implementing our interaction framework.

6.3. Implementation

Immersive VR interfaces require a different approach from that used for traditional desktop interfaces. The data flow approach allows the interface to consist of multiple data pathways all operating in parallel. The CoRgi interaction system implements this data flow idea using *listener* attributes. The listener attribute is simply a pointer to an object in the system that is *informed* when a certain operation occurs. Thus, the data flow network in the CoRgi interaction system comprises objects, linked together by their listener attributes.

The CoRgi interaction system is abstractly separated from the remainder of the CoRgi VR system according to the framework detailed in Figure 3 – 2. This separation allows the different portions of the system to be developed independently, provided the interfaces between these parts are specified and those specifications adhered to. We have detailed the specifications for these interfaces and given details on the current implementation of the system under the CoRgi VR toolkit.

We have provided an example to the operation of the system, and justification of its workings based on the interaction taxonomies detailed in Chapter 2. We have also compared the system with other similar toolkits.

The justification of the framework using the accepted taxonomies of interaction, shows that the system includes all the salient features considered important for interaction in current VR systems. The comparison with other interaction toolkits shows our framework to be more general in its application, but also expandable to any particular specialised case.

6.4. Case Studies

Several applications have been developed using the interaction framework described in the previous chapters. They have been described from the point of view of which parts of the interaction framework they utilise as well as any extensions they made to the original framework. These applications, along with the theoretical justification for the framework given in Chapter 3, server as justification for the usefulness and completeness of the interaction framework, and its implementation – the CoRgi Interaction system.

The applications in Chapter 4 all demonstrate the flexibility of the framework. Each application was inherently different and required different services from the interaction system. Our framework was able to provide all the required services without requiring any modification other than logical extension (e.g. the implementation of new widgets). Additionally, these applications were developed by people with no previous knowledge of the system. They were able to quickly gain a working knowledge of the use of the interaction framework, thus demonstrating its ease of use.

6.5. Distributed System

The CoRgi interaction system was designed to be easily separable from the remainder of the CoRgi VR system. We defined the interfaces between the different parts of the interaction system and the remainder of the VR system at an early stage. CORBA is ideally suited for abstracting away network specific details in object oriented programming. Using CORBA and our predefined interfaces, we were able to extend the CoRgi interaction system to a distributed system.

The main problem encountered when using CORBA for real time applications (like VR) is the extra overhead introduced by the abstraction layer. We found that with careful planning of the interfaces to

minimise data flow across them, we were able to reduce the impact of the CORBA abstraction layer to the point where other factors in the system (namely the distribution of the VE itself) caused significantly more overheads.

The distribution of the system was made easy by using common interfaces to objects and a well designed abstraction layer, CORBA. The ease with which the system distributed is another measure of the flexibility of the framework.

6.6. Achievements

The eventual goal of this research is to produce a interaction framework, which can be used to easily and efficiently create immersive, interactive, VR applications. To this end, we have completed the following:

- A review of the current state of the art in VR interaction. This gives a detailed description of the current state of the art as regards VR interaction. The research includes details about particular interaction techniques, structures for classifying these techniques and overall frameworks and taxonomies of the techniques.
- Identification of the problems associated with interaction in immersive VR. We have used previous research from several different sources to compile a definitive list of the current shortcomings of immersive VR interaction. We have included problems like tracker accuracy and equipment cost, which we are unable to address, as well as problems like lack of tactile feedback and limited depth information, which we are able address by using accepted interaction techniques to solve specific problems.
- A discussion of interaction techniques and design philosophies for overcoming these problems. We have included detailed descriptions of the common interaction techniques as well as taxonomies for the classification of techniques into common groupings. These taxonomies are useful in that they allow us to identify the salient features of each group of techniques. This identification of salient features allows us to justify the completeness of a particular interaction toolkit, by showing that it satisfies all the features of each interaction technique section. Thus, we do not have to show details for each technique, rather we identify common features and show that our framework satisfies these.
- Identification of the requirements for a VR interaction framework to implement these techniques in a generic and extendable way. We have provided details of all the important factors that influence the design and implementation of an interaction framework.
- A review of current interaction toolkits and frameworks. A small number of interaction abstraction toolkits exist in the field, but most are geared towards solving some particular problem. We have examined the design and implementation of each of these toolkits and used this information to extend our own framework.
- The design of a flexible and generic VR interaction framework based on previous research. The framework is important in that it provides the basis upon which all the particular techniques will

depend. We based the design on accepted techniques in the field, thus eliminating the need to reproduce research that has already been carried out. Our eventual framework is based on several of the design methodologies considered. We examined each methodology in detail and chose the best points of each to integrate into our eventual design.

- A working implementation of this framework. The implementation of the interaction framework was completed as independently as possible from the underlying VR toolkit that was used – the Rhodes University CoRgi VR toolkit in this case. In this way, we were able to make our implementation of the framework as generic as possible. We showed the generic nature of our framework and its implementation by creating a distributed version of the implementation, using CORBA.
- Justification of the framework on theoretical basis by comparison against a proven taxonomy. We have illustrated how our framework is able to satisfy each of the requirements identified in the interaction technique taxonomy we utilised. Thus, the framework is able to satisfy all the requirements for each of the particular interaction techniques we detailed, as well as future techniques supported by the taxonomy.
- Justification of the framework on a practical basis by case studies and examples. We show practical use of the framework and its implementation by detailing several projects, completed as part of the Rhodes University postgraduate school, all utilising our interaction framework. The case studies also highlight the generic nature of the framework in that all the examples required some extension to the existing implementation, yet all the extensions are handled within the original framework.
- Proof of the usability of the framework by having case studies designed and implemented by developers not involved in the design and implementation of the interaction system. All the developers who used the system were new to the system, none had any part in the original design and implementation of the system. All were able to gain a working knowledge of the system in a short time and were able to produce applications that utilised and even extended the original implementation.
- Proof of the generic nature of the framework by moving to a distributed platform. The extension of the implementation to a distributed platform (using CORBA) highlights the generic nature of the framework and its implementation. The move to a distributed system was made easy through the design of the interfaces between the interaction system and the VR toolkit. These interfaces were designed so that a minimum amount of information need flow between the interaction system and the VR toolkit, thus changing to a different VR toolkit is made as easy as possible.
- A comparison of our framework against other existing interaction frameworks. Comparison with exiting immersive VR interaction toolkits showed our toolkit to be more generic than the existing ones, and thus able to be useful in a greater number of applications. But, while being more generic, our toolkit still retained the ability to efficiently implement all of the required interaction techniques and applications.

The framework and implementation we have developed are important to the field of VR as they enable application developers to quickly and easily implement interaction in their immersive VR applications. The interaction system handles the details of how the interaction is accomplished at a data level, leaving the developer free to think and implement their application at a higher abstract level. We have demonstrated the usefulness of the system through a series of case studies and the theoretical completeness of the system by comparison with other interaction toolkits and taxonomies of interaction techniques. In addition, we have shown that the framework is sufficiently generic to be easily extendable, but not to generic as to prove difficult to tailor to specific examples.

6.7. Future Work

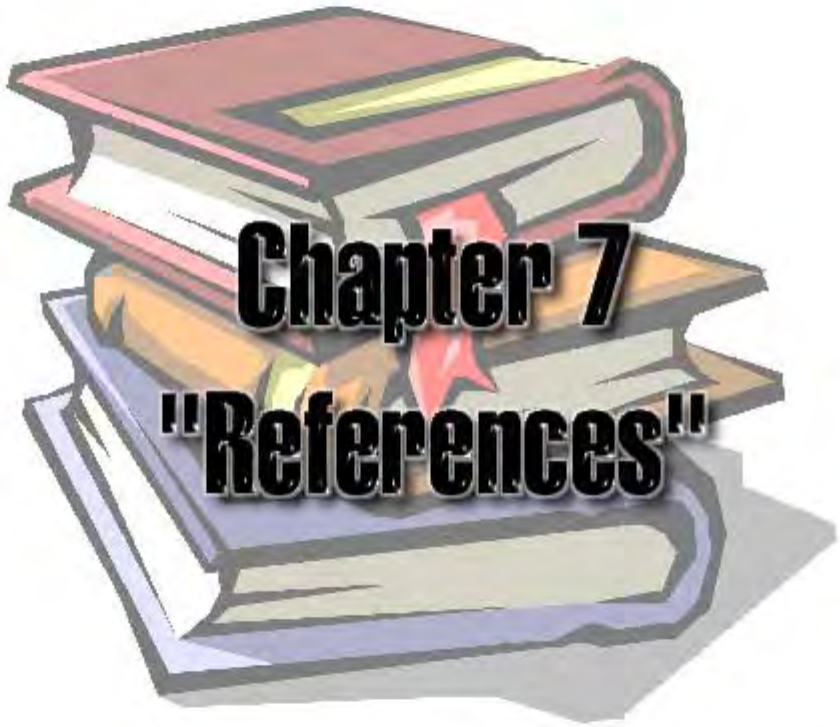
The CoRgi interaction system provides the basic low-level functions required to implement a VR interaction system. Currently, it provides only a very basic set of widgets to the application developer, namely the menu and button widgets. These two widgets can be used to implement usable VR interfaces, but do not constitute a complete set. Further research in this area could concentrate more on the psychological issues of 3D widgets, providing a more complete set of useful widget tools to the developer. Mine [Mine, 97-2] describes several 3D widgets developed specifically for interaction in VR. These widgets range from variations on standard desktop widget themes (e.g. the rotary selector) to purely immersive ideas (e.g. the head-butt zoom). Mundell [Mundell, 99] has done some research in this field, using the CoRgi interaction system. The research involved giving more intuitive representations to the standard menu and button widgets. The system currently implements a very limited set in interaction methods. As more applications are developed, so the interaction methods they require can be implemented using the CoRgi interaction system and incorporated into the overall system.

There has been much research lately in the field of formal specifications for interfaces. Most of this research has involved standard desktop interfaces, but an increasing number of researchers are now looking into formal specifications for immersive interfaces. For example, Jacob [Jacob, 99] has produced a specification language based on the parallel data-flow nature of immersive interfaces. The basis for Jacob's specification (a data-flow model) corresponds very well with the data-flow model used in the CoRgi interaction system. Future work could involve writing interpreters for Jacob's specification language, enabling CoRgi application developers to implement interfaces in a more intuitive manner. Currently, in the CoRgi system all interfaces are either hard coded into the application as raw C++ code, or read in from simple definition files, which define only the physical characteristics of the interface elements (e.g. size, shape, etc.) and contain no details of their data flow specifics.

Jacob also uses a visual tool for constructing his interfaces. Visual tools enable the developer to concentrate on the actual operation of the interface, instead of wasting time on the semantics of the interaction system implementation. The ideal for VR systems is to have an immersive tool for designing the immersive interfaces. Research in this area has been carried out for many years, with

several systems being produced [Conner, 92; Gobbetti, 93; Gobbetti, 94; Steed, 96; Steed, 97; Stevens, 94; Zeleznik, 93]. Many of these systems (e.g. [Steed, 96; Steed, 97]) utilise the same data-flow ideas employed in the CoRgi interaction system. They would thus form a sound basis for the implementation of a similar immersive interface construction application for the CoRgi interaction system.

Distributing the system could also introduce scope for future research. The current distribution system has been developed more as a proof of concept application than a usable system. Distributing the system introduces many new factors that are not covered in the current implementation, for example, the idea of *ownership*. Objects in the system may be owned by different users, and any particular user's ability to interact with a given object may depend on the objects ownership rights, in much the same way as the files in a UNIX file system have ownership attributes that determine what users are able to read, modify, etc. them.


Chapter 7
"References"

[Bowman, 95] Bowman, D. and Hodges, L., “User Interface Constraints for Immersive Virtual Environment Applications”. Graphics, Visualisation and Usability Centre, Georgia Institute of Technology, Technical Report TR95-26, 1995.

[Bowman, 97] Bowman, D. and Hodges, L., “An Evaluation of Techniques for Grabbing and Manipulating Remote Objects in Immersive Virtual Environments”. Proceedings of the 1997 Symposium on Interactive 3D Graphics, 1997.

[Bowman, 97-2] Bowman, D., Koller, D., and Hodges, L., “Travel in Immersive Virtual Environments: An Evaluation of Viewpoint Motion Control Techniques”. Proceedings of the Virtual Reality Annual International Symposium (VRAIS), 1997, pp. 45-52.

[Bowman, 99] Bowman, D., “Interaction Techniques for Common Tasks in Immersive Virtual Environments - Design, Evaluation and Application”. Doctoral dissertation, Georgia Institute of Technology, June 1999.

[Carey, 97] Carey, R., Bell, G. and Marrin, C., “Virtual Reality Modelling Language ‘97 – International Standard (ISO/IEC 14772-1:1997)”. The VRML Consortium Incorporated. Available: http://www.vrml.org/technicalinfo/specifications/vrml97/index.htm. 26 October, 1999.

[Conner, 92] Conner, D.B., Snibbe, S.S., Herndon, K.P., Robbins, D.C., Zeleznik, R.C. and van Dam, A., “Three-dimensional widgets”. Proceedings of the 1992 Symposium on Interactive 3D Graphics, 25(2), ACM SIGGRAPH, March, 1992, pp. 183-188.

[van Dam, 97] van Dam, A., “Post-Wimp User Interfaces: the Human Connection”. In Communications of the ACM 40(2) (1997).

[Dembovsky, 99] Dembovsky, C. and Bangay, S., “VRPhysics Environment - A Framework For Collision Detection and Modelling in a Virtual Reality Environment”. Honours Thesis, Computer Science Department, Rhodes University, November 1999.

[Gobbetti, 93] Gobbetti, E., Balaguer, J. and Thalmann, D., “VB2: An Architecture for Interaction in Synthetic Worlds. Proceedings of the ACM UIST '93, Atlanta pp. 167-178.

[Gobbetti, 94] Gobbetti, E. and Balaguer, J. F., Virtuality Builder II: On the Topics of 3D Interaction". In Virtual Worlds and Multimedia, John Wiley (ASIN: 0471939722), pp. 99-111.

[Henning, 1999] Henning, M., Vinoski, S., "Advanced CORBA Programming with C++". Published in the Addison-Wesley professional computing series, ISBN 0-201-37927-9.

[Hix, 99] Hix, D., Swan, E., Gabbard, J., McGee, M., Durbin, J. and King, T., "User-Centred Design and Evaluation of a Real-Time Battlefield Visualisation Virtual Environment". In Proceedings of IEEE Virtual Reality '99 (formally Virtual Reality Annual International Symposium), March 1999.

[Isdale, 98] Isdale, J., "What Is Virtual Reality?". Available: http://vr.isdale.com/WhatIsVR/. 26 October 1999.

[Jacob, 99] Jacob, R., Deligiandnidis, L., and Morrison, S., "A Software Model and Specification Language for non-WIMP User Interfaces". ACM Transactions on Computer-Human Interaction, Vol. 6, No. 1, March 1999, pp. 1-46.

[Kessler, 97] Kessler, D., Kooper, R. and Hodges, L., "The Simple Virtual Environment Library (Version 2.0) Users Guide". Graphics, Visualisation and Usability Centre, Georgia Institute of Technology, USA, April 1997. Available: http://www.cc.gatech.edu/gvu/virtual/SVE/docV2.0/sve.book_1.html. 26 October 1999.

[Kessler, 99] Kessler, D., "A Framework for Interactors in Immersive Virtual Environments". IEEE VR '99, Houston, TX, Mar. 1999. pp 190-7.

[Lindeman, 99] Lindeman, R. W., Sibert, J. L. and Hahn, J. K., "Towards Usable VR: An Empirical Study of User Interfaces for Immersive Virtual Environments". Proceedings of ACM CHI '99, pp. 64-71.

[Mine, 95] Mine, M. R., "Virtual Environment Interaction Techniques". UNC Chapel Hill Computer Science Technical Report TR95–018.

[Mine, 97] Mine, M. R., Brooks, F. P. (JR) and Sequin, C. H, "Moving Objects in Space: Exploiting Proprioception In Virtual-Environment Interaction". Proceedings of SIGGRAPH 97, August 1997, pp. 19–26.

[Mine, 97-2] Mine, M. R., "Exploiting Proprioception in Virtual-Environment Interaction". Doctoral dissertation, Department of Computer Science, University of North Carolina, Chapel Hill, 1997.

[Mundell, 99] Mundell, M. and Bangay. S., "Towards a Virtual Operating Environment: Exploring Immersive Virtual Interface Design using a Simple VR Image Viewer". Honours Thesis, Computer Science Department, Rhodes University, November 1999.

[Norman, 90] Norman, D., "The Design of Everyday Things". *Doubleday, New York (1990)*, ISBN 0385267746.

[Ozer, 98] Ozer, J., "3D Computing". In PC Magazine, Vol. 17, No. 11 (June 1998), pp 118.

[Polhemus, 96] "Polhemus FastTrak®". Available http://www.polhemus.com/ftrakds.htm. 5 December, 1999.

[Poupyrev, 96] Poupyrev, I., Billinghurst, M., Weghorst, S. and Ichikawa, T., "The Go-Go Interaction Technique: Non-Linear Mapping for Direct Manipulation in VR". In proceedings of UIST '96, Seattle, WA: ACM pp. 79-80.

[Poupyrev, 97] Poupyrev, I., Weghorst, S., Billinghurst, M. and Ichikawa, T., "A Framework and Testbed for Studying Manipulation Techniques for Immersive VR". In proceedings of VRST 97: ACM, 1997.

[Rahn, 98] Rahn, S., "WorldToolKit™ Release 8 Technical Overview". Sense8 Corporation, Mill Valley, CA, February 1998. Available: http://www.sense8.com/products/wtk_tech.pdf. 26 October 1999.

[Robertson, 89] Robertson, G., Card, S. and Mackinlay, J., "The Cognitive Coprocessor Architecture for Interactive User Interfaces". Proceedings of ACM SIGGRAPH/SIGCHI 1989 Symposium on User Interface Software and Technology, 13-15 November, 1989, pp 10 – 18.

[Romer, 1999] Romer, K. and Puder, A., "The Mico CORBA 2.2 Implementation". Available: http://www.mico.org. 26 October 1999.

[Rorke, 98] Rorke, M., Bangay, S. and Wentworth E., "Virtual Reality Interaction Techniques". Proceedings of the 1st Annual South African Telecommunication, Networks and Application Conference (SATNAC), Cape Town, South Africa, September, 1998, pp. 526-532.

[Rorke, 99] Rorke, M. and Bangay, S., "The Virtual Remote Control - An Extensible, Virtual Reality, User Interface Device". In proceedings, South African M. & PhD. conference, June 1999, pp 39-43.

[Shaw, 92] Shaw, C., Liang, J., Green, M. and Sun, Y., “The Decoupled Simulation Model for Virtual Reality Systems”. Published in Proc. CHI '92. Monteray,CA., May, 1992, pp 321–328.

[Shaw, 93] Shaw, C., Green, M., Liang, J. and Sun, Y., “Decoupled Simulation in Virtual Reality with The MR Toolkit”. ACM Transactions on Information Systems, Volume 11 Number 3, July 1993. pp 287-317.

[Siegel, 1997] Siegel, J., “CORBA - Fundamentals and Programming”, John Wiley and Sons, Inc., ISBN 0-471-12148-7.

[Stahl, 97] Stahl, O. and Anderson, M., “DIVE – A Toolkit for Distributed VR Applications”. Swedish Institute of Computer Science (SICS), Stockholm, Sweden. Available: http://www.scics.se/dce/dive/online/ercim.html. 26 October 1999.

[Steed, 94] Steed, A. and Slater, M., “A User-Defined Virtual Environment Dialogue Architecture. Virtual Reality Software and Technology”, Proceedings of VRST 94, 23-26 August 1994, World Scientific. pp87-96.

[Steed, 96] Steed, A. and Slater, M., “A Dataflow Representation for Defining Interaction Within Immersive Virtual Environments”. Proceedings of VRAIS 96, IEEE Computer Society.

[Steed, 97] Steed, A., “Dataflow Languages for Immersive Virtual Environments”. Virtual Environments on the Internet, WWW and Networks. 15–17 April 97, National Museum of Photography, Film & Television, Bradford, UK.

[Stevens, 94] Stevens, M.P., Zeleznik, R.C., Hughes, J.F., “An Architecture for an Extensible 3D Interface Toolkit”. Proceedings of UIST '94, ACM SIGGRAPH, November, 1994.

[Stoakley] Stoakley, R., Conway, M. and Pausch, R., “Virtual Reality on a WIM: Interactive Worlds in Miniature”. Proceedings of CHI, 265-272.

[White, 99] White, L., “MR Toolkit”. Computer Graphics Research Group, Department of Computer Science, University of Alberta, Canada. Available: http://www.cs.ualberta.ca/~graphics/MRToolkit.html. 26 October 1999.

[Winnemoeller, H, 99] Winnemoeller, H. and Bangay, S., “"Investigation into gestures as an input mechanism – A simple 3D modelling application in a virtual reality environment". Honours Thesis, Computer Science Department, Rhodes University, November 1999.

[Wloka, 95] Wloka, M. and Greenfield, E., “The Virtual Tricorder: A Uniform Interface for Virtual Reality”. In Proceedings of UIST'95, ACM Press, November, 1995, pp. 39-40.

[Zeleznik, 93] Zeleznik, R.C., Herndon, K.P., Robbins, D.C., Huang, N., Meyer, T., Parker, N. and Hughes, J.F., “An interactive toolkit for constructing 3D interfaces”. Proceedings of SIGGRAPH '93, 27(4), ACM SIGGRAPH, July, 1993, pp. 81-84.


Appendix A
"UML Diagrams"

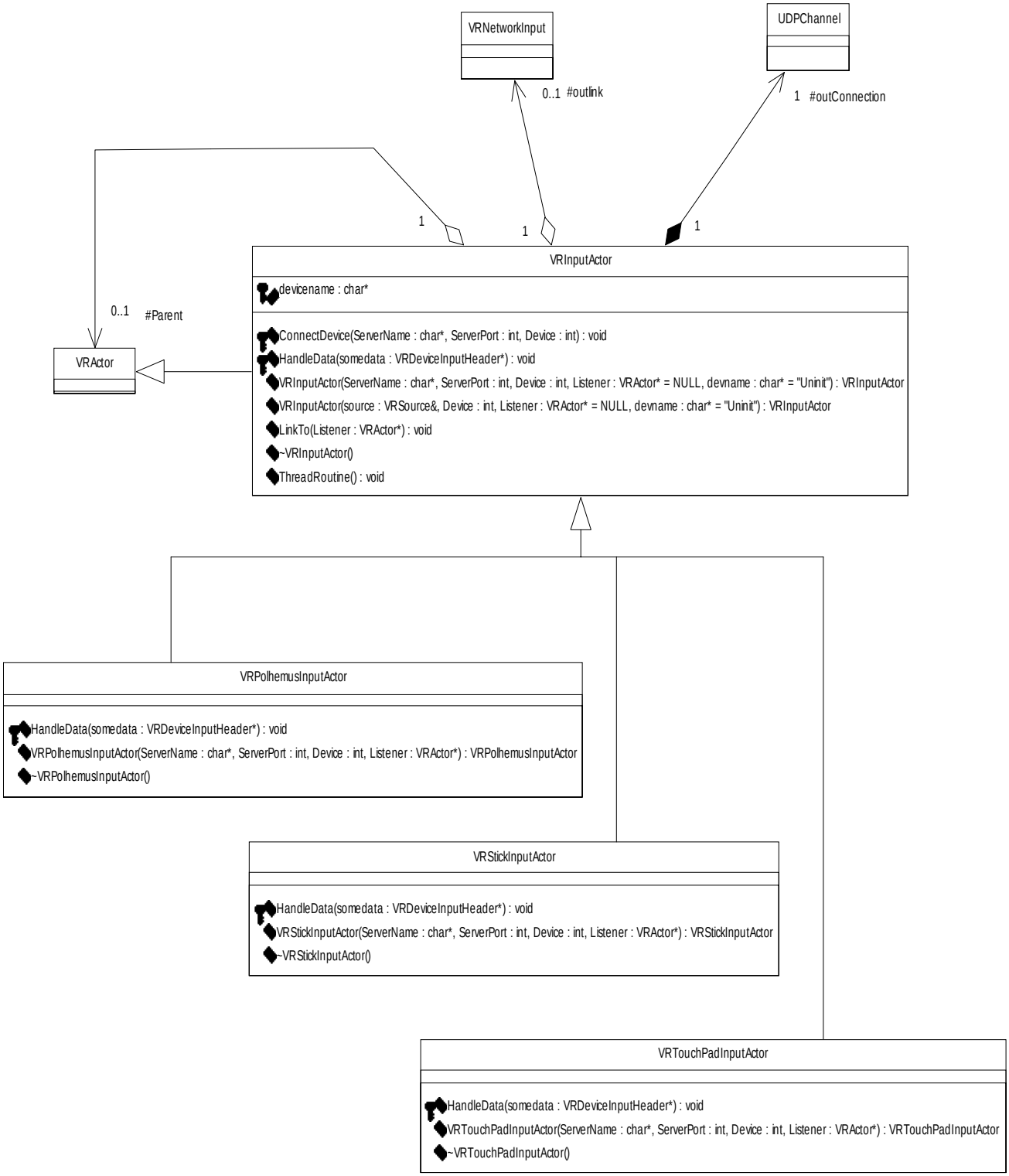


Figure 1 – UML Diagrams for Input Component Classes.

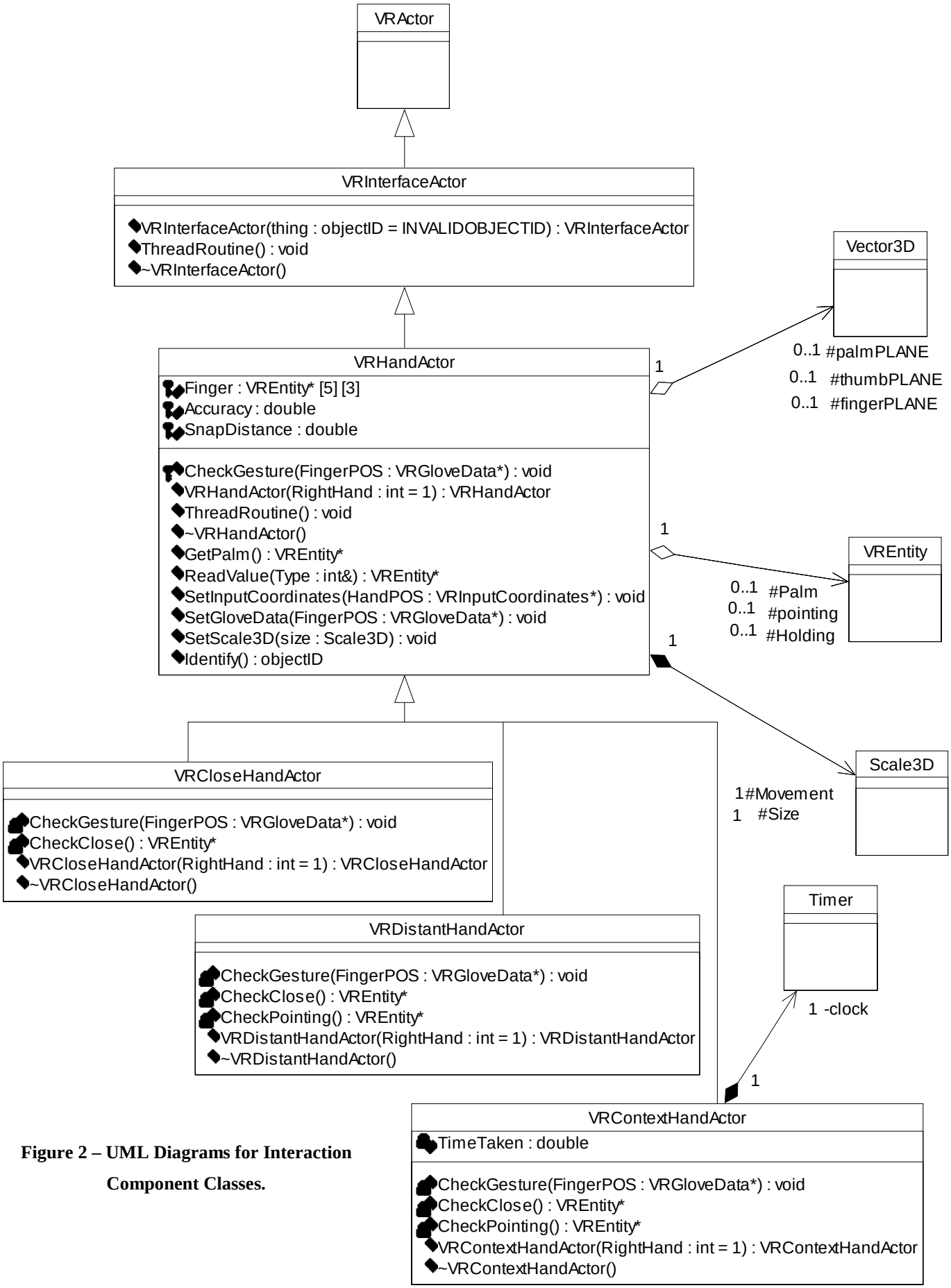


Figure 2 – UML Diagrams for Interaction Component Classes.

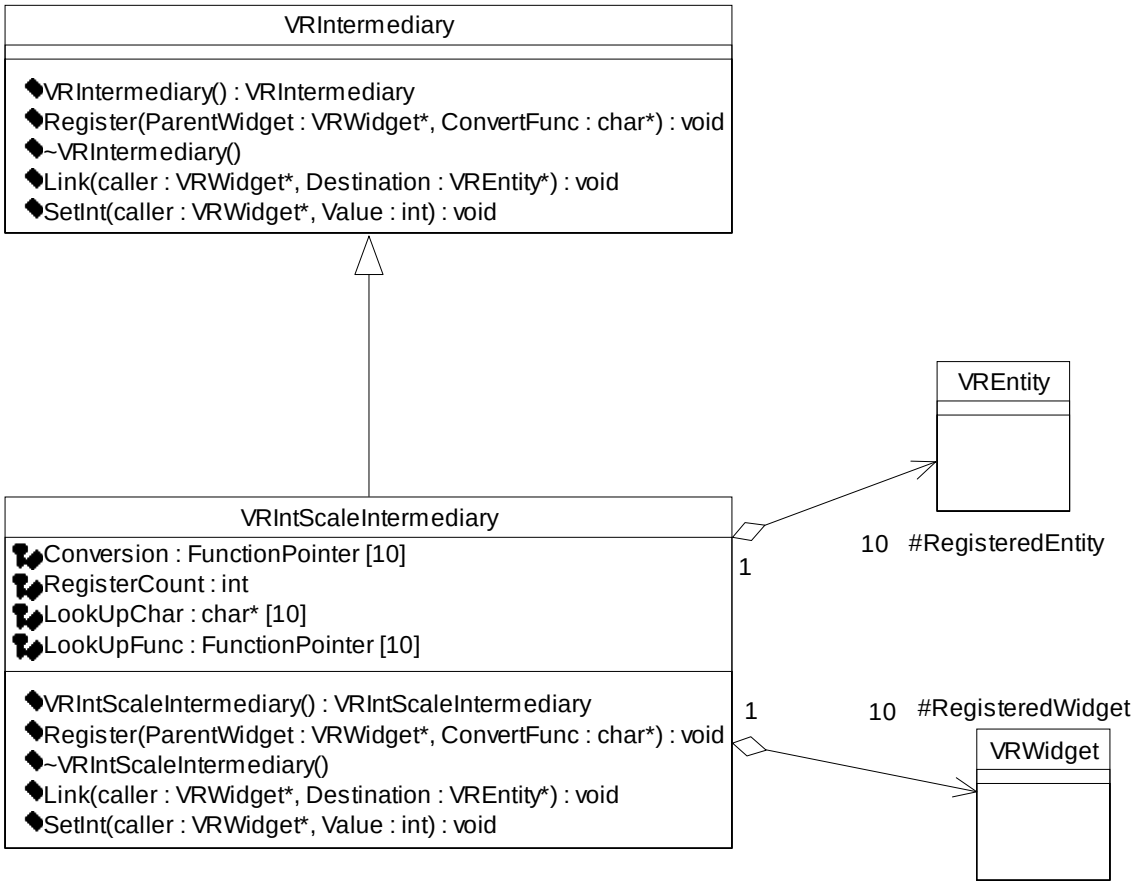


Figure 3 – UML Diagrams for Intermediary Component Classes.

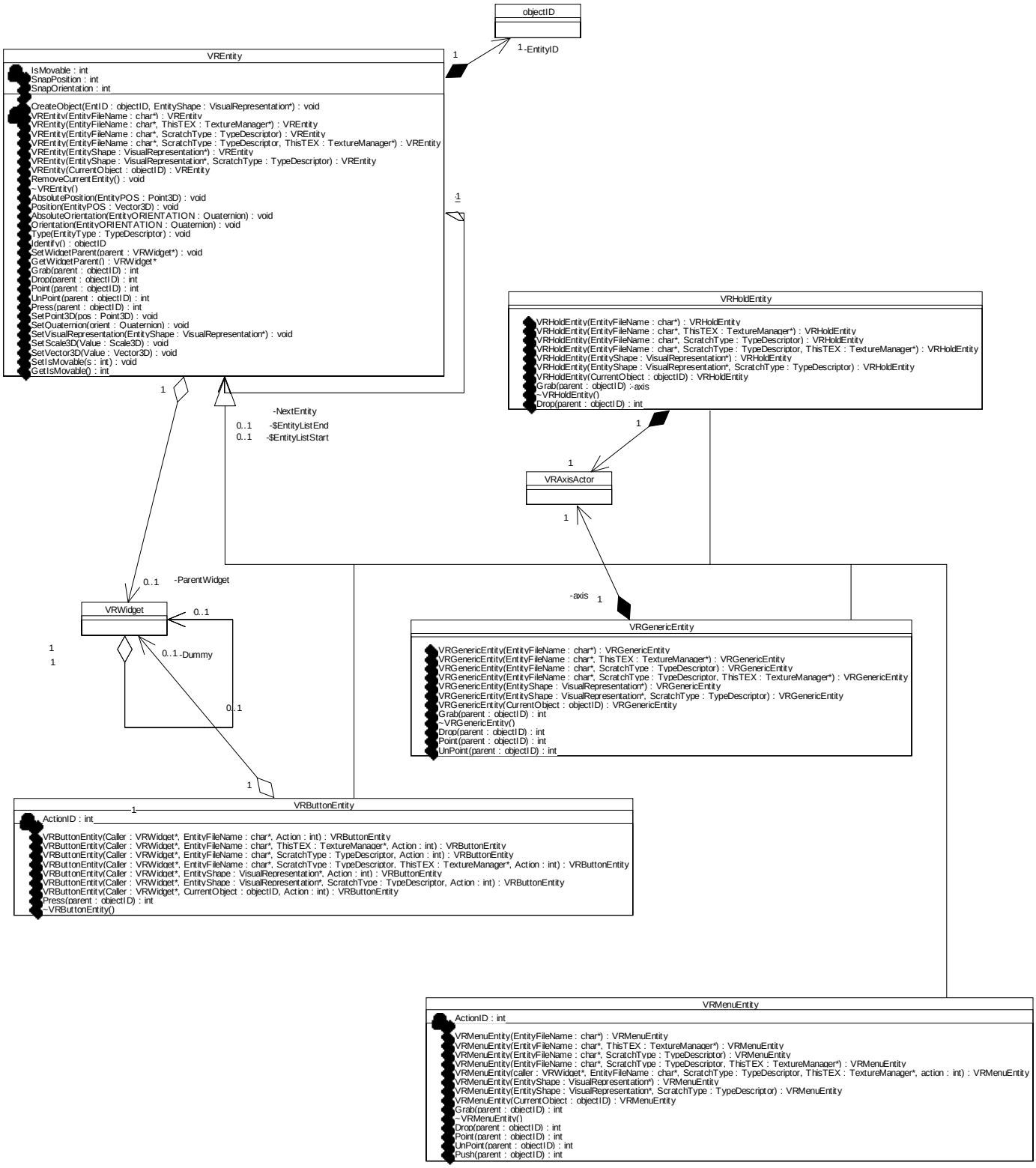


Figure 4 – UML Diagrams for Entity Component Classes.

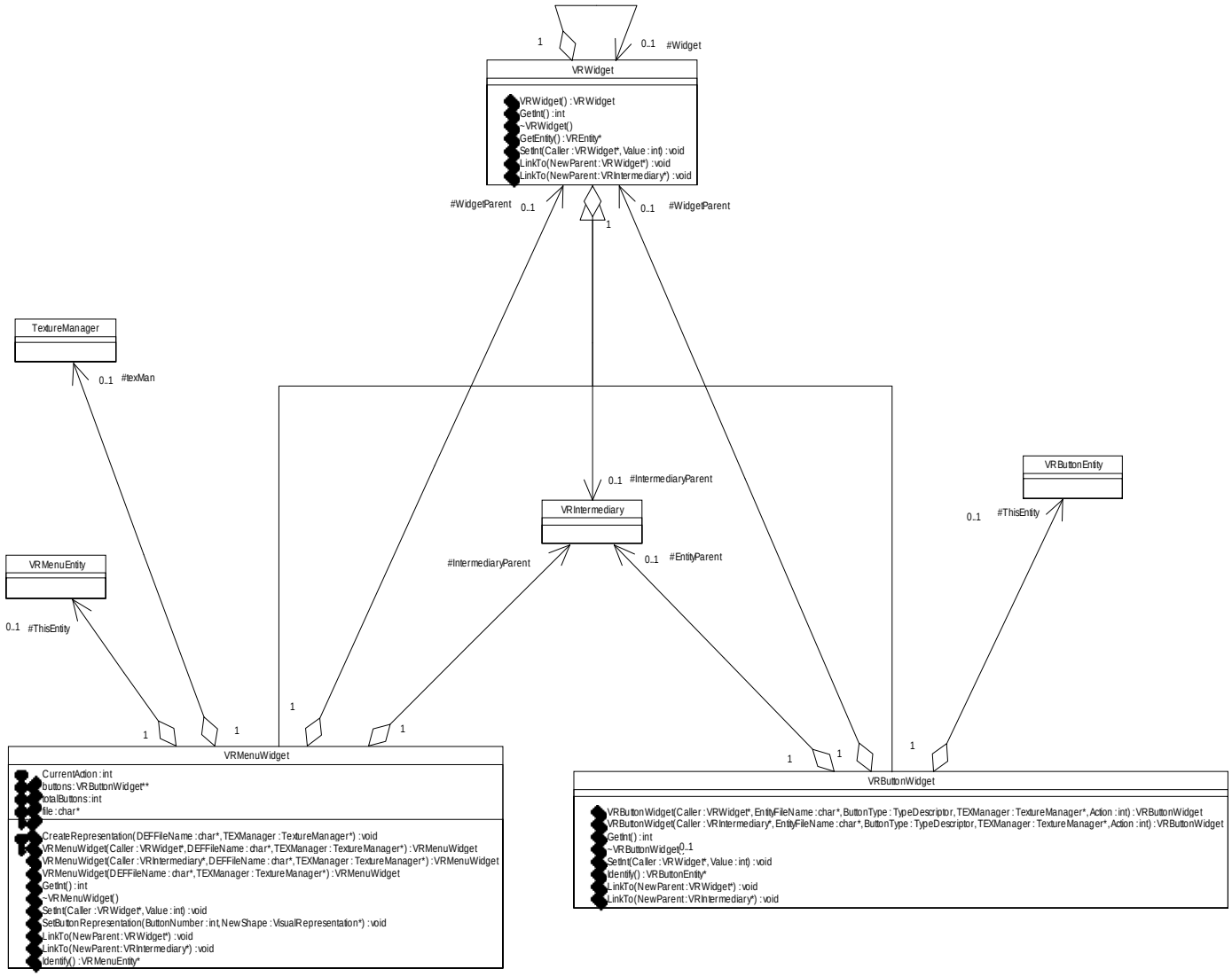


Figure 5 – UML Diagrams for Widget Component Classes.

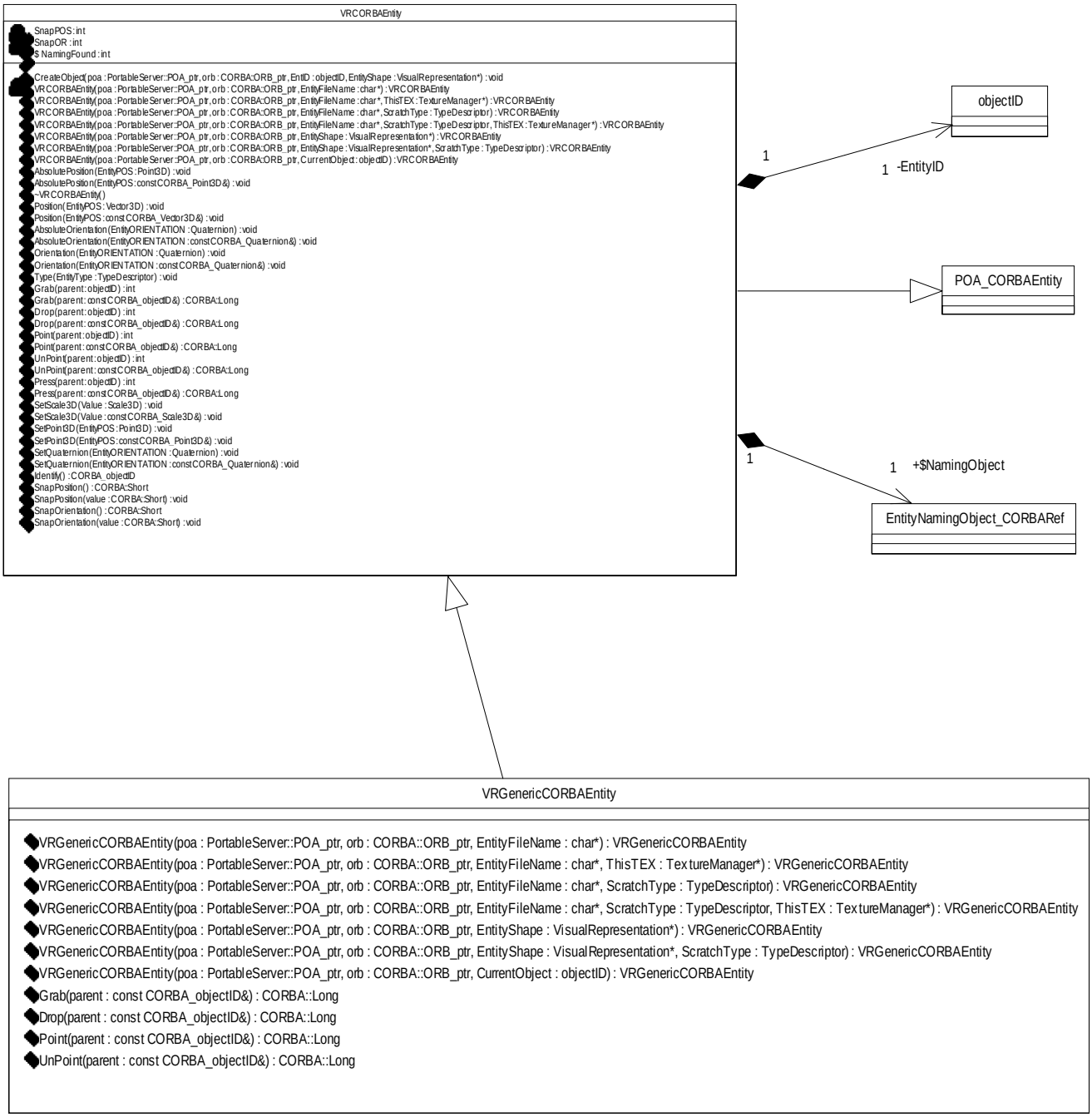


Figure 6 – UML Diagrams for the CORBA Entity Component Classes.

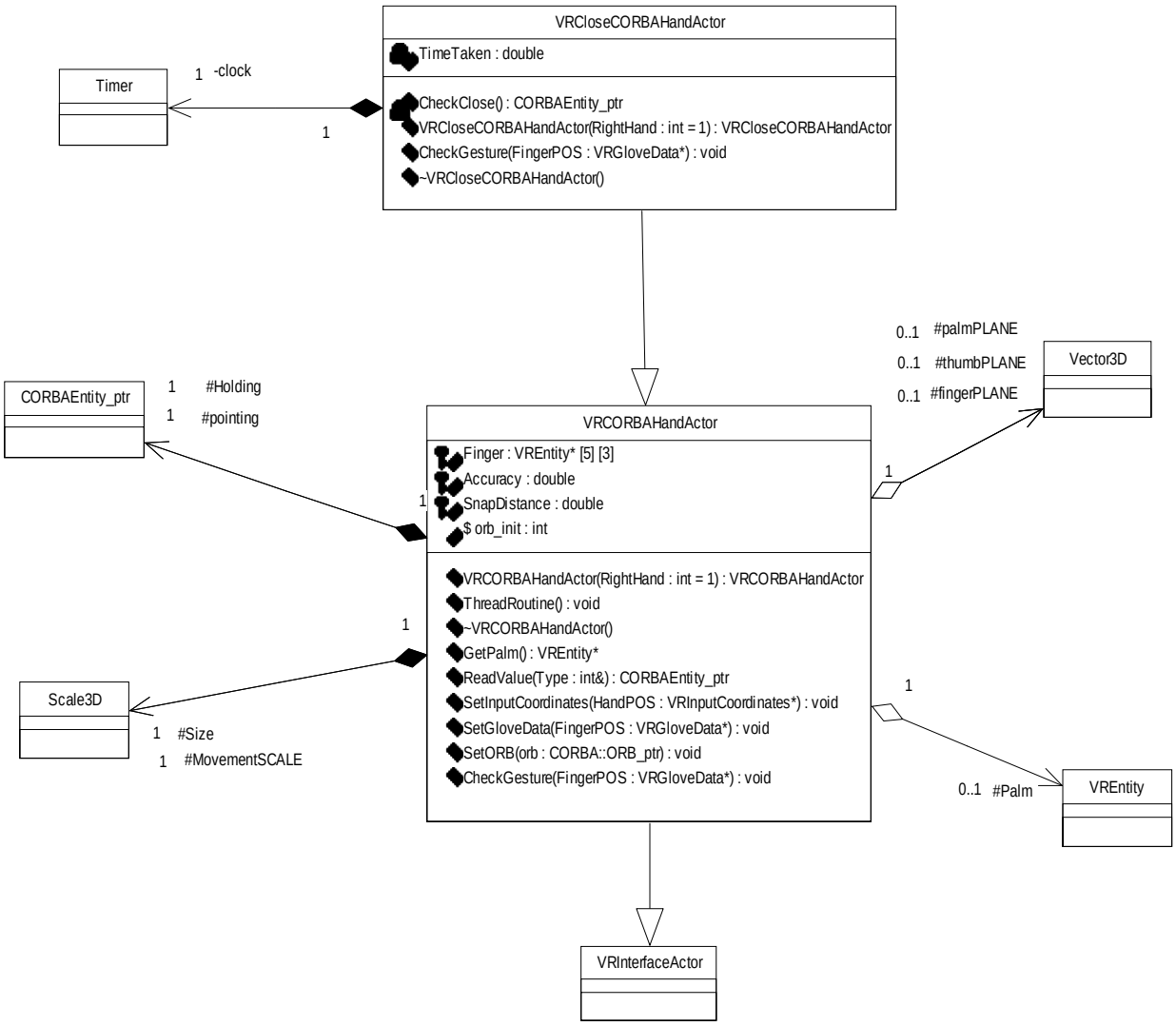


Figure 7 – UML Diagrams for CORBA Hand Actor Classes.

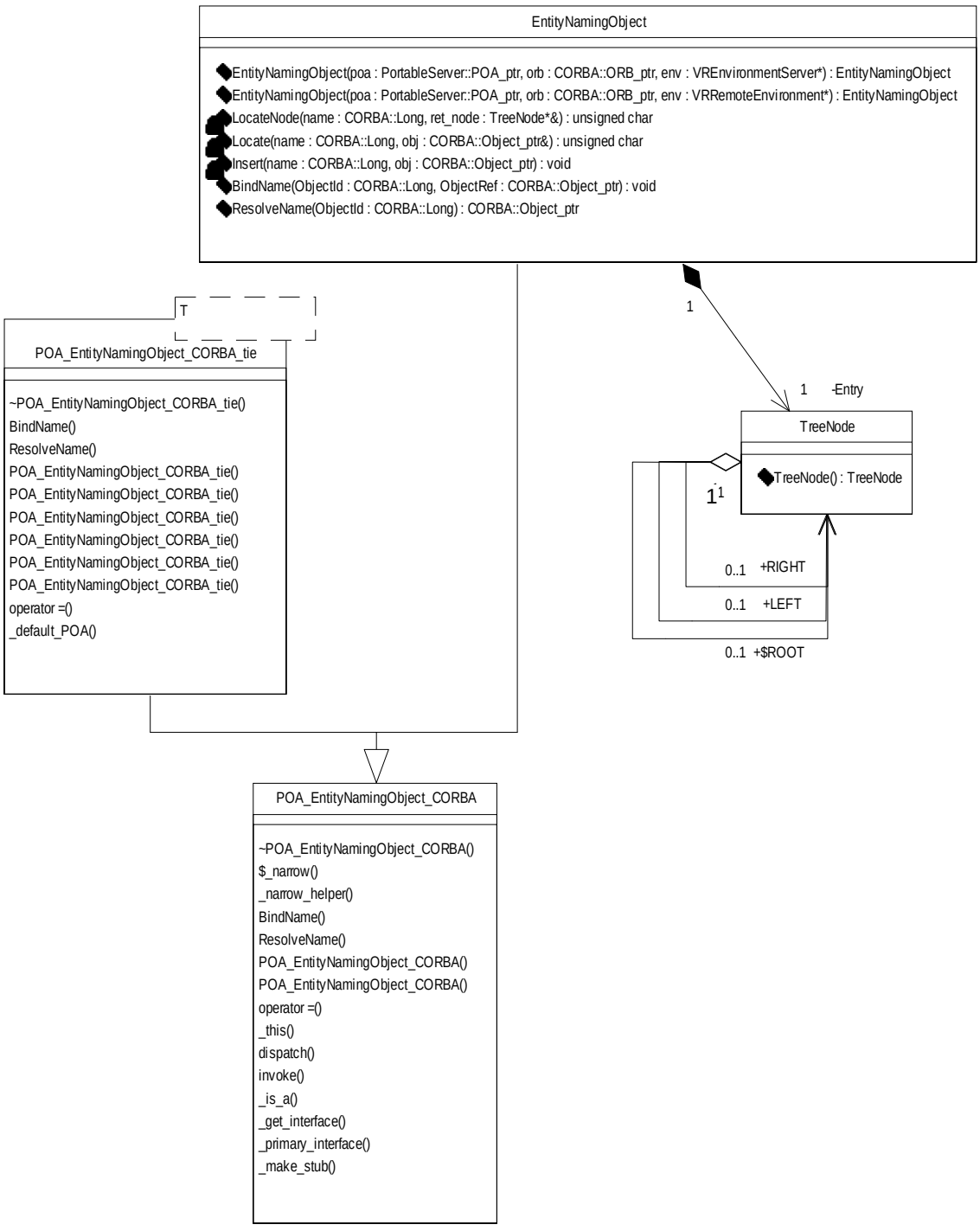


Figure 8 – UML Diagrams for the CORBA Entity Naming Object Classes.

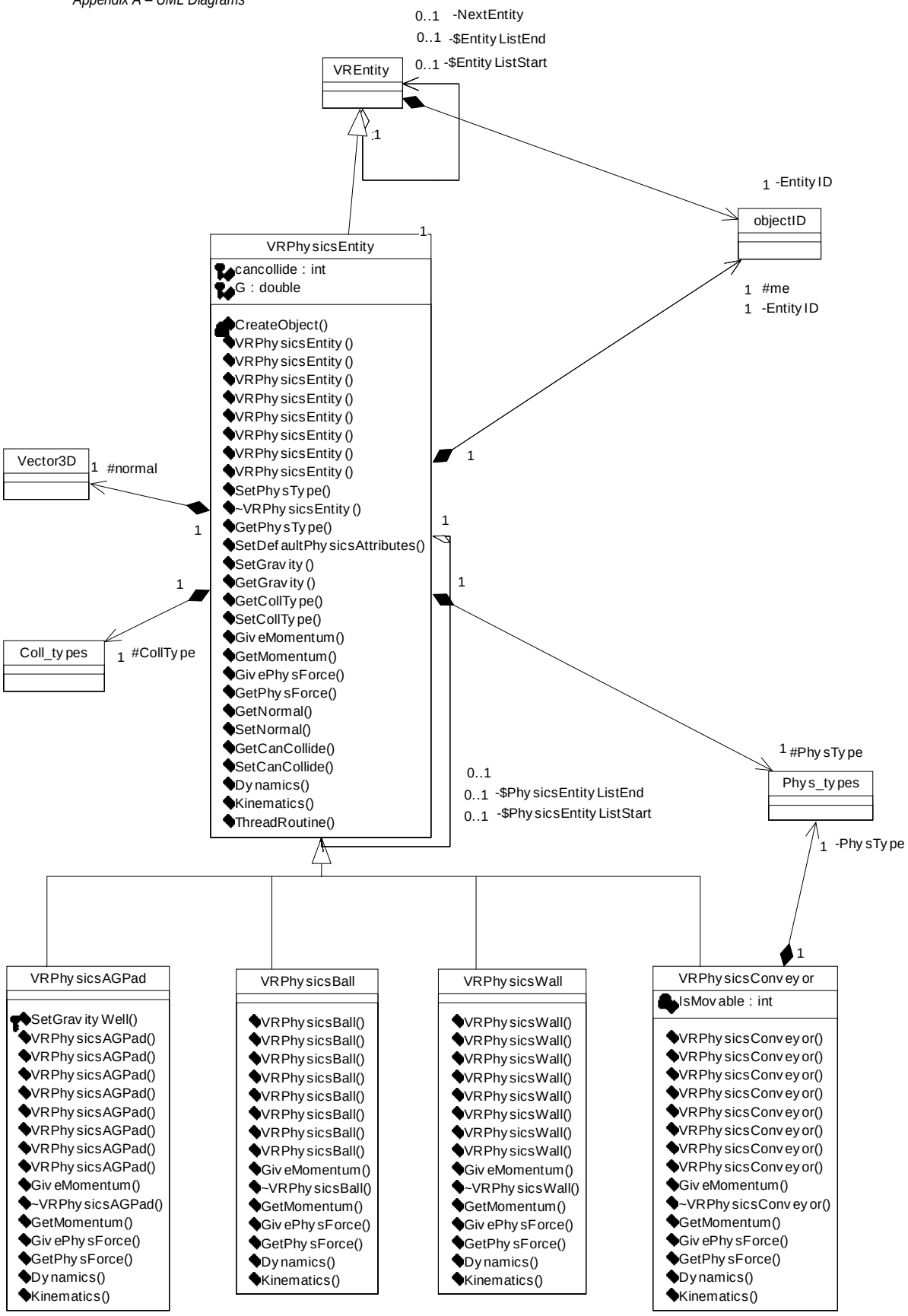


Figure 9 – VRPhysicsEntity UML Diagrams.


Appendix B
"CoRgi Communication
Hierarchy Tutorial"

Communication Hierarchy in CoRgi

Widget and Interaction Component Communication Methods

Currently, there are two types of communication in the CoRgi system - *Interaction Component* (e.g. the `VRHand` or `VRRemoteControl`) and *Widget* (e.g. the `VRMenuWidget`) communication, each of which is handled differently

Interaction Component Communication

An interaction component is able to communicate with objects in the world (including widgets) by sending commands or actions to the objects to execute. The current method for doing this is to get a pointer to the object in question. This pointer may be stored in the program, or received via collision detection, ray casting, etc. Once we have a pointer to the object, we issue the command that we require by calling an action method in the object. The current form of object *Entities* (or objects in the system that have control methods) allows for 5 types of action - *Grab*, Drop, *Point*, *UnPoint* and *Press*. For example, the following code fragment (taken from `VRCloseInteractionActor`) issues a *Press* command to the entity pointed to by the pointer `pointing`:

```
VREntity *pointing;

/* CheckClose : collision detection method */
pointing = CheckClose();
if (pointing!=NULL)
{
  MenuSelect = pointing->Press(Finger[2][1]->Identify());
  if (MenuSelect) GotMenu = 1;
}
```

When issuing the action to the object, the `ObjectID` of the *calling object* is passed. This may or may not be used by the object, but is there if required. The entity replies to the action request with an integer value. The value of this return integer tells the calling method whether the action was carried out or not, and possibly, what value the action returned. A return value of 0 means that the action is not supported by that particular entity (e.g. a *Press* attempt on a `VRGenericEntity` - which only implements *Grab*, *Drop*, *Point* and *UnPoint* - returns a 0). Any positive integer means that the action was successfully carried out. In the case of `VRButtonEntitie`'s, the return value is the `ActionID` for that particular entity.

In order to specify how a particular type of object will react to the various actions, you need to create a `VR*Entity` object which inherits methods from the base class `VREntity` and implements those that

it wishes to override. As an example of this see `VRGenericEntity`, `VRHoldEntity`, and `VRPointEntity`.

Widget Communication

Widgets are simply classes of entity in the world that generate some callback when they receive a valid action from an interaction component in the world. For example, the `VRButtonWidget` generates a callback when it receives a *Press* command. Widgets have associated entities, which define their physical presence in the world and also how they may be interacted with. When creating a widget, a pointer to a *parent* object needs to be passed. The *parent* object must have a `Set*` method which will be called by the entity. For example:

```
void SetInt(Widget* Caller, int Value)
```

This method acts as the *callback* for the widget and is called whenever the widget receives a valid interaction command. An entity can have multiple Set methods, with each method handling a different data type e.g. `SetDouble` or `SetInputCoordinates`. The parameter `Caller` is a pointer to the widget which made the callback, while `Value` is some value indicating what the current state of the widget is or, in the case of a button widget, what action is assigned to that widget. The particular `Set*` method is the parent object must pass the value it receives on to its own parent (if it has one) and perform some action relevant to the value it has received.

When creating menus, buttons, etc. that need their values sent to the main program, we use the `Read*` method with the appropriate data type (e.g. `ReadInt)`, to access their values, since we are unable to pass a pointer to the main program in order to set-up a callback. The `Read*` method should only be used by the main program to get the value of the particular widgets in the world.

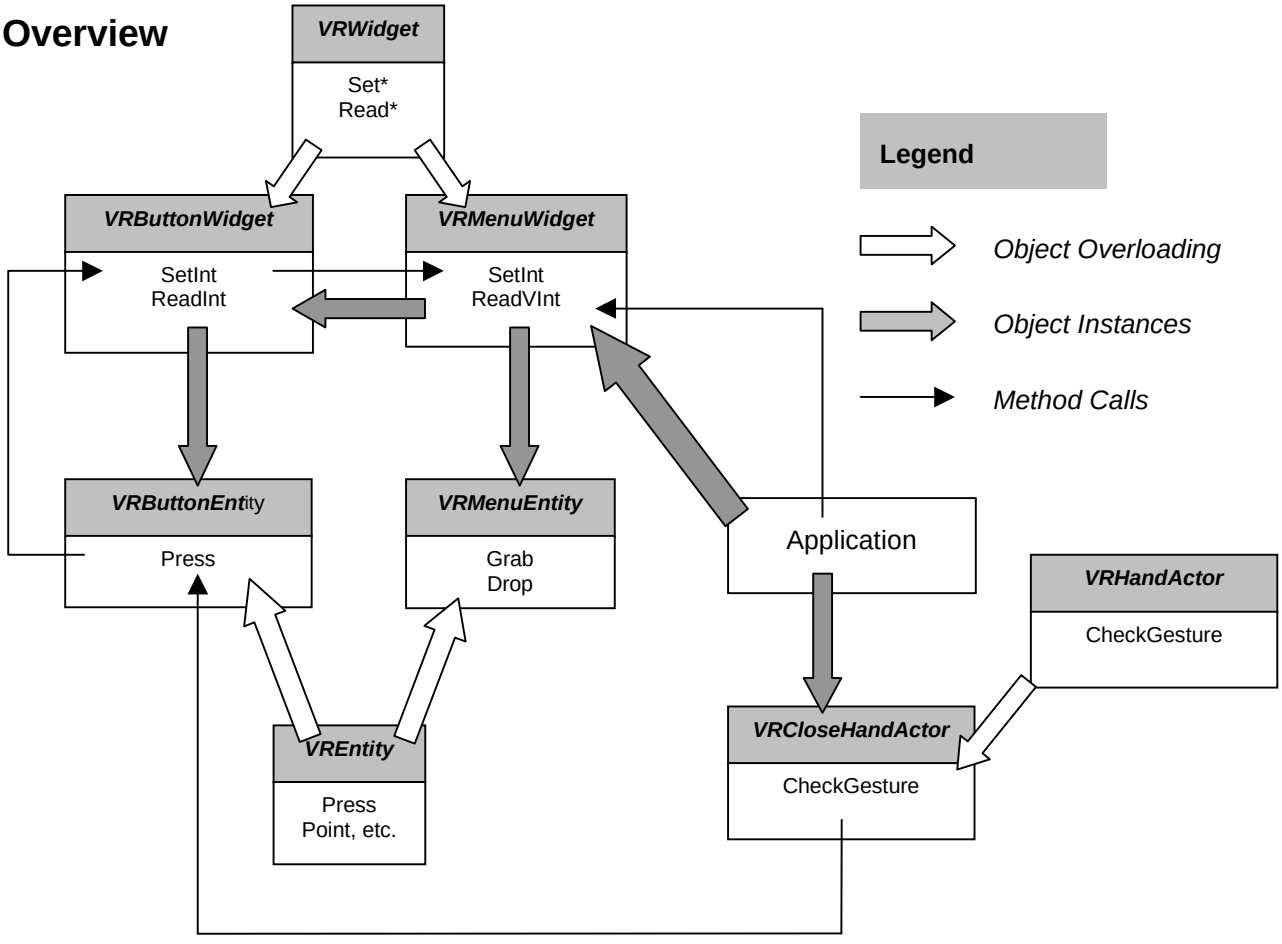
Linking Widgets and Entities

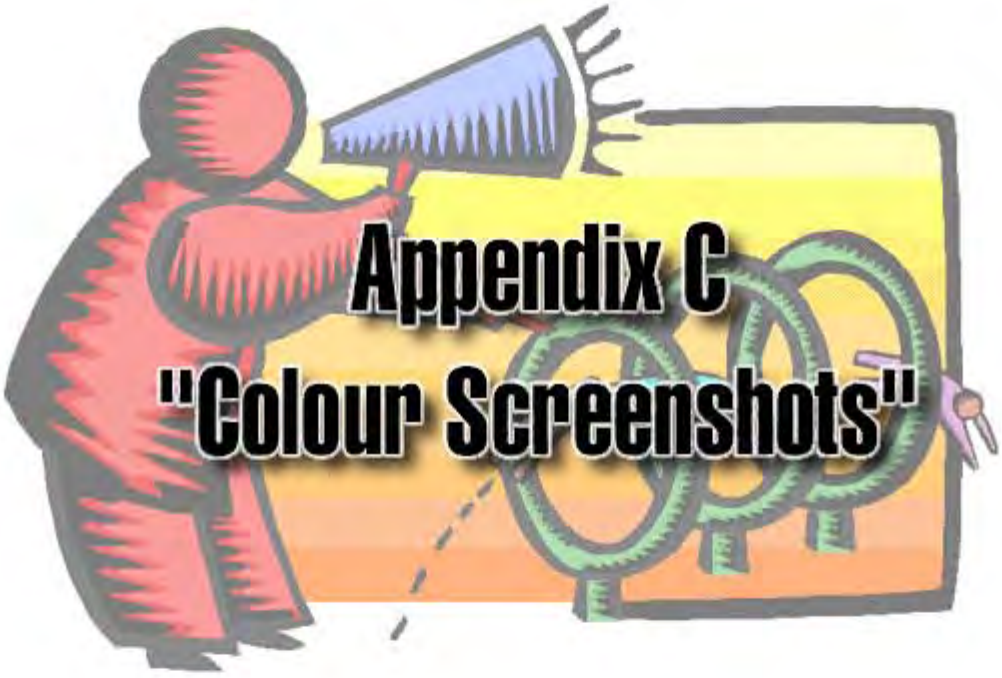
Quite often it is useful to link a widget directly onto some entity. For example, if we had a menu widget with numbers on the buttons corresponding to object sizes, we could link this menu to a particular widget and the values selected from the menu would be immediately transferred to the connected entity to be processed. When linking widgets and entities, be aware of what data type the widget produces and what data type the entity is willing to accept. As an example of this, the menu widget produces integer values i.e. it calls the `SetInt` method in its parent. Entities do not have `SetInt` methods, as it is not obvious how an integer value should affect an entity i.e. should it change its position, size, shape, etc. Instead, entities have methods like `SetSize`, `SetPoint3D`, etc. whose effects are obvious. Thus, when linking a widget to an entity, we need to as a `VRIntermediary`, whose job it is to take in a certain data type and produce another data type, based on some set of rules. As an example of this the `VRIntScaleIntermediary` has a `SetInt` method which the widget calls to pass it some value. In turn, it uses this `integer` value to create a `Scale3D` value, which it

uses to make a `SetScale3D` call in its parent entity. The rules on how to convert input data to output data within the intermediary are encapsulated as methods inside the particular intermediary object. When linking an entity to an intermediary, you are required to choose which of the available methods you wish to use to translate your data. The following piece of code, taken from the `VRWidgetDemo` application, demonstrates the linking of a menu widget with a `VRGenericEntity`:

```
/* Create a new Int-To-Scale intermediary*/
VRIntScaleIntermediary* Int2Scale = new
      VRIntScaleIntermediary()
/* Create a new generic entity */
ConnectedObject = new VRGenericEntity("object_files/king",
      ScratchType, texManager);
/* Create a new menu widget linked to the intermediary*/
VRMenuWidget* Menu = new VRMenuWidget(Int2Scale, "menu1.def",
      texManager);
/* Choose a conversion method from into to scale */
Int2Scale->Register(Menu, "IntToScale");
/* Link the result from the menu to the entity */
Int2Scale->Link(Menu, ConnectedObject);
```

Overview




Appendix C
"Colour Screenshots"

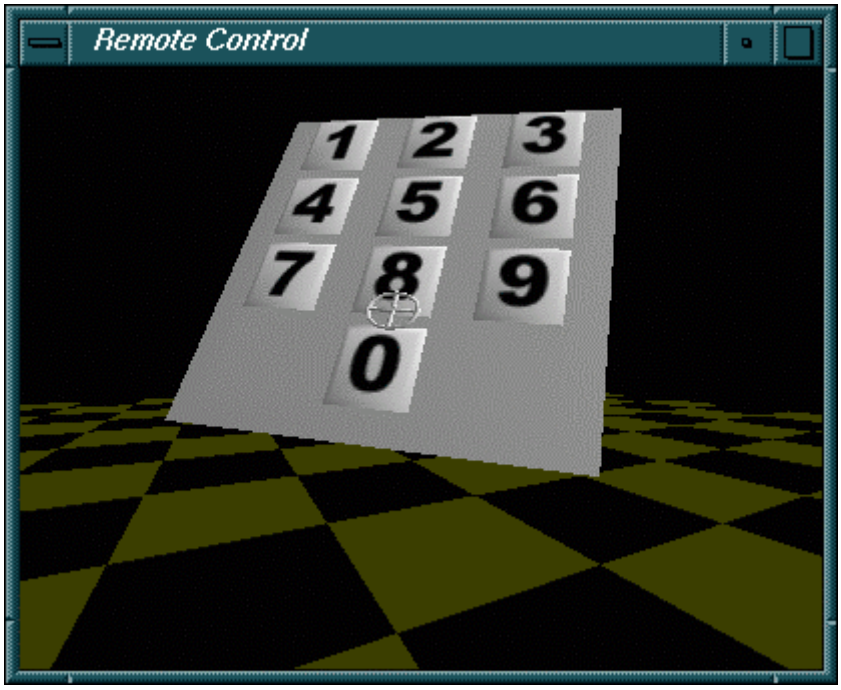


Figure 4 – 1 – The Virtual Keypad.

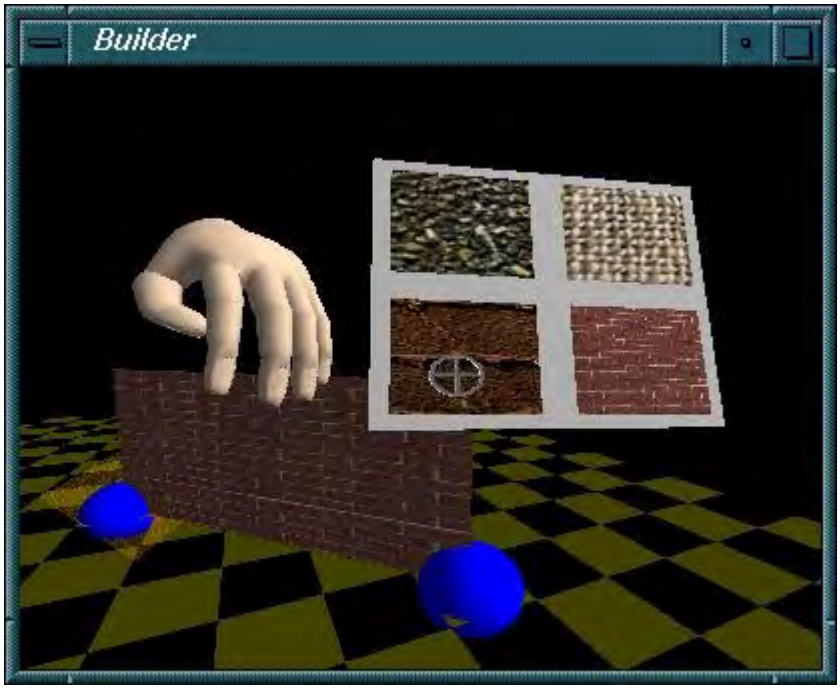


Figure 4 – 2 – The Texture Selector.

Figure 4 – 4 (1) – Screenshots of the VRHandApp.

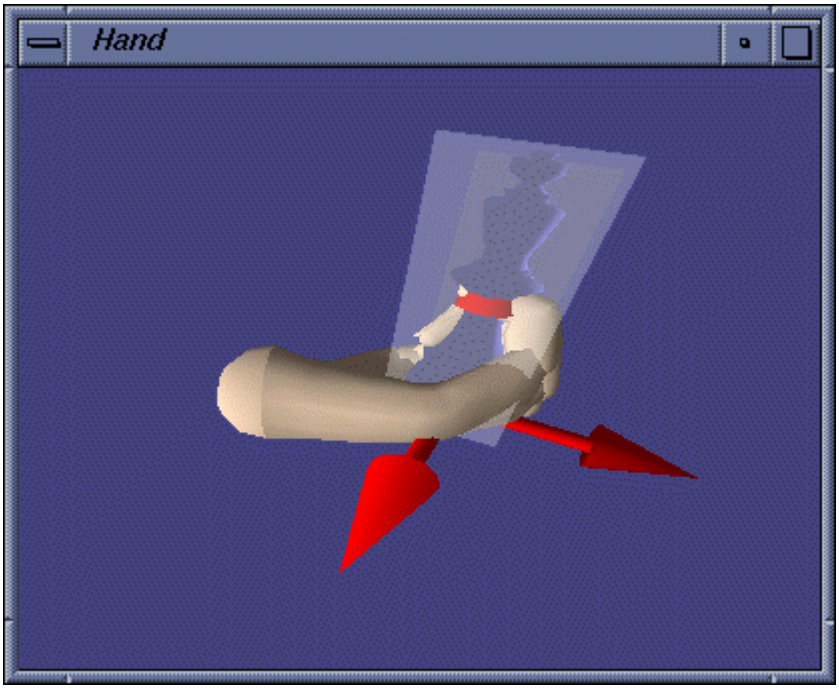


Figure 4 – 4 (2) - Screenshots of the VRHandApp.

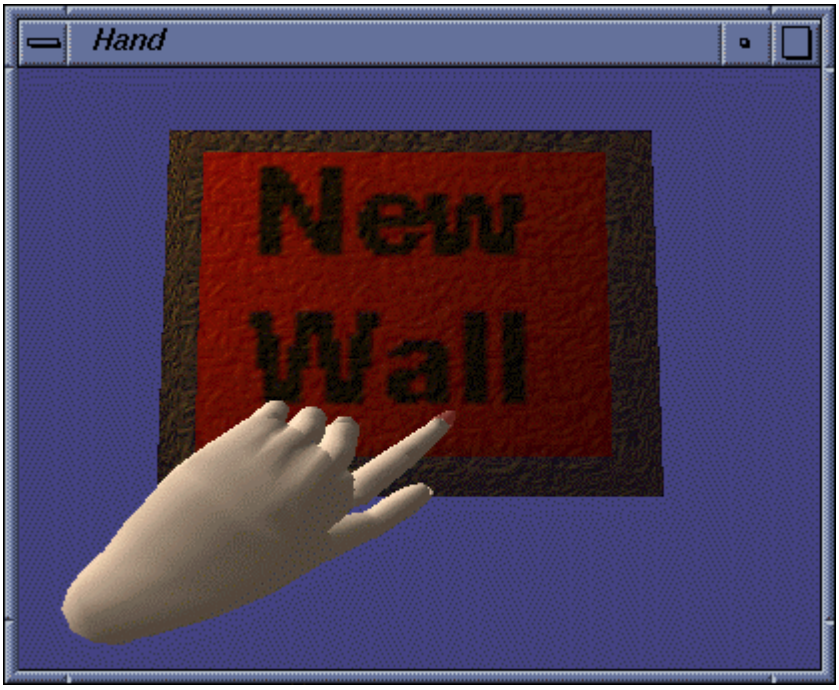


Figure 4 – 4 (3) – Screenshots of the VRHandApp.

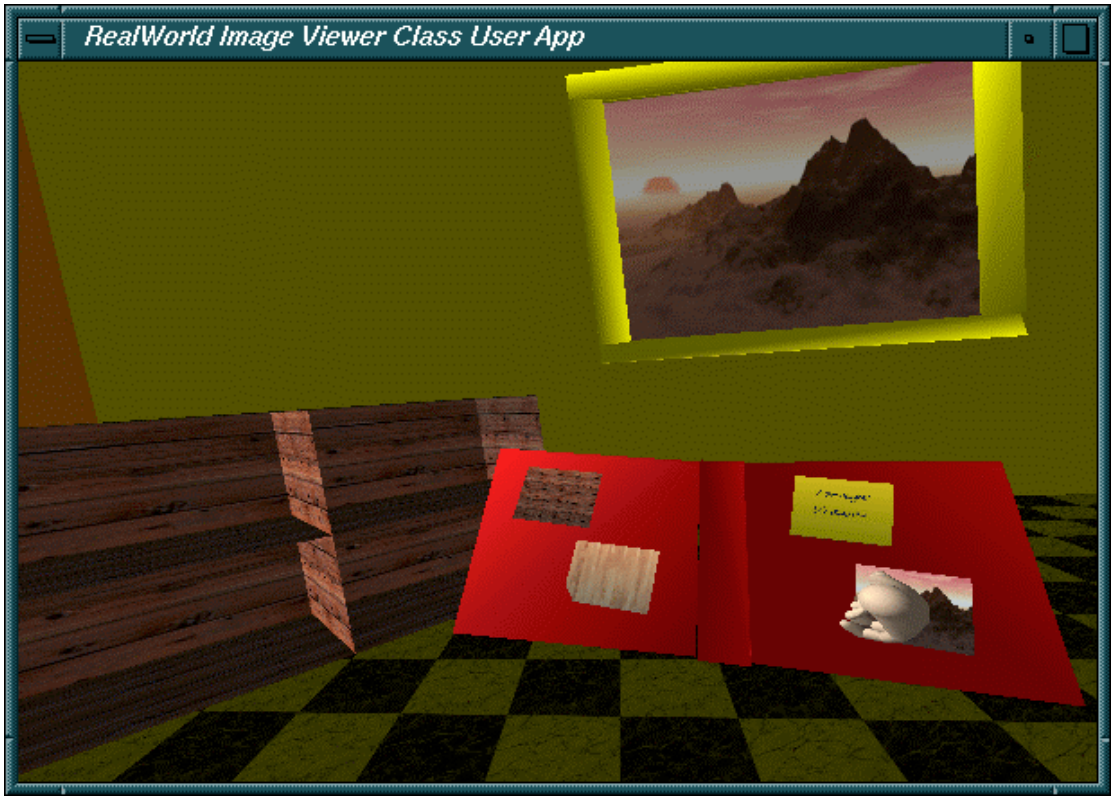


Figure 4 – 6 (1) – The 'Real World' Image Viewer.

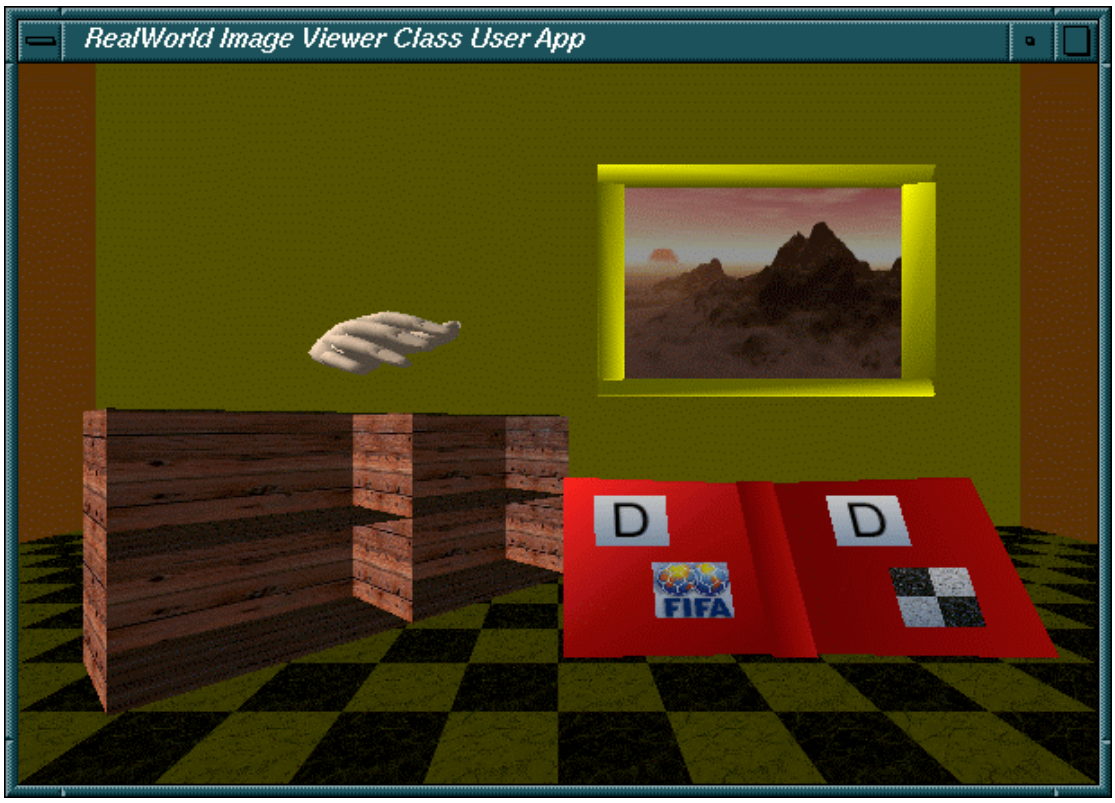


Figure 4 – 6 (2) – The 'Real World' Image Viewer.

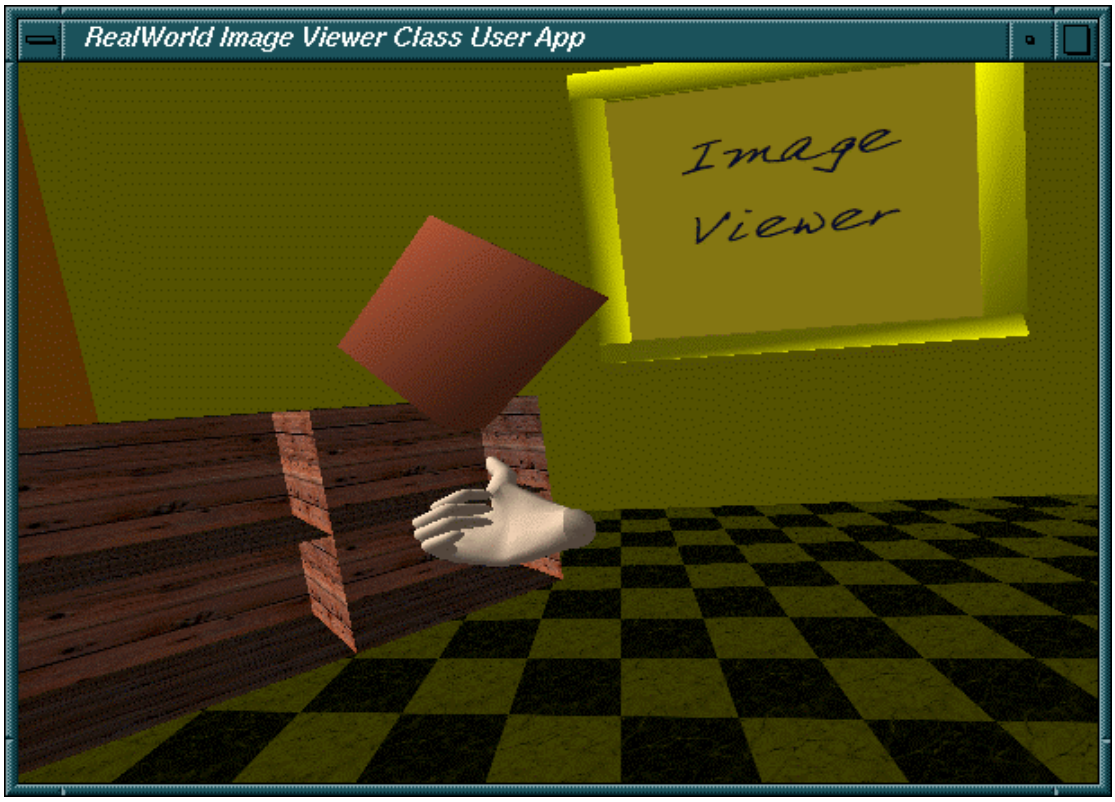


Figure 4 – 6 (3) – The 'Real World' Image Viewer.

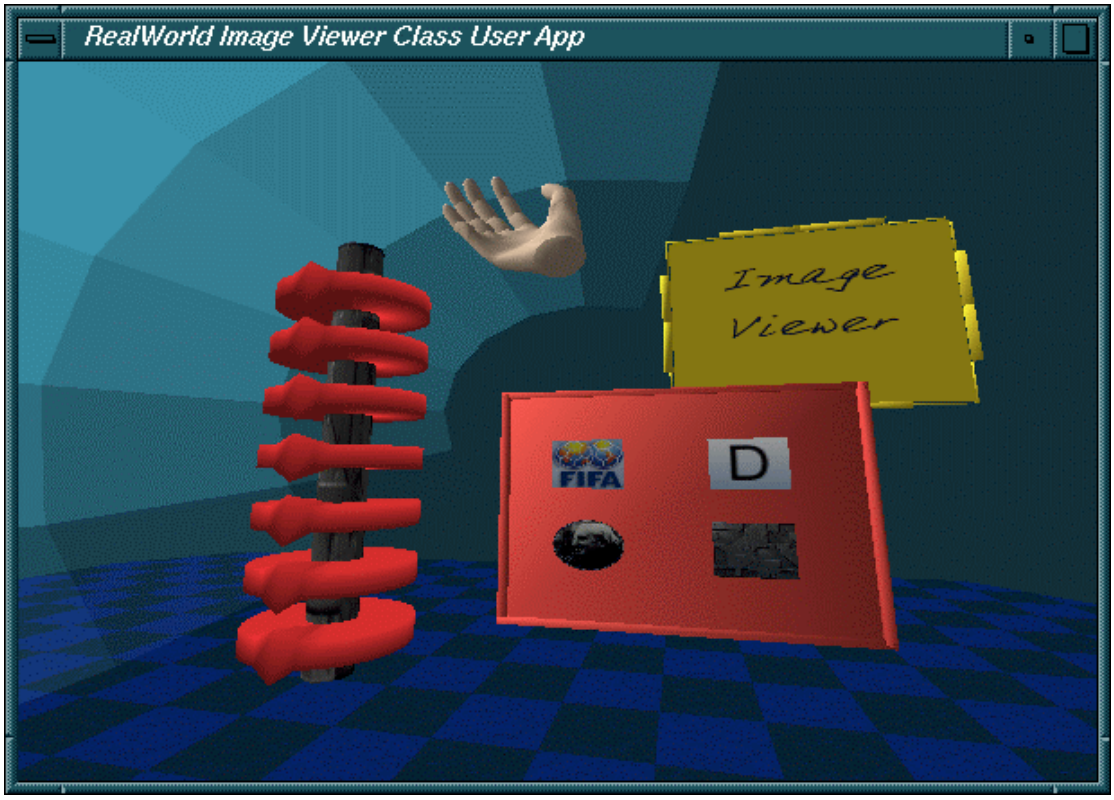


Figure 4 – 7 (1) – The 'Abstract' Image Viewer.

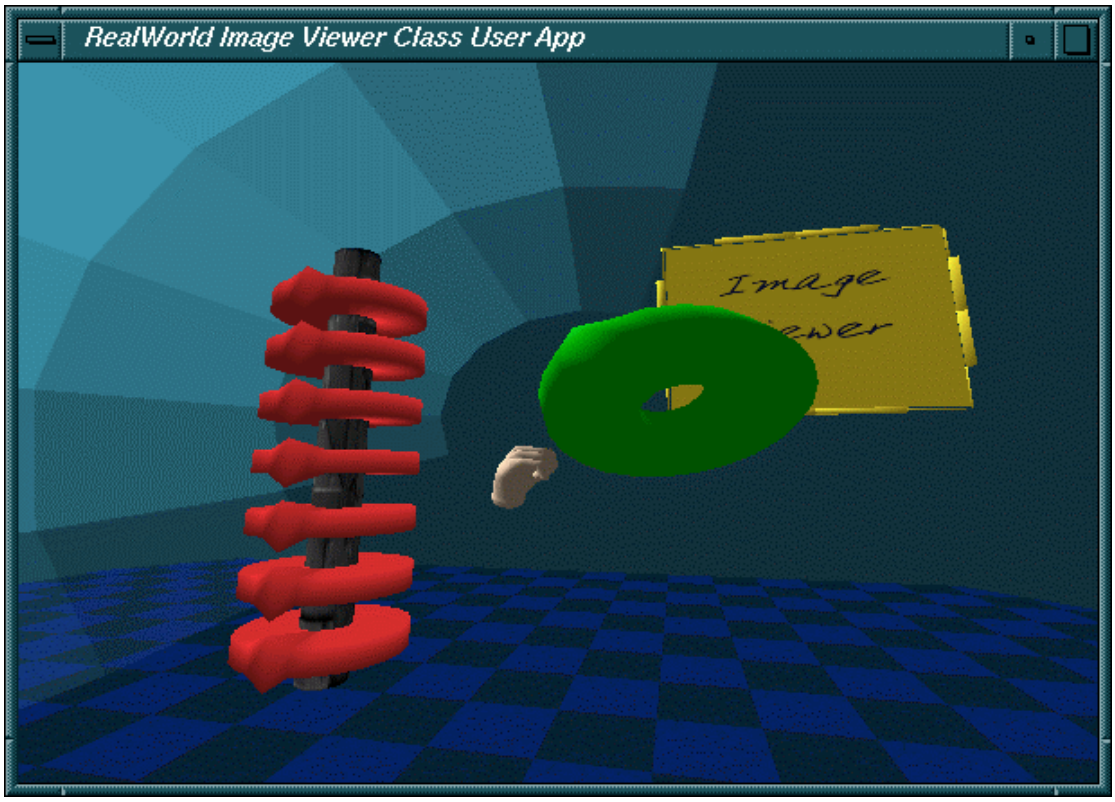


Figure 4 – 7 (2) – The 'Abstract' Image Viewer.

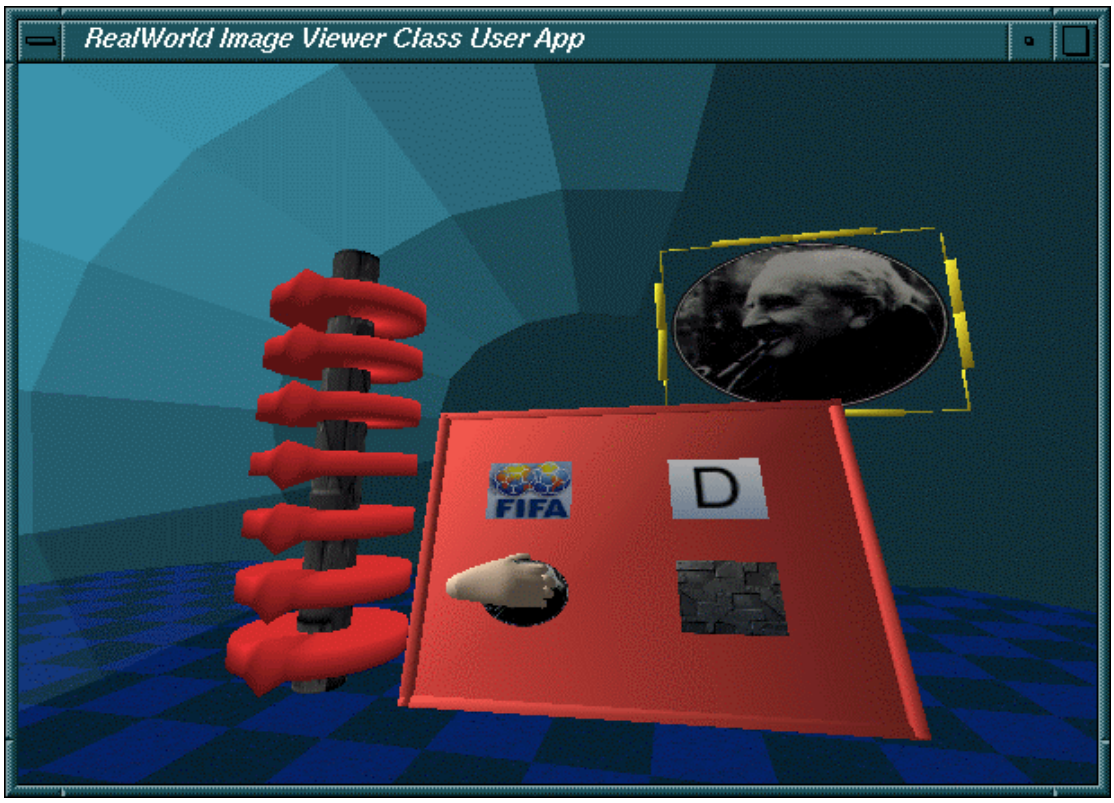


Figure 4 – 7 (3) – The 'Abstract' Image Viewer.

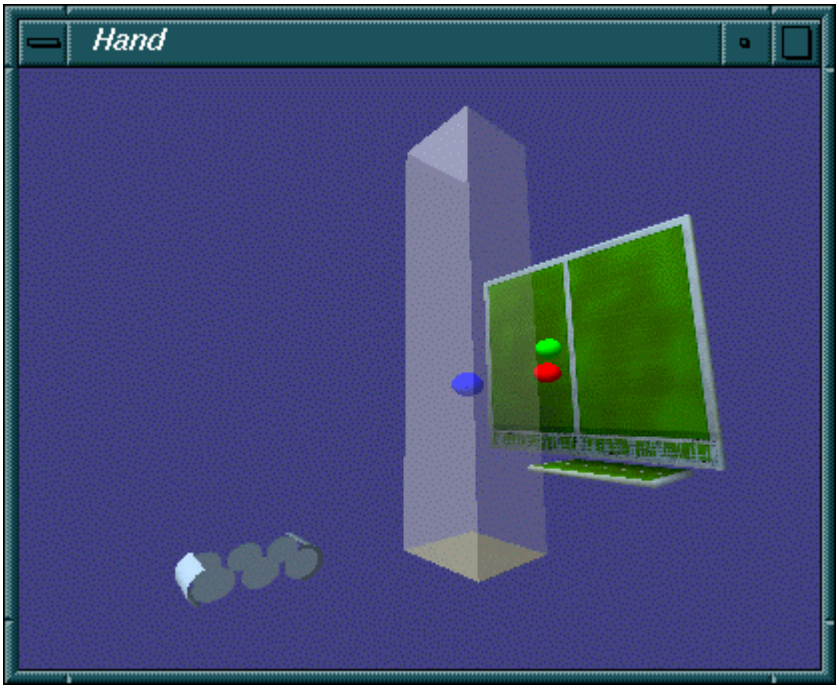


Figure 4 – 8 – The VRPhysicaApp Application.

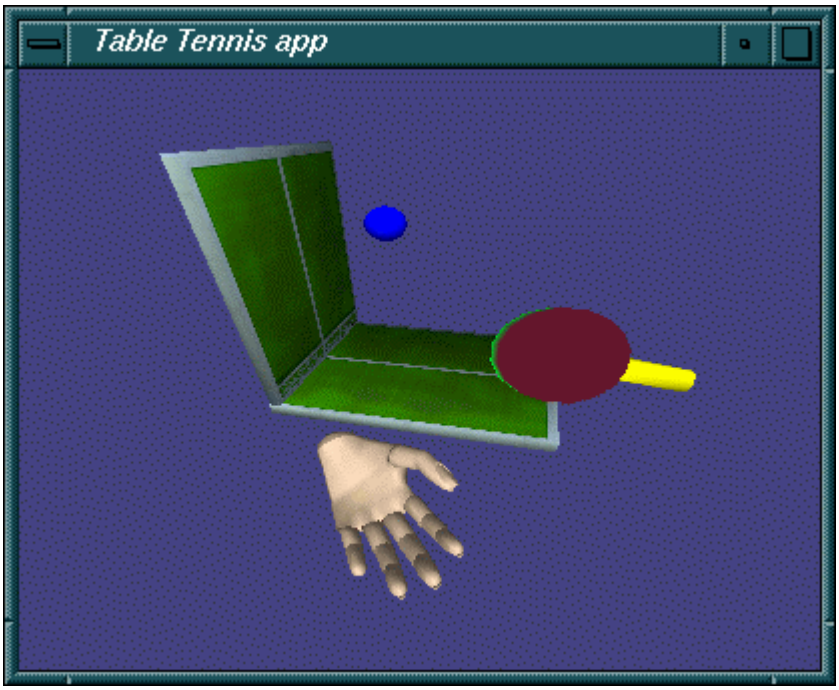


Figure 4 – 9 – The VRTTApp Application.