A Generic Virtual Reality Interaction System and its Extensions Using the Common Object Request Broker Architecture (CORBA)

Michael Rorke, Shaun Bangay

Computer Science Department
Rhodes University
Grahamstown, 6140
South Africa

{mrorke, cssb}@cs.ru.ac.za

ABSTRACT

The paper describes the design and implementation of an immersive Virtual Reality (VR) interaction system. The system aims to provide a flexible mechanism for programmers to implement interaction in their VR applications, making good use of all accepted practices in the field. The paper further describes how the system was extended to a multi-user system using the CORBA middleware layer.

Keywords: immersive virtual reality; interaction; CORBA.

INTRODUCTION

Immersive VR places the user inside the application environment in some way or another. 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 users body (e.g. the hands and head). The information from the trackers is used to generate an environment where the user is 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.

Currently successful virtual reality systems make good use of immersion techniques to enable users to explore virtual worlds. Applications like building walkthroughs, where the user is able to examine the contents of a virtual world, make good use of this technology. On the other hand, applications that allow the user to interact with their surroundings in VR have not been as successful. The reasons for this lack of success when attempting to allow the user to interact with a virtual environment range from lack of haptic feedback (or 'touch') in VR environments to inaccuracies in tracking hardware [1]. All these factors tend to lower the sense of realism that the 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 the problems associated with this field have been proposed [2,3,4]. This research has produced various interaction techniques which, coupled with current advances in hardware, have paved the way for a usable VR interaction system.

The CoRgi interaction system builds on this research to create a generic VR interaction system with the capability to implement current and future VR interaction techniques.

THE CORGI INTERACTION SYSTEM

Rhodes University is currently involved in various different forms of VR research, aimed at creating a usable VR programming system. The current version of the system (called CoRgi) is the basis around which the interaction system is built. CoRgi currently runs under IRIX and Linux and provides a framework for easily creating VR applications. It uses OpenGL and has support for a variety of input and output devices, like HMDs and magnetic trackers.

The CoRgi interaction system uses a simplified data flow model. A data flow model describes 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 data flow model used in the CoRgi interaction system is loosely based on the Virtual Environment Dialogue Architecture (VEDA) [5,6], which is similar to the data flow model used in the Virtual Environment Modelling Language (VRML) 2.0 standard [6]. Additionally, the CoRgi interaction system uses the ideas behind the MR VR toolkit [7] to define the interfaces between the various parts of the system.

As an aid to understanding the system it is presented along with an example application. The example application consists of a simple Table Tennis game where the user controls a bat, which they must use to hit a ball. In the distributed version of this system (explained in the CORBA section) the user could compete against other users. The overall interaction system can be broken up into six main parts as detailed in Figure 1. Each of the six parts operates as follows:

- **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

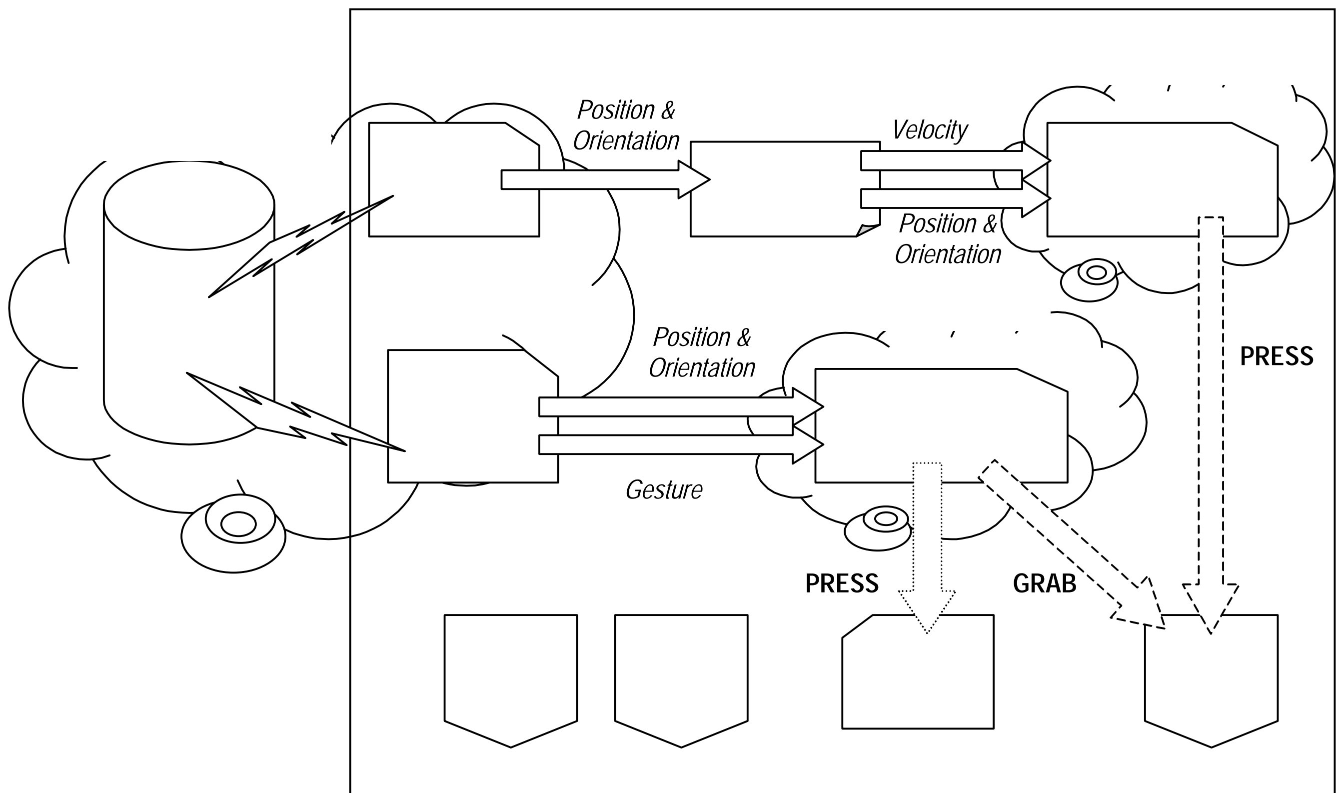

Figure 1 – The Table Tennis Interaction Example

system component is also responsible for producing the output that the user sees.

In the Table Tennis game example, the system component is responsible for implementing the physical systems modelling required to make the objects move in a realistic manner. Specifically, the system component implements collision detection between the ball, table and bat. It also implements gravity and any rules which are required for the game.

- **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.

  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 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.

  The table tennis 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 an input actor in the application. The input actor then supplies data to the application relating to the position and orientation of the user's hand and thus the bat. There are also additional input actors for the hand and head mounted display.

- **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 and a data glove). The CoRgi interaction system currently uses several different interaction components, each implementing some form of recognised immersive VR interaction technique [2,3,4].

  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. The other interaction component is a

virtual hand, which the user can use to retrieve balls that have fallen off the table.

- **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. Intermediaries receive data from some source (or multiple sources) process the data in some way and pass the data 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.

  The Table Tennis game example uses an intermediary to calculate the velocity at which the bat is moving. The polhemus tracker input actor 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. 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 1).

- **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, etc.

  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. The various interaction components, already implemented in the system, use methods such as collision detection to identify the entity, and 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.

  Relating this to the Table Tennis game example, there are several entities in the system - the ball, the table, the net and the floor. The attributes of the ball, for example, include its shape, colour and elasticity (how much it reacts to collisions with other objects e.g. the bat or table). The ball knows how to respond to *grab, drop* and *press* 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. The *press* command is used to relay to the ball that it has collided with the bat and should change direction/velocity appropriately.

- **Widgets:** Widgets in the CoRgi system are simply specialised forms of the basic entity. They take the form of controls, which the user is 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). This data is then sent along a data flow network (usually containing at least one intermediary) to its destination.

  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 system is connected directly to the widget and due to the simple nature of the widget, no intermediaries are required.

Each of the components in the system may be implemented separately and independently of the others, provided that they each conform to the data interface definition which describes what data they must make available to other components, and what data they can expect to acquire from other components.

DISTRIBUTING THE SYSTEM

The basic CoRgi system supports distributed VR in the form of a client-server type architecture. An application can create a central server environment, to which other client environments can connect. The distributed environment is currently implemented using standard TCP/IP and UDP. Since the interaction system was implemented separately from the rest of the system (as proposed in [7]), it is not necessary to distribute both in the same way.

In order to distribute the basic VR system, it was necessary to efficiently disseminate large amounts of data to all the clients. Thus, the system was distributed at the lowest possible level (the TCP network layer) in order to make it as efficient as possible. The interaction system was 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 CORBA middleware layer, to allow the interaction system to be spread over different platforms in a network. The CORBA middleware system is a standard (set by the Object Management Group (OMG)) for distributing object-oriented applications over a network. CORBA provides a standard implementation whereby an application can call methods in another, separate application, over a network [8]. CORBA handles all the low-level details of these object communications (e.g. byte marshalling from little endian to big endian systems and vice versa). Once a reference to a CORBA object has been found, this reference can simply be used (similarly to a pointer in C/C++) to call methods in the object. Whether the object is local, in a different process or thread, or on a different machine entirely, is irrelevant. CORBA implements platform independence by requiring that the interfaces to the objects be specified in a platform independent language (called Interface Definition Language (IDL)) which is then translated into the language of your choice using a program supplied with the CORBA ORB package. The CORBA ORB (or *Object Request Broker*) is the backbone of the CORBA communication system. Applications each talk directly to a local ORB (on the same machine) this ORB then communicates with the ORB on the destination machine, which in turn passes the data on to the destination object. Thus, any data passing between objects in a CORBA network passes through at least two ORBs. The link between the ORBs is where network implementation details are handled, the user is thus freed from having to worry about these details. The CoRgi-CORBA interaction system currently uses the MICO ORB, but this ORB can be easily replaced with one from another vendor. Different ORBs offer different trade-offs between factors like speed and cost.

The CoRgi-CORBA interaction system consists of various entity component objects, residing on the different client machines. Each new entity must register its CORBA *object reference* (an address that allows the object to be accessed via the CORBA middleware layer) with a central object (the *Entity Naming Object*) which resides on the central server. The Entity Naming Object acts as a central point, linking the object references of the various entities in the system, to the various objects representations displayed in the virtual environment. Thus, the user is able to select an object (using the interaction component) and issue a command to it. The object reference for the selected object is retrieved from the Entity Naming Object and the command passed on to the relevant entity via the CORBA layer.

The Entity Naming Object is a simplified version of the CORBA Naming Service. The full implementation of the CORBA Naming Service includes a lot of functionality that was not required by our system. This extra functionality complicates the use of the Naming Service and also makes it less efficient. It was therefore decided to use the simple Entity Naming Object and not the full CORBA Naming Service to store the entity object references.

IMPLEMENTATION DETAILS

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.

The Entity Naming Object is the main difference between the distributed and single user versions of the system. In the single user system, the list of what entities are available is stored in a simple linked-list, which is used when deciding which entity should receive a particular interaction command. The

distributed version required a more complicated system consisting of a central CORBA object (the Entity Naming Object, created by the server VR application) with which each entity was required to *register* in order to be available to receive interaction commands. The object reference of the Entity Naming Object is disseminated, along with other VR system information, through standard TCP/IP network channels, when the client application first connects with the server.

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.

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.

CONCLUSION

The paper presents a generic VR interaction system, which allows programmers to quickly and easily create VR applications, using currently accepted practices for achieving interaction in VR. The system is also built in such a way as to allow new interaction techniques to be easily implemented an assimilated into the system.

The system also shows that, with correct design procedures, CORBA can be successfully (and easily) applied to the field of VR. The extra overhead created by using a middleware layer as opposed to using low level network programming can be offset by careful design of the object interfaces, in order to minimise data flow between them.

REFERENCES

[1] 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.

[2] M. Mine, **Virtual Environment Interaction Techniques**. *UNC Chapel Hill Computer Science Technical Report TR95-018*

[3] 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.*

[4] 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.

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

[6] A. Steed, **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*

[7] 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

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