The Participant System:

Providing the Interface in Virtual Reality

by

Max Minkoff

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in Engineering

University of Washington

1993

Approved by _____
Chairperson of Supervisory Committee

College of Engineering (Inter-Engineering)

July 13, 1993

TABLE OF CONTENTS

FIGURES

INTRODUCTION

Although Virtual Reality (VR) is a complex technology that is being intensively studied, it still seems to be poorly understood.   For example, almost every approach used today treats the task of building each new virtual world as an entirely new endeavor. Very few approaches allow world computation facilities to include programs that have not been written specifically for use in a VR system. But the new capability available within Virtual Reality is the <u>interface</u>, not the computation. For over ten years User Interface Management Systems (UIMS) have been tackling a similar problem of providing a standard interface for a variety of applications as part of an effort to simplify the creation of new application interfaces and reduce their cost. VR technology will not gain wide acceptance until a VR system is created that, like a UIMS, specifically supports the interface and provides for the integration of any processing appropriate for use in a virtual world.

This thesis describes an idealized software system that provides the virtual interface and facilitates integration into a fully-featured VR system. By way of organization, I review first the most common VR systems used today to give a full context for this new paradigm. The Participant System (PS) concept is then introduced, and the required qualities of a PS reviewed. Once this background is established, I propose a model for a Participant System that meets all of these requirements. "Mercury", the Human Interface Technology Laboratory's (HITLab) implementation of the Participant System Model, is then illustrated and evaluated in light of the given requirements. Finally, I discuss issues that may arise in the use of a Participant System and areas of future research.

VR Systems

There are several systems that have been developed over the last five years which help create a virtual environment (Sense8 1993) (Appino, et. al. 1992) (Blanchard, et. al. 1990) (Grimsdale 1992) (Grimsdale and Atkin 1992) (Shaw, et. al. 1992). Though each system has its own way of providing the VR experience, there are several basic components that are required for the creation of any virtual environment. These elements are shown in Figure 1 and discussed below.
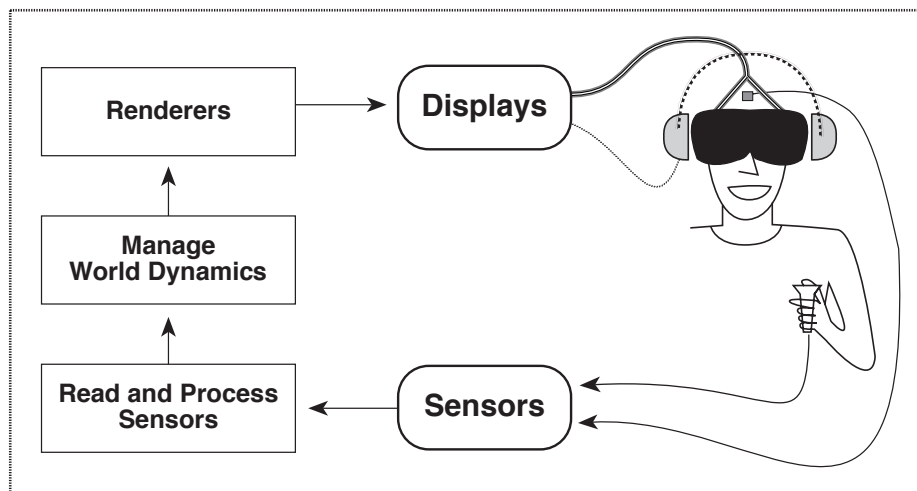
Figure 1:
Generic Virtual Reality System
(arrows represent data flows)

Displays are hardware devices used to present the environment to the user. The display that is most commonly associated with Virtual Reality is the head-mounted display (HMD). HMD's such as the VPL Eyephones™ have two video screens mounted in a helmet or mask used to project a stereographic view to the user. The computer screen, with or without 3D capabilities, is another video display. Headphones and loudspeakers are examples of audio displays. Research is also being done to create and improve haptic,

kinesthetic, and various other types of displays to be included in the VR interface (Lederman and Taylor 1987) (Minsky, et. al. 1990) (Monkman 1992).

Renderers are used to compute the sensations to be presented in the displays. Using the description and positions of objects in the world and the position of the user's body, renderers create the appropriate sensory experience and drive the associated hardware.

Sensors are input devices used to determine the user's physical behaviors. Typically, position sensors are used to determine the orientation and relative position of the user's head and hand. Other sensors include the VPL Dataglove™ which measures the amount of flex in the users hand joints, the "wand", a 6D joystick which might report whether or not buttons on it are pressed and where it is pointing, the standard mouse, and 6D input devices such as the Spaceball.

World dynamics provide the environment with which the user interacts. Almost all systems provide some sort of manipulation of objects in the world as opposed to only enabling the user to fly through a static model.

The system elements above must be **integrated** into a smooth, fast-flowing process to facilitate the virtual environment while introducing as little latency between the sensors and displays as possible (Krieg 1993) (Frank, et. al. 1988).

Example - VEOS

VEOS, the Virtual Environment Operating Shell, is a virtual environment system that the HITLab has been developing for the past three years. It is based on a model of distributed processing with a distributed database across heterogeneous computing platforms. VEOS itself provides the database and the internal communication. FERN (Fractal Entity Relativity Node) has been written as part of a higher level language to facilitate access to VEOS to manage the database and accommodate general interaction.

Each object in the environment is called an entity. Each entity has any number of attributes (e.g. graphical description, spacial location, aural description, etc.) to describe it while maintaining its own processing.

VEOS's basic function is to provide integration between all aspects of the virtual environment. This is accomplished through its distributed nature and its open source code. Additional program libraries may be compiled with VEOS to lend functionality to any specific VEOS program. This is the case for renderers such as the HITLab's (graphical) Imager and Sound Renderer. These renderers output to the appropriate displays directly. SensorLib, a software library used to interface with peripheral devices, is added in the same way. Once these libraries are all compiled, they are then accessed by specific entity programming. Entity code is then used to access the renderers and sensors, process world dynamics, and communicate the resulting information to each other using the distributed database.

UIMS

Creating an application system that must read a variety of sensors, process the resulting data and other application dynamics, and display the results is not a new problem. In the early 1980s, as graphical user interfaces became popular, application developers were faced with the problem of how to provide efficient interface and application processing while using the same interface from application to application without having to start essentially from scratch each time. At a Computer Graphics workshop held in Seattle in 1982, the field of User Interface Management Systems (UIMS) was born (Olsen 1992). At the workshop, the following major conclusions were made:

1. The user interface implementation should be separated from the application code and be implemented using specialized programming tools.

2. Interactive applications should have external rather than internal control. The user interface and its supporting software would control the flow of

the application (external control) rather than the application code itself (internal control).

3. Tools should be developed to assist user interface developers who are not necessarily programmers.

4. User interfaces should be specified for user interface design rather than programmed in a general purpose language.

When applied to the task of building a virtual world, the above recommendations have an exciting impact:

1. The first point suggests that the interface - the displays, sensors, and renderers - be separated out from the world dynamics. This provides benefits well beyond the ease of building the application, as will be seen below.

2. External control (control by the user and the interface rather than control by the application) is a concept that has become pervasive throughout software development with the advent of modeless programming. Virtual environments are intended to allow the highest level of this type of user control.

3. The world builder should be freed as completely as possible from having to build the interface system, and enabled to specify the elements of the interface as easily as possible.

4. Interface elements, and similarly, virtual world entities, are much simpler to describe as compared to typical program elements. A protocol and specification method that specifically facilitate worldbuilding tasks should be created instead of requiring the use of a more general purpose, and potentially less relevant, programming language.

Separating the interface from the application and providing an easy method for virtual world specification are concepts that have lead directly to the development of the Participant System Model. In general, UIMS research and development has produced a variety of systems that support many types of hardware platforms and peripherals. Current UIMS allow the application builder to easily specify and interact with windows, sliders, and other "widgets" found in common graphical user interfaces. While such features are likely to find their way into Participant Systems of the future, the current PS model is limited to providing the interface itself through the <u>integration</u> of the interface components and does not consider the elements of the interface themselves. For that reason, further

UIMS advances and their relevance to Virtual Reality in general and Participant Systems specifically are left for future investigation. This will be discussed further under "Future Research."

Of the five basic virtual environment components discussed above, displays, renderers, sensors, and sensor processing remain extremely similar among all VR systems. The methods these systems use to integrate these components with each other and world dynamics vary greatly and often yield inadequate results.

RB2™

VPL's RB2™ (Reality Built for Two) Virtual Reality System employs a high-end Macintosh computer and two Silicon Graphics Iris workstations (one for each eye), in addition to peripherals such as the EyePhones™ HMD and the Dataglove™ hand flex sensor (Blanchard, et al. 1990). Head and hand positions are tracked by Polhemus™ sensors integrated in the other peripherals and control units (see Figure 2).
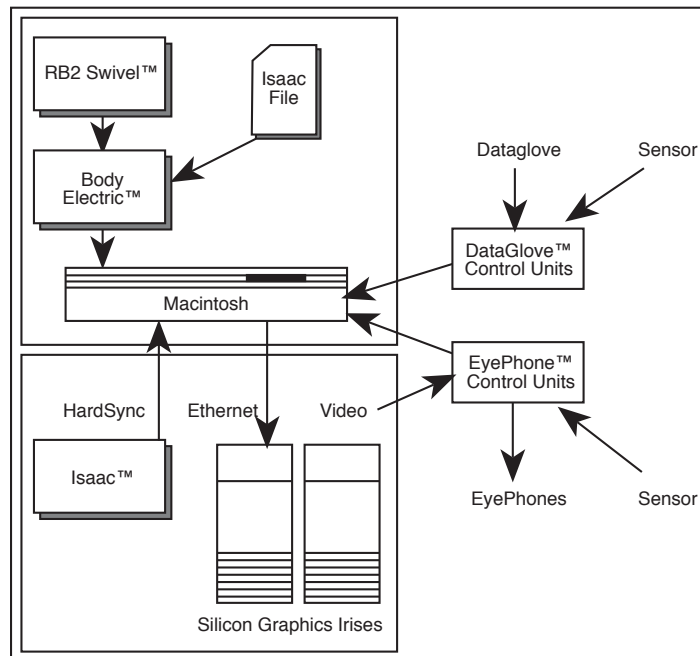
Figure 2:
Reality Built for Two (Blanchard, et al. 1990)

In this system, data from the sensors are read and processed by the Macintosh. At the same time, world dynamics based on programming in VPL's Body Electric™ software

and dynamics built into the RB2 Swivel™ graphical model file are computed. The set of world and participant updates are sent to the Isaac™ renderer on the Irises, and the results are displayed in the EyePhones™.

       Though VPL's Body Electric™ software offers a simple graphical interface for configuring virtual world dynamics, the Macintosh presents a severe bottleneck because its single processor must provide all but the graphics processing. The frame rate experienced by the participant is therefore directly related to the complexity of the world dynamics.

WorldToolkit™

       Sense8's WorldToolkit™ (WTK) is a library of over 400 C language functions to be used to create a virtual world (Sense8 1993). The library includes functions for reading sensors, changing attributes of the environment and world objects, calculating world dynamics, and rendering scenes. WTK libraries are available for SGI, Sun, and enhanced PC platforms.

       The drawback of the WTK system, beyond the complexity involved in tying the many functions together, is that that it uses a single loop for reading the sensors, calculating world dynamics, then rendering the scene as shown in Figure 3. This single loop means that a great deal of latency will be added while calculating world
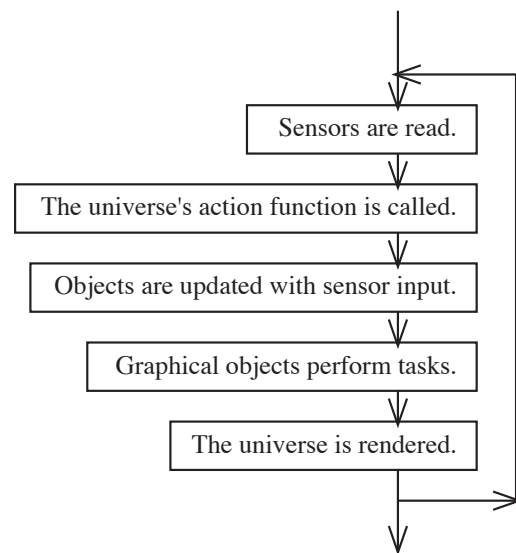
Figure 3:
WorldToolkit's™ Simulation Loop
(Sense8 1993)

dynamics between reading the sensors and rendering the scene.

SIMNET

SIMNET (SIMulation NETwork) is the distributed simulation developed by the Defense Advanced Research Projects Agency (DARPA) through most of the 1980s (McDonough 1992). By 1990, it had 250 full-crew Armored Vehicle and Aircraft simulators in nine sites in the United States and Germany, all interconnected by local area and long-haul networks. SIMNET allows participants to practice all the tasks required during normal combat, either with or against each other.

Each simulator placed the participant in a mockup of a vehicle's interior, for instance a tank. Views out of viewports and appropriate sounds were computed by Image and Sound Generators (renderers) from data read from the controls and instruments (sensors), a model of the terrain, and a world
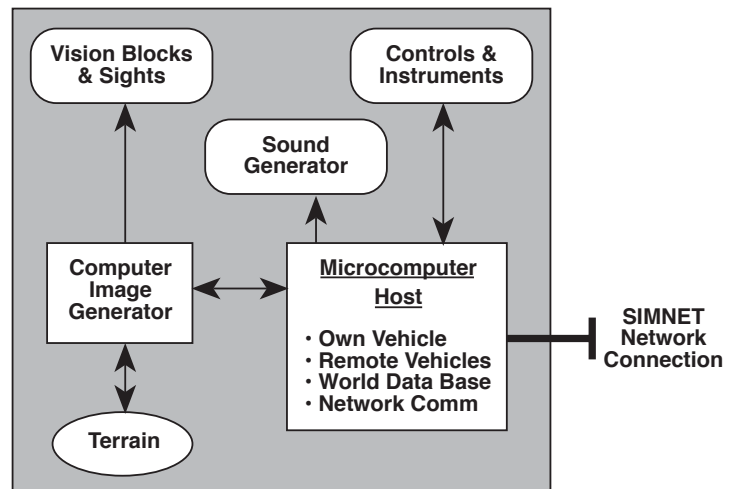
Figure 4:
SIMNET Model (McDonough 1990)

database of the other vehicles in the simulation. Most of the processing occurred locally with updates passed along as necessary across the network. In addition to the simulators themselves, Management, Command and Control (MCC) stations allowed for the command of support services such as artillery batteries and refueling facilities. Semi-Automated Forces (SAFOR) were also created to include a large number of participating forces commanded from a single console without having to include as many human participants in the simulation.

SIMNET is the first large-scale multi-participant virtual environment ever created. It has also been highly successful. It is the first of the systems discussed here to begin to separate and differentiate the local database and the world database. Though each simulator's host must process all local aspects of the environment, the designers have begun to recognize that certain aspects of the experience require more constant attention than others in order to create a reasonable simulation. These concepts reflect the same decisions made in the Participant System model.

dVS

Division, Ltd., a British firm, offers a line of VR systems which include parallel computational hardware and software. Division's hardware platforms include a set of transputers, each of which run dVS, their parallel processing VR operating system (dVS... 1992). dVS uses a metaphor that includes actors and a director. Actors are individual program modules that provide the processing for a single element of the environment. Common actors include 'viz', the module that reads the head sensor and renders the graphics, and 'dGlove', which manages the interface with the Dataglove™. Users may also write their own actors to participate in creating the virtual environment. Each actor outputs and accepts only the data which is appropriate to its function. Data and processing flow is controlled by the director.

dVS is the first of several systems that not only use distributed, parallel processing, but also manage the flow of information. It allows for reusable code, and
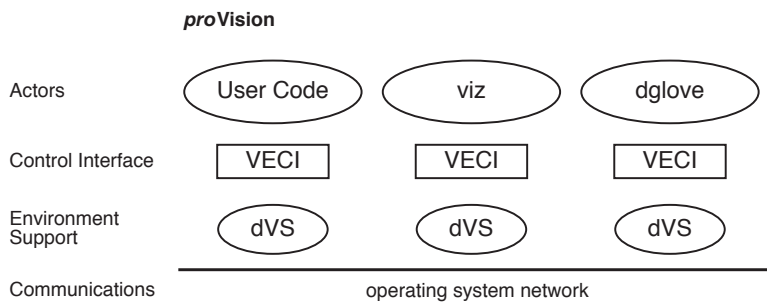
Figure 5:
Software Architecture of ProVision (dVS... 1992)

provides a standard subset of programs (in this case actorss) that manage the processes that

provide the virtual interface. Still these processes must share the same data path as all

world dynamics, and therefore may suffer undue throughput delays. While reading the

head sensor is built into 'viz' (the graphics renderer), reading the hand sensor and placing

the hand in the virtual world is accomplished by a completely separate process (dglove).

As a consequence, the hand interaction is updated only as fast as general world objects.

VEOS

VEOS was described as an example VR system above.
Its design is based on the model shown in Figure 6. Data from
sensors flows into the system at the upper right. It is
interpreted by the physiological model of the human participant
and is passed on to the tools and communicated from one part
of the system to another before being sent out through the
virtual body to the renderers and finally displayed to the
participant. Simultaneously, processing of world dynamics
may take place on the same processors, and information
regarding world entities passed through the same channels as
body data (Bricken 1990).

Figure 6:
The VEOS Model
(Bricken 1990)

Like the Division system, VEOS worlds can suffer a
large amount of unnecessary latency as data passes around the
system, hampered by sharing processing power and bandwidth with world dynamics.

Furthermore, when building a virtual world, the programmer must pay close attention to

how this data gets passed around the system to ensure that is accomplished in the most

efficient manner possible. These problems should not be the concern of the world builder.

VUE

Like VEOS, the Veridical User Environment (VUE) system developed at IBM's Thomas J. Watson Research Center (Appino, et al. 1992) tackles the demands of the VR system by separating and distributing the necessary elements across multiple processors. The VUE system depends on three components: device servers, application
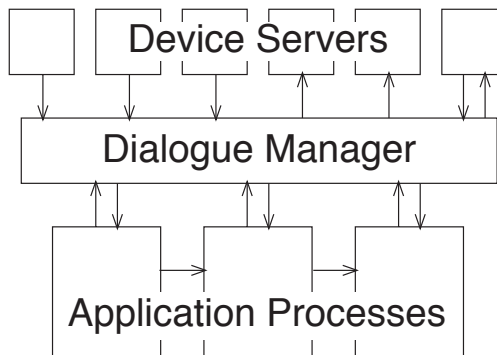
Figure 7:
VUE: The Veridical User Environment
(Appino, et al. 1992)

processes, and the dialogue manager. Device servers are processes that interface with sensors and renderers to provide the user interface. Application processes add the world dynamics. The dialogue manager integrates these processes and servers based on a set of rules to coordinate the flow of data through the system (see Figure 7).

To better understand these system components, an example VUE application, The Vortex World, is presented (Appino, et al. 1992). This application allows the user to interact with data from a fluid dynamics simulation. The device servers and application processors used in the example are presented in Figure 8.

The application processors in this world are the batch simulation and the surface extractor. These processors provide the dynamics of the simulation. The device servers are all involved in creating the virtual reality interface, including the sliders. The virtual sliders provide standard
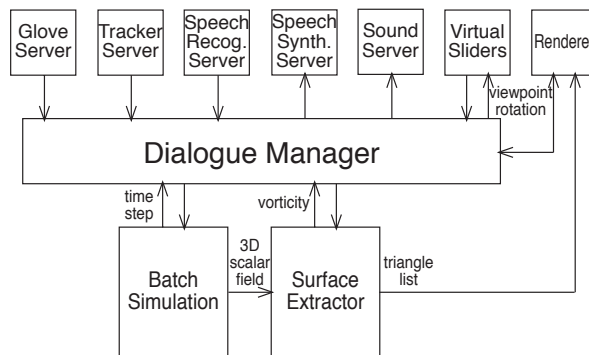
Figure 8:
VUE: The Vortex World
(Appino, et al. 1992)

interface elements that can be setup to interact with other processes, such as the application processors. In this application, five sliders allow the user to modify the simulation time step, surface vorticity, and surface color (red, green, and blue).

The creators of the VUE system cite three major advantages of their architecture in providing a virtual reality experience. The first is the increased computational capacity available through a distributed processing system, like both VEOS and dVS. The second advantage comes from the relatively low inter-process bandwidth requirement attained through partitioning the main tasks involved in creating the virtual environment. Again, both VEOS and dVS provide similar functionality by limiting communications to only that which is necessary. The third major advantage is the ability to overlap device and application processing times by executing them in parallel on multiple processors. Again, the same advantage is found in VEOS and dVS. Unfortunately, the same disadvantages cited for dVS and VEOS apply to the VUE system as well.

MR Toolkit

The MR Toolkit, based on the Decoupled Simulation Model for Virtual Reality Systems, is another VR system that uses distributed processessing (Shaw, et al. 1992) (see Figure 9). As such, it has the same advantages and disadvantages found in VEOS and VUE. The difference, though, is that this model specifically decouples the participant's input/output loop from the processing of world dynamics. Special attention is paid to allowing each component of the system to asynchronously process as rapidly as possible.
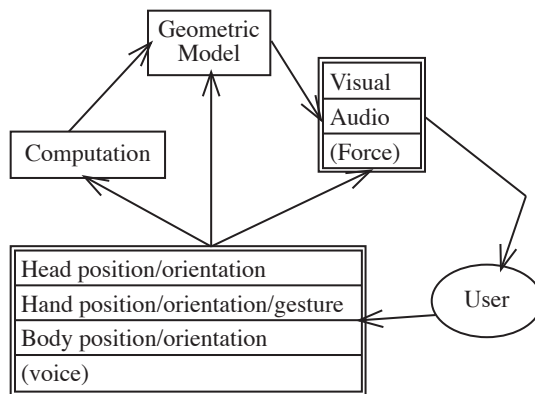
Figure 9:
The Decoupled Simulation Model
(Shaw, et al. 1992)

This means that each part of the system will make the newest data available at any given time. It also facilitates tight registration between renderers.

Though this model does allow the worldbuilder to concentrate solely on the computation of world dynamics, further development of this concept leads to even lower latency, more flexibility, and lower overall bandwidth within the system.

Summary

No matter what the application and no matter what the virtual environment is like, every virtual reality system must provide the virtual interface. When integrating the elements of the interface, there are characteristics that are common to almost all of these systems:

    1) the ability to create high-level reusable elements to be integrated into the full VR system.

    2) partitioning of the system elements for more efficient operation.

    3) parallel and in some cases distributed operation of these elements.

The most common system elements that are created and reused are those involved in the management of the VR interface. In this way, every system in one way or another has implemented a Participant System - a reusable set of program elements used to create the VR interface. But because these systems don't recognize that providing the interface is a task that should be separated from the rest of the required processing and linked as tightly as possible, the interface is doomed to operate only as fast as the rest of the system. My work has been to unify these elements into a single optimized program and provide for a method of integrating it with a VR database and processing system such as VEOS to create a high-speed, flexible, and robust Virtual Reality system.

THE PARTICIPANT SYSTEM CONCEPT

Virtual reality systems may be considered to consist of an application database of world dynamics and a 3D interface to that database. The world dynamics are usually unique to each virtual world, but the mechanics necessary for creating the interface remain constant. This
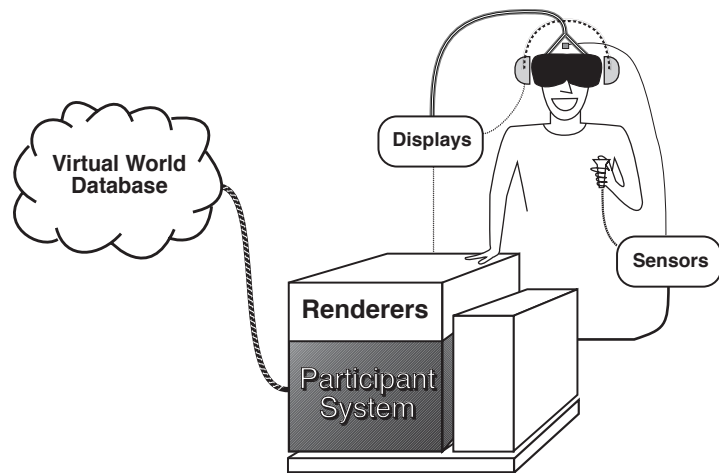
Figure 10:
The Participant System

separation can be exploited to improve not only the way virtual worlds are built, but also to enhance the quality of the participant's experience in virtual reality. In terms of the VEOS model, reading sensors, providing the semantics for data read from them, managing the virtual body, and writing to the renderers are all handled by one component – the Participant System. The environmental entities and the participant's interaction with them are processed in the Virtual World Database and communicated to the Participant System through a standard protocol (See Figure 10).

The motivation for this project comes from two years of virtual world development using both VPL's Body Electric and the HITLab's VEOS. To make virtual worlds easy to build, the low-level program elements that are common to most worlds must be provided by the system itself, freeing the world builder to concentrate on tasks specific to the world being built.

Initially, a set of VEOS programs were created to provide a single common sensor and renderer interface for virtual worlds built using VEOS. This was the first example of a VEOS Participant System. Experience with these programs and their continued success when integrated into a variety of virtual worlds prompted the development of "Mercury," a single high-speed program for accomplishing these tasks. Figure 11 shows the Participant System within the original VEOS model. Whereas before (Figure 6) a set of programs were created to read the sensors, write to the renderers, understand the participant's physical and virtual bodies, and interact with world dynamics and tools, now a single software component exists – the Participant System.

Figure 11:
The PS within the VEOS model

The Participant System integrates all sensors and displays into a single software entity which can then be connected to any system capable of issuing the proper commands to display the world dynamics. For example, processing sensor inputs to create virtual world positions of the head and hand are managed entirely within the Participant System, and therefore beyond the scope of the world builder's concern. Other benefits will be examined below.

REQUIREMENTS

The ideal VR Participant System would give the world builder the kind of flexibility available in systems like VEOS, and include the following performance requirements:

High Speed/Low Latency

High Speed and Low Latency are probably the most important characteristics of any VR system. The speed of a VR system refers to the number of display frames of a particular modality, usually in the visual mode, it can produce in a second. Display update rate is important for several reasons: continuity of displayed motion (BBN, 1992), improved performance (Harris and Parrish, 1992) (Swartz, et al., 1992), and limited latency (Furness, 1992, 1986) (BBN, 1992). It has been found that VR systems must support a display update rate of at least 50 Hz. An update rate as low as 15 Hz may be acceptable for lower-end systems, at least when using current technology, but to provide a high-quality experience, this requirement cannot be ignored (Furness, 1992, 1986).

Latency (or "lag") is the time difference between when the participant moves and when the system displays the appropriate reaction. Studies show that humans can detect a lag greater than 5 msec when it comes to rotational head movement (Furness, 1992). Lags that are longer than 30 msec have been found to cause motion sickness and interfere with operator performance (Frank, et al. 1988). Though other features of a VR system are important, a system that does not provide an experience with both an acceptable frame rate and low latency will not be viable for any serious applications.

There are two fundamental issues that most affect speed and latency requirements. The first concerns the sensor values most sensitive to performance and the second, the elements that comprise total system lag. When a visual display is used in a virtual environment, the most important sensor value is the instantaneous orientation of the participant's head. Though it is also important to provide a smooth update of the rest

participant's body, the orientation of the head will have the greatest impact on the rendering of the virtual image. Translational movements of the body and/or head do not cause the entire display scene to change since an optical flow field with a vanishing point is observed. But rotational movement of the head requires each pixel in the display scene to change in order to appear to be space-stabilized. Rotational movements of the head up to 50-100 degrees per second are typical. Low frame rates and latencies cause the space-stabilized display to appear to "ratchet" or oscillate about a fixed point in space when the user rotates his or her head. The amplitude or subtended angle of these oscillations is a function of the rotational velocity and sum of the update rate and throughput latencies (Furness 1992).

There are several system components that contribute to the time delay between when sensors are read and the appropriate scenes are rendered. Only one of these is under the direct control of the PS (Krieg 1992). The first system delay occurs within the sensor itself and is called Acquisition or Computation Time. Every sensor device has its own internal time lag; for example, a mechanical position tracker has almost zero lag whereas the Polhemus 3Space Isotrak® has over 120 msec delay (Meyer, et al. 1992). Once the sensor has determined the value, it must be communicated to the host processor. This is the Transmit or I/O Time. Depending on the communications method, the format of the data being transmitted, and other factors such as flow control, transmit time may add another 50 msec or more to total system lag. The third step is the processing of the data – the Processing Time. Depending on how the data is going to be interpreted and used, additional time may be added in this step. Finally, the processed sensor data is sent to the renderer so that the current scene may be created. A variety of renderer and display components may then contribute to the Display Time, but that level of detail will not be considered here. Figure 12 represents these time lag components.
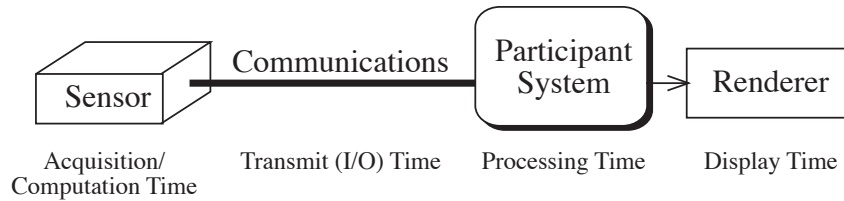
Figure 12:
Components of System Latency

We can see from this analysis that the display update rate is a function of the rendering software and hardware and the scene complexity. Although the PS may contribute to latency as will be discussed below, the overall speed of the environment cannot be determined within the Participant System.

Registration

As a corollary to providing low latency between sensors and displays, multiple modality displays must themselves have little or no latency between them. [It is not acceptable for a visual event to happen at one time and the corresponding sound, for instance, to occur several seconds later.] The ideal system facilitates coordination between visual, acoustic, and other renderers.

Flexibility

While some of the latency and update issues can be solved by hard-coding the features that are common to most virtual worlds, there are exceptions to every rule. The ideal system must not sacrifice flexibility when providing this common functionality. These features must be both easy to use and easy to configure for special cases.

Extensibility

The main function of the Participant System is to move data from the sensors and the application system to the renderers. The form that these data take are independent of the PS. As new sensors and renderers are added into the system, additional processing within

the PS will accommodate the various types of data that the sensors provide and the renderers expect. Additionally, certain types of processing will be specific to the PS and not necessarily to the Application itself, such as the processing of a physiological model of the body. This processing should be provided by the PS.

Lastly, certain types of processing must occur within the PS to provide a smooth experience, such as moving through the world. If such tasks were left to the Application System, the participant's moving view would only be updated as fast as the application communicates to the PS, not at the higher internal speed of the PS itself.

Low Bandwidth

There is a specific data set that must be passed between the participant system and the virtual world database. Bandwidth measures the capacity of the communications path needed to provide this connection in bits per second (bps). It is important to identify and communicate these data using as little bandwidth as possible. In a remote participant virtual world, this data set will cross the remote access link.

Independence

It must be clear that the Participant System is software. By concentrating on the software aspects of the system, a versatile and long-lived system will result. To provide such versatility and longevity, the software must remain independent from all of the following components:

Hardware Platform

A variety of hardware platforms are currently capable of creating a Virtual Reality interface. New platforms are being announced on a regular basis. Furthermore, different applications and associated performance requirements allow a wide variety of processing capabilities to be considered for use in Virtual Reality. The Participant System should be

able to run on any platform that meets the graphical and/or general processing requirements.

Displays and Renderers

It is difficult to predict the software renderers and hardware displays that will be created for use as part of a Virtual Reality interface. By identifying how these elements interface with the Participant System, the ideal solution will be able to integrate new displays and renderers, even ones created after the PS.

Sensors

Like renderers and displays, there are a variety of sensors that may be employed when creating a VR interface. New sensors must be able to be fully integrated into the ideal Participant System.

Application

As discussed earlier, the main objective of this project is to identify the elements common to all VR experiences – the interface, and to build a unified system for managing these elements. The elements that are unique to each individual virtual world cannot be predicted, and as such, the application system that provides them cannot be static. Thus, the Participant System must remain independent from any single Virtual World Application System.

PARTICIPANT SYSTEM MODEL
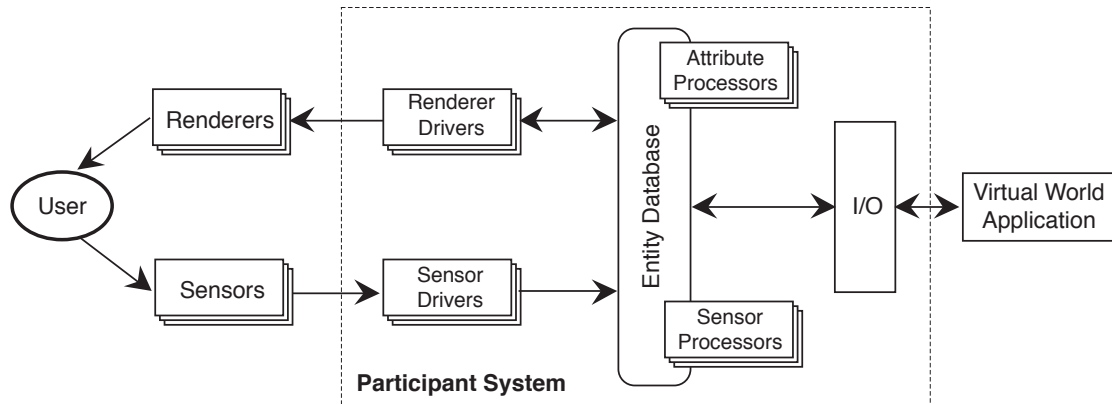
Figure 13:
The Participant System Model

In light of the performance requirements, experience with building and using Participant Systems has lead to the development of the Participant System Model shown in Figure 13. This model provides all of the features required for participant systems listed above. Mercury was developed by the HITLab to embody features and functions of the Participant System model. In this regard, Mercury has been a vital part of the HITLab's VR software suite for the past eight months. In the next paragraphs, the Participant System will be described in detail, followed by discussion about various versions of Mercury. The latest version of Mercury is a full implementation of the Participant System model.

Perhaps the most important aspect of the Participant System is the tight coupling between the input devices and displays. In a typical VR system, sensor data is read and applied at the same rate that the application manipulates all the other world objects. This means that if a user is riding a virtual train, updates from the user's head movement will change the 3D view as often as the train's movement is calculated. While it is desirable to have all movement applied as often as possible, experience has shown that updates from head tracking are much more important and need to be applied more often than updates

from what happens in the virtual world (Furness 1992). In this regard, the PS model applies data from the sensors directly to the display, regardless of how often those changes are communicated to the application. Application speed has no impact on perceived frame rate under this design. In the same way, latency is significantly reduced by coupling the input data directly to the appropriate renderers; therefore, world dynamics calculations are not at all a part of this loop and do not add latency.

In the PS model, all the renderers are executed or driven from a single hardware platform. When using a distributed VR system such as VEOS, it is very difficult to synchronize data delivery between various renderers. This can cause visual and audio events to be displayed at different times, or worse yet, at different spatial positions. By combining the renderers into a single tightly-coupled system, registration problems are minimized. Anomalies due to individual displays and/or rendererers can be compensated for in software.

When a virtual world is created, there are two issues to be considered: what elements will comprise the world and how will the user interact with those elements. While the individual elements within a world are often unique to that world, the issues of reading the sensors and providing displays are common to every world that is created. By managing the user's interaction with the virtual world without imposing any structure or requirements on the virtual world's dynamics, the PS solves half the problem that each world builder faces, that of reducing the problem to the level of any current database management system. Few programmers today are actually concerned about how to create a window on a screen. They take for granted that the elements of their program will be displayed and that they will be able to find out what actions the user has taken merely by using the required protocol. The PS provides this same functionality when building a virtual world.

While the previous discussion represents the initial motivation for creating the PS model, perhaps its greatest feature is that the final system may be driven by any application program that can use the required protocol. Anything from a simple communications package to a full-blown Virtual Environment Database System such as VEOS can issue the proper commands to create and manage a virtual world. Furthermore, with a limited amount of effort, current applications can be adapted to provide a virtual interface. This is the ultimate delivery of the PS promise – to solve all of the issues involving the interface to the physical body so that any application need only worry about what the world contains, not how it is displayed.

Database

All data within the PS are either sent to the database or came from it. Data read from the sensors are sent to the database. Updates for entity attributes from the application are sent to the database. Renderers receive all their data from it, and the application receives its data about the participant from it. In sum, all processors operate on data within the database.

The PS Database has two files. One contains entities and attributes, and the other contains data read from sensors. Records in the entity file have the following format:

Unique ID	Attribute	Value

To input records, full records are submitted. To refer to a record, the Entity-ID and Attribute are given. Values alone are returned.

The sensor file stores information read from both the sensors themselves and from sensor processors. Its records have a format similar to attribute records, with one exception:

Unique ID	Data-Type	Count	Value

Some sensors (e.g. the Polhemus Fastrack™) have multiple sensors with the same data type. To accommodate this, a count must be used to refer to each individual output. Otherwise, the sensor file operates similarly to the entity file.

To facilitate flexibility in mapping sensor readings to body parts, the output of any sensor may be fed directly to the value of any entity attribute, as shown in Figure 14. This is referred to as

Figure 14:
Binding a sensor output to an entity's attribute

"binding" a sensor to an entity's attribute. For example, to create a virtual joystick in the world that moves with a physical joystick attached to a position sensor, the output from that sensor is bound to the position attribute of the entity that is the virtual joystick. Other attributes describing the virtual joystick's appearance are determined by the application, but movement from then on is mediated within the PS itself, with no intervention from the application. Any subsequent updates by the application to an entity's attribute that has been bound to a sensor are ignored.

I/O Manager

The Input/Output manager provides all interaction with the application. It receives updates from the application regarding world entity attributes such as position and graphical description, configures the data flow within the PS at the application's request, and sends updates about the participant from the database to the application at a controlled rate. Updates may be issued for the participant's attributes as well in order to update necessary information for renderers or to provide inputs to sensor or attribute processors. The protocol for this interaction will be discussed in greater detail below.

Application

The application responsibilities may be provided by virtually any program that uses the PS protocol. Using this protocol, the application issues the proper set of commands to the Participant System to create a virtual world and interact with the user. The application is responsible for calculating world dynamics, mediating the interface to other virtual world resources such as other Participant Systems, and the general management of all aspects of the virtual experience that are not directly related to the participant.

Protocol

The protocol for communication between the PS and the application requires the most attention and care of all communication paths within the PS. It will most often be used across a network or other remote link, and must therefore use as little bandwidth as possible. It will also be used by different applications running on a variety of platforms, so it should be extremely simple to implement, both in terms of technical integration and worldbuilder understanding.

There are three types of packets that must pass from the application to the PS, and two types that pass from the PS to the application. Packets flowing from the application to the PS are all commands being issued to the PS, and packets from the PS to the application are data packets either reporting the current state of the PS or reporting values read and computed from the sensors.

Flow commands are issued by the application to specify how data will be processed within the PS. When the Participant System first starts, the I/O Manager tells the application what sensors and sensor processors are available. The application can then use the Flow commands to organize the flow of data from the sensors through the sensor processors into the entity database, determining which sensor data streams will be attached to which entity attributes and which will be delivered to the application. A Flow command

is also used to configure the packet of data that is reported back to the application regarding current readings from sensors and sensor processors.

Attribute commands are used to update attributes of world entities for the PS. This includes entities that are used as part of the virtual body. While certain attribute values, such as position, will come directly from sensors and sensor processors for body parts, graphical, aural, and other descriptions will be determined by the application and updated as any other world entities.

Finally, the application issues **PS** commands to control specific events within the PS itself, including the calibration of sensors and the behavior of the PS, e.g. what data it prints to the screen and how fast it should communicate with the application.

In most cases, the application will not only send updates of world data to the PS, but it will need updates of the participant's actions as well. As mentioned, the application uses a flow command to specify what sensor data will be reported from the PS in its output **Data Packet**. The data packet contains current information from the database regarding the participant's position in the virtual world, etc. In current HITLab worlds, it is up to the application to set the participant's velocity, determined from the rotation of the joystick and the state of the joystick buttons. This data is received by the application in the data packet. This is by far the most common of the two types of packets that flow from the PS to the application.

The other set of packets includes the **State Packets**. These packets are used by the PS to tell the application what sensors, sensor processors, attribute processors, and renderers are available, and what choices the application has made regarding data flow. Typically these packets will only be sent when the PS is starting up. The application will use this information to configure the PS after which the majority of packets flowing to the application will be Data packets and the majority of commands issued to the PS will be Attribute commands.

Processors

In order for the PS to make use of further developed data without requiring that they pass through the application, two types of processors have been added to the model – attribute processors and sensor processors. Both processors accept one or more data streams as input, and return one or more processed data elements to be used by other processors, the application, or the renderers.

The main value of the processors is that they give the user the ability to create and use new processors without having to build them directly into the Participant System. The Participant System's functionality can be expanded long after the system is built. Processors also let the Participant System continue to manage and provide the functionality that is directly associated with the physical body, providing as much detail as is available about body motion to the application. It remains up to the application to determine how these resources are to be applied.

<u>Attribute Processors</u>

Attribute Processors operate on entity attribute data as it is input from the application.   A common example of an attribute processor is the Position Hierarchy.

The position hierarchy is a method for describing and understanding hierarchical spatial relationships between virtual world entities. It allows an entity, or the participant himself, to specify an "origin" entity relative to which it moves (see Figure 15). In other words, an entity that is "origined" to another entity

··········· World 6D
– – – Relative 6D

B

A

Local Coord. Sys.

World Coordinate System

Figure 15:
The Position Hierarchy:
Entity B is "origined" to Entity A.

has a relative position (i.e. a "delta") in the coordinate system of the origin entity (Robinett

1992). When the origin entity moves, any other entity that is attached to it moves along with it. An entity's delta is maintained as the origin entity moves through space. Entities may be origined to entities which are themselves origined to other entities. In a sense, all entities have an origin and a delta. For most entities, their origin is the world origin and their delta is their world position.

In addition to being able to group sets of entities into a single connected object, the main application for this concept is to allow entities to "ride" other entities. For example, if the participant wants to ride a virtual train that is moving through a virtual terrain, the application need only set the participant's origin to be the train entity. Until the origin is changed, the participant will move through the world on the train.

The most basic use of this hierarchy is in the placement of the Participant's body parts in the virtual world. Spatial sensors report positions relative to the sensor source. The Polhemus™ tracker, for example, reports the position of its sensors relative to the source of the magnetic field. To facilitate movement through the virtual world, an entity representing the sensor source is created. All body part entities have this source entity as their origin. Relative positions received from the spatial sensors are then used as deltas from that origin. To move the body through the world only the sensor source needs to be moved. Body part entities retain their deltas relative to the sensor source and thus follow it automatically.

This functionality is supported by attribute processors. In order to provide the proper world positions for all entities, as opposed to positions in local coordinate systems, these functions compose an entity's delta and its origin's world position to create a world position for the entity. This may be a recursive process if there is more than one level to the position hierarchy. Though world entities could manage their positions relative to other entities themselves, this mechanism eliminates any latency between a new position being reported and displayed for one entity and the subsequent position change by the following

entity. The position hierarchy attribute processor will be executed whenever an entity changes its delta, its origin, or an origin entity changes its world position.

Sensor processors

Though raw data may be applied directly to an entity attribute, **Sensor Processors** are used to extend the usefulness of this input. Input data to a Sensor Processor might come from raw sensor data, output data from another Sensor Processor, or from the entity database. Sensor Processors may also use values read from anywhere in the database.

A simple example of a Sensor Processor that would typically be delivered with a Participant System is a prediction processor. This module would take a raw spatial sensor data stream, maintain a set of the most recent samples, and return a predicted position based on those values. This returned value would then be attached to an entity such as the participant's head and be used for placing that entity in the virtual world.

A more complicated example that is also likely to be included in any Participant System is a Movement processor. In order to move through the virtual world, the application sets a velocity for the virtual sensor source. At least once per rendered frame, elapsed time will be determined and a new position for the virtual sensor source will be determined from the source entity's velocity, current position, and the elapsed time. If velocity control is not processed internally, the application must provide the change in the participant's position. While this obviously will work, it means that movement will occur at the same frame rate as the communication between the application and the Participant System, instead of at the internal frame rate. Unless this communication rate is relatively high, this is likely to cause movement to be jerky compared to the frame rate at which head movement is executed. By providing movement through the world as a sensor processor,

the application need only provide a velocity to accomplish smooth traversal of the virtual world.

Other more advanced examples of sensor processors are a Gesture Processor, a Speech Recognition Processor, and a Physiological Model. A Gesture Processor would take inputs from a glove interface, including both position and flex information, and return recognized gestures such as "thumbs up" or "pointing". Similarly, a Speech Recognition Processor would take a digitized audio stream and would return spoken words. The Physiological Model, a computational model of the constraints and dynamics of the human body, would return positions of body parts based on input from some limited number of sensors.

It should be noted that even though the gesture processor would return recognized gestures, the data it receives would still be used for positioning the parts of the hand in the world. Typically, recognized gestures will only be used by the application or by other processors, not directly by any renderer.

Renderers

Renderers are software modules that drive the display hardware to create the sounds, graphics, or other physical manifestations of world entities. Entities may have a variety of attributes that may be displayed by one or more renderers. The position attribute, for example, is used by any renderer that displays a spatial quality of an entity, such as a graphical or aural renderer. Other attributes such as a graphical description or a loudness may be specific to a single renderer. Renderers are responsible for driving related display hardware and software.

Renderer Drivers

Renderer drivers are responsible for issuing the proper renderer calls to display the information in the database. Each renderer driver delivers relevant information from the database as often as the corresponding renderer can accept it. Renderers themselves may or may not be written to use the database directly. Those that do use the database will have a minimal driver, only used to manage the renderer. Renderers that do not share the entity database will have drivers that continually copy relevant information from the database and send it to the renderer.

Renderer drivers must support the following functions:

The **Configure** command reads the renderer's configuration file to determine which attributes this render will be interested in and what function to call when a new value for that attribute is received.

Init starts up the renderer. In the case of HITLab's graphics renderer, a second process is spawned specifically for rendering the graphical images.

Shutdown quits the renderer.

Calibrate adjusts the renderer to the current user. Such adjustments might include setting the interocular distance (the distance between the user's eyes) or the size of the user's head (used by the sound renderer).

Enable tells the renderer where the data it depends on is located, (e.g. both the graphics and sound renderers depend on the position of the head in the virtual environment). The enable command tells such renderers where the current position of the head, computed from sensor data, can be found.

Disable tells the renderer that its dependent data can no longer be found in the previous location.

Render is the command used to signal the renderer that all data for the current frame has been calculated and it is time to render the frame.

Sensors

Sensors are typically hardware devices that provide raw data about the participant to be used by the renderers and the application to provide the virtual environment. Common sensors detect the positions of various body parts, the state of switches on a 6D joystick, or flexes of joints in a dataglove.

Sensor drivers

Sensor Drivers are program modules used to interface with sensor hardware. By making sensor drivers modular, any new sensor that becomes available can be used merely by writing a new driver for it instead of having to rewrite the whole Participant System. Furthermore, the same applications could be used with new sensors without changing the application code. Like renderer drivers, each sensor driver must support seven standard device functions:

Configure parses the configuration file to setup the available sensors and sends a list of configured sensors to the application.

Init opens a connection device, establishing the proper communication.

Shutdown closes the connection to the device.

Calibrate allows the participant to set the sensors to return values that are appropriate to each given user.

Read returns new data from the sensor.

Enable and **disable** functions turn off and on the reading of each sensor attached to a given device (some devices such as the Polhemus Fastrack™ provide more than one sensor).

IMPLEMENTATION

As mentioned above, the Participant System began as a set of LISP programs used to create VEOS entities responsible for reading sensors and managing renderers. These programs were included in the creation of every virtual world. To take advantage of the static nature of this code, and, moreover, to reduce the latency inherent in sending data from computer to computer through VEOS, the first compiled single-component version of the Participant System was built in August and September of 1992 and dubbed "Mercury" for its high speed. Mercury 1.0 had its first major exhibition in October of 1992 at the Human Interface Technology Laboratory's Second Industrial Symposium on Virtual World Interfaces.

Mercury

Three versions of Mercury have been created, representing the Participant System model to varying degrees. Each version has been programmed for use on the HITLab's Silicon Graphics Iris 320 VGX graphics workstation, but the code is intended to be portable. The HITLab's two major renderers – the Imager for graphics and the Sound Renderer for spatialized audio – have also been integrated into each version, though in diffferent ways. Sensors that have been integrated for use with Mercury include the Polhemus Fastrack™ and Isotrack™, the Logitech acoustic position tracker, the VPL EyePhones™ and Dataglove™ control units, the Spaceball™, and a 6D joystick button sensor built specifically for this project. Specific versions had the following features:

Mercury 1.0 was created solely for high speed, low latency, and tight registration between renderers. It did not have any of the modular features that have come to be part of later versions. Every aspect of the system was hard-coded, from reading the sensors to making calls to the renderers. Additional sensors, renderers, and processing could not be

added without reprogramming at the lowest levels. Though successful in demonstrating the initial Participant System concept, it had extremely limited expandability.

Mercury 1.5 was built in early 1993. It was the initial embodiment ofthe Participant System concept, providing modular sensor drivers and sensor processors but still maintaining hard-coded renderer interfaces. This version used an enhanced version of the application interface protocol created for version 1.0.

Mercury 2.0 represents an almost complete rewrite of the Mercury software. It incorporates the full Participant System model allowing for both modular sensor drivers and modular renderer drivers, plus allows for the addition and configuration of sensor and attribute processors. The application interface has also been updated to allow for the expanded communication necessary to take full advantage of Mercury's capabilities.

Language

Mercury 2.0 was written in C using the MIPS compiler on a Silicon Graphics Iris 320 VGX computer. The Mercury XLISP interface was also written in C and compiled with the MIPS compiler on a Digital Equipment Corporation 5000/240 workstation. It uses a standard UNIX sockets library with TCP/IP as the transport mechanism between the "front end" XLISP interface on the DEC and Mercury itself running on the Iris. Appendix A provides an example XLISP program that interfaces with Mercury to produce a virtual world.

Database

Mercury's database uses one-dimensional arrays to store all necessary data, including data coming from sensors, produced by processors, and entity attribute values. Each array may store any number of elements and each element may be any type and any length (depending on the type). Arrays that store information that is not directly related to an entity (e.g. sensor or processor data) are called "datasets". Arrays that store entities

have specific formats determined at startup when the attributes that will be used within Mercury and by the renderers are configured. When configuration files are read, attribute names are assigned numbers according to their corresponding element in the entity array. These attribute names and ID numbers are then reported once to the front end. Subsequently, only the attribute IDs need be sent to Mercury with entity updates. Furthermore, the front end will then know what attributes Mercury will be concerned with and can cull out any irrelevant attribute values before sending them across the network. This method also allows Mercury to be completely flexible in the attributes it will be using. Previous versions hard-coded the attributes and therefore would only display those attributes and no others. Either type of array may be passed to a processor as input, output, or both (as in the case of the sensor source's position used for both input and output of the movement processor).

Protocols

Sensor and renderer driver protocols are exactly as described in the model. Commands that are additional to the functions described in the protocols may also be added by any driver. To add a command, a driver specifies a string command name and lists a function to be called when that command is issued by the application. Such commands may include setting environmental conditions such as the color of the background, sending MIDI commands from the sound renderer, or engaging a special feature specific to a certain sensor device. Like attribute names, each command is assigned a number and the resulting table is sent to the front end at startup. The front end will then send command numbers instead of string names.

The application protocol is also implemented much as it is described in the model itself. The application begins by issuing the "hg-init" command which starts the Mercury process running on the remote computer (Note that all Mercury commands begin with

"hg", the atomic symbol for Mercury). Mercury responds with a state packet containing information about the sensors, processors, and renderers that are currently available as well as the attributes and data types that it knows about. Using this information, the application issues flow commands to add processors, bind sensor and processor outputs to entity attributes, and specify the information packet that will be returned from Mercury to the application.

Once the data flow tree has been established, the application will use attribute commands to impart to Mercury the values of entity attributes in the world. Attribute command packets are marked with a 0 in the highest bit and contain the attribute number, the entity number, and the relevent data.

PS commands will be used to issue commands directly to renderer and sensor drivers. These are the commands to which the drivers will have attached functions at startup, as described above. These packets will have a 1 in the highest bit and will contain the command number (retrieved from a table of command names and numbers) and any required data.

Lastly, data packets created from datasets specified earlier by a flow command are sent from Mercury to the front end as often as the application can accept them, as described in the next section.

Timing and Flow control

Timing of communications and processing is a major issue in the Participant System. It is extremely important for each element of the system to be processed as often as needed, but not processed more often for fear of degrading other processing in the system. Similarly, data and commands should be communicated as often and only as often as they can be processed. Over-communication results in data getting stale while being held in buffers. The two most delicate points of concern with regard to timing are the

communications between the Participant System and the application, and the processing of sensor and attribute data.

Communications between the PS and the Application

Managing the proper communications rate between the Participant System and the application is accomplished by means of a relatively simple algorithm. The PS begins sending the application data at some low but reasonable rate. Every few seconds, the PS increases the number of packets sent per second by one. The application wants to find data about 80% of the time it looks for it in its input buffer. If data is present more than 80% of the time, the application alerts the PS and the packet rate drops back by three packets per second. The application sends "I'm alive" messages to the PS at regular intervals. If the application ceases to send these messages, Mercury will assume that application processing has stopped and will wait until it begins receiving them again before resuming communications. It should be noted that the values used in this algorithm were arrived at empirically. Though other values may prove to be more efficient, these seem to work sufficiently.

This algorithm allows the PS to send data only as often as the application can accept it, while requiring the least amount of acknowledgment from the application to the PS. By halting communications with the application if feedback is not received, Mercury can maintain the appropriate communications rate for when the application comes back on line. Otherwise, stale packets would accumulate in the application's input buffer and the send rate would drop off, only having to be ramped back up when communication resumes.

Participant System	Application
1. Increase rate by 1 every few seconds. 2. Reduce rate by 3 if application alerts. 3. Halt sending without periodic acknowledgements.	1. Read data as fast as possible 2. Send alert if data is found > 80% of the time. 3. Send periodic acknowledgements.
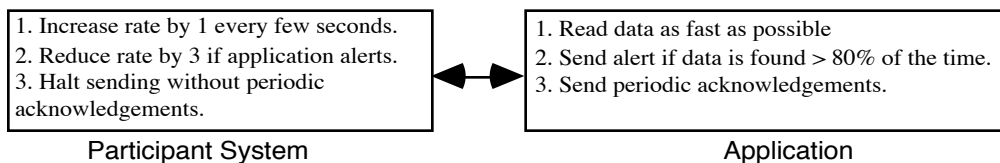
Figure 16:
Summary of PS-App Communications Algorithm

Data Flow Tree

When to process data is a much trickier problem than when to communicate. As the application configures the data flow from sensor drivers through the processors to the renderers and output packet, the PS sets up a flow tree to manage the data as it moves through the system. Proper configuration is extremely important because each element of the Participant System may run at a different rate. Various sensors and renderers may each have their own maximum frame rates, and more often than not, each one's frame rate can vary greatly depending on a wide range of factors. Poor management results in, at best, superfluous processing and, at worst, greater latency. The algorithm used by Mercury 2.0 for configuring and processing the tree appears to account for all of these factors and provide the proper solution in most common cases.

This processing algorithm is a hybrid of "eager" and "lazy" processing of data (i.e. data-driven and demand-driven processing). In other words, at some times data is processed as it is received, and at other times it is not processed until it is about to be used. Such processing choices must be made to account for certain types of data, such as sensor data, that may often be received much faster than it can be used and thus should only be processed as necessary, as opposed to world data that is not likely to be updated again in the same frame, and therefore should be processed immediately. Furthermore, it is extremely important to process the data in the proper order to assure that only the freshest data is used at each step.

To accomplish the appropriate processing, each node of the data flow tree is assigned a number based on the number of processing steps between it and the top of the tree (see Figure 17). As data arrives, its flow tree value is determined. If it is at level 0, it is processed immediately. For instance, world data coming from the application that does not need to be processed is assigned a 0, and would therefore be sent immediately to the renderers on arrival. This is the eager processing end of the tree. Data with flow tree

values greater than 1 are stored in buffers associated with their tree value. When it comes time to render a frame, data in buffers are processed beginning with buffer 1 and working up. This is the lazy end of the processing, i.e. the processing that only occurs as necessary. This processing is only necessary when something is going to be rendered and not before. Without this arrangement, all the processing might occur as fast as the sensors are read, even if the renderers would only use a portion of the resulting data.

The algorithm as described handles all but one aspect of the required processing: certain processors, such as a prediction processor, must receive data as often as possible, no matter how often the renderers need it. To accommodate this requirement, processors may be set up to require data as often as possible. The result is that such processors will behave in the tree much like renderers in that they will ensure that all data below them will be processed as often as they can receive it.

Example Data Flow

By way of illustration we will follow the somewhat complicated yet common example of the data flows needed to run the PS in any general virtual world:

Figure 17:
Assigning Flow Numbers

Sensors: One of the most common sensors used for virtual reality interfaces is the position tracker. We will use two position trackers in this example – one to measure head position and orientation and the other to measure operator hand inputs using a six degree-of-freedom joystick. Additionally, the time sensor reports the current time.

Individual processes will compare this time to the time they last processed data and act accordingly, as explained below.

      Sensor Processors: There are two sensor processors used in this example. The first is the Movement processor, as discussed in the description of the model. This processor takes as input the participant's current position and velocity from the database and current time from the time sensor. The output is the participant's new world position. The other sensor processor used is a prediction processor to help better determine the head position and orientation at any given time. The effect of prediction is to reduce the throughput delay usually apparent in the visual display as a result of a head rotation. Note that this processor receives both sensor and time data, and outputs a predicted head position. While the hand moves too rapidly and with too many degrees of freedom to accurately predict where it might be at any given moment, the head's position may benefit greatly from using even a simple prediction algorithm. Under the PS model, as well as when using Mercury 2.0, various prediction processors, as well as any other kind of processor, can be exchanged and/or added to the processing loop.

      Attribute Processors: The attribute processor used in this example is the Position Hierarchy processor also discussed in the model above. This processor composes the relative positions of the body parts and the world position of the virtual sensor source. Positions read from the head and joystick sensors will be sent directly to the head and joystick entities' relative positions. These relative position attributes are then processed by the Position Hierarchy to create world positions for these entities.

      Application: The application's role in this example, beyond configuration, is to describe the given entities (e.g. the appearance of the joystick) and to maintain participant attributes such as velocity. These updates, as long as they don't require any processing by the PS, are sent directly to renderers.

Renderer Drivers: This example will only use a graphical renderer, though any renderer will behave similarly. In this case, the renderer will only be interested in the position of the entities and initialization data such as graphical descriptions.

Configuration: The application configures the data flow within the PS as shown in Figure 18. The PS then uses the resulting tree to manage all processing.

Processing: In the figure representing the data flow in our example, each datum has been marked with its flow tree value. There are three interesting parts of this example. The longest branch of the tree shows a common configuration of sensors and sensor processors used to manage both the participant's virtual body and movement through the virtual world. Data is read from the Time sensor and the database at the top to create a new sensor source position delta, which is composed with the world origin to create a world position. This world position is then composed with the position deltas from the sensor source read from the head and joystick sensors to create the head and joystick world positions. As discussed in the description of the algorithm, data past the first level are not computed until just before they are to be rendered.

The second interesting part of this example is that data from the application that does not depend on any attribute processors is sent directly to the renderers without waiting
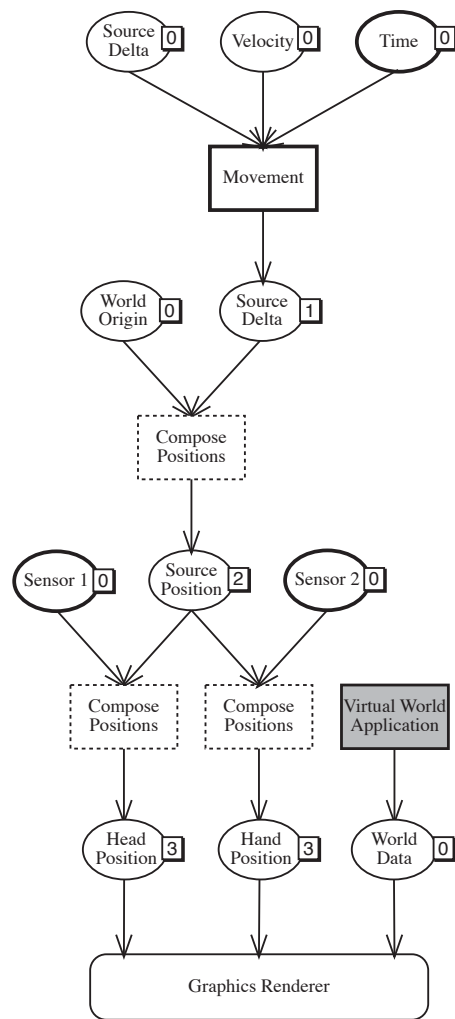
Figure 18:
Data Flow example configuration

for a render loop to come around since these values are unlikely to be updated before the renderers will next need them. Attributes received from the application that do require processing, such as position deltas, would not be processed until required by the renderers if they fall above the first level of the tree. By waiting until the renderers will actually use the data, we avoid using data that might change before the next frame is rendered.

To implement a prediction processor, samples of sensor data must be taken as often as possible. To provide these samples, even under a low renderer frame rate, the prediction processor is set to process the full tree behind it as often as possible (see Figure 19). In this example, the data that the prediction processor uses is all level 0 and would therefore be processed anyway, but it should be noted that even if there was an extensive tree before this processor, the full tree would be computed, and more importantly, values below this processor in the tree are counted from 0 again. If necessary, this part of the tree may get its own set of data buffers to properly compute the data needed by the processor.
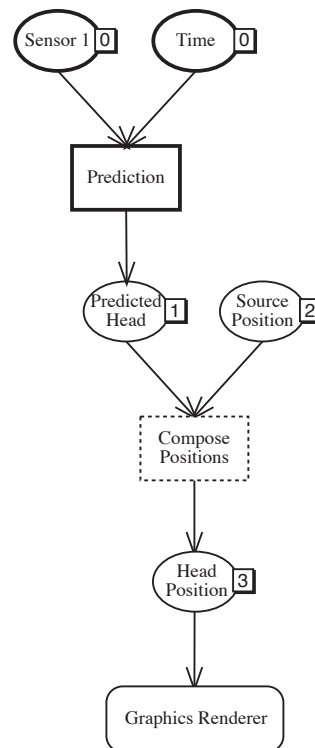
Figure 19:
The Prediction Processor

EVALUATION

Mercury 2.0, the HITLab's most recent version of the Participant System, has recently been completed. This is the version that most closely represents the PS model described above and the first to meet all of the requirements outlined above. Below, Mercury is evaluated in terms of these requirements.

Speed

As has been stressed since the beginning of this paper, speed and latency are probably the two most important characteristics affecting a comfortable and believable virtual environment. Indeed, speed and latency are the main qualities by which VR systems are judged.

When running on a Silicon Graphics Iris 320 VGX computer in a configuration similar to the one described above, Mercury 2.0 processes at well over 1000 Hz. This frame rate is the number of times per second that Mercury executes its main loop of reading the network, reading sensors, processing, and writing to renderers. Actual frames per second experienced by the participant are determined by the renderer used, the hardware it runs on, and the complexity of the environment. At least for the near future, Mercury will run as fast or faster than any renderer that is likely to be linked to it, so the issue of frame rate is placed in the hands of the renderers themselves.

Latency

The PS does contribute to the latency experienced by the participant, as explained in detail when low latency was first discussed as a desired quality. To determine the time lag between when a sensor is read by the participant system and the results are displayed by the renderers, special data types were created that attach a time value to normal data used by the system. As data are read from sensors, they are marked with the current time. This time

stamp is maintained and passed along as this and resulting data are processed by the PS. When the next renderer frame is drawn, the time stamp is then compared with the original time stamp to determine the latency imposed by the Participant System. When running on the HITLab's Silicon Graphics IRIS 320 VGX and using a common data processing tree such as the one described above, Mercury 2.0 was found to add an average of 10 msec to the total time lag.

Registration

Mercury is a program that runs under UNIX. Because UNIX is not a real-time operating system, there is little control over exactly when a given process will occur. The best that Mercury can do is make required renderer calls as fast as possible. Even when renderer calls are executed from a single high-speed process as often and as quickly possible, it is unlikely that better registration can be achieved, at least on a non-real-time platform.

Flexibility

By simplifying the world building process, VR systems can reduce flexibility available to the world builder. While different systems offer varying degrees of flexibility, the Participant System is meant to encompass processing that must occur to provide a VR interface, no matter which system is providing that interface. Regardless of how world objects and their dynamics are specified, every VR system must read sensors, interpret user behavior, and write to renderers. It is up to the individual implementations to ensure that flexibility is not lost along the way. Mercury, an accurate implementation of the Participant System model, is as configurable and flexible as the model allows (it's modular design and the resulting flexibility in interfacing with new applications and using new sensors and renderers will be discussed even further under "Extensibility"). Below, we examine

Mercury's flexibility specifically in terms of building worlds and configuring the participant's virtual body.

Building Worlds

The ultimate goal of the PS is to impose as little of its own structure on world data as possible. As long as data that is sent from the application can be understood by the renderer that receives it, Mercury will not interfere. Mercury does not even know what attributes will be used by linked renderers until it starts up and reads the configuration files. By concentrating on ensuring prompt and efficient data delivery from sensors and the application through processors to the renderers, and not adding any semantics other than those chosen by the application and configured through the processors, Mercury 2.0 allows maximum flexibility in specifying the virtual environment.

Building the Body

There is no place in the PS model that adds any physiological semantics to data that are not chosen by the application. As explained earlier, any sensor or sensor processor data may be applied to the value of any entity's attribute. The model's design allows the worldbuilder to configure available sensors and processors in any way. The participant's virtual body is only limited by the sensors and processors that are available. The flexibility of the participant's body will be discussed in more detail below.

Extensibility

Every aspect of the Participant System excluding the database and the internal structure is completely modular. The PS can be configured to use any sensor driver, renderer driver, processor, and application that uses the given protocols. Using one or more of these program modules, the PS may be integrated with practically any hardware or software with a minimum amount of work.

Bandwidth

By reducing the data required to pass between the application and the PS, bandwidth is conserved. The ideal bandwidth is a function of the types of attribute data, the size of the data packet sent from the PS to the application, and the rate at which data is passed back and forth. Although the model attempts to reduce the data size to the smallest necessary, it does not put a limit on what can be passed to and from it, nor does it limit the speed of communications.

Independence

Mercury has been written in the C computer language and can therefore be compiled and run on virtually every popular computer platform. Again, renderers, sensors, and any application can be incorporated into the main program. Other modules used by Mercury may or may not be independent of certain hardware, but if not, most of the code used will probably be reusable.

Summary

Mercury 2.0, and the Participant System model itself, was built on experience gained from the implementation of version 1.0, and to a certain extent version 1.5. The original Mercury concept was to reduce latency and increase speed, something each system does fairly well. Through our better understanding of the virtual body and the processing necessary to make it useful we were able to include the extended features discussed above. Version 1.0 was a beginning step toward improving the quality of the interface itself, but version 2.0 goes beyond that to improve the way virtual worlds are built, the flexibility in the way the participant's virtual body may be configured, and the ease with which new renderers and sensors may be included in the VR experience. A summary of Mercury's evolution in meeting the performance requirements discussed above is given in Table 1.

Table 1:
Summary of Mercury versions and associated qualities

Requirement	Mercury 1.0	Mercury 1.5	Mercury 2.0
High Speed	GOOD	GOOD	GOOD
Low Latency	GOOD	GOOD	GOOD
Registration	GOOD	GOOD	GOOD
Flexible World	FAIR	FAIR	GOOD
Flexible Anatomy	POOR	FAIR	GOOD
Extensible	POOR	FAIR	GOOD
Low Bandwidth	FAIR	FAIR	FAIR
Hardware Independence	FAIR	FAIR	GOOD
Displays/Renderers Indep.	POOR	POOR	GOOD
Sensors Independence	POOR	GOOD	GOOD
Application Independence	GOOD	GOOD	GOOD

APPLICATION CONSIDERATIONS

The PS vs. the Application

It is important to understand the responsibilities of the Participant System relative to the application. The PS not only provides a high frame rate for internal processes, but it also allows a wide variety of internal processing to create the virtual environment. Its speed and transparency to the world builder make it tempting to place some world dynamics processing into the PS. While initially this may yield a smoother experience for both the user and the world builder, the PS will eventually lose most or all of the benefits it is supposed to provide because it will be spending its processing power on computing world dynamics. The world builder must keep in mind the major motivation and benefits of the Participant System:

High Speed/Low Latency

The main purpose of the PS is to provide as close a link between the sensors and the displays as possible. Any processing that must be performed before data is received from the sensors is reflected in the displays adds to latency and reduces the quality of the participant's experience. As such, only processing that leads to properly displaying to the participant his or her view of the environment should be added to the PS. For example, though it may be tempting to give world objects velocities and let world positions be executed in every frame by the PS, this will reduce frame rate and increase latency, ultimately reducing the quality of the experience.

Participant Management

The PS is responsible for the participant only. It only knows and cares about reading sensors, processing data, and displaying entities. It is the interface only. The application must take responsibility for managing the user's interaction with world dynamics. In other words, the PS will tell the application that, for instance, a button has

been pressed on the wand. It is the responsibility of the application to determine how that button press will impact the virtual environment, if at all.

Single User Interface

The Participant System provides an interface to a virtual environment for a single user. Any processing that occurs within the PS must be relevant only to that single user. In the case of the world entity with a velocity mentioned above, new positions for that entity would only be known to the participant that is processing that velocity. Other participants in the environment would either never see the updated positions or would run the risk of having inconsistencies between participants. The bottom line is that if the functionality in question will only impact the given participant then it should be managed by the PS. If it may be experienced by other participants as well, then it should be performed by the application.

Flexibility in Physiology

An early question in Mercury's development was the extent to which the participant's physiology would be predetermined by the PS. Indeed, Mercury 1.0 did have a fixed physiology, allowing only for head and wand movement. In contrast, the PS model and later versions of Mercury have been expanded to allow for any physiology either through the programming of processors or from physical sensors. In other words, since the PS only manages the flow and processing of information, and doesn't attach physiological semantics to any data flow, any sensor or sensor processor can be configured to represent any body part. For example, if the world builder wanted to use a Fastrack's™ third and fourth sensors to provide the positions of the participant's feet, entities representing the feet would be described to the PS and the data flows from those sensors would be bound to the position attributes of those entities. After that, the entities

representing the feet would be positioned according to the data retrieved from those sensors.

A more specific challenge is the desire to attach the participant's view to his or her fingers instead of the head or provide any other non-standard virtual physiology. The HITLab's graphics renderer, for example, determines the eyepoints relative to the position of the head in the virtual world. To accomplish the desired effect, the application need only tell the renderer to use the position of the hand or finger to determine the viewport instead of the head. Further views will then be drawn from the hand's perspective. Future renderers are likely to provide more detailed control over physiology, but it remains unclear how useful such capabilities will be.

Multiple Participants

As VR hardware prices fall and network bandwidth increases, multiple participants in a shared virtual environment will soon become the norm rather than the exception. The Participant System is directly applicable to multiple participant issues. Since each PS will display any virtual world entities that are presented to it and report on the data that it reads and processes from sensors, all the application must do is communicate the data read from one PS to another PS and vice versa in order for the participants to see each other. Other dynamics may also be provided by the application as desired. As distributed database systems such as VEOS continue to improve, communicating current information from each PS to all others becomes trivial.

There are issues that are unrelated to the PS that must be solved to improve the quality of a shared virtual experience. While a PS ensures a smooth experience for the given participant, it cannot ensure smooth world dynamics. This is the responsibility of the application. If the application updates the position of a world object that is relatively far away from its previous position, the illusion of motion will be lost and the object will

appear to jump from place to place. This problem is exacerbated when multiple participants are involved. Participants may move at high speeds in order to traverse the virtual landscape within a reasonable amount of time. While the PS will make this movement as smooth as possible for the moving participant, other participants will get only a fraction of the number of updates the moving participant experiences, and thus will perceive the moving participant at best to "jump" and at worst to "disappear." More advanced routines are needed to account for such disparities between Participant Systems.

Networking

Communications between the PS and the application may be provided in a number of ways. Mercury currently uses an ethernet connection to interface with VEOS, but other communications methods such as serial lines or even SCSI may eventually be used. If Mercury were to be run on the same computer as the application, intra-machine communication methods such as shared memory may be used. When providing this connection to the application, and perhaps more importantly the connection to other participants through the application, the worldbuilder must understand how the given application will operate.

For example, examine an application that puts two participants in remote locations into a single virtual world. There are two specific communication links that must be considered: the connection from PS to the application and the connection between the two PSs within the application. Perhaps the application requires a tight connection with the participant so that it may fully interpret and react to all of the participant's behaviors. Furthermore, participant interaction may be minimal. An example of such an application would be SIMNET, in which local computation and interaction was to be performed in detail, but information regarding remote participants was reduced to velocities and periodic

database synchronization. In this case, an application module should be placed at each site. Each module would then determine what information should be sent to other modules.

Another application might require a high degree of interaction between participants and therefore would need a higher bandwidth connection. In this case, the connection between the PS and the application itself might best be performed over the remote link. One PS might be local to the application and the other remote, or both PS might "dial up" the application each over remote links. Either way, the application would act chiefly as a data router, sending pertinent information between Participant Systems. The proper choice will depend on the application.

FUTURE RESEARCH

Physiological Model

Increasing the computer's understanding of how the body works, and thereby enhancing the way we use our bodies in a virtual environment, is one of the next major steps in Virtual Reality research. Some critics of VR maintain that this technology will further separate mind from body, person from person. Anyone who reads such "cyberpunk" classics as William Gibson's <u>Neuromancer</u> finds that indeed bodies, or "meat" as they are commonly called in Gibson's vision of the future, have become almost obsolete (Gibson, 1984). I don't believe that this is true. On the contrary, VR is a means of finally bringing our bodies with us on our electronically mediated journeys, instead of just sitting on the couch. Furthermore, we may be accompanied on these journeys by friends and family, regardless of their physical location. To reach this vision, we will need more than just our head and a joystick.

Physiological models such as ones used for Human Modeling Systems like Jack (Phillips and Badler, 1988) should be incorporated into Participant Systems to extend a small set of sensor readings into a full representation of the participant's body in the virtual world. The result will not just be a better representation of the participant for other entities in the application, but it will greatly enhance the participant's sense of presence in the virtual environment to be able to look down and see a representation of their whole body. The Participant System's modular design makes it uniquely poised to use such a model by allowing it to be incorporated into a sensor processor. Sensor positions would be sent to this processor and the resulting data describing the whole body would be sent to both the renderers and the application.

Human Factors Research

The Participant System project has concentrated on the mechanics of providing the virtual interface based on specific performance criteria discussed earlier. However, while configuration of the PS is extremely flexible, little study has been done to determine what arrangements will produce an appropriate user interface. The following are issues that need to be explored in the future:

Virtual Body - Though the PS allows for any part of the physical body to be mapped to any part of the virtual body, some mappings will be more beneficial than others. For example, attaching the viewpoint to the participant's hand position may serve to cause motion sickness in short order. On the other hand, certain mappings may allow the physically disabled to have a presence in the virtual realm that would not be physically possible in reality.

Renderer and Sensor combinations - Thus far only graphical and acoustic renderers, spatial trackers, and binary switches have been used with the PS. When additional other types of input devices and renderers are included, human factors research will be necessary to determine the best ways to combine this variety of components.

Minimum configurations - Though the future will bring us virtual bodies that are far more robust than the head and head body that is typically used now, we will need to understand how much of the body is required to allow the user to fully participate in various types of virtual worlds.

Now that the means for easily configuring and mediating the virtual interface has been created, these and other issues will require attention to complete our understanding of how virtual interfaces should be built.

Dialog Manager

As mentioned in the beginning of this paper, the field of UIMS quickly grew to include the management of specific interface objects as well as interface interaction.
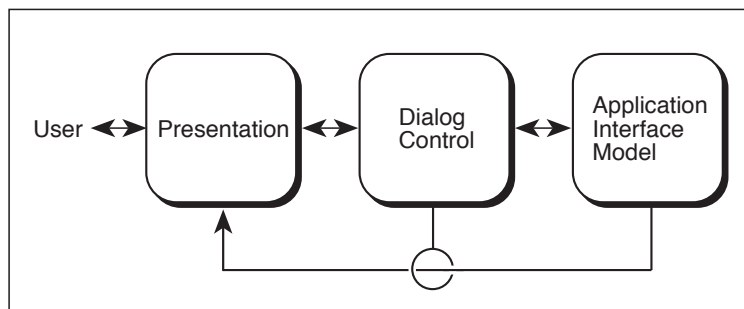
Figure 20:
The Seeheim Model

At the second UIMS workshop held in 1983 in Seeheim, Germany, the "Seeheim Model" was developed, shown in Figure 20. Though this model was later abandoned for being too simple, it illustrates the general issue well. User Interface Management Systems provide not only a standard application interface for input and output devices, they include a Dialog Control component that provides a standard set of interface objects for use by the application. For example, the application doesn't need to know and manage the elements that comprise a scrollbar. It only needs to know where the user has set it at any given time. Similarly, a set of interface elements, or "widgets," can be created to provide more easily the user interface for the application.   This may be implemented as shown in the diagram as a component between the application and the presentation – in this case the participant system – or it may be a set of sensor and attribute processors. While implementing a new component may seem to be the right solution, care must be taken to ensure that a further separation of the participant's behavior and any resulting world reaction does not result in greater latency. If such functionality is implemented using PS processors, the designer must also be careful not to place too much processing between reading the sensors and sending new positions to the renderers.

Database Manipulation

Some virtual world applications are "walkthroughs" in which all the participant does is move through and view a static model. The required processing can be accomplished completely in the PS by making a sensor processor that uses the state of the buttons in the joystick and the joystick's direction to create a velocity which is then used by the movement processor. Most applications, though, will include far more world dynamics than this example, and require a much higher degree of interaction between the participant and the environment. Interaction with world objects in programming terms means manipulation of the environment database. The question is how this database manipulation is accomplished.   Unfortunately, the answer involves more than where to perform world dynamics processing. Somewhere between reading the sensors and changing the database, the participant's behaviors must be interpreted to determine what changes need to be made. All current examples of worlds built using Mercury place all database interaction in the hands of the application. Even computing current velocity is outside of the PS. Joystick direction and button states are sent out to the application which then determines the current velocity and sends it back to Mercury. The price, of course, is that depending on the speed of the application and the rate of communication between it and the PS, significant delays may occur between behavior and resulting reactions. As more people begin to develop sensor and attribute processors, the optimum blend of PS and application processing will be reached.

Processor Programming Language

All processors must currently be programmed in the C language to be linked in with Mercury 2.0. As more world builders begin to use Mercury, they will want to be able to create their own internal data processing without necessarily having to know how to program in C. Because of this, and also because there are a limited number of functions

that are necessary for this type of computation, this is a natural place for using a simpler programming language to specify sensor and attribute processors.

Alternative Operating System

Mercury currently runs under UNIX. The main benefit of UNIX is that it is a standard operating system that runs on almost all workstations. Unfortunately, traditional UNIX adds overhead when communicating both across the network and with input devices. Furthermore, it gives the user limited control over when certain processes will operate such as when devices will be read and when renderers will render. This is the price we pay for compatibility and utility. However, now that the function of the PS has been well-defined, it would be worthwhile to investigate developing the PS on an alternate operating system such as a real-time version of UNIX or even a dedicated OS designed specifically to support the Participant System. Such systems would allow for tight registration between renderers and lower latency. In addition, the parallelism inherent in the PS model may be better supported by an operatng that provides more fine-grained tasking. Unfortunately, such systems are less common, more exotic, and more expensive, especially in the case of a dedicated operating system that would need to be created entirely from scratch.

CONCLUSION

The goal of Virtual Reality is to connect more closely humans with computers. The
Participant System is the software component that mediates this connection for a virtual
world database. Its ultimate purpose is to create a synthetic experience for the user that
may someday be imperceptible from reality. By displaying at appropriate update rates and
with system latencies below those described above, users no longer see a stream of
flickering pictures in front of their eyes but instead believe themselves to be in a new,
computer-generated world. The implications of this new set of perceptions – this new state
of conciousness – will continue to be explored far into the future. Because VR allows us to
change the very configuration of our bodies, to remap one body part to another, it may well
redefine what it means to have and perceive a body.

I have introduced a new paradigm for providing the virtual interface within any VR
system. Based on the general trend toward reusable and decoupled interface components,
it seems that this is the next logical step for most VR systems being used today. When
evaluated in terms of the Participant System requirements presented early in the paper, the
PS model and its implementation provide every quality needed to provide a successful
virtual world experience with high flexibility and little effort by the worldbuilder. Although
there are several paths for future enhancement of the model, the current implementation,
even in its present state, offers a fast, flexible, and simple virtual interface system that can
be incorporated into almost any VR application.

REFERENCES

Appino, P. A., Lewis, J. B., Koved, L., Ling, D. T. , Rabenhorst, D. A. and Codella, C. F. (1992). An Architecture for Virtual Worlds. <u>Presence: Teleoperators and Virtual Environments, 1</u>(1), 1-17.

Blanchard, C., Burgess, S., Harvill, Y., Lanier, J., Lasko, A., Oberman, M., & Teitel, M. (1990, March). Reality Built for Two: a Virtual Reality Tool. In Mark Levoy, Edwin Catmull, and David Zeltzer (Eds.), <u>Proceedings of Symposium on Interactive 3-D Graphics</u> (pp. 35-36). New York, NY: ACM.

Bricken, W. (1990) <u>Software Architecture for Virtual Reality</u> (Technical Report No. P-90-4). Seattle, WA: Human Interface Technology Laboratory.

Frank, L. H., Casali, J. G., and Wierwille, W. W. (1988) Effects of Visual Display and Motion System Delays on Operator Performance and Uneasiness in a Driving Simulator. <u>Human Factors, 30</u>(2), 201-217.

Furness, T. (1992). Class Notes, <u>MEIE 599C -Virtual World Systems</u>.

Furness, T. F. (1986). <u>Putting Humans into Virtual Space</u> (Technical Report No. HITL-R-86-1). Seattle, WA: Human Interface Technology Laboratory.

Gibson, William. (1984) <u>Neuromancer</u>. New York, NY:ACE Science Fiction.

Grimsdale, C. (1992) <u>dVS: Distributed Virtual Environment System</u>. London, England: Division, Ltd.

Grimsdale, C. and Atkin, P. (1992) <u>SuperVision: A Parallel Architecture for Virtual Reality</u>. London, England: Division, Ltd.

Harris, R. L., and Parrish, R. V. (1992). Piloted Studies of Enhanced or Synthetic Vision Display Parameters. SAE Technical Paper Series No. 921970, Warrendale, PA: Society of Automotive Engineers.

Krieg, J. C. (1993). Accuracy, Resolution, Latency and Speed; Key Factors in Virtual Reality Tracking Environments. In Sandra K. Helsel (Ed.), <u>Proceedings of Virtual Reality '92. VR Becomes a Business</u> (pp. 90-100). Westport, CT: Meckler Publishing.

Lederman, S.J., and Taylor, M.M. (1987). Hand Movements: A Window into haptic Object Recognition. <u>Cognitive Psychology</u> 19(3), 342-368.

Lewis, J. B., Koved, L., Ling, D. (1991). Dialogue Structures for Virtual Worlds. In S.P. Robertson, G.M.Olson, and J.S. Olson (Eds.), <u>Human Factors in Computing Systems. Reaching Through Technology. CHI '91</u> (pp. 131-136). New York, NY: Association of Computing Machinery.

McDonough, J. (1992). Doorways to the Virtual Battlefield. In Sandra K. Helsel (Ed.), <u>Proceedings of Virtual Reality '92. VR Becomes a Business</u> (pp. 104-114). Westport, CT: Meckler Publishing.

Minsky, M., Ouh-yourn, M., Steele, O., Brooks, F.P. Jr., Behensky, M. (1990) Feeling and Seeing: Issues in Force Display. In <u>Symposium on 3-D Interactive Graphics</u>. Snowbird, Utah: ACM.

Mogal, J. S. (1993) Immersive Visualization System Architectures . In Sandra K. Helsel (Ed.), <u>Proceedings of Virtual Reality '92. VR Becomes a Business</u> (pp. 118-124). Westport, CT: Meckler Publishing.

Monkman, G.J. (1992) An Electrorheological Tactile Display. <u>Presence: Teleoperators and Virtual Environments, 1</u>(2), 219-228.

Olsen, Dan R. Jr. (1992). <u>User Interface Management Systems: Models and Algorithms</u>. San Mateo, California: Morgan Kaufmann.

Pausch, R., Crea, T., and Conway, M. (1992). A Literature Survey for Virtual Environments: Military Flight Simulator Visual Systems and Simulator Sickness. <u>Presence: Teleoperators and Virtual Environments, 1</u>(3), 344-363.

Phillips, C. B. and Badler, N. I. (1988). Jack: A Toolkit for Manipulating Articulated Figures. In <u>Proceedings of the ACM SIGGRAPH Symposium on User Interface Software</u> (pp. 221-229). New York, NY: ACM Press.

Pimentel, K. and Blau, B. System Architecture Issues Related to Multiple-User VR Systems: Teaching Your System to Share. In Sandra K. Helsel (Ed.), <u>Proceedings of Virtual Reality '92. VR Becomes a Business</u> (pp. 125-133). Westport, CT: Meckler Publishing.

Pountain, R. (1991) ProVision: The Packaging of Virtual Reality. <u>Byte</u>. pp. 53-63.

Robinett, W. and Holloway, R. (1992) Implementation of Flying, Scaling, and Grabbing in Virtual Worlds. In Mark Levoy, Edwin Catmull, and David Zeltzer (Eds.), <u>Proceedings of Symposium on Interactive 3-D Graphics</u> (pp. 189-192). New York, NY: ACM.

Sense8 (1993) <u>Technical Brief: WorldToolkit™ Version 2.0.</u> Sausalito, CA: Sense8 Corporation.

Shaw, C., Liang, J., Green, M., and Sun, Y. (1992). The Decoupled Simulation Model for Virtual Reality Systems. In Penny Bauersfeld, John Bennett, and Gene Lynch (Eds.), <u>Proceedings of CHI '92. Striking a Balance</u> (pp. 321-328). New York, NY: ACM.

Swarz, M., Wallace, D., and Tkacz, S., The Influence of Frame Rate and Resolution Reduction on Human Performance. (1992). In <u>Proceedings of the Human Factors Society 36th Annual Meeting, volume 2</u>. (pp. 1440-1444).

The Virtual Environment and Teleoperator Research Consortium (VETREC) (1992). <u>Virtual Environment Technology for Training</u>. (BBN Report Number 7661). Cambridge, MA: MIT.

APPENDIX A: XLISP EXAMPLE

The following is a sample XLISP program that interfaces with Mercury to create a virtual world with a hilly terrain and a train riding on tracks. It illustrates the protocol used to communicate with Mercury and configure the data flow within it.

```
;;
;; merc-metro.fent
;;
;; Copyright (C) 1992  Washington Technology Center
;;
;; by Max Minkoff and Andrew MacDonald at the HITLab
;;
;; This runs the metro demo using only Mercury (and a little FERN)

;;--------------------
;; OVERVIEW
;;--------------------

;; Here is the layout of this file:

;; 1. Initialize
;;    A. variables
;;    B. entity-IDs
;;    C. Mercury

;; 2. Define support functions

;; 3. Setup Process loop for
;;    A. reading data from mercury
;;    B. processing joystick inputs
;;    C. moving train around


;;----------------------------------------------------------------------
;;                           INITIALIZATION
;;----------------------------------------------------------------------

;;------------------------
;; initialize variables
;;------------------------
(setq tooting nil
      driving nil
      riding nil
      train-sound-off nil
      speed 10.0
      old-train-6D #(#(0.0 1.8 0.0) #(1.0 #(0.0 0.0 0.0)))
      old-world-6D #(#(0.0 30.0 0.0) #(1.0 #(0.0 0.0 0.0))))
```

```
;;------------------------
;; Initialize entity IDs
;; These numbers are arbitrary.
;;------------------------
(setq train 2000
      terrain 2001
      rails 2002
      lake 2003
      trees 2004
      tunnel 2005
      horn 2006)

;; create body part entity IDs.
(setq source 1000    ;; the virtual sensor source
      head 1001      ;; the participant's head
      hand 1002      ;; the participant's hand (joystick)
      flytool 1003)  ;; the joystick attachment that represents flying.


;;------------------------
;; initialize mercury
;;------------------------

;; hg-init returns Mercury's current configuration, including:
;; available processors, renderers, and sensors;
;; valid attributes, types, and commands
;; (remember - hg is the atomic symbol for Mercury!)
(let* ((merc-config
      (hg-init "iris2"                      ;; machine to run merc
            :display-host "iris2"       ;; where to display merc's
                                        ;; window
            :binary "/home/mercury/bin/mercury"
            :config-file "/home/mercury/config/mercrc"))
       head-dataset)

  ;;-------------------------------------------------------------
  ;; create a dataset to store the current velocity.  This dataset
  ;; will be used by the movement processor.
  ;;-------------------------------------------------------------

  ;; hg-make-dataset takes a vector of dataset values, in this case only
  ;; one.  Each dataset value will be a vector of dataset type,
  ;; identifier, and value.  In this case, the type is "type" because
  ;; we'll be storing data here as opposed to this being an entity or a
  ;; sensor (see below), the identifier is "triple" since we'll be
  ;; storing a vector of three values, and the initial value is
  ;; #(.0 .0 .0) because we won't be flying at first.  identifiers were
  ;; reported originally by hg-init.

  (setq vel-dataset (hg-make-dataset
                  (vector
                   (vector "type" "triple" #(.0 .0 .0))
                   )))
```

```
;;----------------------------------------------------------
;; initialize body entity attributes
;;----------------------------------------------------------

;; Note that the first command uses a generic "hg-set-attr"
;; command which takes the attribute name as an argument.  After
;; that, we'll be using the built in functions such as
;; "hg-set-picture" which already know about the standard attributes.
;; Either way will work.

;; make a grey joystick picture for the hand entity
(hg-set-attr "picture-desc" hand "/home/data/dogs/wands/joystick.dog")
(hg-set-attr "color" hand #(0.8 0.8 0.8))

;; make a green flytool and attach it to the joystick.
(hg-set-picture flytool "/home/data/dogs/wands/flyring.dog")
(hg-set-color flytool #(0.28 1.0 .27))
(hg-set-origin flytool hand)

;; move the source to 30 meters above the terrain to start
(hg-set-attr "6D" source #(#(0.0 30.0 0.0) #(1.0 #(0.0 0.0 0.0))))


;; bind sensor datasets to entity attributes.
;; sensor-find-tag returns the sensor id associated with the
;; names in the config files.  After this, the head and the hand
;; source values will be directly attached to the "6D" attributes
;; of the corresponding entities.
;; Note that dataset 0 contains all raw sensor inputs.
(hg-bind-attribute head "6D" (vector "dataset" 0
                                     (sensor-find-tag "head")))
(hg-bind-attribute hand "6D" (vector "dataset" 0
                                     (sensor-find-tag "hand")))

;; origin sensors to virtual source
(hg-set-origin head source)
(hg-set-origin hand source)


;;----------------------------------------------------------
;; bind output of movement processor to the source's 6D
;;----------------------------------------------------------

;; hg-add-processor takes an entity-ID, an attribute, the processor
;; name, and a vector of inputs to the processor.  Each input is
;; either an entity attribute vector:
;;                    #("entity" entity-ID attribute-name)
;; or a dataset vector: #("dataset" dataset-ID index).

;; the movement processor will use the source's 6D, the value in
;; the velocity dataset, and the time sensor.
;; sensor-find-tag finds the number associated with the "time"
;; name.
```

```
    (hg-add-processor source "6D" "movement"
                  (vector (vector "entity" source "6D")
                          (vector "dataset" vel-dataset 0)
                          (vector "dataset" 0 (sensor-find-tag "time")))))


  ;;-----------------------------------------------------------
  ;; setup imager
  ;;-----------------------------------------------------------

  ;; create a dataset with the head's world 6D in it
  (setq head-dataset (hg-make-dataset (vector
                                  (vector "entity" head
                                          "world-6D"))))

  ;; turn on the renderer and tell it where to get the data it needs -
  ;; the world 6D of the head
  (hg-bind-renderer (renderer-find-tag "HitlImager") head-dataset)


  ;;-----------------------------------------------------------
  ;; setup data packet
  ;;-----------------------------------------------------------

  ;; create a dataset to store the whole packet of information that
  ;; will be send to the application
  ;; "left", "middle", "right", and "trigger" are names associated
  ;; with the joystick buttons.
  (setq packet-dataset
      (hg-make-dataset
       (vector
        (vector "entity" source "world-6D")
        (vector "entity" hand "6D")
        (vector "dataset" 0 (sensor-find-tag "left"))
        (vector "dataset" 0 (sensor-find-tag "middle"))
        (vector "dataset" 0 (sensor-find-tag "right"))
        (vector "dataset" 0 (sensor-find-tag "trigger"))
        )))

  ;; tell mercury to use this dataset as the packet
  (setq merc-packet (hg-specify-packet packet-dataset))
  )

;;------------------------------
;; initialize background
;;------------------------------
;; using the "set-void-color" command, set the background color
;; in the graphical renderer.
;; like attributes, certain standard commands have been wrapped in
;; their own functions.
(hg-command (command-find-tag "set-void-color") #(0.2 0.2 0.9))
pp
```

```
;; set up all of the environmental entities attributes.
(hg-set-picture train "/home/data/dogs/train.dog")
(hg-set-color train #(0.0 0.2 0.5))
(hg-set-sound train "tracksout")
(hg-set-loudness train 0.6)

(hg-set-picture terrain "/home/data/dogs/Train_demo/train_terrain.dog")
(hg-set-color terrain #(0.0 0.25 0.0))

(hg-set-picture lake "/home/data/dogs/Train_demo/train_lake.dog")
(hg-set-color lake #(0.0 0.2 0.5))

(hg-set-picture rails "/home/data/dogs/Train_demo/train_rails.dog")
(hg-set-color rails #(0.3 .16 .16))

(hg-set-picture tunnel "/home/data/dogs/Train_demo/train_tunnel.dog")
(hg-set-color tunnel #(.3 .3 .3))

(hg-set-picture trees "/home/data/dogs/Train_demo/train_trees.dog")
(hg-set-color trees #(0.0 0.2 0.0))

(hg-set-picture horn "/home/data/dogs/horn.dog")
(hg-set-color horn #(.7 .2 .2))
(hg-set-sound horn "horn")
(hg-set-loudness horn 0.0)
(hg-set-origin horn train)
(hg-set-6D horn #(#(-.65 1.4 2.4) #(1.0 #(0.0 0.0 0.0))))




;;------------------------------------------------------------------
;;                          DEFINITIONS
;;------------------------------------------------------------------

;;-------------------------------------------------
;; make a velocity from the given 6D, turbo flag,
;;    and direction (-1.0 -> forward; 1.0 -> reverse).
;;-------------------------------------------------
(defun make-velocity (ppq dir)
  (let ((point
          (vector 0.0 0.0 (* dir speed)))
        (result #(0.0 0.0 0.0))
        (quat (if ppq
                  (aref ppq 1)
              #(1.0 #(0.0 0.0 0.0)))))
    (pointxquat point quat result)
    result))

;;-------------------------------------------------
;; set the sensor source's velocity
;; use hg-poke-dataset to change the value in the dataset
;;-------------------------------------------------
(defun fly (wand-6D direction)
  (hg-poke-dataset vel-dataset 0 (make-velocity wand-6D direction)))
```

```
;;----------------------------------------------
;; horn controls
;;----------------------------------------------
;; to turn on the horn
(defun toot ()
  (hg-set-loudness horn 1.0))

;; to turn off the horn
(defun untoot ()
  (hg-set-loudness horn 0.0))


;;----------------------------------------------
;; collision detection
;;----------------------------------------------
(defun test-in (test-pt x-low x-high
                    y-low y-high
                    z-low z-high)

  ;; test to see if the hand is within the boundaries
  (and (< x-low (aref test-pt 0) x-high)
       (< y-low (aref test-pt 1) y-high)
       (< z-low (aref test-pt 2) z-high)))


;;----------------------------------------------
;; test to see if the hand has collided with the horn
;;----------------------------------------------
(defun horn-test (hand-pt)
  ;; test to see if hand is the horn
  (let ((this-test (test-in hand-pt -.75 -.5 1.3 1.5 2.2 2.5)))
    (cond ((and this-test (null tooting))
           (toot))
          ((and tooting (null this-test))
           (untoot)))
    (setq tooting this-test)))


;;----------------------------------------------------------------
;;                      PROCESSING
;;----------------------------------------------------------------


;;----------------------------------------------
;; process hand data received from Mercury
;;----------------------------------------------

;; we want to remember the old state of the middle button so that
;; if it was on last time we won't keep flipping on and off
;; the train.  I.e. we only use new button presses
(setq old-but-1 nil)

(defun process-hand (hand-6D buttons source-6D)
```

```
;;-----------------------------
;; get off and on train
;;----------------------------
;; if you press the middle button the participant toggles
;; between riding the train (being origined to it) and
;; flying in the world.
(if (and (aref buttons 1) (null old-but-1))
    (if riding
        ;; if you're currently on the train
        (progn

            ;; stop riding the train
            (hg-set-origin source nil)

            ;; save your current position on the train
            (setq old-train-6D (aref (aref merc-packet 0) 0))

            ;; set your position to your old position in the world
            (hg-set-6D source old-world-6D)

            ;; remember that we're not riding anymore
            (setq riding nil)

            ;; reset the speed for flying
            (setq speed 10.0))

      ;; if (not riding) - you're currently flying in the world
      (progn
        ;; ride the train
        (hg-set-6D source train)

        ;; save your current position in the world
        (setq old-world-6D (aref (aref merc-packet 0) 0))

        ;; set your position to your old position on the train
        (hg-set-6D source old-train-6D)

        ;; remember that we're riding now
        (setq riding t)

        ;; reset the speed for on the train
        (setq speed 1.5))))

;; remember current state of middle button for next time
(setq old-but-1 (aref buttons 1))

;;------------------------------------------------
;; test to see if the participant is blowing the horn
;;------------------------------------------------
;; only works when the wand trigger is pulled and
;; the user is riding the train.
```

```
(if (and (aref buttons 3) riding)
     (progn
       ;; get hand position in the train's coordinate system.
       ;; Remember that the source and the horn are both
       ;; origined to the train - i.e. they are both in the
       ;; train's coordinate system.
       (setq tr-hand-pt
             (aref (composeposquats source-6D hand-6D) 0))
       (horn-test tr-hand-pt)))

  ;;----------------------------
  ;; fly
  ;;----------------------------
  ;;    left button flies forward
  ;;    right button flies backward
  ;;    pass a -1.0 for forward and 1.0 for reverse

  (cond
   ;; fly forward if left button is pressed
   ((aref buttons 0)
    (fly hand-6D -1.0))

   ;; fly backward if right button is pressed
   ((aref buttons 2)
    (fly hand-6D 1.0))

   ;; otherwise, turn off velocity
   (t
    (hg-poke-dataset vel-dataset 0 #(.0 .0 .0))
    )))


;;------------------------------------------------------------
;; data receive loop
;;------------------------------------------------------------
;; this is the function we will loop with to continually retrieve
;; the merc-packet and operate appropriately

(defun get-merc-data ()
  (if (hg-get-data merc-packet)
      (process-hand
        (aref merc-packet 1)
        (vector (aref merc-packet 2) (aref merc-packet 3)
                (aref merc-packet 4) (aref merc-packet 5))
        (aref merc-packet 0))
    ))

;;------------------------------------------------------------
;; move the train - the guts of how the train moves is irrelevent
;; for our purposes.  Notice that the train only updates itself
;; 10 times per second, even if it could go faster.
;;------------------------------------------------------------
(set-up-train)
```

```
;; position train
(hg-set-6D train (setq train-6d (2d-ppath-calc tr-path 0.0 0.0)))

(init-time)

(setq tr-base-time (read-total-time))

;; this is the looping function the train will use for movement
(defun train-persist ()
  (setq tr-elapsed-time (- (read-total-time) tr-base-time))

  ;; test to see if .1 seconds have passed since last update
  (if (>= tr-elapsed-time 0.1)
      (let ((tr-dist (+ tr-old-dist (* tr-elapsed-time tr-speed))))
      (hg-set-6d train (setq train-6d
                         (2d-ppath-calc tr-path tr-dist tr-old-dist)))
      (setq tr-old-dist tr-dist)
      (setq tr-base-time (+ tr-base-time tr-elapsed-time)))))


;;--------------------------------------------------------------
;;  persist procedures
;;--------------------------------------------------------------
;; set up the persist procs, i.e. the main loop to process as
;; fast as possible.
(fern-persist '(progn
             (get-merc-data)
             (train-persist)))
```