Decoupled Simulation in Virtual Reality with The MR Toolkit *†

Chris Shaw, Mark Green, Jiandong Liang and Yunqi Sun

Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2H1
{cdshaw,mark,liang,yunqi}@cs.ualberta.ca

Abstract

The Virtual Reality (VR) user interface style allows natural hand and body motions to manipulate virtual objects in 3D environments using one or more 3D input devices. This style is best suited to application areas where traditional two-dimensional styles fall short, such as scientific visualization, architectural visualization, and remote manipulation. Currently, the programming effort required to produce a VR application is too large, and many pitfalls must be avoided in the creation of successful VR programs. In this paper we describe the Decoupled Simulation Model for creating successful VR applications, and a software system that embodies this model. The MR Toolkit simplifies the development of VR applications by providing standard facilities required by a wide range of VR user interfaces. These facilities include support for distributed computing, head-mounted displays, room geometry management, performance monitoring, hand input devices, and sound feedback. The MR Toolkit encourages programmers to structure their applications to take advantage of the distributed computing capabilities of workstation networks improving the application's performance. In this paper, the motivations and the architecture of the toolkit are outlined, the programmer's view is described, and a simple application is briefly described.

CR Categories and Subject Descriptors: D.2.2 [Software Engineering]: Tools and Techniques - software libraries ; user interfaces ; H.5.2 [Information Interfaces and Presentation]: User Interfaces - input devices and strategies ; theory and methods; I.3.6 [Computer Graphics]: Methodology and Techniques - device independence; interaction techniques; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - virtual reality;
General Terms: Design, Human Factors
Additional Key Words and Phrases: User Interface Software, Interactive 3D Graphics.

---

*This paper appears in *ACM Transactions on Information Systems*, Vol 11, Number 3, 287-317, July 1993
†A preliminary version of this paper was presented at the ACM SIGCHI 1992 Conference at Monterey, California in May 1992.

1 Introduction and Motivation

The Virtual Reality (VR) user interface style involves highly-interactive three-dimensional control of a computational process. The user enters the virtual space of the application, and views, manipulates and explores the application data using natural 3D body motions. The key advantage of this user interface style is that the user's instinctive knowledge of the physical world can be transferred to manipulations in the virtual world. To support this type of interaction, the user typically uses non-traditional devices such as head-mounted displays (HMDs) and hand measurement equipment (gloves). These devices give the user the impression that the application is running in a true 3D space. Application objects can be manipulated by normal hand motions such as pointing and grabbing. VR has been successfully used in a wide range of applications including architectural and design visualization [15], scientific visualization [1, 4], remote manipulation [9], and entertainment (e.g., the W industries Virtuality game). While the utility of the VR style has been demonstrated by these applications, new interaction techniques and software tools to support VR must now be developed.

A wide range of software development tools exist for the more traditional WIMP (Windows, Icons, Menus, and Pointing) style of user interface. These tools include graphics packages, window systems (such as X), toolkits (such as Motif and OpenLook) and UIMSs. These tools significantly simplify WIMP user interface development. VR user interfaces need a similar set of software development tools.

This paper describes our VR toolkit, the Minimal Reality (MR) Toolkit. We have four main reasons for developing this toolkit:

1. Investigate the software architectures that can be used to support VR user interfaces on currently available hardware. Very little general purpose software has been produced to support the development of VR user interfaces. By producing this toolkit we hope to provide one software architecture that effectively supports VR.

2. Support testing of new devices. The continuing development of new devices for VR user interfaces demands that they be evaluated in real applications. The MR Toolkit allows the easy integration of new devices into existing applications to evaluate their effectiveness.

3. Provide a testbed for developing and evaluating new interaction techniques for VR user interfaces. The MR Toolkit provides a framework for developing new interaction techniques, and for integrating them with existing applications without developing large amounts of extra software. The MR Toolkit also provides a means of evaluating several competing interaction techniques in the same environment.

4. Provide a usable tool for the development of VR applications. The production of good VR user interfaces requires a considerable amount of programming. The MR Toolkit significantly reduces the amount of programming required to produce a good user interface and allows one to tackle a wider range of applications.

We have taken the approach of developing a toolkit instead of a higher level tool, such as a UIMS, for two reasons. First, developing a high level tool requires a good understanding of the types of interactions that the tool must support. For example, high level tools for WIMP user interfaces didn't appear until researchers and developers had considerable experience in developing user interfaces based on this style. At the present time the VR style of interaction isn't well enough understood to consider developing high level tools. Second, high level tools, such as UIMSs, often rely on toolkits to provide low level support. In the case of WIMP user interfaces, UIMSs produce code calling the routines in one or more standard toolkits. High level tools for VR user interfaces will likely require the use of a toolkit like MR.

In developing the MR Toolkit, we built several complex sample applications, such as the fluid dynamics visualization example outlined in section 4.1. We found that proper structuring of the application is vital to its success. A key goal in structuring complex applications is ensuring that the simulation component of the application doesn't slow down the user interaction, making it unusable. The MR Toolkit represents our structural model of VR applications, called the Decoupled Simulation Model (DSM), which enables VR application developers to create successful VR interfaces to complex simulations.

Once a VR application has been produced, it may need to be distributed to other sites. Portability is the key software issue. We have been distributing the MR Toolkit to other sites since September 1991, and this experience has shown us a list of system dependencies that must be considered to ensure application portability. The MR Toolkit encourages a program structure that increases application portability.

2 Previous Work

Zeltzer and his colleagues at MIT produced a general purpose package called `bolio` for building interactive simulation systems [24]. In `bolio`, a constraint network connects all the objects in the application, including the DataGlove. The DataGlove-dependent constraints evaluate the current gesture, causing subsequent constraints to be executed, producing the reaction to the user's activity, which is displayed. One disadvantage of this system is that the simulation is single-threaded, meaning that the visual update rate depends upon how fast the constraint satisfier runs.

Card, Mackinlay and Robertson at Xerox produced an architectural model for VR user interfaces called the Cognitive Coprocessor Architecture [18]. The purpose of this architecture is to support "multiple, asynchronous, interactive agents" and smooth animation. It is based on Sheridan's Three Agent Model of supervisory control and interactive systems [21]. These agents are: the User, the User Discourse Machine and the Task Machine, or application.

The basic control mechanism in the Cognitive Coprocessor Architecture is the animation loop, which has a task queue, a display queue, and a governor. The task queue maintains all the incoming computations from different agents; the display queue contains all the objects to be drawn; while the governor keeps track of time and helps the application produce smooth output. The animation loop starts with processing the user's input and other tasks that are independent of the user's activities. Then it processes tasks in the task queue until the queue is empty. The loop ends with drawing all the objects in the display queue. This architectural model is similar to our DSM (outlined in section 4.2).

VPL Research sells a software toolkit with their RB2 system [2] that uses multiple workstations and a visual programming system to manage device data. A central server drives the devices and sends device and polygonal model data to one or two pairs of graphics rendering machines. Unfortunately, the visual programming facility reported by Blanchard et al. [2] is limited to data flows between predefined data transformers on the central machine. Subsequent marketing literature indicates that new data transformation nodes can be programmed in C.

Koved, Lewis, Ling and their colleagues at IBM have been using multiple workstations to support the real-time requirements of VR user interfaces [1, 8]. Their VUE system assigns a workstation to each of the devices in their user interface, including a server process for each graphics renderer. The MR Toolkit uses a similar input device management approach, as described in section 6.1 of this paper. Simulation processes, such as the Rubber Rocks demo [8], are also servers that communicate with the renderer server(s).

The top level of VUE is an event-based UIMS that coordinates the input events coming from device and simulation servers, and routes transformed data to the output servers. At the UIMS core is a dialogue manager containing a set of rules that fire when the appropriate event arrives. A rule, containing fragments of C code, can dispatch new events to the UIMS, and to any of the servers. The application model embodied by the VUE system is similar to the DSM outlined in section 4.2.

The primary benefit of the VUE UIMS is that the presentation component style can be separated from the content. For example, one user may use a 3D tracker to send 3D position events to the UIMS, while another may use screen-based sliders. The dialogue manager doesn't care about the source of the events, allowing a high degree of flexibility in application presentation.

Researchers at UNC at Chapel Hill [19] have developed a general purpose 3D tracker library that presents a unified programmer's model of a tracker, thus hiding device-dependent details. The MR Toolkit also presents a unified view of a tracker, but also allows the device drivers to be distributed over many machines.

Bryson and Levit at NASA Ames [4] have been working on VR techniques for visualizing steady and unsteady fluid flows. They use the Three Agent Model to develop applications. Jacoby and Ellis, also at NASA Ames [14], have evaluated various types of virtual menus, planar menus in three-space that use a ray shot from the hand for selection. The intersection point of the ray with the menu is the 2D cursor, and menu selection can be performed using a hand gesture. The panel package in the MR Toolkit provides a similar facility.

The Sense8's WorldToolKit (WTK) provides an object-oriented modeling system allowing the display and animation of hierarchical collections of objects. WTK also provides higher-level interactions such as terrain following and collision detection. Sense8 supplies drivers for a wide range of devices, but does not support distribution of the devices across multiple machines on a local area network. This system is most useful on platforms with minimal 3D graphics support, such as IBM PCs.

Division's dVS software also provides object-oriented modeling facilities, with explicit support for multiple coordinating processes on multiple processors. The event-based central simulation engine accepts state changes from any object and forwards this new information to all other objects in the system. Similar to `bolio`, the DataGlove and HMD are objects in the system that can add events to the simulation, but visual update doesn't rely on the complete evaluation of the DataGlove's events before anything is drawn. Similarly, the HMD rendering object draws all other objects based on their state and the HMD's location.

Both WorldToolKit and dVS maintain an internal geometric model of the objects, since the hardware platforms these toolkits use don't support 3D graphics. The advantage of an internal geometric modeling system is that it allows a feature like collision detection to be implemented as part of the toolkit. The disadvantage is that it can be restrictive, making some applications harder to write. The MR Toolkit does not provide an internal geometric model, relying instead on the workstation's graphics library to perform 3D rendering. Future extensions of the MR Toolkit will provide a higher-level modeling facility to support automatic simulation.

Our own previous work addressed the problem of distributing the low level device software over multiple workstations and the production of a skeleton VR application [13, 20]. The toolkit described in this paper is an extension of this work.

3 Requirements

We have identified nine major requirements based on our own experiences with developing VR applications [13, 20] and the experiences of others [12], five related to the interactive performance of VR applications, and four related to issues of writing software for virtual environments.

3.1 User Interface Requirements

The following five requirements are properties that a VR application must have in order to be both usable and pleasing to the user. If these requirements are not met, the application will not be a success since the user won't tolerate the application's performance for very long.

1. VR applications must generate smoothly animated stereoscopic images for HMDs to maintain the key VR illusion of *immersion* – the sense that the user is "really there". To appear *smooth* [1, 18], the visual update rate must exceed 10 updates per second, based on Card, Moran and Newell's *Middleman* perceptual performance metric [6]. The application structure must provide a high visual update rate, independent, if possible, of the application update rate, since the application could take more than 100 milliseconds to update. This capability isn't directly supported by most commonly-available 3D graphics workstations.

2. VR applications must react quickly to the user. If the lag between movement and image update is too long, users change their behavior, either by slowing down their movements, or by adopting a move-and-wait motion strategy [21, 22]. If the user must micro-plan his or her actions, then the user interface has failed because it is no longer interactive. A *responsive* [1] VR system must have image lags of under 100 milliseconds, again based on *Middleman* [6, 7].

3. Provide support for distributing an application over several processors. When a HMD is used, one workstation is used in our VR applications to produce the images for each user's eye. Distributed computing expertise shouldn't be necessary to produce efficient VR user interfaces. The underlying toolkit should provide facilities that manage the distribution of computations over several processors, ensuring that each process in the application gets the data it needs.

4. An efficient data communications mechanism is needed, since most VR applications use multiple workstations. Most communication involves data that is shared between two of the processes in the application. For example, the two HMD rendering processes must share application data. In addition, for rapid display update, data communications must be as efficient as possible. The toolkit should hide as many data communications details as possible so the programmer can concentrate on developing the user interface.

5. Performance evaluation of VR applications is needed. Existing performance monitoring tools are not very helpful for two reasons. First, the programmer needs to know where the real time is spent, not just the CPU time, since in a VR application, a process that waits for I/O or interrupts is just as bad as a process that spends its time computing. Second, since the application is distributed over several processors, the loads on the individual processors must be taken into account, along with communications delays. A VR toolkit must provide mechanisms for monitoring both the real time used by a process and the total performance of the application.

By far the most important requirements for a successful VR interface are the need for high visual update rate and low visual lag to convey immersion in the environment. Requirements that flow from these are support for distributed computation, efficient data communications, and evaluation of real-time performance.

Maximizing update rate and minimizing lag are two related but not identical challenges. The update rate of a particular set of geometric data depends strongly on the rendering power of the 3D graphics system, and also on the presence of other loads, and on system throughput. The usual strategy for maximizing update rate is to reduce graphics rendering time, which sometimes involves simplifying the geometric model.

Overall visual lag is the sum of tracker lag, system lag (including synchronization waits) and graphics update lag. The strategies for dealing with lag are to minimize tracker and software system lags, and to compensate for lag by predicting user actions. Our approach to lag compensation for HMDs is to predict future tracker orientation using a Kalman filter [16].

In some cases, the application causes high image lag and low update rate. For real applications, with update rates below 10 Hz, smooth animation is impossible unless the VR software eliminates unnecessary synchronization of the visual update with the application. In most cases, the application doesn't require the user to work at the application's update rate, so the visual update rate can be different from the application update rate. If the user sees an application updating 2 Hz while the visuals update at 20 Hz, the feeling of immersion is preserved, because the virtual environment acts visually like the real world.

3.2 Software Engineering Requirements

The four requirements outlined in this section deal with the ease of programming VR applications. These requirements became clear to us when we rearranged the hardware in our lab to accommodate new equipment, and distributed the MR Toolkit to other sites. They all flow from the need for software portability.

1. Portability of applications from one site to another. VR applications are too often strongly tied to the environment they were developed in. Porting an application to another site should at most require a recompilation of its source code, not an extensive retooling. A VR software system should not restrict the range of possible applications in order to meet this goal. For example, a VR toolkit that supports only VR-style polygon mesh viewing is likely to be portable, but not very useful.

2. Support for a wide range of input and output devices. The hardware technology used in VR applications has not matured to the point where there is a standard set of devices an application can assume

all workstations have. A VR toolkit should be able to accommodate newly developed devices and facilitate their addition to the package. Low-level support for new devices such as position trackers and gloves must be efficient and minimize lag, while high-level abstractions are required by the application programmer.

3. Independence of applications from room geometry and device configurations. Trackers used in VR applications use their own coordinate systems, depending upon where they are located in the room. Three-dimensional output devices also have their own coordinate systems. In order to use these devices, an application must transform coordinates they report to a standard coordinate system, corresponding to the application coordinates. When the device configuration changes, either by rearranging the room, or porting the hardware or software to a new site, this coordinate transformation must be updated. A VR toolkit should handle geometric configuration so that applications are not dependent on room geometry and physical device location.

4. A flexible development environment for VR applications. Quite often a VR application is developed in a different environment than the one it will be used in. Two workstations are required to drive a HMD, but during program development the programmer may not want to monopolize two workstations. Application developers can use a single screen and then move to the HMD near the end of the development cycle. Flexibility is needed when critical resources such as tracker devices and graphics screens are needed by more than one user at a time. The toolkit should facilitate this type of development by requiring only minor changes to the code to move from a single display to a HMD. Similarly, during program development the programmer may want to try different devices to determine the optimal device set for the application. The toolkit should support this type of experimentation without requiring the programmer to change more than a small amount of code.

The key software engineering issues are portability from various site and software dependencies, and flexibility in the development of VR applications. Site dependencies include communications setup and geometric configuration, and if these are not accounted for in a flexible manner, the porting effort can be large. A VR Toolkit should also support the ability to easily switch from one hardware configuration to another as conflicts arise.

4 Structuring Applications

The DSM can be used to structure a VR application for smooth animation. We illustrate our discussion using a scientific visualization example.

4.1 Fluid Dynamics Example

We have written a simple fluid dynamics user interface using the MR Toolkit. This user interface forms the front-end to an existing fluid dynamics program written in FORTRAN [5]. It provides the user with a 3D stereo view of the fluid surface using the HMD. The user can walk through the flow to view it from different directions. The DataGlove interacts with the flow and establishes boundary conditions for the computation. The user makes a fist to impose a force on the surface of the fluid. The closer the glove is to the surface, the greater the force. When the glove exits the fist gesture, the force boundary condition is turned off, and the fluid returns to equilibrium (flat surface).

The simulation code of the fluid dynamics example runs on a CPU server at its own update rate. It accepts boundary condition commands from the DataGlove at the beginning of each simulation time step, and sends force vector data at the end of each time step to the machines that draw the fluid's surface. The images in the HMD update at 20 Hz, while the simulation is able to compute at 2 Hz.

The user can explore a slow-moving simulation with limited lag, since the wait from head motion to image update is only 50 milliseconds. Similarly, the image of the DataGlove is also updated using its most recent position and orientation. Only the simulation runs slower than 20 Hz, since its maximum speed is 2 Hz.

Photographs Go HERE

Figure 1: Two fluid dynamics interfaces: (a) The DataGlove puts a force boundary condition on the point on surface marked by the arrow. DataGlove X and Y location and current surface height are noted on the text panel. (b) X, Y and Pressure are input using sliders controlled by the DataGlove. The X and Y location is marked on the surface with a circle on the left. The DataGlove doesn't interact directly with the surface.

It is important to distinguish between the two levels of user interface design that are present in the fluid dynamics application. The two levels are the *semantic* design and the *syntactic* design [10]. Semantic design has to do with the functionality of the application. In the fluid dynamics application, the semantic input is the boundary condition on the simulation, and the semantic output is the surface of the fluid.

Syntactic design defines the form that the application's commands take, and the form that the output from the application takes. To illustrate this, the fluid dynamics application can take many forms, two of which are shown in figure 1. In the top picture, when the user makes a fist, the DataGlove puts a force boundary condition on the surface point marked by the arrow. The DataGlove X and Y location, snapped to surface grid coordinates, are output on the text panel, along with the current surface height. In the second interface, figure 1(b), the X, Y and Pressure values are input using sliders controlled by the DataGlove. The X and Y location is marked on the surface with a circle. The DataGlove doesn't interact directly with the surface.

These two examples have the same semantics but different syntax. Traditionally, the separation of semantic processing from syntactic processing allows programmers to create different user interfaces to the same application, or to customize the user interface without having to change the application code. This encourages modularity and code reuse.

This separation is especially important in VR user interfaces, because there are also strong *temporal* constraints on user interface behaviour. Recalling the requirements from section 3.1, a VR user interface must have smooth animation and low image lag. In the fluid dynamics example, the application is updated at 2 Hz, which is unsatisfactory. However, these two examples can be visually updated at 20 Hz, because the scene can be drawn fast enough, and the user interface has no temporal dependence on the application.

Temporal decoupling can be achieved by structuring a VR application such that the visual update rate doesn't depend on the simulation update rate. Our model for structuring VR applications to accomplish this is called the DSM.

4.2 The Decoupled Simulation Model

The Decoupled Simulation Model structures a VR application so that the user perceives smooth animation of the environment. It is primarily intended as a guide for programmers to build VR applications that meet the requirements outlined in section 3.1. It does this by breaking the application into the four components shown in figure 2. Some applications are simple enough to require only the Presentation component and the Interaction component, while others require all four parts. The arrows represent information flows within the system, again with the proviso that some flows are quite minimal in simpler applications. We call this the Decoupled Simulation Model, because the Computation component proceeds independently and asynchronously of the remaining components.

The Computation component contains the application's core, and is usually a continuously running simulation, such as that of section 4.1. This component runs independently of the rest of the system, and manages all non-graphical computations. It is the application programmer's main concern. Typical simulations evaluate a computational model in a series of discrete time steps, periodically outputting the model's state. When a time step is complete, the Computation component forwards its output to the Geometric Model component. The Computation component receives syntactically valid user commands as input from the Interaction component, and updates the application data asynchronously from the Presentation component's visual and other update rates.

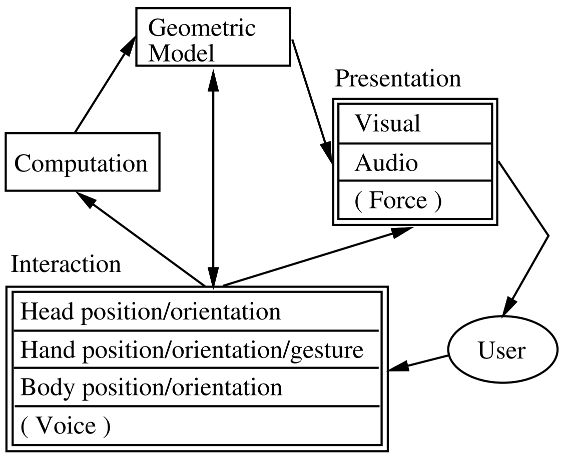

Figure 2: The Decoupled Simulation Model

The Interaction component is responsible for managing all input from the user, and coordinating all output to the user. It manages at a high level the input devices available to the user, and dispatches commands to output devices based on user actions. The sub-boxes in the Interaction component of figure 2 indicate that multiple input devices are used. The parenthesized items indicate items that the MR Toolkit doesn't currently support, but can be supported in the toolkit framework.

In the first fluid dynamics example, the Interaction component is responsible for creating and updating the panel with the current X and Y position of the DataGlove, and the Height of the surface at that point. The position of the arrow from the DataGlove to the surface is determined by the Interaction component. In the second example, the Interaction component manages the X, Y, and Pressure sliders using the DataGlove, and determines the position of the circle on the surface.

The Geometric Model component maintains a high-level representation of the data in the computation. This component is responsible for converting the application data in the Computation component into a form amenable to visual, sonic, and force display. One important function of the Geometric Model component is simplifying the geometric model of the application in cases where the current model is too complex for real time display. This type of simplification on the fly could arise from two sources, an increase in geometric complexity, or an increase in angular velocity of the HMD. In the second case, it may increase user satisfaction to simplify the model while the head is moving quickly to eliminate strobing effects in the dynamic images. One example of geometric simplification occurs in a geographic fly-through application, where more distant features of a geographic database are approximated by larger polygons than the closer features. This approximation process depends on the current update rate, so if the model is updating too slowly, it should be simplified, and if it updating faster than necessary, the complexity can safely increase.

The mapping from application data to geometric data can be static in simple applications, and can be controlled by the user. Like the Computation component, the Geometric Model component accepts time data and user commands from the Interaction component. Update time in the Geometric Model component is more important than in the Computation component, since one of the main functions of the Geometric Model component is to achieve smooth animation.

The other important function of the Geometric Model component is syntactic feedback to the user. In both of the fluid dynamics examples, the representation of the fluid surface as a rectangular mesh is performed by the Geometric Model component. When the Interaction component determines the height of the fluid underneath the DataGlove, it queries the Geometric Model for this information. It doesn't need to ask the Computation component for this information. Feedback to user input can be used to reflect user-proposed changes to the computational model that haven't been incorporated into the simulation. For example, the arrow in the first example tells the user that a force is being applied to the surface, and also tells exactly where it will be applied.

The Presentation component produces the views of the application data that the user sees, along with the sonic and force aspects of the application data.

In the visual domain, the Presentation component is the rendering portion of the application, taking input from the Geometric Model, and the viewing parameters (such as eye position) from the Interaction component. It also gets syntactic data from the Interaction component, such as the X, Y, and Height information, and the location of the arrow from the first example. The output is one or more images of the current application data from the current viewpoint. These images must be updated each time the application data or the viewing parameters change, and must be updated frequently enough to achieve smooth animation. When a HMD and DataGlove are used, the images must be updated as soon as the head or the hand(s) move. This is included in the Presentation component because these tasks are application independent, while syntactic feedback is application dependent.

In the sonic domain, the Presentation component presents sonic feedback and application sounds based on the application data and, if 3D sound is used, based on the user's head position and orientation. Again, sounds must be updated as the application data and/or the user's position and orientation change.

4.3 Discussion

Robertson, Card, and Mackinlay's [18] Cognitive Coprocessor Architecture is a single-process architecture that has one interaction loop updating the screen with database or application changes. The Task Machine's job is to animate changes from one database view to another, to animate database updates, and to indicate progress in the computation. Two central assumptions of this architecture are that the scene can be drawn fast enough, and that application updates do not take a long time. Of course, fast graphics update must be supported if true interactivity is to be achieved.

The assumption of fast application update isn't always true in practice, however, and our fluid dynamics example is a situation where the application simply isn't fast enough to support smooth animation. Robertson et al's answer to this is that the application should be changed to support finer-grained computation. This is a rather strong requirement on the application, and one that we feel isn't always justified, since some applications are difficult to modify.

In our approach, the update rate can be adjusted by the Interaction component through the Geometric Model component. If the images are updating too slowly, the Geometric Model component can be told to speed up. Importantly for "dusty deck" applications, this doesn't require a change in the application.

Also, there is no direct provision for an independent continuously-running simulation task in the Cognitive Coprocessor Architecture. Our model has two loops running asynchronously, and therefore has direct support for both discrete event and continuous simulation.

Robertson et al. explicitly use the Three-Agent Model [21] in the Cognitive Coprocessor Architecture. Researchers at NASA [4] implicitly structure their software using the Three-Agent Model, because they build their applications using a simulation machine communicating with a graphics machine. However, it isn't clear from viewing their videotapes that the visual update rate isn't slaved to some aspect of the application update rate.

The DSM is an elaboration of the Three-Agent Model that decouples the dependency of the visual update rate from the application update rate. The DSM splits the User Discourse Machine into the Interaction, Geometric Model, and Presentation components, since these components provide clearly usable functionality within most VR applications. This splitting of the User Discourse Machine helps programmers structure applications that have difficulty updating fast enough.

IBM's VUE system [1, 8] is closer to the spirit of the DSM, since VUE splits the application into simulation servers, output servers, and the dialogue manager. The simulation server corresponds to the Computation component, the output servers correspond to the Presentation component, and the dialogue manager corresponds to the Interaction component of the DSM.

The two important differences are that VUE doesn't have a component corresponding to the Geometric Model component, and that in VUE, there is only a weak temporal decoupling of the output servers from the simulation. Close single-frame examination of the Rubber Rocks videotape indicates that the time between visual updates depends on the complexity of the simulation. One example shows two hands in a room that updates at 15 frames per second, or 66 milliseconds per update. However, when one hand is in the room with 3 rocks, anywhere between 100 to 300 milliseconds (6 to 18 NTSC fields) may pass until the next visual update. Some updates will move just the hand, while others will move the hand and the rubber rocks. Clearly, since the application can render the scene within 100 milliseconds, and since the hands update within 66 milliseconds, the only cause of much longer update intervals is an unnecessary temporal dependence of the output servers upon the simulation, since one would expect that a new hand update would result in an update of the scene. The DSM eliminates this dependency of the visual update upon the simulation update.

In summary, the DSM is an evolution of the Three-Agent Model with specialized components such as the Geometric Model performing functions vital to the success of a VR interface. One should carefully note that DSM doesn't fix every problem of image lag, since syntactically valid commands sent to the Computation component will necessarily have to wait for it to execute them.

We feel that this is appropriate in most cases. Card et al. [7] refer to three human time constants that an interactive system must support: *perceptual processing*, *immediate response*, and *unit task*. The *perceptual processing* time constant is about 100 milliseconds, and is directly linked to human perceptual-motor loop times [6]. Operations that rely directly on the human perceptual-motor loop must have lags less than 100 milliseconds to be usable, such as the lag between head or hand motion and visual update. These types of operations fall into the category of syntactic feedback, since from the application's perspective, nothing of note is really happening. Syntactic feedback, such as the arrow highlight in the first fluid dynamics program, is also needed to help the user create well-formed commands,

The *immediate response* time constant is on the order of 1 second [17], and relates to the time needed to make an unprepared response to some stimulus. For example, if a computer responds within one second of typing return at a command-line prompt, the user concludes the computer is running. In the VR context, semantic input to the application (e.g., invoking a command by making a hand gesture) should generate a response within one second, otherwise the user will think that the application is dead. Operations in a VR application that pass through the Computation component require semantic feedback, and must respond within the time allotted for immediate response. In the fluid dynamics example, semantic feedback is required to indicate that a new boundary condition is being processed by the simulation. The surface of the fluid deforms showing that a semantic action has been taken.

The *unit task* response time constant is between 5 and 25 seconds, which is the time required to complete some elementary task. In the fluid dynamics example, running one small simulation to completion is an example of a unit task.

The MR Toolkit we describe in the remainder of this paper assists in the building of VR applications using the DSM. Some aspects of the DSM are supported in only a rudimentary way by the MR Toolkit. In particular, the programmer must build most of the Geometric Model, due to the strong dependence of this component on the application. For example, the MR Toolkit doesn't supply routines for simplifying geometry in the presence of low visual update rates, although it does detect that the scene is updating slowly.

The MR Toolkit does implement the decoupling of the application from the presentation. In fact, the prime intent of the DSM is to help the programmer to break a VR application into manageable parts, and we believe that a clear strategic guideline of this nature provides a solid basis for building VR applications. These guidelines are bolstered by the toolkit routines, which perform much of the drudgery entailed in managing a VR interface.

5 The MR Toolkit

An MR application is a collection of processes working together to implement the VR interaction style. The basic strategy is to collect the latest input from the user, send commands to the application, collect output from the application, and display the current state of the application with respect to the user's view position and orientation. Because the application may update slowly, if there is no new application data, MR displays the current (old) application data.

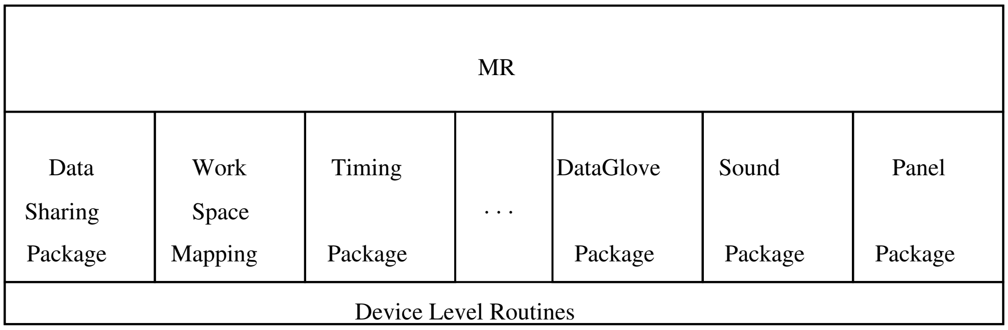

Figure 3: Three level software structure of the MR Toolkit

Our first software effort in this field was to create effective software for driving VR devices such as 3D trackers and the DataGlove. Our next step was to create an application skeleton for building simple VR user interfaces [13], and in light of this experience, we designed and built the MR Toolkit.

Our approach therefore is to build on a solid foundation, attacking more complex problems once the "simple" problems have been solved with some success. We believe that maintaining the interaction of two or more independent users working together requires a software system that adequately solves the problems for one person. This means that the requirements outlined in section 3 should first be met without confusing the issue with entirely separate problems such as resource contention between users, command conflicts, and communications lags between users. Therefore MR provides direct support for only a single user. The one-person model of interaction allows application developers to focus on the problems of person-to-application interaction independently from the problems of person-to-person interaction. We are developing a software system called the Peers package, which supports multi-person VR applications across both local area and wide area networks. The Peers package builds on the MR Toolkit to provide data and device sharing between users.

5.1 The Software Structure of The MR Toolkit

The MR Toolkit consists of three software levels, shown in figure 3. The bottom level consists of the routines that interact with the hardware devices supported by the MR Toolkit. The structure of the bottom level of the toolkit is described in section 6.1.

The next level of the MR Toolkit consists of a collection of packages. Each package handles one aspect of a VR user interface, such as providing a high level interface to an input or output device, or routines for sharing data between two processes. The standard packages, such as data sharing and performance monitoring, are always used by MR applications, and provide services that are required by all VR applications. Optional packages provide extra services required by some VR applications. Typically there is one optional package for each device supported by the MR Toolkit and for each group of interaction techniques. The use of packages facilitates the graceful growth of the MR Toolkit as new devices and interaction techniques are added. All of the packages are briefly described in sections 6.3 through 6.8.

The top level of the MR Toolkit, introduced in the next section, consists of the standard procedures that are used to configure and control the complete application. The routines at this level provide the glue that holds the MR Toolkit together.

5.2 Application Process Structure

This section describes how the programmer approaches user interface development using the MR Toolkit. Our toolkit is more flexible than indicated in this explanation, and the programmer isn't forced to follow this program structure. But, following this structure simplifies program development and allows the programmer

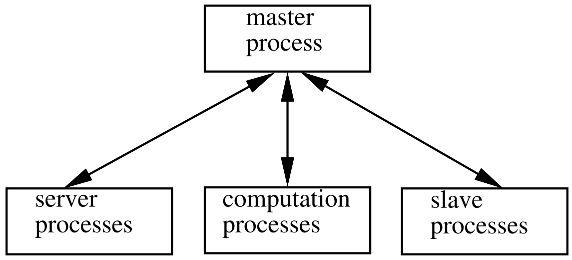

Figure 4: Process Structure of an MR Application

to take maximum advantage of the facilities the toolkit provides. A complete description of the MR Toolkit is in The MR Toolkit Programmer's Manual [11].

The MR Toolkit imposes a process structure on applications that use it. An MR application consists of one or more UNIX-style processes, with one designated as the master process, and the others as slave or computation processes. The designation *master*, *slave* or *computation* is called the *role* of the process. An MR application also establishes connections to one or more *server* processes, each uniquely responsible for the management of an I/O device such as a DataGlove, a position tracker, or sound output.

The usual purpose of a slave process is to perform output tasks. For example, a HMD requires two sources of NTSC video to simultaneously supply a distinct image for each eye. If special hardware isn't available to supply these signals from a single graphics machine, then two graphics machines supply them in synchrony. In this case, the master process on one machine supplies one image, and a slave process on another machine generates the other. Slave processes are part of the Presentation component of the DSM.

A computation process corresponds to the Computation component in the DSM. Computation processes do not require direct interaction with the user. In the fluid dynamics example, the computation process performs a fluid dynamics calculation while the user interacts with the results of the previous time step. The distinction between slave and computation processes assists the application programmer in structuring the application, and in the future the MR Toolkit may perform optimizations that depend upon the type of process.

The master process performs the duties of the Interaction component and the Geometric Model component, along with any graphics element of the Presentation component that can reside in the master's local machine. While this tight binding of graphical display and interaction isn't required, it is desirable for the purposes of reducing lag. The master process is the first process started and is responsible for initiating the slave and computation processes and establishing the communications with these processes. The master process is also solely responsible for establishing communications with the server processes. The MR Toolkit supports a limited version of distributed computing in which the slave, computation and server processes can communicate with the master process, but they cannot directly communicate with each other. This process structure is illustrated in figure 4.

The MR Toolkit detects the process role automatically when the master and slave processes use the same source code. For a computation process, the program must call a role-setting procedure. All of the procedures within the MR examine the role of the calling process before they perform any actions. In this way MR procedures can ignore calls that are not appropriate for a particular role, or perform actions that depend upon the role of the program. Similarly, the application code can determine the process's role by making a procedure call. The use of one set of source files for all the processes in an application solves a number of software maintenance problems.

5.3 Configuration Section

An MR program is divided into two main sections. The *configuration section* describes the structure of the application, the data passed between the processes in the application, and the devices used by the application. The *computation section* contains the code for the computations that the process performs.

The configuration section of the program starts with a procedure call that initializes internal MR data structures and determines the process's default role. Next, slave and computation processes are declared. The declaration procedures return a pointer to a data structure containing all the information required to access the slave or computation process from inside the current process.

Shared data items are declared next. A shared data item is a data structure declared and allocated in both the master process and one of the slave or computation processes. Shared data provides a network-based inter-process communication mechanism between the master and the other processes. A shared data item can contain any C data type, except for pointers. The shared data abstraction was chosen because it is fairly intuitive, and it allows the automatic routing of data into the destination's local storage. The programmer doesn't have to write a packet parser.

The programmer uses procedure calls in the configuration section to specify the active set of devices used in the application. Both the master and the slave(s) declare these devices, but only the master communicates directly with the devices, passing device data to the slave(s).

The final procedure call of the configuration section starts the declared slave and computation processes, sets up the shared data items, and in the master, opens communications to the devices in the active device set.

5.4 Computation Section

The computation section does any necessary device calibration, such as for the HMD, and then enters an interaction loop. The interaction loop continues until the user decides to quit, and is typically divided into three parts.

If a computation process is present, the master process will collect the output from it and update the Geometric Model appropriately. Next, the master will evaluate the input devices and execute the syntactic processing present in the Geometric Model component. Once the Geometric Model has been processed, syntactically valid commands are sent to the computation process.

Finally, the interaction loop generates the images presented to the user. The image generation code is started by a call establishing the viewing and perspective transformations for the user's head location. This computation is performed by the master process and sent to the slave process that is generating the other image. Once all graphics processing is complete, a second procedure call synchronizes the two rendering workstations. The frame buffers on both workstations are swapped when both have completed drawing their images.

One important point to make is that if no data has arrived from the computation process by the time that the master is ready to draw the next update, the master *does not block* waiting for input from the computation. The slave also doesn't block on the computation. This supplies direct support for the requirements of low lag and high image update rate as shown in section 3.1.

6 Hardware and Software Architecture

The development environment for the MR Toolkit consists of SGI, IBM, and DEC workstations running Unix. The MR Toolkit therefore supports development using SGI's GL graphics, and PHIGS.

The MR Toolkit currently supports applications developed in the C, C++ and FORTRAN 77 programming languages. The FORTRAN 77 support allows the addition of VR front ends to existing scientific and engineering computations.

6.1 Device Level Software

The MR Toolkit manages devices such as 3D position and orientation trackers, hand digitizers, and sound I/O using the client-server model. In most cases, one server drives each hardware device.

The sole exception to this is the DataGlove, which is internally two devices: a hand digitizer and a 3D tracker.

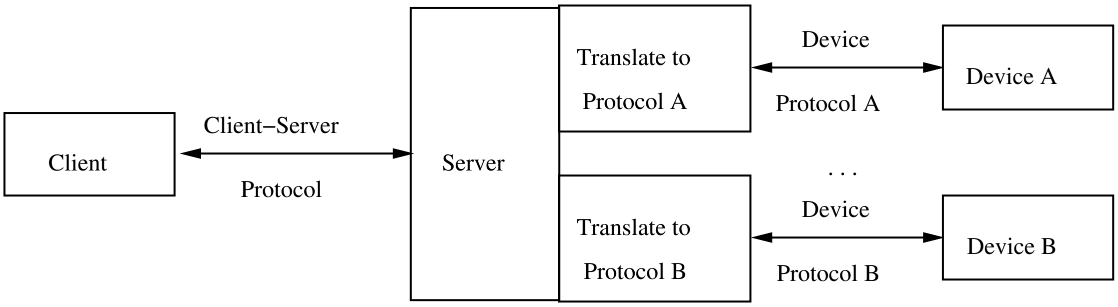

Figure 5: Client-Server process structure

Each server is solely responsible for all interaction with its device: input servers continually collecting from input devices, and output servers continually updating the values on output devices. The server process converts the device's output into a form that is more convenient for the other parts of the application. The servers also perform certain standard operations that may change from one client to the next. For example, the servers for 3D trackers filter [16] the 3D tracker output before it is delivered to the rest of the application.

The client side of the device level software is a library of procedures that are loaded with the MR application. When a client needs service from a particular device, it makes a socket connection to the device's server (using TCP/IP over the local-area network). The client then specifies device-server communications and any special server-based processing, such as filtering. When the server gets a connection, it commences collecting data from input devices, or sending data to output devices. Figure 5 shows the process structure for one client-server pair.

There are several reasons for adopting the client-server model for low level device interactions.

1. Using a separate process to handle each device facilitates distributing the application over several workstations. Workstations with a lighter computational load can host server processes.

2. Device sharing is facilitated by this approach. The application can use any device on the same network. This is particularly useful during program development, since several programmers can work on different workstations, yet share the same set of devices when testing their programs.

3. If improvements are made to either the client or server software, these changes usually don't affect programs that use the client code. For example, a new filter was added to our tracker server with no change in existing application code.

4. The client-server model provides an easily extensible device interface. In fact, the client-server protocol for a particular device type defines the standard interface to that device type. Each 3D tracker server is constructed from two sets of source code. The first set implements the standard client-server protocol, manages the connection with the client, and handles filtering. The second set translates between the server's internal representation of the client-server protocol and the individual hardware device's serial line protocol. Figure 5 illustrates the structure.

5. The sample rate of the device is decoupled from the update rate of the application. One major benefit is that noise reduction filters and predictive filters can operate at the highest possible frequency – the sampling frequency of the device – not at the slower visual or application update frequencies. Filtering at the highest sampling rate results in the best filtering effect. Also, filter performance is invariant under application load.

   Another benefit is that the client is guaranteed to get the latest filtered data by putting the server into *continuous send* mode, where the latest data is collected by the server, filtered, then sent to the client automatically. Our Isotrak lag experiments [16] indicate that continuous mode from Isotrak to server reduced lag by 20-25 milliseconds, and a similar benefit can be had with continuous data traffic from server to client. Network bandwidth is not wasted on poll requests from client to server. This approach may add processing required to deal with excess incoming packets, but our experiments indicate that this cost is minimal.

The potential disadvantage of the client-server approach is that there is a small performance penalty involved with transmitting data through the client and server software layers. We evaluated the performance penalty of the client-server approach, and found that on modern hardware using one-way data communication, the cost is between 3.5 to 8.5 milliseconds. This point is only true if the data producer and the data consumer are on the same machine. If they are on different machines, then the client-server architecture must be used. Furthermore, if client and server are on the same machine, much faster shared memory operations can be used, bypassing the socket based communications entirely.

Since the 3D tracker client drives many different servers of similar type, the server can do anything as long as it obeys the protocol. So, one of our tracker servers doesn't drive a tracker at all, but is a front end to a 2D Graphical User Interface that imitates a 3D tracker. The advantages are that the GUI tracker allows many users to debug VR applications without tying up a device, or indeed, without even being in the same room as a physical tracker. The GUI tracker also gives precise numerical control over a 3D location when needed, and if a portable MR application relies on more trackers than are available, a GUI tracker can be substituted.

The client-server structure has many benefits, in terms of device independence and application portability. The client-server structure also provides support for optimal predictive and noise-reduction filters. Efficient data communications and application distribution are also achieved with this structure.

6.2 Device Names

A second device-handling issue that arose in the design of the MR Toolkit was naming the available devices. A naming scheme provides a way of referring to the key features of a device without regard to the device type. Some 3D position and orientation tracker systems track only one point, while others track as many as 16 individual points. Some hand digitization systems also track multiple gloves [1]. Each sensor needs a unique name, and this name should be independent of the tracking method, the number of trackers associated with one central electronics box, and the computer the electronics box is connected to. A naming system should associate a device name name with the device and site dependent configuration information required to make the toolkit software portable.

The first version of the MR Toolkit introduced the concept of a device set, using a static translation table from device set entries to device names. The device set adds a second level to the naming scheme. The bottom level names sensors and associates communications information, while the top level associates application sensor names with external sensor names.

The obvious problem with a static name for each server is that every change to the hardware configuration requires recompilation of the toolkit. It has the slight advantage that the application programmer and the user don't have to know the details of the configuration. Our current approach allows for a dynamic mapping at the bottom level, solving some other problems with device and site dependency. The device dependencies are:

1. Each device manufacturer has its own communications protocol, requiring unique server software for each device type.

2. Some devices, such as the Polhemus FasTrak, track multiple sensors and use one communications line to transmit the data. Thus, one server may drive multiple sensors, and each sensor has a unique name.

The site dependencies are as follows:

[1] For convenience, we will use the word *sensor* to refer to an individually tracked point of a 3D position and orientation tracking system, and/or an individually sensed hand or glove.

1. Each device must be connected to a serial communications port on some machine, which differs from site to site.

2. Each device server must have a unique TCP/IP port number on the machine. The client software must be able to find the TCP/IP port number in order to connect to the server.

3. Each server typically has a log file where it logs connection information and debug output.

4. Each tracker reports data with reference to its own coordinate system. The transformation from device to application coordinates is site dependent.

Our solution to these problems was to create a file called `servertab` (for *server tab*le) that encodes all of the site and device dependencies for the servers except for the geometric dependency. Geometric dependency is handled in a separate file called `workspace`. The system-wide `servertab` file is always available, and users may copy it for personal modification. When an MR application starts up, it uses `servertab` to determine how and where to connect to servers.

The human-readable `servertab` file lists all of the MR Toolkit servers that are running at this site. This file is a dictionary that maps the name of a sensor or server to other spaces such as network communications space, device space, and log file space.

Confrontation of the naming issue helped satisfy some of the software engineering requirements listed in section 3.2: the key requirement of application portability is supported because the naming system can change from site to site. Portability for devices of like type, and development flexibility are supported by allowing the user to replace the names in the default `servertab` by names of his/her own choosing at application runtime.

6.3 Data Sharing Package

The data sharing package allows two processes to share the same data structure. The purpose of the data sharing package is to place a shared data abstraction on top of message-based TCP/IP socket communications, hiding the technical details of sending data from one process to another. Shared data items are also used internally by MR to manage automatic update and synchronization of the HMD parameters between the two rendering processes.

The data sharing package is structured so that one process is a producer of data and another process is a consumer. This fits well with the DSM, in which most data communications flow in one direction between components. Two way communications can be achieved by setting up two one-way links in opposite directions. For simplicity, one of the communicating processes must be the master, but this can be easily extended to allow communication between arbitrary pairs of processes.

There are three reasons for providing a shared data package instead of having the programmer directly send data between the processes.

1. The data sharing package provides the programmer with a higher level interface to data communications than is provided by the standard socket software. This reduces the amount of time required to develop applications and also reduces the possibility of bugs.

2. The data sharing package can optimize the data transfer in ways that the programmer may not have the time or knowledge to perform. For example, if the master and a slave are on a shared-memory multiprocessor machine, the data sharing package could use shared memory instead of socket I/O.

3. The data sharing package increases the portability of the application. If the application is moved to another set of workstations using different networking software, only the data sharing package needs to be rewritten, not all the applications that use it.

To commence data sharing between the master process and a slave or computation process, the programmer declares the shared items in the configuration sections of processes sharing the data. The declaration procedure returns a data descriptor that is used by the data sharing calls. Each shared item has a unique id, used to identify the item when it is passed from one program to another. This id is common to both processes sharing the data item. The master and slave or computation processes may share any number of data items.

A non-blocking data sharing receive call accepts all messages that are sitting in the input queue, and fills each item based on its message id. If there are no waiting data items, this procedure returns immediately. This allows a process to use the most up-to-date data without blocking. This provides direct support for the DSM, since the Computation component can proceed at its own pace updating the downstream processes asynchronously. A blocking data sharing receive procedure can be called when the consumer process must have a particular data item before it can proceed.

Since a shared data model is used, the programmer is usually not concerned with the timing of the data transfers. This greatly simplifies most applications, since the programmer only needs to state the shared data, and when to transmit the data. The process that receives the data doesn't need to specify when to receive it, or take part in any hand-shaking protocol with the sending process. Moreover, messages arriving out of order are correctly handled by the package.

There are usually two situations where data synchronization is necessary, both associated with the problem of presenting two consistent images simultaneously in the HMD. The first requirement, *consistency*, requires both master and slave processes to render identical copies and views of the database. *Simultaneity* requires consistent images to be presented at the same time to both eyes.

To get consistency, the master calculates the view parameters and sends them to the waiting slave. Therefore, if database update is performed before view parameter update, the databases will be consistent after the viewing parameters are synchronized on the slave.

To achieve simultaneity, the master waits until the slave has indicated that it has finished rendering. The slave sends a sync packet to the master, and then displays its image. When the master receives the slave's sync packet, the master displays its image.

There are times when the consuming process needs to know when the shared data is updated, so the programmer may attach a custom trigger procedure to any shared data item. When a new value for the shared data item is received, the trigger procedure is called by the data sharing package.

6.4 Workspace Mapping

The VR style depends strongly on the geometric configuration of the input and output devices used in the interface. For example, trackers used in VR applications use their own coordinate systems, depending upon where they are located in the room. Three-dimensional sound output devices also have their own coordinate systems, and 3D graphics displays have an implicit default view direction. The workspace mapping package removes the application's dependency on the physical location of the devices that it uses.

The workspace mapping package performs two sets of geometric transformations. The first set maps the coordinate system of each device into the room's coordinate system. The second transformation set is a single transformation that converts room coordinates into environment or "virtual world" coordinates.

The mapping matrices for every device (including workstation monitors, a 3D trackers, joysticks, etc.) are stored in a system-wide `workspace` file. When an application starts up, it reads this file to find the geometric configuration of all of its input and output devices. When a device such as 3D tracker is installed in a different location, the `workspace` file is updated and all applications will automatically use the new device position. Because each tracker is mapped to the common room coordinate system, tracked items drawn in virtual space maintain the same geometric relationship they do in real space.

The room-to-environment mapping depends upon the coordinate system used in the application. This mapping is specified by a sequence of procedures, and can be changed dynamically while the program is running. This transformation is specified in two parts, a rotation matrix specifying how the axes of the room map onto the axes in the environment, and a translation matrix that maps the physical origin of the room coordinate system onto a point in the environment. The translation component can be used to move the user around in the environment. By changing either the position of the room origin or the environment point it maps to, you can either move through the environment or have the environment move by you. Since all devices map to room coordinates, they maintain the same relationship they do in real space. Therefore, the MR Toolkit can achieve complete independence from site-dependent geometry.

The workspace mapping package was initially envisioned as a means of solving the problem of each tracker having its own coordinate space. However, workstation monitors were added because some single-screen applications such as our DataGlove calibration program use the same object code on multiple workstations. Having a single fixed viewpoint means the DataGlove must be held in a particular position to be visible, no matter what screen is used. Instead, applications now read the `workspace` file to find the room position and orientation of the monitor, and adjust the viewing parameters so that the DataGlove is visible when it is held in front of the screen.

6.5 Timing Package

There are two types of analysis available from the timing package. The first type allows the programmer to time stamp key points in the application program, and thereby monitor the amount of real time elapsed between each time stamp call. The time stamp call allows the programmer to associate a text string with each section of code for identification purposes. When the program ends, the timing package outputs a summary of the average real time consumed in each section of the code. One summary appears for each process that runs under the MR Toolkit, and gives the programmer a clear starting point for program optimization. This type of analysis exacts a very small overhead on modern workstations.

Of course, this timing analysis is only part of the story, since the issue of data communications is ignored. The second type of timing analysis deals with the communications time used by the entire application. The data sharing package records in a log file the id and timestamp of each packet that is sent or received. At the end of a run the logs from all the workstations used in the application can be analyzed to determine the location of the communications delays.

6.6 DataGlove Package

The DataGlove package provides routines for simple DataGlove and CyberGlove interaction. It collects the latest values from the DataGlove server, and transforms the position and orientation into environment coordinates using the workspace mapping package. The DataGlove server performs the conversion from raw finger joint readings to degrees, based on a calibration file. Users may use a personal calibration file, if necessary. The DataGlove package also supplies a routine that will draw the hand in the current position, in stereo if necessary.

An interactive DataGlove calibration and gesture editing program is also part of the DataGlove package. This program allows the user to define and name any number of static hand postures, including recognition of hand orientation. A three level data structure is used to represent the gestures used with the DataGlove. The top level is the gesture file, which contains a collection of gesture tables. Each gesture table contains list of related gestures. The gesture recognition procedure considers only the gestures in the gesture table passed as a parameter, which gives control over what gestures are active at one time.

The gesture editing program allows the user to calibrate the DataGlove to his or her hand, and save or load the calibration file. It also allows the user interface designer to design and test gestures by demonstrating acceptable gestures to the gesture editor. Gesture editing is done using the calibrated joint angles, therefore, the resulting gestures are largely user independent.

Although the DataGlove package is optional, it extends the range of input devices that the MR Toolkit supports.

6.7 Sound Package

Sound can be a very important part of a VR user interface. It provides a very good feedback mechanism for gesture recognition, command and operand selection. For these operations it can be difficult to produce good graphical feedback for the following two reasons. First, it can be very hard (if not impossible) to determine where to put the feedback in three-dimensional space so the user will see it, and it won't obstruct his or her view of important parts of the environment. Second, it can be very difficult to design a graphical form of feedback that fits in with the environment the user has been placed in. By associating a distinct sound with each gesture and command, we can provide feedback without interfering with the graphical images.

The optional Sound package provides a standard front end to the many possible sound output techniques available. We use the client-server scheme outlined in section 6.1, with the client producing data and the server consuming data. The MR Toolkit's current assumption is that sound output will be used mainly as a means of signaling events. When an application needs to signal the user that a command has been received, a single sound package call can be used to dispatch a sound to the server. This is similar to the "Earcon" approach [3]. Overlapping events are mixed sonically by the server. An interactive editing program allows the user to define and name sounds that will be generated as events in an application.

Our present sound server uses the audio device on the SGI 4D/20 and 4D/25 workstations. This device is essentially a D/A converter that plays a precomputed sound sample. A simple software FM synthesis package has been developed along with an interactive sound editor to produce the sound samples used by the sound package. This allows the user interface designer to design the sounds used in the application independent of the programming of the user interface. Our colleagues at the Banff Centre for the Arts are in the process of building a sound server based on the IRCAM Sound Workstation, which has spatialized sound capability.

6.8 Panel Package

Three-dimensional presentations of standard two-dimensional interaction techniques are provided by the optional Panel package. A panel is a flat rectangle in 3-space that functions as an arbitrarily oriented 2D screen containing hierarchical menus, graphical potentiometers, buttons and text windows. The text is drawn using the Hershey vector fonts. The programmer declares a panel, its 3D position and orientation, and the width and the height of the surface in terms of environment coordinates. To program 2D interaction techniques, the programmer declares its width and height in 2D screen coordinates, and then allocates screen space, callback routines, and events in the same way as a in 2D menu package. Therefore, once the panel's 3D attributes in virtual space are set, interaction programming is purely 2D. A standard event handling mechanism underlies the implementation of the panel package's 2D interaction techniques, so programming these facilities is quite easy for those familiar with ordinary 2D toolkits.

The selection pointer is the DataGlove, whose position and orientation is used as the origin of a ray cast from the hand into the 3D space of the application. The nearest front-facing panel intersecting the hand ray determines the panel for 2D interactions. Within this panel, the intersection point of the hand ray, shown by a cursor, determines the active 2D interaction technique. One static hand gesture named *select* is then used to perform the operations normally assigned to a mouse button. When the *select* gesture is entered, it is interpreted by the intersected panel as a *button down* operation, and exiting the *select* gesture is interpreted as a *button up* operation.

When more than one panel is active, the panel that intersects the hand ray closest to the hand is the one that is selected. Pop-up panels are also supported, with the default orientation aligned with head tilt, and perpendicular to the line of sight. Pop-up panels can be invoked by entering the *select* gesture, or another gesture can be used to pop the panel up and leave the panel visible until a *select* gesture has been entered.

The ray-casting technique is in contrast to Weimer and Ganapathy's approach [23], where the glove index finger intersects with a panel having menu buttons. Weimer and Ganapathy bolstered this interaction technique by mapping the virtual location of the menu onto the surface of a wooden block, giving the user tactile feedback when the finger intersect the menu. We chose ray-casting because the finger intersection technique requires the user's hand to intersect the plane of the panel, requiring the menu to be within arm's reach. The advantage of the ray-casting technique is that it works reasonably well when the panel is beyond the user's reach. Also, in the immersive VR context, one can't rely on the careful placement of tactile cues like a wooden block in the environment.

Jacoby and Ellis reported a similar ray-casting technique [14] for use with virtual menus. They use one gesture to pop up a menu, and another to highlight and select a menu item. There is no support for standard 2D interaction techniques like sliders, buttons, or ordinary text in their package, although they do support hierarchical menus. They previously tried a hit determination scheme that used an invisible bounding box surrounding the hand to intersect with the menu, and the intersection point determined the menu item to be highlighted. Jacoby and Ellis found that users experienced arm fatigue, and there was sometimes a problem with menus being out of reach. Users also found the lack of a cursor to be somewhat annoying, so Jacoby and Ellis changed the selection method to use the ray-casting scheme.

While ray-casting from the hand works reasonably well, our experience has been that it could stand some improvement. Part of the problem is the presence of noise and delay in the tracker data. The noise results in pointing inaccuracy, while the delay makes target finding more difficult than one would like.

7 Discussion and Conclusions

We have described the Decoupled Simulation Model of real-time 3D animated simulation systems, and a software system to support this model, the MR Toolkit. Our toolkit provides the programmer with the high level facilities needed to develop VR applications. Version 1.2 of the MR Toolkit is available under licence to academic and research institutions, and is currently licenced by 125 sites. To obtain the PostScript licence form, execute anonymous `ftp` to `menaik.cs.ualberta.ca`, and retrieve the file `/pub/graphics/licence.ps`.

At this point, MR's support for the DSM is incomplete since the Geometric Model requires some statement about the application's syntax, and it is not immediately clear how to provide one. This is an area for future research. The MR Toolkit is adequate for our purposes, however, and is an excellent start towards developing software development tools for VR user interfaces. With respect to the requirements outlined in section 3, the packages and facilities of the MR Toolkit directly support application distribution, efficient communications, and performance evaluation. The key requirement for application portability is supported by good device management, geometry independence, and application and device flexibility.

The requirement for low lag is partially supported by predictive filters on tracker devices, and by the extensive use of unidirectional communication of data from the device to the application. Low lag is indirectly supported by the efforts spent on making MR facilities and packages efficient.

Latency and update rate strongly depend upon the application. We believe that the best way to attack these twin challenges is by supplying structural models and tools for application building. Low update rate is usually caused by high graphics load, so the Geometric Model component is used to adjust the graphical load to achieve interactivity. Some complex applications can update slowly, so the Computation component removes the strong temporal coupling of the visual update to the application update. The problem of syntactic lag is further addressed by moving syntactic processing into the Geometric Model component.

In describing the MR Toolkit we have tried to outline our major design decisions and the reasoning behind these decisions. It is our hope that other researchers will question these decisions and either confirm their validity or suggest better approaches. More research is needed on software development tools for VR user interfaces.

We are currently working on higher level design tools for VR user interfaces which will further flesh out the software support of the DSM. We also plan to improve the efficiency of the MR Toolkit and include better performance modeling tools.

7 Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada. One of the authors was supported by a scholarship from the Alberta Microelectronics Corporation. The Banff Centre for the Arts provided invaluable equipment support. The authors also wish to thank Warren Ward and Steve Feiner for their copy editing expertise.

References

[1] Perry A. Appino, J. Bryan Lewis, Lawrence Koved, Daniel T. Ling, and Christopher F. Codella. An architecture for virtual worlds. *PRESENCE: Teleoperators and Virtual Environments*, 1(1):1–17, Winter 1991.

[2] Chuck Blanchard, Scott Burgess, Young Harvill, Jaron Lanier, Ann Lasko, Mark Oberman, and Michael Teitel. Reality built for two: A virtual reality tool. *Proceedings 1990 Symposium on Interactive 3D Graphics*, pages 35–36, 25-28 March 1990.

[3] Meera M. Blattner, D. A. Sumikawa, and R. M. Greenberg. Earcons and icons: Their structure and common design principles. *Human-Computer Interaction*, pages 11–44, 1989.

[4] Steve Bryson and Creon Levit. The virtual wind tunnel. *IEEE Computer Graphics and Applications*, 12(4):25–34, July 1992.

[5] U. Bulgarelli, V. Casulli, and D. Greenspan. *Pressure Methods for the Numerical Solution of Free Surface Fluid Flows.* Pineridge Press, Swansea, UK, 1984.

[6] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.

[7] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. *Human Factors in Computing Systems CHI'91 Conference Proceedings*, pages 181–188, May 1991.

[8] Christopher F. Codella, Reza Jalili, Lawrence Koved, J. Bryan Lewis, Daniel T. Ling, James S. Lipscomb, David A. Rabenhorst, Chu P. Wang, Alan Norton, Paula Sweeney, and Greg Turk. Interactive simulation in a multi-person virtual world. *Human Factors in Computing Systems CHI'92 Conference Proceedings*, pages 329–334, May 1992.

[9] S.S Fisher, M. McGreevy, J. Humphries, and W. Robinett. Virtual environment display system. *Proceedings of ACM 1986 Workshop on Interactive 3D Graphics*, pages 77–87, 1986.

[10] James D Foley, Andries VanDam, Steven K Feiner, and John F Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, Mass., 1990.

[11] Mark Green and Dani Beaubien. Minimal reality toolkit version 1.2: Programmer's manual. *Available by anonymous ftp from menaik.cs.ualberta.ca:/pub/graphics/MRdoc/MR.ps.Z*, November 1992.

[12] Mark Green and Robert Jacob. Siggraph '90 workshop report: Software architectures and metaphors for non-wimp user interfaces. *Computer Graphics*, 25(3):229–235, July 1991.

[13] Mark Green and Chris Shaw. The datapaper: Living in the virtual world. *Graphics Interface 1990*, pages 123–130, May 1990.

[14] Richard H. Jacoby and Stephen R. Ellis. Using virtual menus in a virtual environment. *Proceedings of the Symposium on Electronic Imaging, Science and Technology*, vol. 1668, SPIE, February 11-12 1992.

[15] F. P. Brooks Jr. Walkthrough - a dynamic graphics system for simulating virtual buildings. *Proceedings 1986 Workshop on Interactive 3D Graphics*, pages 9–21, 1986.

[16] Jiandong Liang, Chris Shaw, and Mark Green. On temporal-spatial realism in the virtual reality environment. *UIST 1991 Proceedings*, pages 19–25, November 1991.

[17] Allen Newell. *Unified Theories of Cognition.* Harvard University Press, Cambridge, Massachusetts, 1990.

[18] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. The cognitive coprocessor architecture for interactive user interface. *UIST 1989 Proceedings*, pages 10–18, November 13-15 1989.

[19] Warren Robinett and Richard Holloway. Virtual-worlds research at the university of north carolina at chapel hill. *Proceedings of 1992 Symposium on Interactive 3D Graphics*, March 29 - April 1 1992.

[20] Chris Shaw, Jiandong Liang, Mark Green, and Yunqi Sun. The decoupled simulation model for virtual reality systems. *Human Factors in Computing Systems CHI'92 Conference Proceedings*, pages 321–328, May 1992.

[21] Thomas B. Sheridan. Supervisory control of remote manipulators, vehicles and dynamic processes: Experiments in command and display aiding. *Advances in Man-Machine Systems Research*, Vol 1:49–137, 1984.

[22] Thomas B. Sheridan and W. R. Ferrell. Remote manipulative control with transmission delay. *IEEE Transactions on Human Factors in Electronics*, HFE-4(1), 1963.

[23] David Weimer and S.K. Ganapathy. A synthetic visual environment with hand gesturing and voice input. *Human Factors in Computing Systems CHI'89 Conference Proceedings*, pages 235–240, May 1989.

[24] David Zeltzer, Steve Pieper, and David Sturman. An integrated graphical simulation platform. *Proceedings of Graphics Interface '89*, May 1989.