The Decoupled Simulation Model for Virtual Reality Systems

Chris Shaw, Jiandong Liang, Mark Green and Yunqi Sun

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

Abstract

The Virtual Reality user interface style allows the user to manipulate virtual objects in a 3D environment using 3D input devices. This style is best suited to application areas where traditional two dimensional styles fall short, but the current programming effort required to produce a VR application is somewhat large. We have built a toolkit called MR, which facilities the development of VR applications. The toolkit provides support for distributed computing, head-mounted displays, room geometry, performance monitoring, hand input devices, and sound feedback. In this paper, the architecture of the toolkit is outlined, the programmer's view is described, and two simple applications are described.

Keywords: User Interface Software, Virtual Reality, Interactive 3D Graphics.

1 Introduction

The Virtual Reality user interface style denotes highly-interactive three dimensional control of a computational model. The user enters a virtual space, and manipulates and explores the application data using natural 3D interaction techniques. This style usually requires the use of non-traditional devices such as head-mounted displays, and hand measurement equipment (gloves). The core requirement of this style is support for real-time three dimensional interactive animation.

This results in the following issues:

1. The real-time generation of synchronized stereoscopic images for a head-mounted display is not supported by most commonly-available 3D graphics workstations. As a result, two workstations must be operated in tandem to provide two video signals. Consistent images must be presented in synchrony to the user.

2. Low-level support for new I/O devices such as position trackers, gloves, and so on must be provided for efficiency and lag minimization, while high-level abstractions are required by the application programmer.

3. Applications must be designed independently of the tracker geometry, room geometry and device configuration, yet correct handling of geometric data is vital to avoid user confusion.

4. The real-time nature of the task demands that a performance monitoring tools be available for performance optimization and for debugging.

The MR toolkit we describe in this paper is developed to address these concerns.

2 Previous Work

Other groups have worked on support for this user interface style. Zeltzer and his colleagues at MIT produced a general purpose package for building interactive simulation systems, especially for task level animation systems [Zeltzer 1989]. The key element in the package is a constraint network to which all the objects are connected. Once the status of an object is updated, all the constraints which involve this object are informed and evaluated. By using constraint propagation, the gestural input from the DataGlove can also be viewed as an object. New DataGlove gestures trigger gesture-dependent constraints which then produce the reaction to the user's activity.

Card, Mackinlay and Robertson at Xerox have produced an architectural model for VR user interfaces called the Cognitive Coprocessor Architecture [Robertson 1989]. The purpose of the Cognitive Coprocessor Architecture is to support "multiple, asynchronous, interactive agents" and smooth animation. It is based

on a three agent model of an interactive system. These agents are: the user, the user discourse machine and the task machine. The basic control mechanism 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 sorts of agents; the display queue contains all the objects to be drawn; while the governor keeps track of the time and helps the application to produce the smooth output. This architectural model is similar to the Decoupled Simulation Model outlined in section 3.

Researchers at IBM have been using multiple workstations to support the real-time requirements of VR user interfaces [Wang 1990, Lewis 1991]. They have assigned a workstation to each of the devices in their user interfaces and an event based UIMS is used to coordinate the input coming from several devices. MR uses a similar device management approach, as described in section 5.2 of this paper.

Holloway at UNC at Chapel Hill has developed a general purpose tracker library called trackerlib for 3D position and orientation trackers.

Our own previous work has addressed the problem of distributing the low level device software over multiple workstations and the production of a skeleton VR user interface that can be used as the basis for user interface development [Green 1990]. The toolkit described in this paper is an extension of this work.

3 Decoupled Simulation Model

A VR application can be broken into four components, represented by boxes in figure 1. 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 of the remaining system components.

The Computation component is the item of central interest to the application programmer. This component manages all non-graphical computations in the application, and is usually a continuously running simulation. Typical simulations evaluate a computational model of some process in a series of discrete time steps, periodically updating the application data into a consistent state. When the application data is consistent, the Computation component forwards its data to the Geometric Model component. The simulation can receive two classes of input from the Interaction component. The first input class is user commands. The second input class is the current time, which is used to pace the update rate of the application with the update rate of the graphical presentation. The time updating is needed to maintain a constant scaling factor between real time and simulation time, since there is no necessary connection between the Presentation component update rate and the Computation component update rate.

The Interaction component is responsible for managing all input from the user, and for coordinating all output to the user. It manages at a high level the various devices available for user input, and dispatches commands to the output devices based upon user actions. The sub-boxes in the Interaction component of figure 1 indicate that multiple devices are used for input. The parenthesized items indicate items that MR does not currently support, but can be supported in the MR framework.

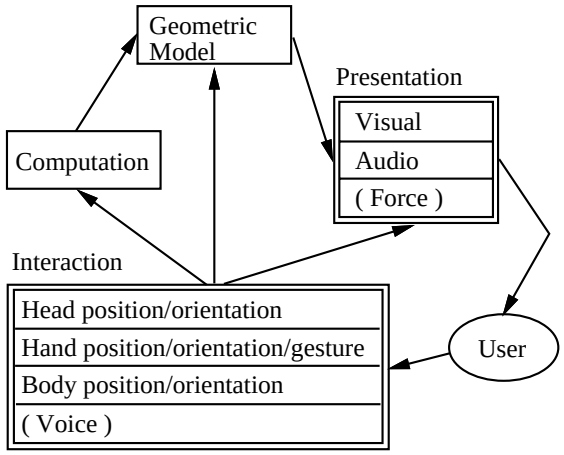

Figure 1: Decoupled Simulation Model.

The Geometric Model component maintains a high-level representation of the data in the computation. The component is responsible for converting the application data in the Computation component into a form amenable to visual, sonic, and force display. For example, the mapping from application data to graphical data could be static, and thus defined at compile time, or it could be dynamic, and under control of the user. The input from the Interaction component can be used to reflect user-proposed changes to the computational model which have not yet been incorporated into the simulation.

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, whose input is the graphical data from the Geometric Model, and the viewing parameters (such as eye position) from the Interaction component. 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.

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. Again, sounds must be updated as the application data and/or the user's position change.

To compare to previous work, Robertson et al's Cognitive Coprocessor Architecture has only one interaction loop that updates a database with small changes. The Cognitive Coprocessor Architecture Task Machine's job is to animate changes from one database view to another, and to animate database updates. There is no direct provision for a continuously-running simulation task in the Cognitive Coprocessor Architecture. The Decoupled Simulation Model has two loops running asynchronously, and therefore has direct support for both discrete event and continuous simulation.

The MR toolkit we describe in this paper was developed to assist in the building of VR applications using the Decoupled Simulation Model. We believe that a clear strategic guideline of this nature can provide a solid basis for building VR applications. These guidelines are bolstered by the toolkit routines, which perform much of the drudgery entailed in managing an interaction of this kind.

4 Application Process Structure

An MR application consists of one or more UNIX-style processes, with one designated as the master process, and others as slave or computational processes. The designation *master*, *slave* or *computation* is called the *role* of the process. An MR application will also establish connection to one or more *server* processes, each of which is uniquely responsible for the management of an I/O device such as a DataGlove, a position tracker, or sound output.

Typically, slave processes perform output tasks. For example, one of the images for a head-mounted display is generated by a slave process. This corresponds to the Presentation component of the Decoupled Simulation Model.

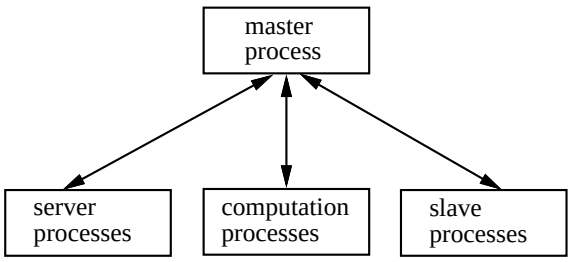

Figure 2: Process structure of an MR application.

A computation process corresponds to the Computation component in the Decoupled Simulation Model, typically performing the simulation part of the application. For example, a computation process could be performing a fluid dynamics calculation, while the user is interacting with the results of the previous time step in the computation.

The master process performs the duties of Interaction 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 with the Interaction component is not required, it is desirable for the purposes of reducing lag. The master process is the first process started and is responsible for initiating the other processes and establishing the communications between these 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 2. Figure 3 shows the process structure of the example code given in the appendix. Figure 3 shows four processes: The master, the almost-identical slave, and two server processes.

5 Hardware and Software Architecture

The hardware environment that the MR toolkit was developed in consists of a Silicon Graphics 4D/35 workstation, a Silicon Graphics 3130 workstation, a VPL DataGlove (tracked by a Polhemus 3Space Isotrak), a VPL EyePhone (also tracked by an Isotrak) and a third Isotrak digitizer. All the timing figures mentioned in this paper are based on this hardware configuration. We are in the process of porting the MR toolkit to Decstation 5000/200PXG workstations.

The MR toolkit currently supports applications developed in the C programming language, and in the near future at least partial support for applications written in Fortran77 will be provided. The Fortran77 support will allow the addition of VR front ends to existing scientific and engineering computations.

5.1 Internal Structure of MR

The MR toolkit consists of three levels of software. The bottom level of the structure consists of the routines that interact with the hardware devices supported by MR. The structure of this level of the toolkit is described in the next section. 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 packages are divided into two groups, which are called

standard and optional. The standard packages are always part of the MR toolkit and provide services, such as data sharing and performance monitoring, that are required by all VR applications. The optional packages provide services that may not be required by every VR application. Typically there is one optional package for each device supported by MR and each interaction technique or group of interaction techniques. There can be any number of optional packages, the exact number depends upon the devices that are locally available and the interaction techniques that have been produced. The use of packages facilitates the graceful growth of MR as new devices and interaction techniques are added. In the following sections some of the more important packages are briefly described. The top level of the MR toolkit consists of the standard MR 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 Device Level Software

The MR toolkit manages devices such as 3D trackers, gloves, and sound using the client-server model. Depending on the device, one or more server processes are used to drive the device hardware. In our configuration for example, one server manages the head-mounted display's Isotrak, one server manages our DataGlove, and one manages the sound driver.

Each server is solely responsible for all interaction with its device, continually collecting from input devices, and continually updating the values on output devices. When a client wishes to get service from a particular device, it makes a socket connection with the device's server (using TCP/IP over ethernet). The client then sets up how the server should communicate with the device, and instructs the server to commence any special processing that should be performed by the server.

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. The server processes can be placed on workstations that have a lighter computational load.

2. Device sharing is facilitated by this approach. The devices used by an application don't need to be attached to the workstation that the application is running on. The application can use them as long as the workstation they are attached to is on the same network.

3. When a new device is added to the system, existing applications are not affected. Applications that don't need the new device simply don't connect to its server.

4. 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 Isotrak server with no change to any existing client code.

5. The rate at which the device is sampled is decoupled from the update rate of the application.

To expand on the fifth point, there are two major benefits to sample rate decoupling. The first benefit is that noise reduction filters and predictive filters can operate at the sampling frequency of the device, not at the application update frequency. Since the server can perform filter calculations at device update rate, filter performance is invariant under application load.

The second benefit is that the client is guaranteed to get the latest filtered data. Moreover, the client-server interaction can be constructed so that the client puts the server into *continuous send* mode, in which the latest data is received by the server, filtered, then sent to the client automatically. Our Isotrak lag experiments [Liang 1991] indicate that continuous mode from Isotrak to server reduced lag by 20-25 milliseconds, and our preliminary measurements show that a similar benefit can be had with continuous data traffic from server to client. Also, network packet traffic between client and server is cut in half, a significant savings.

5.3 Data Sharing Package

The data sharing package allows two processes to share the same data structure. 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 Decoupled Simulation Model, in which data communication is one way. Two way communications can be achieved by setting up two one-way links in opposite directions. To simplify the implementation, 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 the 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.

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

To commence data sharing between the master process and a slave or computation process, the programmer declares in both processes the data structure to be shared. The declaration procedure wraps a header around the data structure to be shared, and returns an id that is used by the data sharing calls. This id is common to both processes sharing the data item. Any number of data items may be shared between the master and slave or computation processes. The data sharing action calls automatically update the appropriate data structures. To send data to the other process, the producer process calls send_shared_data, and the consumer process calls one of receive_shared_data or shared_data_sync.

The receive_shared_data call accepts and properly routes all shared items that are sitting in the input queue waiting to be received. If there are no data structures ready to be received, this procedure returns immediately. This allows a process to use the most up-to-date data without blocking for a lack of input. This provides direct support for the Decoupled Simulation Model, in that the Computation component can proceed at its own pace updating the downstream processes asynchronously of the Presentation component.

On the other hand, the shared_data_sync procedure is called when the consumer process must have a particular data item before it can proceed. While shared_data_sync is waiting in the expected item, it will process and update any incoming shared data item that is sent to the consuming process.

Since a shared data model is used, the programmer is usually not concerned with the timing of the data transfers. For most applications this greatly simplifies the programming process, the programmer only needs to state the data that is shared by the two processes, and when the data is to be transmitted. The process that receives the data doesn't need to specify when it is to be received, or take part in any hand-shaking protocol with the sending process.

There are usually two situations where data synchronization is necessary, both of which are associated with the problem of presenting two consistent images simultaneously in the head-mounted display. The first requirement, *consistency*, implies that before image rendering starts, both the master process and the eye slave process must have identical copies and views of the polygonal database to be rendered. The second requirement, *simultaneity*, means that the consistent images must be presented to both eyes at the same time.

The database consistency operation could be performed by having the slave execute a shared_data_sync call to synchronize databases with the master before it starts drawing. However, since the views must also be consistent, MR automatically has the master calculate the view and send it to the slave, which is waiting for the view parameters with a shared_data_sync call. Therefore, if database update is performed before view parameter update, the databases will be consistent after the viewing parameters are synchronized on the slave.

The simultaneous display requirement is usually met by having the master wait for the slave to indicate that it has finished rendering. When the master receives the slave's sync packet, it sends a *return sync* pulse to the slave, and displays its image. When the slave receives the *return sync*, it can display its image. In hardware configurations where the slave workstation is significantly slower than the master workstation, the *return sync* packet is not needed, since the master will always finish first.

There are times when the consuming process needs to know when the shared data is updated. In MR a programmer-defined trigger procedure can be attached to any shared data structure. When a new value for the shared data structure is received, the trigger procedure is called by the data sharing package.

5.4 Workspace Mapping

Because the VR user interface style depends so strongly on the collection of geometric data from the user based upon the user's position in space, this style creates a new demand for geometric accuracy not previously considered by most 3D graphics packages. For example, trackers used in VR applications use their own coordinate systems, which depend upon where they are located in the room. Three-dimensional sound output devices also have their own coordinate systems, and even standard 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 coordinate system of the room in which the device is situated. The second transformation set is a single transformation which converts room coordinates into environment or "virtual world" coordinates.

The mapping matrices for every device (including workstation monitors, 3D trackers, joysticks, etc.) are stored in a system-wide `workspace` file. 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 position of the device. Because each tracker device 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 can be altered by the application's navigation code, but since all devices map to room coordinates, all devices in the room maintain the same relationship as they do in real space.

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 and the like 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 that the DataGlove must be held in a particular position to be visible, no matter what screen is used. Instead, such 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 that is running the program.

5.5 Timing Package

There are two types of analysis available from the timing package. The first type allows the user to time stamp certain key points in the application program, and thereby monitor the amount of real time that was consumed 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 MR, and so gives the programmer a clear starting point for program optimization. This type of analysis exacts a very small overhead, only 12 microseconds per time stamp call on our SGI 4D/35. On our SGI 3130, each call takes 410 microseconds, so we turn off the time stamping calls on this machine when we are not tuning code for it.

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. In this situation, the data sharing package records each packet that is sent or received. A log file contains the id number of the shared data structure and the real time when it 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 where the communications delays are in the application.

5.6 DataGlove Package

The DataGlove package provides routines for simple DataGlove interaction. It collects the latest values from the DataGlove server, and transforms the position and location into environment coordinates using the workspace mapping package. The package also supplies a routine that will draw the DataGlove in the current position, in stereo if necessary.

An interactive DataGlove calibration and gesture editing program is part of the package. This program will allow the user to define and name any number of static hand postures, including recognition of hand orientation if necessary.

5.7 Sound Package

The Sound package is an optional package which provides a standard front end to the many possible sound output techniques available. We use the client-server scheme outlined in section 5.2, where in this case, the client is the data producer and the server is the data consumer. MR'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 [Blattner 1989, Gaver 1989]. Overlapping events are mixed sonically by the server. The sound package also includes an interactive editing program that allows the user to define and name any number of sounds to be later generated as events in an application.

5.8 Panel Package

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, onto which the application programmer can map menus, graphical potentiometers, buttons and the like. The programmer simply declares a panel and its 3D position and orientation, then allocates screen space, trigger routines, events and so on in the same way the he/she would do with a 2D menu package. The pointer in this case is the DataGlove, whose orientation is used to cast a ray from the hand to any panel in the application. The intersection of the hand ray with the panel is used to determine which interaction technique is active. Hand gestures are then used to perform the operations that would normally be assigned to mouse buttons.

When more than one panel is active, the panel that intersects the hand ray closest to the hand is the one that gets activated. Pop-up panels are also supported, with the default orientation aligned with head tilt, and perpendicular to the line of sight.

6 Examples

In this section, we briefly describe two examples of MR applications. The accompanying videotape has these two examples also.

The first one is the very simple program listed in the appendix. It is the equivalent of the "hello world" program for a programming language. The application consists of a master process, and a slave process. There is no computation process. The flowchart of the application is shown in figure 3, with dashed lines denoting data communications between processes.

The master process first configures the shared data structure, and device set, and starts the slave process. Then both the master and slave processes enter a loop. In the loop, the master process first updates the hand data structure by communicating with the DataGlove server, and performs gesture recognition. Then it updates the EyePhone information by interacting with the EyePhone server, sets up the viewing parameters, and sends them to the slave process. After the shared data is updated to a consistent state, both master and slave processes draw the hand in the environment as an echo of the state of the DataGlove. The update of images are then synchronized before entering next iteration of the loop.

The second example is a simple user interface for fluid dynamics. This user interface forms the front-end to an existing fluid dynamics program written in Fortran [Bulgarelli 1984]. This user interface provides the user with a stereo three dimensional view of the fluid surface using the EyePhone. The user can walk through the flow to view it from different directions. The DataGlove is used to interact with the flow and establish the boundary conditions for the computation, which proceeds while the user is interacting with it.

In terms of software structure, the fluid dynamics example closely follows the Decoupled Simulation Model. The Computation component runs on a CPU server machine at its own update rate. It accepts boundary condition commands from the Interaction component, and sends force vector data to the Geometric component. The Geometric component creates the checkerboard surface that the user sees, and passes it on to the Presentation component. The Presentation update rate is much higher than the Computation update rate, allowing the user to explore slow-moving simulations with limited lag. The fluid dynamics visualization example was produced in three days using this toolkit, with most of the effort being spent on interfacing the Computation component with the Geometric component and the Interaction component.

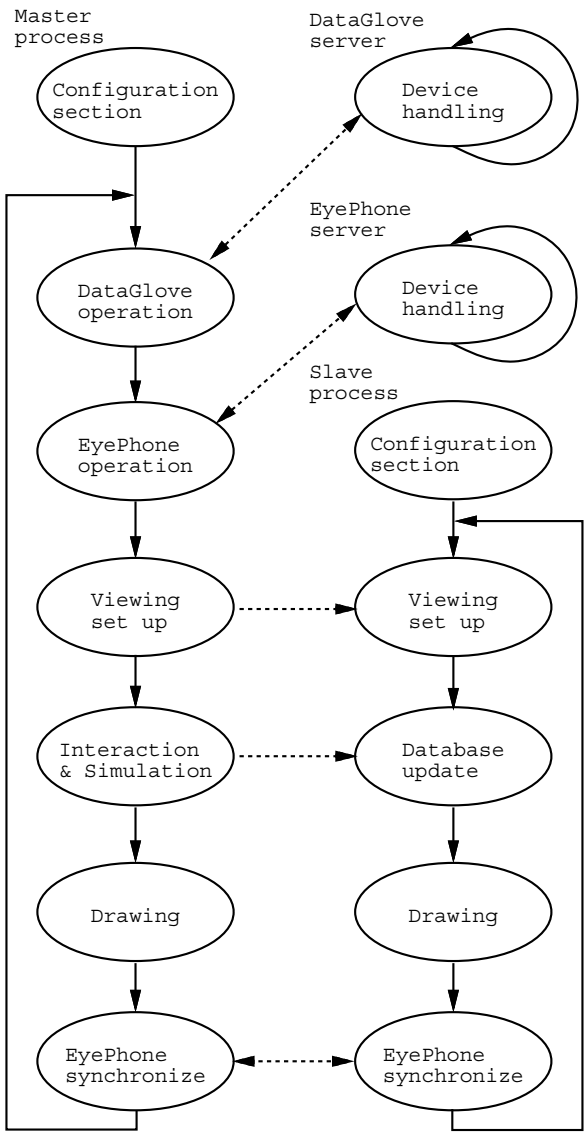

Figure 3: A typical flowchart for an MR application.

7 Conclusions

We have described the Decoupled Simulation Model of real-time 3D animation systems, and a software system to support this model, the MR toolkit. Although it is difficult to demonstrate the flexibility of the toolkit in a paper of this length, it can be seen that it provides the programmer with relatively high level facilities that are most needed in developing a VR application. All the features described in this paper have been implemented. Version 1.0 of MR is available to academic researchers.

We don't make the claim that MR is the best possible toolkit for developing VR applications. Our claim is that it is adequate for our purposes and an excellent start towards developing software development tools for VR user interfaces. 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. There definitely needs to be more research on software development tools for VR user interfaces.

We are currently working on higher level design tools for VR user interfaces. We are also working on better performance modeling tools to be included in the MR toolkit and improving the efficiency of the toolkit.

References

[Blattner 1989] M. M. Blattner, D. A. Sumikawa and R. M. Greenberg, Earcons and Icons: Their Structure and Common Design Principles, *Human-Computer Interaction*, pp. 11–44, 1989.

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

[Gaver 1989] W. W. Gaver, The SonicFinder: An Interface That Uses Auditory Icons *Human-Computer Interaction*, pp. 67–94, 1989.

[Green 1990] M. Green and C. D. Shaw, The Datapaper: Living in the Virtual World, *Graphics Interface'90 Proceedings,* 1990, pp. 123–130.

[Lewis 1991] J. B. Lewis, L. Koved, and D. T. Ling, Dialogue Structures for Virtual Worlds, *CHI '91 Conference Proceedings,* pp. 131–136, 1991.

[Liang 1991] J. Liang, C. Shaw and M. Green, On Temporal-Spatial Realism in the Virtual Reality Environment, *UIST'91 Proceedings,* 1991.

[Robertson 1989] G. G. Robertson, S. K. Card, and J. D. Mackinlay, The Cognitive Coprocessor Architecture for Interactive User Interface, *UIST'89 Proceedings,* pp. 10–18, 1989.

[Wang 1990] C. Wang, L. Koved, and S. Dukach, Design for Interactive Performance in a Virtual Laboratory, *Proceedings of 1990 Symposium on Interactive 3D Graphics, Computer Graphics 24, 2 (1990)*, pp. 39–40.

[Zeltzer 1989] D. Zeltzer, S. Pieper and D. Sturman, An Integrated Graphical Simulation Platform, *Proceedings of Graphics Interface'89,* 1989.

Appendix

```
#include <MR/mr.h>
#define machine "tawayik"
#define program "hello_world"
extern  Gtable gtables[];

main(argc, argv)
int     argc;
char    *argv[ ];
{
int             quit_id, count = 0;
Program         slave;
Data            shared_cnt;
Hand            hand;
Gtable          gst_tbl;
Gesture_event   usr_gst;

    /* Configuration section */
    MR_init(argv[0]);
#ifdef MASTER
    MR_set_role(MR_MASTER);
#else
    MR_set_role(MR_SLAVE);
#endif MASTER
    slave = MR_start_slave(machine, program);
    shared_cnt = MR_shared_data(&count,
              sizeof(count), slave, MR_FROM);
    MR_add_device_set(EyePhone);
    MR_add_device_set(DataGlove);
    EyePhone_slave(slave);
    MR_configure();

    /* Computation section */
    read_gesture_file("my.gst");
    assign_gesture_ids();
    gst_tbl = gtables[0];
    quit_id = get_gesture_id("select");
    set_room_reference(1.0, 1.5, 2.0);
    map_reference_to(0.1, 0.0, 0.5);

    while (1) {
        update_hand();
        hand = get_hand();
        usr_gst = recognize_gesture(gst_tbl);
        if (MR_get_role() == MR_MASTER)
            if (usr_gst->id == quit_id)
                MR_terminate(0);
        MR_start_display();
        count++;
        if (MR_get_role() == MR_MASTER)
            send_shared_data(shared_cnt);
        else
            shared_data_sync(shared_cnt);
        draw_hand(hand);
        MR_end_display();
    }
}
```