Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

KEY REQUIREMENTS FOR CAVE SIMULATIONS

Scott M. Preddy
Richard E. Nance

Department of Computer Science
and
Systems Research Center
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061, U.S.A.

ABSTRACT

Virtual reality offers a new frontier for human interaction with simulation models. A virtual environment, such as that created with a CAVE, imposes either real-time or quasi-real-time performance on the simulation model. Beyond that general requirement, what others can be identified for simulation programs that drive a virtual reality or virtual environment interface? Based on experience with the Virginia Tech CAVE augmented by a literature search, we propose three key requirements for successful CAVE-based simulations: (1) Portability among CAVE-specific input/output devices, (2) effective and efficient interprocess communication, and (3) overcoming the limitations associated with input/output device interaction. Each requirement is described in some detail to both explain and justify its inclusion. Limitations and near- and intermediate-term research needs are identified.

1 INTRODUCTION

1.1 A Paradigm Shift

When we have learned how to take good advantage of it, it may--indeed, I believe it will--be the greatest boon to scientific and technical communication, and to the teaching and learning of science and technology, since the invention of writing on a flat surface (Licklider 1967).

The above quotation comes from a paper written in the mid-1960s. The author is referring to the difference between a static model and an interactive dynamic model. The former produces the same results on every execution, or allows variation in the values of input variables that lead to different *values* for a fixed set of output variables. The model is defined in terms of static components that do not change throughout model execution. A dynamic model, by Licklider's definition, is a model that possesses components, which change during model execution. An interactive dynamic model allows a human to dynamically change model components while the program is executing. Licklider (1967, p. 281) claims that interactive dynamic models hold the key to "communicating ideas about complex systems and processes."

The static model defined above is the classical simulation model, vintage 1960-1970. The rudimentary capability of an interactive dynamic model is introduced in OPS-3, a mid-1960s research system well known to Licklider (Nance 1996, p. 366-367). In the early 1970s a version of GPSS developed by Reitman's group at Norden (Nance 1996, p. 399) couples the dynamic interactive modeling with vector graphics capability. GPSS/NORDEN represents the first example of visual interactive simulation that becomes a prime research topic in the mid-1980s. Licklider's vision is confirmed today in the enormous number of commercial software products that use interactive graphical simulations as their primary mode of user interaction.

We are now entering a second or extended phase of the Licklider prophecy: one or more humans exchanging data with and within a 3-D virtual environment. The human element is not confined to the modeler as Licklider describes; in fact the technical knowledge of the human participants might be unimportant in the interactive experience.

This form of dynamic interactive modeling is provided by the CAVE automatic virtual environment, or simply CAVE, first implemented in 1991 by Carolina Cruz-Neira, at the University of Illinois at Chicago and described in Cruz-Neira *et al.* (1993). The CAVE is composed of three to six projection screens driven by a set of coordinated image-generation systems. It is assisted by a head and hand tracking system that produces a stereo perspective and isolates the position and orientation of a spatial 3-D input device. Audio interaction can also be used in a CAVE simulation, but that is not the focus of this paper.

1.2 Motivation

The utility of a simulation model driving a VR interface seems almost obvious in a training exercise. A simulation of a planned mission drives an interface that represents the buildings, streets, and enemy emplacements that might be encountered in a future urban combat setting. Trainees physically move through this virtual environment, via some specialized locomotive device (Darken 1997). Moreover, the potential uses motivating the coupling are extensive:

- An architect, now providing an impressive 2-D rendering of a building design through a desktop interface, enables the prospective client to access the interior, experiencing many of the stimuli affected by the completed structure (Cruz-Neira, *et al.* 1993).
- A computer scientist navigates through a source code visualization system depicting the code execution in 3-D space, in which geometric shapes not only represent memory being consumed by the system but also the function calls and objects that are created and destroyed (Maletic *et al.* 2001).
- An individual or group in hotel or tour management experiences virtual travel in an open-air elevator located in the Waldorf Astoria Hotel in downtown Manhattan. The simulation program driving the 3-D environment enables the feel of the jerk from upward acceleration and creates the virtual panoramic view from ten stories above the main lobby of the hotel floor (Preddy 2002).

1.3 Objectives

Opinions likely vary on the criticality of universal requirements for CAVE simulations. We attempt to be restrictive in the designation of key requirements, and the focus is limited to three:

- *Portability* among virtual environment interfaces is achieved through an application-programming interface (API) accommodation of multiple levels of abstraction.
- *Inter-process communication* among the concurrent processes composing the CAVE simulation must be effective and efficient. Data transfer and synchronization requirements among processes on a single computer, or distributed over several computers, can be imposed on a CAVE simulation.
- *Device flexibility* refers to the ability to incorporate specialized CAVE input/output (IO) devices as interfaces with the simulation model. The importance of this criterion is easily underestimated, but we argue that it is essential.

2 PORTABILITY AMONG VIRTUAL ENVIRONMENT INTERFACES

2.1 The Importance of Portability

The motivation to create simulations that operate in the CAVE leads to the question: What concerns should a developer anticipate when designing a simulation for this environment? Portability among the devices intended to work in the CAVE is an issue that needs to be addressed during the early developmental stages of the simulation. A major portion of the development for a CAVE simulation can be allocated to a desktop machine if an API exists that abstracts the underlying details of CAVE-specific IO devices (CIODs).

On a fundamental level, CIODs can be divided into two categories: continuous and discrete. Discrete devices, as the name implies, take samples of input at specified discrete time intervals. Input devices like pinch gloves, wands, and keyboards are examples. Continuous devices produce a steady stream of data that are sampled at various times by the system for a "snapshot" of state (Bowman 2002). Understanding how a simulation program can potentially interact with CIODs and what is entailed in achieving the mandated smoothness of interplay is critical. The API used to develop CAVE applications should abstract most of the details of CIOD usage. A developer should be able to work with abstractions that enable an easy port of CIODs from the desktop to the CAVE. The savings from desktop development of a CAVE simulation increases with the size and complexity of the project.

The claim that CIOD portability is a key requirement is based on its influence on development time and cost for a CAVE simulation. Utilizing a workstation, or desktop environment, for development is almost mandatory. This potential raises the question: Are CIODs used in the CAVE correctly abstracted into a desktop interface? Fortunately, the majority of current API's used for CAVE development do this reasonably well. Two API's that do an exceptional job of abstracting CIODs are VR Juggler from Iowa State University (http://www.vrjuggler.org), and DIVERSE from Virginia Tech (http://www.diverse.vt.edu). The former is the product of a group under the leadership of Carolina Cruz-Neira, and the latter, a group led by Ron Kriz at Virginia Tech.

VR Juggler and DIVERSE (from Device Independent Virtual Environments -- Reconfigurable, Scalable, Extensible) employ similar approaches in the CIOD abstraction. VR Juggler provides abstractions based on functionality, assigning a CIOD to one of several basic classes such as positional, orientation, digital, analog, glove, etc. An important feature of this approach to abstraction is the use of inheritance from base classes to represent varieties of the same type of device. Therefore any instance of CIOD ob-

ject typed as "positional" presents the same software interface to the developer (Bierbaum *et al.* 2001). VR Juggler's inheritance mechanism substantially simplifies the device interface presented to a developer.

The DIVERSE API extends CIOD abstraction even further by treating all CIODs as "services." The developer of a CAVE simulation need not be concerned with the specifics of the underlying CIOD hardware. In fact, the developer need not possess a priori knowledge of the *type* of CIOD used in the simulation. The DIVERSE abstraction of the service concept for CIODs is implemented through protocols embedded in Dynamic Shared Objects (DSOs) (Arsenault and Kelso 2001). A DSO is implemented as a small pre-compiled program that is loaded into the executable system at runtime. Using the DSO-enabled abstraction of service reduces development time by hiding the implementation details of the CIOD.

2.2 More on Dynamic Shared Objects

The DSO implementation of the service concept, specific to DIVERSE, is capable of completely abstracting hardware interaction from the developer, allowing a simple, common interface to be presented for all CIODs. The DSO representing the abstraction for a specific CIOD is loaded into the executable program through the command line or by hard coding its name in the application. The effect is transparent; irrespective of where the specification of the DSO occurs, it is always loaded at runtime as a separate program (Arsenault and Kelso 2001).

An example is the interaction with the wand, the CAVE equivalent of a mouse. The main application loads the appropriate DSO associated with wand interaction, which, using DIVERSE syntax, is achieved by *DPF app(desktopCaveSimGroup,NULL)*, where the name of the DSO is *desktopCaveSimGroup*. Next, the application needs to specify how the input from the wand, which is obtained from the DSO, is received into the main application. A simple method for doing this is *DPFInButton *button = caveInput->button*, which simply creates a button object that the main application uses to read from the wand. This example illustrates the significant level of abstraction that DIVERSE achieves in using CIODs. More examples are readily seen at <http://www.diverse.vt.edu/share/dgiPf/examples/>.

2.2.1 VR Juggler

VR Juggler has a concept similar to DSO for CIOD abstraction. Providing a system known as the *Device Store*, VR Juggler enables the separation of all CIOD from the main library. This "selection catalog" allows devices to be loaded dynamically. The Device Store supports the addition of new devices at run-time, or at link time, without having to recompile the application that uses them. A CIOD can be reconfigured at run-time without affecting the main application. A reconfiguration request is sent to the VR Juggler input manager calling for a specific proxy to point to a different CIOD. Using proxies to abstract the CIOD implementation details provides a uniform interface for the using application. The application developer never directly interacts with the hardware or the specific input classes that control the device. New devices can be added by deriving a new class to manage the new device. The advantage of VR Juggler's approach to device abstraction, like that of DIVERSE, is the delayed binding of specific runtime configuration. The CIOD drivers do not have to be compiled with the application that uses those (Arsenault *et al.* 2001).

2.3 Shared Libraries

Shared libraries furnish effective mechanisms for standardizing CAVE applications. The goal of any API supporting CAVE development should be to provide a standard for developers of *every* CAVE application. The API should not be specific to an individual CAVE. Few standards guarantee developers that integration is successful for any CAVE platform. Without a standardized API, CAVE applications cannot be guaranteed to possess portability. The lack of standardization stems in part from the fact that most CAVEs are located in academic institutions that view API development as a creative challenge.

Despite the non-uniformity in the API's that litter the virtual reality field, some standards have emerged. One such standard is a proprietary software product called CAVELib, provided by VRCO (Costigan 2002). CAVELib represents one of the few standards that CAVE developers depend on when developing applications that are truly portable. CAVELib supports a large variety of operating systems: SGI Irix, Sun Solaris, Hewlett Packard HP-UX, Red Hat Linux and Win32. The disadvantage in using CAVELib is that it is proprietary, and the distribution is in the form of pre-compiled libraries that only expose header files as an API interface.

2.3.1 CAVELib

VRCO claims that CAVELib is the industry standard API for support and creation of CAVE applications. CAVELib is OpenGL based, and requires only three functions for a basic CAVE application. The API is extensive, and VRCO maintains that the developer can use only what is needed. CAVELib is externally configured, with windowing and graphics contexts provided for the developer. Automatic data synchronization between processes, threads, and cluster nodes are provided, and the product furnishes multi-channel and multi-pipe support (Costigan 2002).

2.3.2 The Non-Proprietary Alternatives

Most of the features in CAVELib are also provided by VR Juggler and DIVERSE. An important restriction of CAVELib is that it only exposes the header files of pre-compiled binaries to the developer. While currently tolerated in the VR community, this limitation might have a short future. Developers are likely to avoid a work-around of a pre-compiled API that cannot be customized according to their needs when non-proprietary alternatives like DIVERSE and VR Juggler are widely known.

2.4 Closing Remarks

The importance of portability in CAVE development cannot be overstated. An API such as DIVERSE or VR Juggler, which gives a CAVE developer the freedom to develop the majority of a CAVE application at the desktop is invaluable. An API should provide a clean, portable interface to CIODs to alleviate the time spent learning the specifics of device hardware. CAVELib provides a clean interface to CIODs, but the code is not customizable and therefore restrictive. We are of the opinion that open-source API is the future in CAVE development.

3 INTER-PROCESS COMMUNICATION

3.1 Motivation

Whether local to the machine, or occurring among multiple machines, communication is a particular concern. Communication can take the form of data exchange among processes or process synchronization. A discrete event simulation using the CAVE is replete with examples of both forms in the requirement to impose temporal causality (the correct representation of the time ordering of events). RoboCup (Renambot *et al.* 2000), a distributed soccer simulation that allows multiple users to play a soccer match in a CAVE environment, serves as an example from the entertainment domain. The authors describe the game best:

> *The simulation consists of the Soccer Server and a set of processes modeling the players. The server keeps track of the state of the game, provides the players with information on the game, and enforces the rules. The players request state information and autonomously calculate a behavior, sending the server commands that consist of dashes (accelerations), turns, and kicks. The server discretizes time into slots of 100 msec. Only one command of a player is executed per time slot. The kick command requires the player to be close to the ball (1 m).*

Obviously a simulation program of this nature requires tight coordination between the client and server nodes of the distributed system. High latency times for computation and/or data transfer are unacceptable. The designer of the simulation model must give attention to the communications capabilities of the API used for CAVE development. The capabilities of the CAVE hardware must be assessed before attempting to develop the simulation. If network bandwidth is unacceptably low, or if processor speed is inadequate, then a distributed CAVE simulation might not be feasible. Consequently, a realistic assessment of CAVE hardware affecting network performance should be carried out before the decision to develop a distributed CAVE simulation.

Not all inter-process communication is distributed. Local inter-process communication is typically required for synchronization. Shared memory is a popular architecture for achieving local inter-process communication. The DIVERSE API has abstracted shared memory to the degree that its usage is relatively straightforward. In fact, the DIVERSE API abstraction in the shared memory architecture supplies some practical advantages over message passing. Although quantitative comparison has not been performed, experience shows that well structured shared memory architecture is more than adequate for many CAVE simulations (Arsenault 2001b).

CAVE applications of minimal complexity require some means of inter-process communication. Again using the RoboCup example, precise inter-process communication among distributed processors executing the simulation must be achieved or the simulation quickly loses any sense of realism. A second example is the elevator simulation where tightly integrated inter-process communication must be provided between the graphics application and the program that feeds position coordinates to the motion platform. If an unacceptable lag is encountered while communicating data, then the physical movement of the simulation is not coordinated with the visual portrayal.

3.2 Shared Memory for Inter-Process Communication

3.2.1 The Shared Memory Debate

One of the most hotly debated topics in collaborative simulation is the use of shared memory. In concept, shared memory is a physical "address space" with an interconnection mechanism that permits access by multiple processes. The multiple processes might employ a synchronous or asynchronous access protocol, and the "address space" might be a shared data segment. A universal system clock regulates synchronous shared memory access by cooperative simulation processes; e.g. using a *signal* and *wait*. Asynchronous

access is more complicated to implement but avoids the blocking imposed in the process synchronization.

Both synchronous and asynchronous access is needed, and each has its proponents. The synchronous camp argues that a clock must regulate shared memory access to control the busy waiting incurred by the blocked process(es). Data loss is an important issue for some applications, particularly computationally-intensive ones where the consequences might invalidate the results.

The asynchronous camp argues that blocking by any process is unacceptable in certain situations. Real-time simulations involving multiple participants may not allow a single node of the simulation to block while maintaining proper state information. Instead, the asynchronous camp argues that data continuity should be traded for decreased accuracy (Carter *et al.* 1995).

The access protocol debate is fueled by the fact that the wrong choice can have obvious adverse effects on execution time, results or both in a CAVE simulation. For the majority of cases asynchronous access serves well, especially considering that most shared memory usage is for CIOD signaling. Note that operating system services are likely to be required to implement the protocol correctly. In the cases where a shared address space is used in a computationally-intensive CAVE simulation, data loss is an issue, and synchronous shared memory access is desired.

3.2.2 Example Scenarios of Shared Memory use and Efficiency

CAVE simulations requiring shared memory usage can also be categorized as synchronous or asynchronous. Computationally-intensive CAVE applications involving scientific visualization at first glance might be grouped into the synchronous camp. After all, how realistic is a simulation that models fluid dynamics when the graphical refresh rate is faster than the rate of arrival of the data coordinates at a shared memory location? The more revealing question is: What is the nature of this data? If the answer is "graphics data," schemes that compensate for lost data (such as dead reckoning (Macedonia 1997) could be adopted to reduce the need for blocking on a data read. Most CAVE applications are graphically-intensive, and many do not rely on exacting accuracy. These applications become candidates for asynchronous shared memory usage. If data in a shared memory segment is periodically overwritten, then this problem can be rectified by the output display. The trade-off is that the asynchronous shared memory access should not cause the application(s) to block on the shared memory access. If efficiency is measured in access time, then the asynchronous protocol is more efficient.

CAVE simulations promote user interaction via some sort of input device such as the wand. This lends to asynchronous shared memory coordination because wand input is normally not time-critical and definitely not computationally-intensive. Furthermore, most distributed CAVE simulations are real-time, so a process (running on a single processor) cannot afford to block on data access and remain coordinated with the other nodes of the simulation. For this reason the current trend in real-time simulation programs such as those provided by the CAVE is towards asynchronous shared memory access (Arsenault 2001b).

3.3 Networked Distributed Simulation in the CAVE

Currently most CAVE applications are not constructed to require extensive networking capabilities. In the rare case that the CAVE simulation is distributed, few users (nodes) are typical, and the amount of data shared between nodes of the simulation is small. However, the existence of many nodes or high data transfers among nodes forces two issues to be addressed: (1) the feasibility of adequate network performance to support a CAVE environment, and (2) the necessity for embedding the CAVE simulation in a network architecture.

If a CAVE simulation is created with the requirement of non-local collaboration, then shared memory might not be the best alternative. A distributed CAVE simulation could require the collaboration of multiple non-local entities, even including other CAVE's. The traditional client/server model is acceptable if few non-local entities are planned. However, dealing with many nodes and the consequences of significant data transfer latency requires the consideration of alternatives. Once again the notion of asynchronous versus synchronous data sharing becomes an issue.

Asynchronous data sharing is the preferred protocol if the simulation is real-time (Macedonia 1997). The reasoning here is that a synchronous protocol requires all entities of the distributed simulation to have a state consistent with a global clock. Maintaining consistency with a global clock inevitably causes blocking at some point in time due to lost packets, latency, network congestion, etc. An asynchronous protocol does not require processes to wait on state changes from their distributed counterparts. However, the issue of model validity with data loss becomes a major concern. Resolution of this issue can be challenging, and many schemes exist to reduce the amount of lost data and the latency effects.

A method to reduce data loss and latency in a distributed CAVE simulation of moderate size is entitled "area of interest" (AoI). AoI assumes a client/server architecture, where clients send state information to a server which in turn notifies other clients of the state change. AoI partitions the virtual space of the distributed simulation into subsections where clients of the same subsection exchange information exclusively with each other. This information is sent to the server, however the server only sends the state information to clients that are in its area of interest.

The presumption is that each client or user has its own "aura" that is relevant only to members of its subsection.

An example using AoI is a CAVE simulation composed of a group of users who move cooperatively through a virtual space. Those users within close proximity to each other become a subsection, sharing their "aura." The simulation is partitioned so that members of different subsections do not exchange information.

Although this protocol works when the total number of clients is moderate, it fails when the number of clients becomes high, and the aura of each client becomes extensive (Hori 2001). One solution to this problem is to increase the number of servers that manage the virtual spaces (subsections) composing the simulation. Servers must maintain states of other servers and decide whether information needs to be exchanged between them (Houatra 2000). In the typical situation previously described, CAVE simulations are local. Distributed applications involve few users (nodes) and the data exchange rate is low. In such cases AoI is an acceptable protocol for distributed CAVE simulations.

4 CAVE-SPECIFIC IO DEVICES: USES AND LIMITATIONS

4.1 The Motivation for using CAVE-Specific IO Devices

As technology continues to evolve, CIODs are becoming more complex. An emerging class of CIODs provides physical motion capabilities to simulation developers. CAVE-specific motion devices provide a new dimension to simulation, one that cannot be achieved through graphics alone. Graphics capture much that influence the human perceptions of reality, and a deficiency in graphical content diminishes the verisimilitude (appearance of reality). However, if a graphically sound CAVE simulation incorporates CAVE-specific motion devices into a simulation, the simulation produces an effect beyond that achievable through graphics alone.

Glove devices are another example of a CIOD. Glove input is quite similar to the wand; glove output creates tactile sensations for the wearer. This sensory illusion can be extended to include pressure, temperature, twist, etc. – reactions mediated by skin, muscle, tendons or joints (NCITS 1999).

Envisioning the ultimate collection of CIODs leads to a virtual world that is probably not realizable with current technology. Increases in verisimilitude gained by the inclusion of CIODs exact an increasingly demanding penalty in simulation execution time. Clearly, for a given application a point exists where the gain from alternative mechanisms for enhancing the CAVE simulation fails to match the penalty exacted.

4.2 Limitations of CAVE-Specific IO Devices

The effect on an API of providing abstract interfaces to a broad range of CIODs is an issue that has experienced limited attention. One frame of reference is obtained through the experiences of one of the authors with an elevator simulation program, which is discussed in the next section. A conclusion derived from this experience it that a key issue with motion devices is the need of the API to furnish a sufficient set of abstracted services. Advanced simulation developers who concentrate exclusively on a motion device might not require a specialized abstraction to that device. However, less advanced developers, and/or those targeting a wide range of CAVE simulations, benefit from a well-abstracted, highly functional interface to the underlying hardware.

Beyond providing a wide-range of services to CAVE-specific motion devices, advanced developers should always have the option of bypassing a standardized interface if they require lower-level features not accessible from the API. A second aspect of this key requirement is to enable a compromise between the two poles in design strategy: (1) limited, easily used, standardized services or (2) broad, more complicated, specialized capabilities.

A third aspect of this requirement is the provision of an automatic synchronization mechanism between CIODs and the graphical coordinate system imposed by the simulation. In general, the refresh rate of the graphical component of the simulation is faster than the refresh rate of a CAVE-specific motion device. If the API cannot provide either a wide-range of abstracted services or an automatic mechanism for graphical synchronization, then development of the CAVE simulation may prove too great a challenge.

4.3 Examples of Effects on Simulations using Motion Devices

An example of a CAVE simulation utilizing a motion CIOD is the Omni-Directional Treadmill (Darken *et al.* 1997). The Omni-Directional Treadmill (ODT) is a locomotion device that enables a human (subject) to physically walk or run through a graphical environment while remaining in a fixed space. The ODT is composed of two perpendicular treadmills, one inside of the other. The construction of the ODT allows a subject to move forward, backward, perpendicular to the left or right, and diagonally. While this device is not necessarily CAVE-specific, its size permits placement in the floor of a CAVE.

Integrating the ODT into a simulation is not an easy task. The subject can walk or run in any direction at varying degrees of velocity. Any application requiring the use of the ODT requires an interface that abstracts a significant portion of the complexity of the ODT hardware. Without a sufficient level of abstraction, any useful incorporation of

the device into a CAVE simulation is futile. Coordination between the platform and the graphical components of the simulation introduces another serious problem. Darken states that the coordination between the graphics of a simulation and the movements of the treadmill are approximate at best (Darken *et al.* 1997 p. 220).

The interface to the ODT should provide easy access to the coordinates generated by the device. A well-designed ODT interface should allow access to the coordinates and supply procedures for automatic transformation of these coordinates to a scale that is useful for the graphical portion of the simulation. Clearly progress is needed in facilitating the synchronization requirements for using the ODT in other than elementary simulation models.

Another example of using motion CIOD is the Elevator Simulation Program (ESP). The ESP uses the CAVE as a simulation platform. ESP is unique in that it uses a motion platform to simulate the physical effects of acceleration. The motion platform is a hydraulic device slightly smaller than the base of the CAVE. The platform can rise about a meter from the CAVE floor. The motion platform moves in an x-y-z coordinate system and incorporates head, pitch, and roll.

An adequate interface to the motion platform does not exist. Usage of the platform, even for the most rudimentary tasks, can be tedious. ESP requires the motion platform to move only in a vertical direction. The simplicity of single directional movement belies the difficulty in using the device efficiently. The motion platform does not synchronize with the graphical portion of the simulation, requiring a compensation between the two entities. Therefore a set of generalized services must be provided by an API that serves as an interface to the motion platform.

The examples serve to affirm that the specific functionality of a CIOD should be independent of its usage. If an API lacks the capability to provide a reasonable abstraction to the device and a set of basic services for using it, then serious questions should be raised concerning the feasibility for inclusion in a CAVE simulation. A device might demonstrate high potential in providing complex functionality, but if the functionality is inaccessible, then the device is likely to be underutilized.

5 CURRENT METHODS AND FUTURE NEEDS: A SUMMARY

The current methods of CAVE simulation development appear to be generally adequate for the intended applications. However, the vision of what can be is likely constrained by what is possible. That is to say, CAVE-based simulation at this stage is probably influenced to a greater degree by "technology pull" than by "needs push." The newness of the VR and VE technology limits the human ability to conceive of usable and useful applications.

The CAVE interface can be used to advantage in situations where the driving application has much less complexity than is exemplified in contemporary discrete event simulations. The research focus described herein dwells on the key requirements for simulation applications: (1) portability among CIODs, (2) effective and efficient inter-process communications, and (3) simple, usable methods that extend the API to more complex devices.

Portability among CIODs is a requirement that has a high potential for relaxing barriers to innovative advances. Popular APIs used currently in CAVE applications achieve adequate levels of portability only among relatively simple CIODs. In contrast, portability among CIODs specializing in physical motion is lacking. Software is needed that emulates the functionality of motion-specific CIODs. A device such as the ODT cannot be brought to the desktop during development; thus emulation in software is mandatory. The software emulation approach does not eliminate the need for testing the ODT hardware, but can reduce the cost associated with direct implementation of interfaces to the ODT hardware.

The second key requirement for a CAVE simulation is effective and efficient inter-process communication. Local data sharing is an important concern. Meeting this requirement in current applications is challenging, even when limited to local communications. Shared memory is the most popular protocol for inter-process data sharing, however its scalability is a concern. As graphics usage continues to increase, rendering a more computationally-intensive demand, the need expands to share data with an increasing number of computational nodes.

The future is already appearing on the horizon: multiple CAVEs participating in a distributed simulation. Distributed shared memory introduces a significant increase in complexity. More ambitious forms of process architectures, such as that described in Varadarajan (2002), are likely to offer advantages. Such a protocol requires low overhead associated with memory accesses, while allowing the data to be shared by a number of nodes. While near-term research is expected to concentrate on protocols emphasizing local communication, the context for this requirement is anticipated to extend to systems distributed on a moderate scale.

The third key requirement covers the necessary interaction between more advanced CIODs, particularly those involving subject motion, and the CAVE simulation. Current interaction methods enabled by API's are inadequate, burdening the model developer with too much responsibility for effecting interoperability . Developers should not be required to have expert knowledge of a device in order to use it. In some cases, the creators of CIODs are unsure about its full capabilities. We argue that the complexity of the device needs to be abstracted from developers who require only a subset its services. Lacking a reasonable inter-

face to a CIOD, even one that offers very attractive services, developers in many cases are likely to forego the anticipated cost of creating their own. Interaction with advanced CIODs is a critical research area in CAVE simulation.

REFERENCES

Arsenault, L., J. Kelso, R. Kriz, and F. Das Neves. 2001. DIVERSE: A software toolkit to integrate distributed simulations with heterogeneous virtual environments i.e. the DIVERSE kitchen sink paper university visualization and animation group. Virginia Tech, Blacksburg, VA. <http://www.diverse.vt.edu/papers/2001-whitePaper/Main.html> [accessed April 11, 2002].

Arsenault, L., and J. Kelso. 2001. The DIVERSE Toolkit: A toolkit for distributed simulations and peripheral device services. Department of Computer Science, Virginia Tech. <http://www.diverse.vt.edu/papers/2001-09-01_DTK_IEEEVR2002/> [accessed April 22, 2002].

Bierbaum, A., C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. 2001. VR juggler: a virtual platform for virtual reality application development. Virtual Reality. *Proceedings. Institute of Electrical and Electronics Engineers*, 89 -96.

Bowman, D. A. Spring 2002. VE input devices Course lecture notes for Computer Science 5984, Virginia Tech. <http://people.cs.vt.edu/~bowman/cs5984/lectures/input.pdf> [accessed April 3, 2002].

Carter, J. B., D. Khandekar, and L. Kamb. 1995. Distributed shared memory: where we are and where we should be headed, *in Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, 1995, 119-122.

Costigan, Jim. VRCO Online Presentation: <ftp://vrtigo.com/pub/ODU_CLASS/oduClass03282002.ppt > [accessed April 5, 2002].

Cruz-Neira, C., J. Leigh, M. Papka, C. Barnes, S. M. Cohen, S. Das, R. Engelmann, R. Hudson, T. Roy, L. Siegel, C. Vasilakis, T. A. DeFanti, and D. J. Sandin,. 1993. Scientists in wonderland: A report on visualization applications in the CAVE virtual reality environment. Virtual Reality. *Proceedings, Institute of Electrical and Electronics Engineers Symposium on Research Frontiers*, 59-66.

Darken, R. P., W. R. Cockayne, and D. Carmein. 1997. The omni-directional treadmill: a locomotion device for virtual worlds. *Proceedings of the Association for Computing Machinery Symposium on User Interface Software and Technology*, 213-221.

Hori, M., T. Iseri, K. Fujikawa, S. Shimojo, and H. Miyahara. 2001. Scalability issues of dynamic space management for multiple-server networked virtual environments. *Institute of Electrical and Electronics Engineers Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada. <http://ieeexplore.ieee.org/iel5/7568/20620/00953557.pdf> [accessed April 13, 2002].

Houatra, D. 2000. QoS-constrained event communications in distributed virtual environments. Antwerp, Belgium: Distributed Objects and Applications. <http://ieeexplore.ieee.org/iel5/7022/18922/00874178.pdf> [accessed April 10, 2002].

International Committee for Information Technology Standards (NCITS) 1999. *American National Standard Dictionary for Information Technology*, <http://www.ncits.org/press/k5press.htm> [accessed July 2, 2002].

Licklider, J. C. R. 1967. Interactive dynamic modeling. In *Prospects for Simulation and Simulators of Dynamic Systems*, eds. G. Shapiro and M. Rogers, New York: Spartan Books.

Macedonia, M. R. and M. J. Zyda. 1997. A taxonomy for networked virtual environments. *Institute of Electrical and Electronics Engineers Multimedia*, 48-56. <http://ieeexplore.ieee.org/iel4/93/12582/00580395.pdf> [accessed April 15, 2002].

Maletic, J. I.; J. Leigh, A. Marcus, and G. Dunlap. 2001. Program visualizing object-oriented software in virtual reality. *Proceedings International Workshop on Program Comprehension*, 26 -35.

Nance, R. E. 1996. A history of discrete event simulation programming languages. In *History of Programming Languages*, eds. T. J. Bergin and R. G. Gibson, 369-427. New York: Association for Computing Machinery Press and Addison-Wesley Publishing Company.

Preddy, Scott and R. Hall. 2002. <http://filebox.vt.edu/users/spreddy/CaveElevReport.doc> [accessed July 9, 2002].

Renambot, H. J. W. L., D. Germans, and H. E. Bal. 2002. Man multi-agent interaction in VR: a case study with RoboCup Spoelder, Virtual Reality, *Proceedings. Institute of Electrical and Electronics Engineers*, 291.

Varadarajan, S. 2002. Weaving a code tapestry: a framework for reconfigurable programming, Technical Report, Dept. of Computer Science, Virginia Tech.

AUTHOR BIOGRAPHIES

SCOTT M. PREDDY is a first year graduate computer science student at Virginia Polytechnic Institute and State University (VPI&SU) in Blacksburg, Virginia. He graduated from VPI with a Bachelors degree in Computer Sci-

ence in the fall of 2001. Mr. Preddy's current interests include virtual reality aided simulations, and other cutting edge computing technologies. His email address is <spreddy@vt.edu>.

RICHARD E. NANCE is the RADM John Adolphus Dahlgren Professor of Computer Science and the Director of the Systems Research Center at Virginia Tech (VPI&SU). Dr. Nance is also Chairman of the Board of Orca Computer, Inc. He has served on the faculties of Southern Methodist University and Virginia Tech, where he was department head of Computer Science, 1973-1979. He held a distinguished visiting honors professorship at the University of Central Florida for the spring semester, 1997. Dr. Nance has held research appointments at the Naval Surface Weapons Center and at the Imperial College of Science and Technology (UK). He has held a number of editorial positions and was the founding Editor-in-Chief of the ACM *Transactions on Modeling and Computer Simulatio*n, 1990-1995. Currently, he is a member of the Editorial Board, Software Practitioner Series, Springer. He served as Program Chair for the 1990 Winter Simulation Conference. Dr. Nance received a Distinguished Service Award from the TIMS College on Simulation in 1987. In 1995 he was honored by an award for "Distinguished Service to SIGSIM and the Simulation Community" by the ACM Special Interest Group on Simulation. He was named an ACM Fellow in 1996. He is a member of Sigma Xi, Alpha Pi Mu, Upsilon Pi Epsilon, ACM, IIE, and INFORMS. His email address is <nance@vt.edu>.