# JavaCAVE: A 3D IMMERSIVE ENVIRONMENT IN JAVA

by

Michael Lazar Milvich

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

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

July 2004

APPROVAL

of a thesis submitted by

Michael Lazar Milvich

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Dr. Gary Harkin

Approved for the Department of Computer Science

Dr. Michael Oudshoorn

Approved for the College of Graduate Studies

Dr. Bruce R. McLeod

## STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U. S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Michael Milvich

July 16, 2004

TABLE OF CONTENTS

# LIST OF FIGURES

# ABSTRACT

Three-dimensional immersive environments have traditionally been developed using the C and C++ programing language. Do to the increasing performance of the Java platform, the Java language is becoming more accepted for scientific and graphical applications. Currently developers who choose to use Java are being excluded from visualizing the results of their programs in a rich three-dimensional immersive environment. This thesis will work towards correcting this problem by implementing a Java library called JavaCAVE to control a CAVE$^{TM}$ immersive environment.

In addition to being a Java library JavaCAVE also tried to reduce the costs of a CAVE$^{TM}$ by being designed to run on a cluster, which is more affordable than a super computer. In order to be cross-platform and to support a variety of hardware manufacturers a plugin system was used. Special care was also taken to provide a simple and easy interface for the users of JavaCAVE.

Two test applications were created to test the functionality of JavaCAVE. They prove that JavaCAVE is able to control the necessary hardware and that the Java Platform ran quickly enough to be a viable choice for controlling a three-dimensional immersive environment.

# INTRODUCTION

A three-dimensional immersive environment is a virtual environment generated by a computer system to immerse the user in a computer-generated world. The user can move about the virtual world and inspect objects from different perspectives and interact with the environment.

Currently three-dimensional immersive environments are being created using the C or C++ libraries, with CAVELib and VRJuggler being the most popular. This creates a problem for the increasing number of developers who choose to use the Java platform. Without a Java visualization library to run a three-dimensional immersive environment, developers are forced to write complex bridges to communicate with their Java program, or create an entirely separate C/C++ program to perform the visualization. This leads to duplication of work and increases the complexity of a project. Having a Java library to perform the visualizations would benefit developers since they can leverage the source code they have already written.

For example, the Advanced Radiotherapy Project (ARTP) team at Montana State University is developing a radiation simulation and treatment planning program entirely in Java. Team members have already written Java code to load data sets and to display the data, but they also need to visualize the data. An immersive environment would allow a richer presentation to the doctors and technicians using the program

and making decisions affecting the course of radiation treatments given to a patient. This is the motivation for this work.

This thesis is concerned with the creation of a library, called JavaCAVE, to run a CAVE™ immersive environment, although it could be expanded to handle other types of immersive environments. A CAVE™ is a small room where a virtual world is projected onto the walls in order to surround the user. The goals for JavaCAVE are to show that Java is indeed capable of running an immersive environment, provide a simple interface to the developer, abstract away as many hardware dependencies as possible, and to reduce the cost of a CAVE™ environment.

## Previous Work

### Ivan E. Sutherland

Sutherland is one of the founders of computer graphics. In 1963 he created the first interactive computer graphics program called Sketchpad. In the early sixties most computers ran in batch mode, where you would give the computer a job and come back in a few hours, or the next day, to retrieve your results. Sutherland wrote Sketchpad to use one of the few interactive computers in order to graphically create engineering drawings.

Next Sutherland became interested in virtual reality and wrote a paper, "The Ultimate Display" where he brainstorms about the possibilities for computer graphics and describes an environment that resembles a CAVE™.

> The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal. With appropriate programming such a display could literally be the Wonderland into which Alice walked [1].

Sutherland never did create a CAVE™-like environment that he described but he did start the virtual reality field by creating the first head mounted displays (HMD) in the late 1960s. His HMD used CRTs that where suspended from the ceiling and the user would sit and position the displays in front of their eyes [2].

Carolina Cruz-Neira

In 1991 at the Electronic Visualization Laboratory, at the University of Illinois in Chicago, Carolina Cruz-Neira along with several fellow graduate students from created the first CAVE™ (CAVE Audio Visual Experience), by projecting stereo graphics onto three walls and the floor of a small room. They also tracked the movement of the user and developed methods to correct the perspective based on the location of the user [3].

CAVELib™

The CAVELib™ library is currently being sold by VRCO which is the leading commercial library for CAVE™ environments. VRCO was founded by three graduates of the Electronic Visualization Laboratory (Jim Costigan, Matt Szymanski, and Jason Leigh) in 1996 when they acquired the distribution rights to CAVELib™ from the University of Illinois [4].

## VRJuggler

VRJuggler is an open source virtual reality toolkit being developed by Dr. Carolina Cruz-Neira at Iowa State University's Virtual Reality Applications Center. It is a scalable VR system that can run on a variety of platforms including, single desktop, single screen systems, and super computer, multi-screen systems. VRJuggler uses a Java program to configure the CAVE™ environment and to provide an interface for the programs that are running, but it does not use Java for the actual visualization [5].

## Why Java?

Java is making inroads into scientific computing due to several factors. It is cross-platform, which allows researchers to develop systems on powerful Unix-based computers while being able to deploy, these systems on less expensive platforms. Java has robust language features such as automatic memory management, opaque pointers, index safe arrays, strong typing, enforced exception handling, and built-in synchronization, making the Java language more reliable than C or C++. This is especially important in the medical field where mistakes can be fatal. Java also comes with large and full featured support libraries which makes development easier and faster [6].

The Execution Speed of Java Programs

When Java was first released in 1995 the virtual machines suffered from performance problems and those early experiences have stigmatized the language. Advances in computer speed, compilers, and virtual machines have greatly increased the execution speed of Java programs so that it can come close to and sometimes even surpass that of a natively compiled language (see the section on SciMark 2.0).

HotSpot Virtual Machine

The most important advance in Java's speed is the HotSpot virtual machine. The HotSpot virtual machine is a just-in-time compiler that compiles the Java byte-code into native machine code while the Java program is running. The HotSpot compiler saves the native machine code so it does not have to re-compile when the same parts of Java program is executed again.

Another advantage of the HotSpot virtual machine is that it can use runtime information to decide when to inline a method. Inlining is the process of copying the code required to run a called method directly into the calling method at the point of the call, thus eliminating the overhead of making a method call. A traditional compiler can only rely on programmer hints and guesses about what methods would be good ones to inline. The HotSpot virtual machine can also inline methods from libraries, which is currently not being done by any C/C++ compiler.

Finally, the HotSpot compiler can take advantage of processor specific enhancements. For example, Intel's x86 architecture has changed over time, but in order to

make sure their programs work on the older processors developers typically don't take advantage of the newer and potentially faster instructions. The HotSpot compiler, on the other hand, can take full advantage of the newer processors because it is translating a program into native code at runtime, at which point it knows the processor type [7].

SciMark 2.0

The SciMark benchmark was created at the National Institute of Standards and Technology with the objective of evaluating different Java implementations for applicability for scientific computing. In addition to the Java benchmark they also created a C version which can be used to compare the performance of Java to C. SciMark benchmarks five common scientific operations (fast fourier transform, Jacobi successive over-relaxation, Monte Carlo, sparse matrix multiply, dense LU matrix factorization) and measures the floating point operations rate. The author ran both the C and Java versions of SciMark with a large data set on a dual Opteron 240 computer. After compiling the results the Java version was slightly faster then the C version. This is shown Figure 1, where Java is faster on two of the five benchmarks, and for the rest either tied or came close to the performance of the C version.

## Java and Visualization

Since its creation Java has always had visualization capabilities. In order to make Java cross-platform Sun created the AWT (Abstract Windowing Toolkit) a

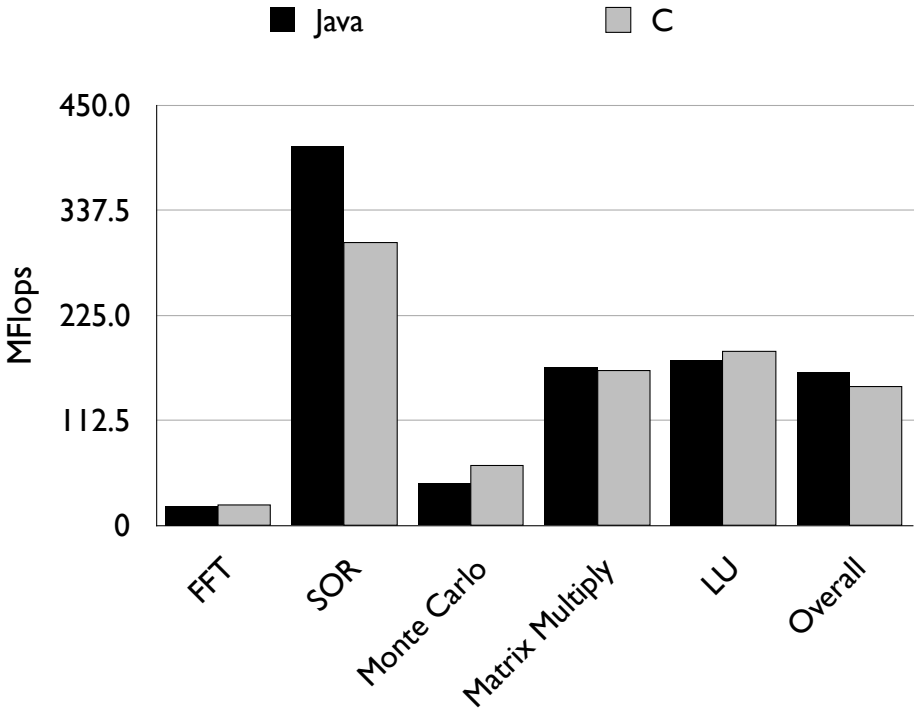


Figure 1. SciMark 2.0 Benchmark Comparing Java and C.

GUI framework that worked across all supported platforms. AWT used the native controls of each platform, so it only implemented a common subset of controls. This was limiting, so Sun later created Swing, which also included a two-dimensional vector graphics package. Swing is a lightweight GUI since it does not use native controls and implements the GUI controls entirely in Java and therefore is not constrained by the native toolkit of any platform.

Visualization Libraries

In addition to the built-in visualization capabilities of Java, several libraries have been created both by Sun and other third party organizations to enhance Java.

Java Advanced Imaging (JAI). JAI is a Java library from Sun that adds support for a variety of image file formats and provides a framework for image processing. It comes with over one hundred image processing operations, such as scale, rotate, color conversion, and statistical functions such as a histogram generation. JAI also allows developers to subclass base operations and extend them to create their own operations. Like many other Java libraries JAI works on Mac OS X, Linux, Solaris, and Windows.

Java Media Framework (JMF). JMF is another graphics library from Sun that adds support for playing or displaying audio, video, and other time-based media. It supports most of the common audio and video formats such as AIFF, AVI, MPEG, Quicktime, and Waves. JMF is distributed both as a Java-only implementation that will work on any platform that supports Java, and specialized versions for Linux, Solaris, and Windows where parts of the library have been rewritten in native code to increase performance.

Java3D. Java3D was an effort by Sun to bring a three-dimensional graphics API to the Java platform. It uses a scenegraph to render the virtual world. A scenegraph

is a collection of three-dimensional objects that the developer builds by adding primitives, such as boxes, triangles, and transformations into a data structure. After the scenegraph is created, Java3D displays everything in the scene, a traditional graphic pipeline requires the developer to explicitly draw all of the elements in the scenegraph each time the display needs an update. Unfortunately, Java3D proved to be complex and unwieldily. Thus, it did not gain acceptance in the marketplace and Sun has discontinued development of Java3D in favor of sponsoring JavaOpenGL (see the next section). Since Java3D is no longer being supported by Sun it was not used for this project.

Java OpenGL (JOGL). JOGL is an open source community project sponsored by Sun. It is a thin binding between Java and the popular OpenGL three-dimensional graphics API. Unlike Java3D, JOGL provides no built-in scenegraph and is much simpler and more low-level. Several higher-level, three-dimensional APIs, such as Xith3D have been built on top of JOGL. JOGL is also cross-platform and supports Mac OS X, Linux, 64-bit Linux, Solaris, Irix, and Windows. It is currently the only three-dimensional API in which a single executable file can run on six different platforms without recompilation. JOGL was chosen for this project because it is open-sourced, has Sun's official sponsorship, and many developers are already familiar with OpenGL and that experience will carry over to JOGL.

Lightweight Java Game Library (LWJGL). LWJGL is another binding between Java and OpenGL. LWJGL also has bindings to OpenAL (a three-dimensional sound library) and provides for input control devices such as gamepads and joysticks. LWJGL is even more low-level than JOGL and cannot be embedded into a Swing interface, so it was not used with this project.

Maestro Case Study

One of the most recent success stories regarding Java's visualization capabilities has to do with the recent NASA missions to Mars. JPL used many of the Java visualization libraries listed above to create a program called Maestro. Maestro does a combination of data visualization, collaboration, and command and control. It analyzes the images that the rovers send back from Mars and creates three-dimensional reconstructions of the terrain. Those models are then used by scientists all over the world to plan the rover operations. A scaled down version can be downloaded for free from the JPL website allowing amateurs to easily analyze the images that NASA has made available. Since it is implemented in Java, Maestro runs on Mac OS X, Linux, Solaris, and Windows.

# ENVIRONMENT

A three-dimensional immersive enviroment is an environment in which the user is fully immersed inside of a computer-generated virtual world where he or she is able to move around and observe the objects in the environment from different angles, and interact with the environment. In addition to visual immersion, the user should also have the sensation of presence in the environment. This provides a sense of realism. The two most common ways to create an immersive environment are with a head-mounted display or a CAVE™ [8].

## Head-Mounted Displays

Head-Mounted displays, or HMDs, have two small screens that are held in place in front of the user's eyes by a harness mounted on the user's head (Figure 2). The display in front of the user's left eye shows the view of the virtual world as it would be seen by the left eye, and conversely the display for the right eye shows how the virtual world would look to the right eye, thereby creating a life-like, three-dimensional image.

HMDs are popular and common because they are relatively cheap and use little space. The small size of the screens restricts their resolution making it more difficult to display crisp images, and their location immediately in front of the user's eyes can


Figure 2. Head Mounted Display.

cause eye strain. They can also cause cybersicknes, which is a form of motion sickness caused by the delay between users moving their heads and the display updates.

## Cave

A CAVE™ accomplishes the same result as a HMD, but instead of having two displays in front of the user's eyes, the images are projected onto the walls of a small room (Figure 3). To create the three-dimensional stereo effect the displays quickly flicker between displaying the left eye and the right eye views. The user must wear LCD shutter glasses that block the right eye while the left eye view is being displayed and vice-versa.

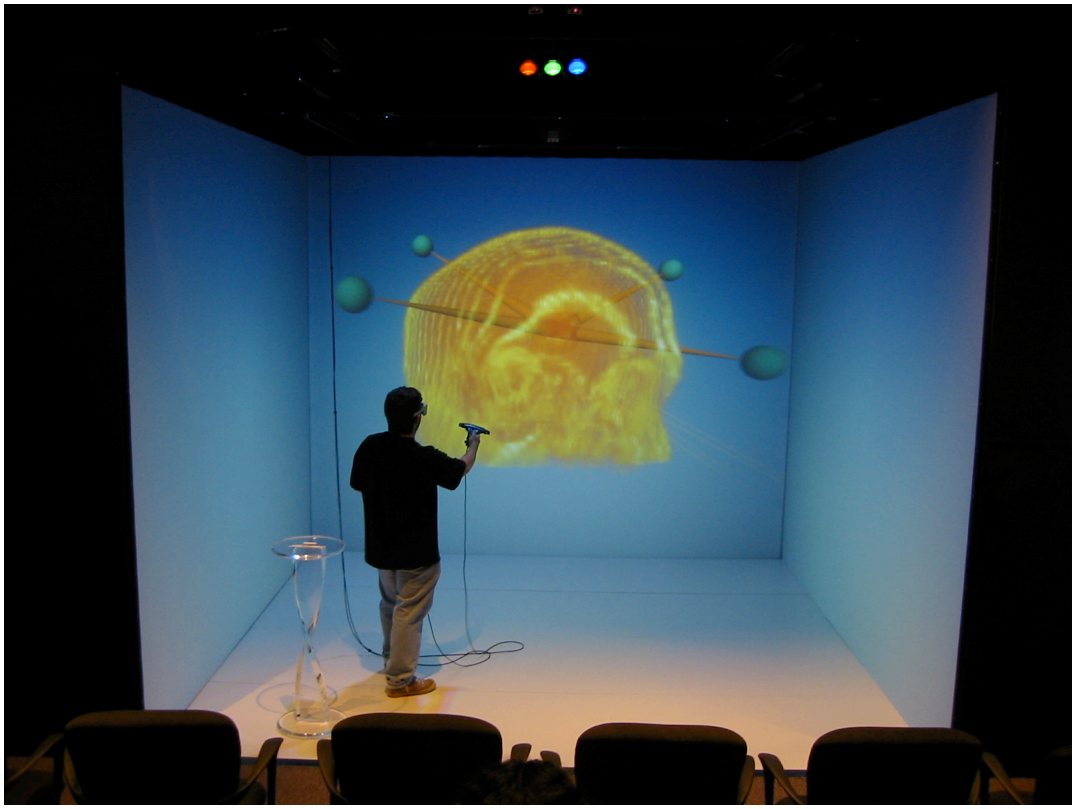

Figure 3. 3D Immersive Cave at the BP Visualization Center.

The CAVE™ method does have some advantages over the HMDs. More than one person can be in the immersive environment at a time, allowing for group collaboration. Cave users have also reported that they experienced less eye strain in the CAVE™ than with a HMD, mainly because the screens in the CAVE™ are farther away from the user. The occurrence of cybersickness is also reduced since the system does not have to display a new image every time the users move their heads thus eliminating the delay problem [9]. Cave environments require a significant amount of space since they need both the room that is the CAVE™ and room for projectors

behind the walls. Caves are quite expensive due to the computer systems, the back projection screens, and the projectors.

Input

The user interacts with a CAVE$^{TM}$ environment through input devices. In order to be completely effective the input devices need to be able to report their current location and their orientation so that the user can point at an object. In addition, the input devices may have additional buttons or other sensory inputs. For example a wand (Figure 4) is useful for pointing and has buttons that can be used to select and modify objects. A data glove on the other hand might have sensors for the joint angles of the fingers or touch sensors to detect a pinching action. Speech recognition could also be used to navigate menus or activate certain actions.

Figure 4. InterSense IS-900 Wand Tracker.

Site Description

The testing of the JavaCAVE library was done using the three-dimensional CAVE™ immersive environment located at the British Petroleum (BP) Center for Visualization in Boulder, CO. The center was established in October of 2000 when BP donated its visualization lab to the University of Colorado at Boulder. This CAVE™ is a three wall, one floor, re-configurable MechDyne MD-Flex™unit. In its closed configuration it is 12x12x10 feet. Its side walls can be opened up to create a 36x10 foot screen that can be used by large group. To eliminate shadows, back projection is used and the floor projector is suspended above the floor. Input is provided by an InterSense 900 VET tracker, with a head tracker and one wand.

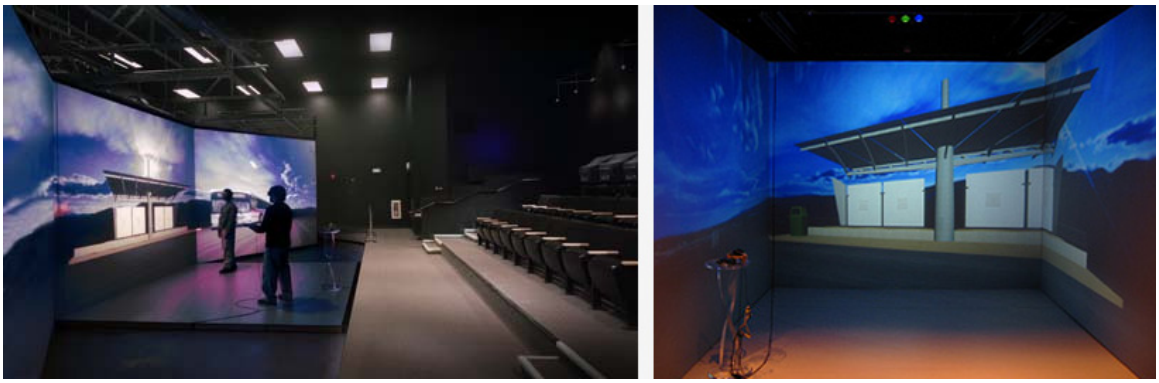

Figure 5. The BP Cave Environment in its Open and Closed Configurations.

# DESIGN

The design of JavaCAVE was heavily influenced by the goals set for this library. A cluster is used to reduce the cost of a CAVE™ environment. The use of a cluster then led to a client-host design. To abstract away any hardware dependencies, a plugin system was created to isolate the dependencies from the core of the library, and to provide a consistent interface for the users of JavaCAVE. The delegate design pattern was used to simply the interface to make it easy for the users of JavaCAVE.

## Cluster

A cluster is a group of computers connected with a network to collaborate on a single problem. Clusters are attractive because they can be built using inexpensive processing components that, when combined, can provide an impressive amount of computing power. For example, according to Top500, an orginzation that tracks the top 500 supercomputers, Virginia Tech recently placed third by connecting 1100 Apple Power Mac G5 computers together with a high speed Mellanox network [10]. The cost to build their supercomputer was only $5.2 million while the other supercomputers in its class cost between $80 and $100 million [11].

For JavaCAVE a cluster is used to run a CAVE™ immersive environment where each node in the cluster is responsible for the displaying on one wall of the CAVE™ environment. A cluster is used because a normal desktop computer has insufficient

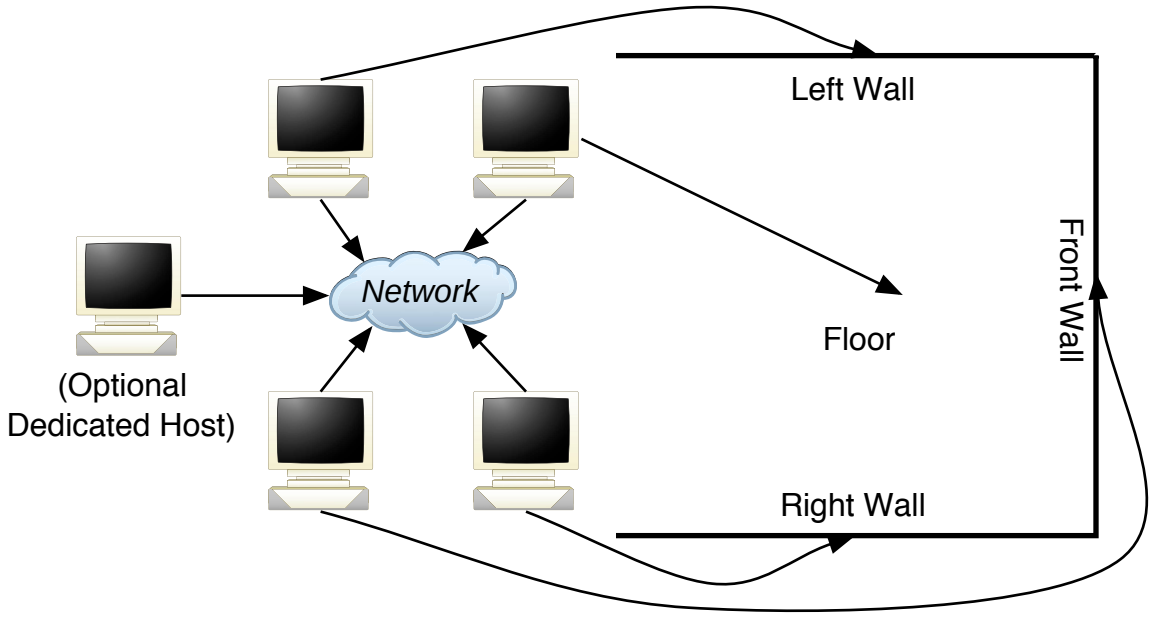


Figure 6. Cluster Diagram for a Cave Environment.

power, and faster computers, from venders such as SGI, that are capable of running the CAVE$^{TM}$ on one computer are expensive. A single desktop computer can easily handle the rendering for a single wall. So an obvious solution to the price versus computing power problem is to simply cluster enough computers so that there is one computer for each wall (Figure 6).

The cluster architecture provides challenges that would not exist if just a single computer was used. With a single computer, all of the rendering for all of the walls will be done on that single computer in-combination with high speed synchronization facilities to control the projectors. Synchronization becomes much more difficult with

a cluster because the nodes are connected by a much slower network with high latency and no built-in synchronization.

Using a cluster also complicates communication with input devices. With a single computer the input devices can be connected directly to that computer and the single process running the CAVE$^{TM}$ environment can claim ownership of the input devices and directly use them. In a cluster, input devices can only be connected to one computer, which then must retrieve and forward the input data to all of the other computers in the cluster.

### Client-Host

The use of a cluster naturally leads to a client-host design pattern. There is exactly one host per cluster, and one client per wall. The host software can run on any of the computers in the cluster, as long as that computer is connected to the input devices. The host can run on its own computer, as depicted in Figure 6 or it can run on the same computer as one of the clients. A dedicated host is sometimes helpful when the user starts a computationally expensive operation. Running the host software on a dedicated computer will not slow down the computers responsible for displaying the data. The host is also responsible for retrieving the input from the hardware and for updating the clients with that new input. The clients, on the other hand, need to render their data as quickly as possible.

## Network

The computers in the cluster are connected through a standard Ethernet network and can use one of several different protocols. These included the standard TCP and UDP protocols, as well as some more exotic networking protocols provided by the Java frameworks such as Remote Method Invocation (RMI).

RMI is a high level network interface built on top of TCP that lets one Java process invoke a method in another Java process over a network connection. It does this transparently to the developer, making it very easy to use once the connections between the two Java virtual machines has been set up. RMI also has facilities for automatically downloading class files from a server if needed, simplifying the distribution and installation of clients.

RMI would have been the easiest protocol to implement for our project, but it was rejected because of the amount of overhead required to make a method call across the network and to resolve all the class types. UDP was a strong contender since it has the lowest overhead, but it does not guarantee delivery, so it was also rejected. TCP was selected since it guarantees delivery and doesn't add as much overhead as RMI.

## Plugins

There are several different manufacturers of three-dimensional immersive environment hardware, and each of those manufacturers has a different way of communicating

with their products that is platform specific. In order to support the different types of hardware we use plugins. A plugin is just a file that contains some executable code that the executing program can load while it is running. This allows a program to be customized and expanded at a later date. JavaCAVE currently has two types of plugins, one for the input devices and another for synchronizing the displays. Users of JavaCAVE can create their own plugins to support their hardware without having to change any of JavaCAVE's code. The plugins have to conform to a specified interface which allows the users of JavaCAVE to move their program between different CAVE™ installations and platforms without having to recompile or change their program.

To make things a bit easier on the developer and the user, JavaCAVE plugins are distributed as Java Archive (jar) files. A jar file is a compressed archive of files. Java has built-in libraries to load executable code from jar files and allows the program to retrieve any file that is stored in the archive. This allows the developer to bundle any extra resources, such as icons, pictures, or anything else with their code in one file.

### Delegation

Delegation is a software design technique in which a portion of the program hands a task over to another part of the program. This simple but very powerful concept allows developers to build libraries that can be customized by the users of the library by suppling a delegate to the library. The library will then turn to the delegate at

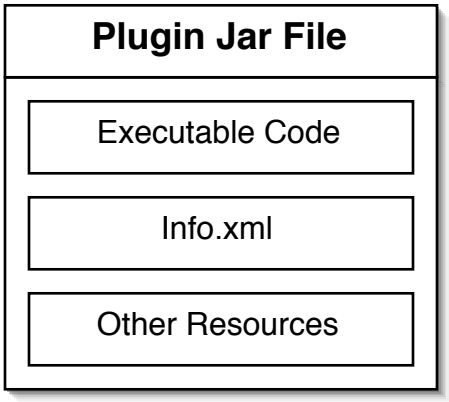


Figure 7. Plugin Enclosed in a Jar File.

critical times and ask it to perform a task. JavaCAVE uses delegates for both the host and client processes. The host's delegate allows the user of JavaCAVE to send extra information to all of the clients, and allows the user to filter events and respond to them at the host level. The client uses its delegate to do the actual rendering.

Delegation makes it easy for a developer to use JavaCAVE. The only thing a developer must do is supply a client delegate with three methods: one to initialize the delegate, another that will render the delegate's image data using JOGL, and one that will update the data based on any input changes received from the host.

### Run Loop

A run loop is the main loop of a program. It is responsible for retrieving events from the operating system and dispatching those events to the proper parts of the

program to be processed. When processing is complete, control passes back to the run loop and it restarts by waiting for the next event.

## Host

The JavaCAVE host is responsible for setting up the network, reading in the configuration, configuring the clients, dispatching the input, and synchronizing the clients. When started, the host listens for TCP connections and once enough clients have joined, the host reads in a specified configuration file and broadcasts the configuration to all of the clients. Next it allows the host's delegate to send any additional information to the clients. When the host has configured all of the clients, the host tells them to start their run loops, and then it enters its own run loop.

### The Host's Run Loop

The host's run loop is shown in Figure 8. It consists of four major stages. First the host polls the input devices for any changes. Next it updates any cluster-wide global variables. Then the host's delegate is given a chance to respond to any updates. Finally the host updates all of the clients.

Input. The first step in the host's run loop is to poll the input devices to determine their current status. Polling the input devices is typically a platform specific operation, and breaks the cross-platform compatibility of JavaCAVE. To make this

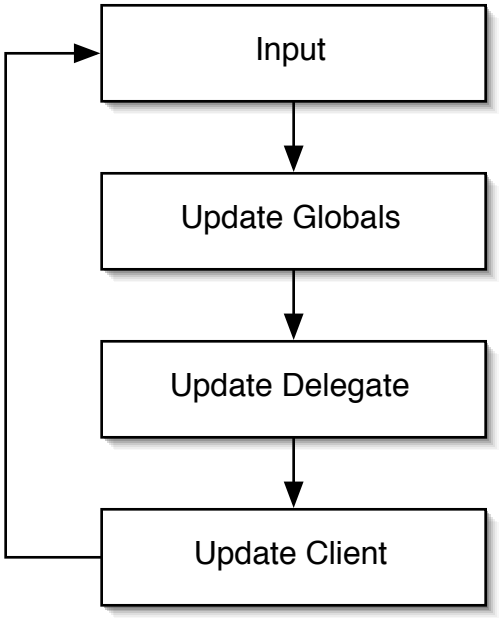


Figure 8. Host Run Loop.

less of a problem the code required to poll the input devices is implemented as a plugin.

Global Variables. After polling for input changes the host updates cluster-wide global variables. A cluster-wide global variable is a variable that must have the same value in all of the nodes. An example of a cluster-wide global variable is the "current time" variable. JavaCAVE cannot rely on the clocks of the individual computers since they will not maintain synchronization independently. So JavaCAVE samples the host's clock during this step and updates the cluster-wide time variable properly.

Delegate Update. Before the host sends the environment changes to the clients the host's delegate is given a chance to respond to any of those changes. A delegate

may want to do this if the user took an action that requires it to load data. It might be faster to have the host load the data once, and then redistribute the data to the clients, rather than having each individual client load the data. This is especially true if all of the clients are trying to retrieve the data from a single database.

Update Client. The last step in the host's run loop is to distribute any changes in the environment to the clients. To do this the host uses the cluster's network and broadcasts the changes to all of the clients. After updating the clients with any input and global variable changes, the host allows the host's delegate to send any custom updates to the client. Finally the host waits for an acknowledgment from the client saying that it got the update. The acknowledgment insures that the host will not overrun the client with data.

## Client

A JavaCAVE client is responsible for rendering data onto a single wall, using the delegate to do the actual drawing. In addition, the delegate is responsible for loading and storing the data – either retrieving the data from the host's delegate or loading the data itself.

Once started, the client first connects to the host and waits for the host to send it the configuration for the CAVE™ environment. After it has received the configuration the client allows the delegate to receive any information from the host. Next the JavaCAVE client creates a JOGL context and allows the client to initialize any

OpenGL settings. After everything has been initialized the client enters its run loop and stays in the run loop until the user stops the program.

The Client's Run Loop

The client's run loop is shown in Figure 9. It is composed of five major stages. First the client waits for any update information from the host. Next the client's delegates are given a chance to respond in the frame update stage. Then the client calculates and applies the view and projection transformations given the user's position and the wall that the client is in charge of. Next the client's delegate is told to display its data. Finally the clients synchronize the swapping of their buffers.

Client Update. During the client update stage, the client receives any environment updates that the host sent. This includes any changes to the input and cluster-wide global variable changes. After receiving the update the client allows its delegate to receive any additional information that the host's delegate wants to send. After all of the updates have been received the client sends a small acknowledgment to the host to let it know that it got the update.

Frame Update. For the frame update stage the JavaCAVE client calls the delegate's frame update method. The delegate should check for any changes to the environment and update its internal representation of the data to respond to any changes. These changes could be changes in user input, a timer that has expired, or a previously started sub-task completed. For example, if the user can spin an

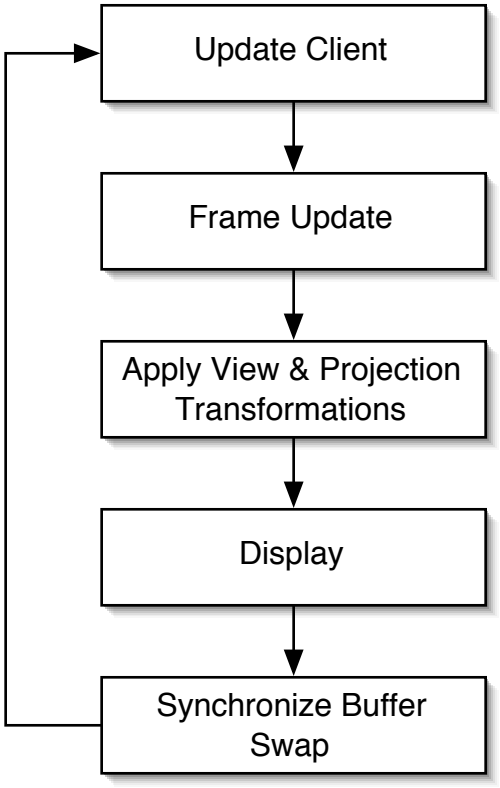


Figure 9. Client Run Loop.

object by holding down a button and waving a wand input device, then the frame update method should check to see if that action has occurred. If it occurred then the delegate should calculate the new rotation angle for that object and save the results for the display method. The delegate's update frame method is only called once per frame, while the display method could be called multiple times. So, any computationally expensive operation should be conducted in this stage.

View Transformations. As shown in Figure 10 the viewing volumes for each wall are different and will change based on the location of the user. JavaCAVE needs

to establish the correct perspective projections to make sure that each wall joins seamlessly with all of the others. We will use the methods described by Dave Pape in his paper "Transparently supporting a wide range of VR and stereoscopic display device."

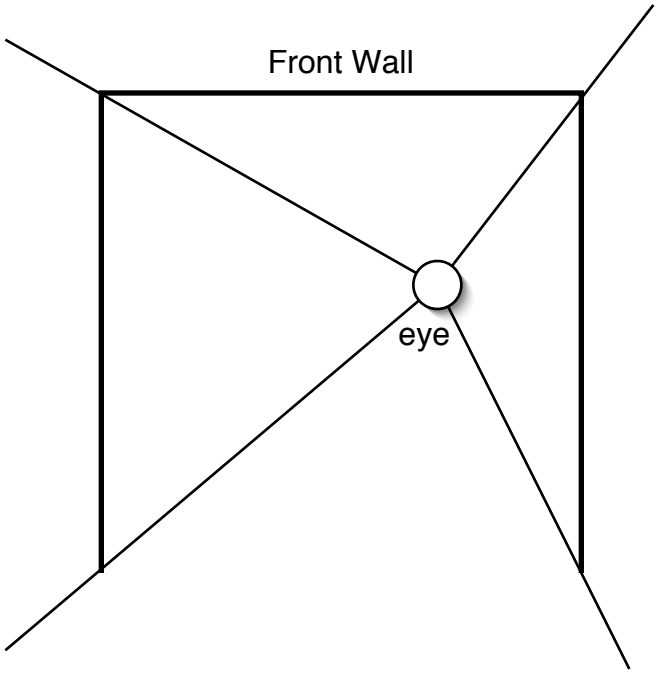


Figure 10. Cave View Volumes.

To establish the projections, JavaCAVE uses OpenGL's glFrustum() function. This function is used to specify the points on the near clipping plane that are mapped to the lower left, and upper right corners of the screen. To figure out these points the user's eye position is projected onto the screen and the distances to the left, right, top, and bottom of the screen are computed (Figure 11).

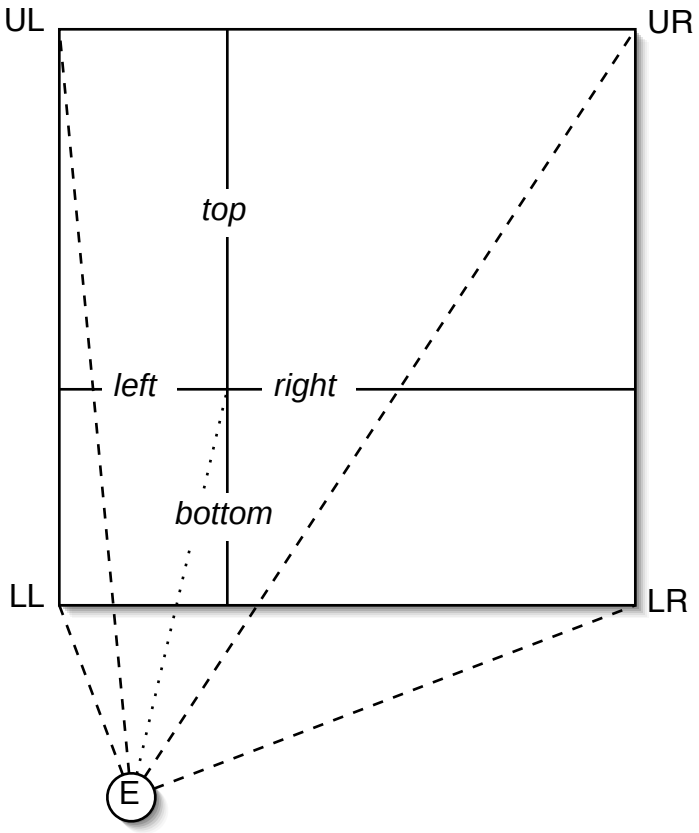


Figure 11. Eye Projection onto the Screen.

In addition to the projection transformation, JavaCAVE also has to rotate the coordinate system so that each wall is displaying a different section of the world. The coordinate system of the wall can be calculated by using the three normalized axis vectors $\vec{X}$, $\vec{Y}$, and $\vec{Z}$. They define a transformation from world-space to screen-space. The inverse of this transformation is therefore the rotational part of a transformation matrix. We also have to translate the world to center the user's position at (0,0,0),

which is accomplished by the following calculation [12].

$$\vec{X} = \frac{LR - LL}{width},\ \vec{Y} = \frac{UL - LR}{height},\ \vec{Z} = \vec{X} \times \vec{Y}$$

$$M_{view} = \begin{pmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \vec{X}_x & \vec{Y}_x & \vec{Z}_x & 0 \\ \vec{X}_y & \vec{Y}_y & \vec{Z}_y & 0 \\ \vec{X}_z & \vec{Y}_z & \vec{Z}_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1}$$

Display. The client draws a new frame of animation by telling the client's delegate to perform the drawing. By this time all of the viewing and projection transformations have been established, so the delegate needs to be careful not to reset OpenGL's transformation matrices. JavaCAVE uses OpenGL's double buffering to draw to a back buffer while OpenGL is still displaying the contents of the front buffer. This allows the program to draw a complete frame of animation before showing anything to the user, and allows JavaCAVE to synchronize displaying the new frame among all of the clients.

If stereo mode is enabled, the delegate's draw method will be called twice, once for each eye. The client usually does not have to change anything to support stereo mode because JavaCAVE sets up the projection matrix for each eye before calling the draw method. A slight offset in the projection matrix will cause OpenGL to render the scenes from two different points, thereby creating a stereo effect. Since the delegate is called twice, the delegate needs to be careful about doing any serious calculations in the draw method. It should instead perform those calculations in the frame update stage and save the results to use in the draw methods. This will prevent

these calculations from being computed twice which would unnecessarily slow down rendering speed.

Buffer Swap. Once the clients have drawn the new scene to the back buffer, the clients must all simultaneously swap the back buffer with the front buffer. Due to differences in the complexity of each client's scene and the unpredictable nature of process scheduling, each client will finish drawing its view at different times. If JavaCAVE did not try to synchronize the buffer swaps, some of the walls would be displaying the old frame while others would be displaying the new frame. At the very least, this would cause disorientation, at the most, the entire stereo effect could fail. So, JavaCAVE forces all of the clients to wait until the slowest client finishes drawing before doing the buffer swap.

In addition to synchronizing the buffer swaps, JavaCAVE also has to synchronize the vertical refresh rate of all of the projectors. Each refresh switches the left and right eye projections. It would be disconcerting if one wall were half way through drawing the left eye view while another wall started displaying the right eye view.

With a single computer these problems are relatively easy to overcome. Synchronizing the buffers swaps can be implemented with a simple spin lock shared between all of the clients. Once all of the clients have checked-in, they can all perform the buffer swaps at the same time. This technique does require a multi-processor computer since the delay caused by swapping between processes will cause some delay

between the buffer swaps. On a multi-processor computer all of the clients can run at the same time and therefore will not have any delays due to processes swapping.

One of the responsibilities of a computer's video card is to provide a synchronizing signal to the display device that tells it when it should start painting from the top of the screen. On a single computer with multiple video cards the developer can control this synchronization signal and make sure that all of the video cards are outputting the same signal, thus synchronizing the vertical refresh rate.

On a cluster both the vertical refresh rate and buffer swapping synchronization become more complicated. The Ethernet network used to connect the cluster together is not deterministic so any synchronization information sent over an Ethernet network will be invalid by the time the clients got it. In order to solve this problem, video card manufacturers (such as Nvidia with the Quadro 3000g) have built Frame Locking (buffer swap synchronization), and GenLock (refresh synchronization) into their video cards. Each video card in the cluster is daisy-chained together with standard CAT-5 cable creating their own network. Nvidia then uses a custom deterministic network protocol to synchronize the video cards. Unfortunately the Nvidia Quadro 3000g retails for about $3000, which is more than the cost of the computers! Although expensive it is a needed component. Plugins are used to implement the frame synchronization to allow JavaCAVE to support more manufacturers and different systems of synchronization.

# TESTS

Two applications were created to test JavaCAVE: the first one was a simple application to verify that the view and projection transformations were being calculated correctly; the second test was a more complex application that displayed a medical model and allowed the user to interact with the model to test the input system.

The easiest metric to acquire, and a critical one, is the number of frames per second (frame rate) that the system is capable of displaying. The more frames the application can draw in one second the smoother the simulation will feel to the user. For each of the two test applications the frame rate was computed when rendered on one, two, three, and four walls, respectively.

## Basic Test

The first test is a basic application that draws a few cones and cubes and moves them around the user's head. It is used to verify that the view and projection transformations are correct by making sure that the cones move smoothly from screen to screen. It does not respond to any input beyond tracking the user's head location to calculate the new view and projection transformations.

As shown in figure 13 with three nodes the frame rate was virtually unchanged, when the fourth node is added there was a decrease in the frame rate. With each additional node the host has to spend a little more time sending out the updates to all

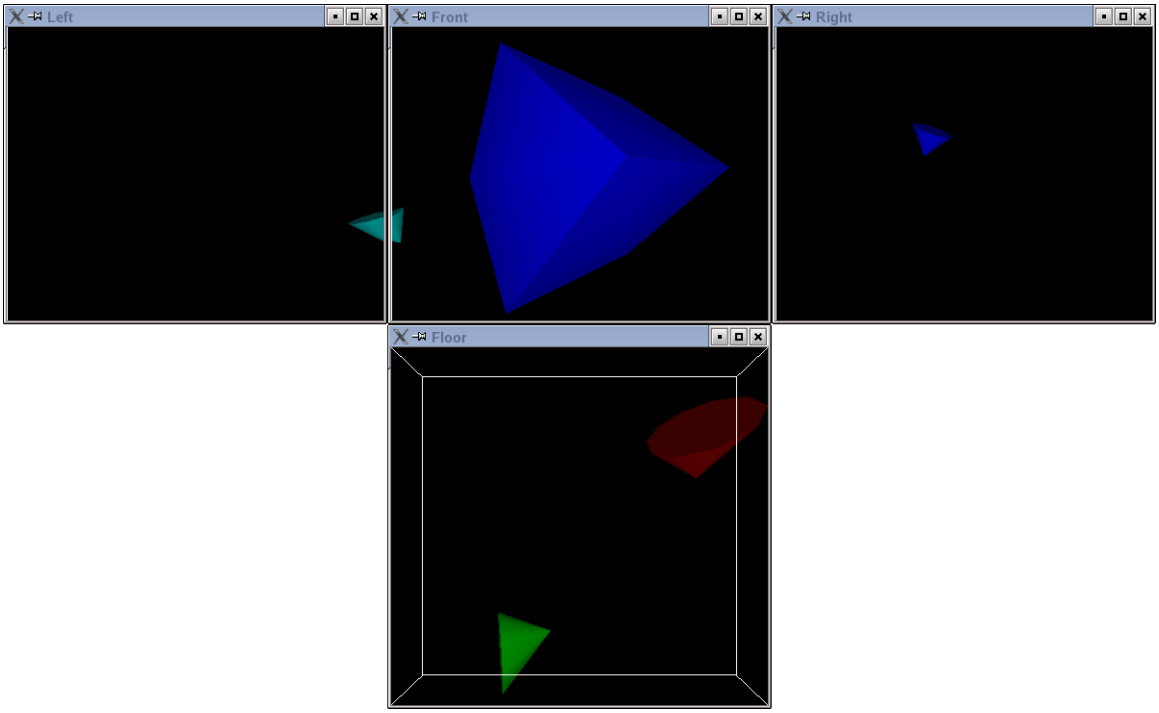


Figure 12. Test Application Displaying Cones.

of the clients. With the fourth node we hit a threshold where the host was spending more time to complete it's run loop then the clients were. This caused the frame rate for the entire cluster to decrease slightly. Although there was a decrease, the cluster is still displaying 445 frames per second, which is far beyond what the human eye can register, making the decrease meaningless.

## Medical Visualization

A more complex test application leverages the work done by the Advanced Radiotherapy Project (ARTP) at Montana State University. ARTP's existing Java data

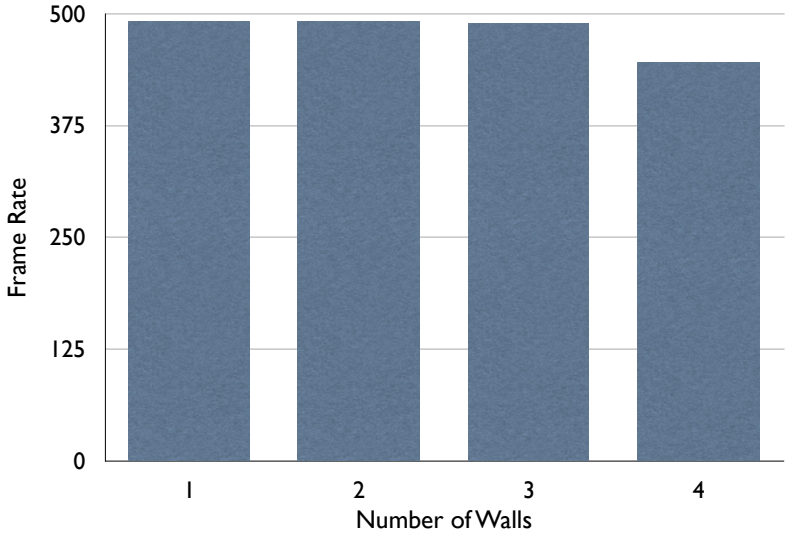


Figure 13. Basic Test Frame Rate.

classes and three-dimensional visualization tools are used to load a model of a human head that has been segmented into different regions and render it. The user can rotate and move the head by using the wand input device. In addition the user can select a region by using the wand to point at the desired region and clicking one of the buttons. Then the user can alter the display of the model by changing the transparency level of the selected region.

The same frame rate metric used with the first test application is used to evaluate this application. The results are shown in Figure 15. Unlike the first application this one renders a very complicated scene with many vertices. The cost of rendering greatly offsets the cost of the network, so the frame rate remains unchanged as the size of the cluster increases.

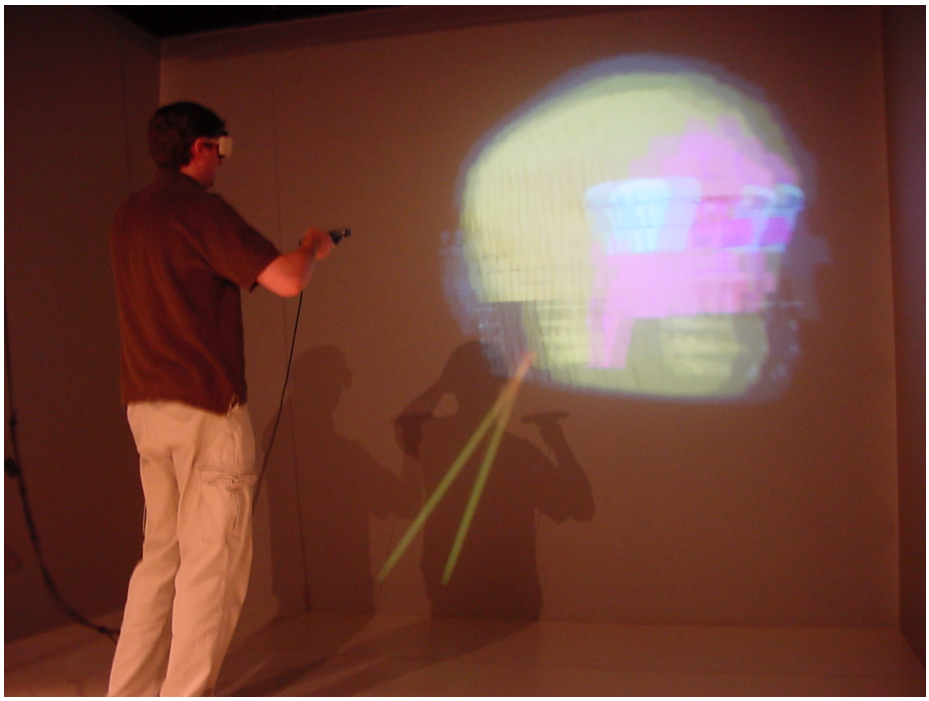

Figure 14. Author demonstrating ARTP visualization at the BP Visualization Center.

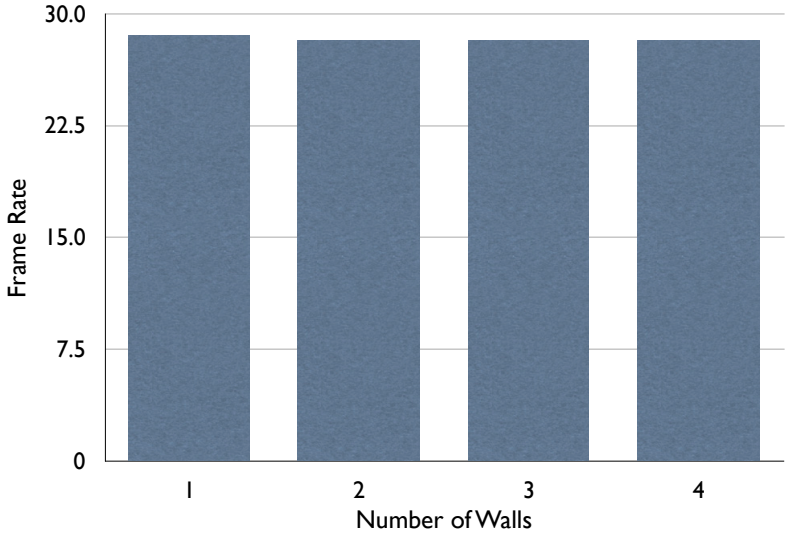


Figure 15. Medical Visualization Frame Rate.

# CONCLUSION

As demonstrated by the two test applications Java is capable of running a CAVE™ environment. JavaCAVE did meet its goal of being easy to use by allowing the developer to create a simple delegate that implements three method calls to initialize, update, and display. An example of a small implementation that draws a spinning cube in front of the user is given in appendix A. JavaCAVE also allows the developer to easily distribute data to all of the clients from the host delegate, and allows the developer to decide where to handle events coming from the user, either in the host delegate or in the client delegate.

JavaCAVE also abstracts out any hardware dependencies. While this is not unique (both CAVELib and VRJuggler provide hardware abstractions) JavaCAVE does this without requiring the developer to recompile and distribute different binaries for various computer platforms (as do CAVELib and VRJuggler). The plugin system also allows JavaCAVE to be easily customized for a specific installation.

JavaCAVE utilizes a cluster to reduce the cost of a CAVE™-like environment. Using a cluster has reduced the cost of the computer platform from being the most expensive component of the system, to being one of the least expensive. The required back projection screens are now the most expensive component of the CAVE™.

# Future Work

## Single Computer

In some cases the speed of a super computer is needed. There are at least two possible ways JavaCAVE could be modified to work on a single computer. The easiest would be to convert the TCP network into pipes or shared memory and run a different process for each wall. Another way would be to modify the client program to create multiple OpenGL contexts, spawn threads for each wall, and integrate the host into the client. While more work is required, using threads would reduce the overhead of communication and speed of the renderers.

## SoftGenLock Support

One of the goals of JavaCAVE was to make CAVE™-like systems more affordable. One way to make them even more affordable would be to replace the expensive Nvidia Quadro 3000g video cards, since they cost just as much as the cluster computers. The SoftGenLock project could be a possible replacement for the expensive video cards. It works by creating a custom network over the parallel ports of Linux computers and, with the help of a real-time Linux distribution, it is able to synchronize consumer level video cards. In order to make JavaCAVE work with the SoftGenLock project a frame synchronization plugin would have to be written to communicate directly with the SoftGenLock's API.

Support Head Mounted Displays

Head mounted displays are a good deal cheaper than CAVE$^{TM}$ environments, but they lack the ability to perform group collaboration. The cost of a collaborative immersive environment could be reduced by using a cluster of computers, in which each node drives a single HMD. Then the positional information about each user could be distributed between the nodes and avatars could be used to insert the members of the group into the virtual world to simulate a group environment.

# REFERENCES CITED

[1] Ivan E. Sutherland. The ultimate display. In *Proceedings of the International Federation of Information Processing Congress*, pages 506–508, 1965.

[2] Ivan E. Sutherland. A head-mounted three dimensional display. In *In Proceedings of the Fall Joint Computer Conference*, pages 757–764, 1968.

[3] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. The cave: Audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):65–72, 1992.

[4] VRCO. Vrco company history.
Available at: http://vrco.com/about/history.html.

[5] Christopher Just, Allen Bierbaum, Albert Baker, and Carolina Cruz-Neira. Vr juggler: A framework for virtual reality development. In *Second Immersive Projection Technology Workshop*, 1998.

[6] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking java against c and fortran for scientific applications. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 97–105, 2001.

[7] Sun Microsystems. The java hotspot virtual machine whitepaper.
Available at: http://java.sun.com/products/hotspot, Sep 2002.

[8] Andries van Dam, Andrew S. Forsberg, David H. Laidlaw, Joseph J. LaViola, and Rosemary M. Simpson. Immersive VR for Scientific Visualization: A Progress Report. *IEEE Computer Graphics and Applications*, 20(6):26–52, November 2000.

[9] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. *Proceedings of the 20th Conference on Computer Graphics and Interactive Techniques*, pages 135–142, 1993.

[10] TOP500. Top500 list for november 20003.
Available at: http://www.top500.org/list/2003/11, Nov 2003.

[11] John Markoff. Low-Cost Supercomputer Made With 1,100 PC's. *The New York Times*, Oct 2003.

[12] Dave Pape, Dan Sandin, and Tom DeFanti. Transparently supporting a wide range of vr and stereoscopic display devices. In *Proceedings of SPIE, Stereoscopic Displays and Virtual Reality Systems VI*, pages 346–353, 1999.

# APPENDIX A

Minimal JavaCAVE Program

This is a minimal implementation of a client delegate, which will draw a spinning cube in front of the user. A host delegate is optional, so it was not implemented.

```
public class TestRenderer implements CaveRenderer
{
    double  rotate;

    public void display(GLDrawable drawable) {
        GL      gl = drawable.getGL();
        GLUT    glut = new GLUT();

        // move back into the screen
        gl.glTranslated(0,0,-5);

        // spin the cube
        gl.glRotated(rotate, 0.0, 1.0, 0.0);

        // draw a blue cube
        gl.glColor3f(0.0f, 0.0f, 1.0f);
        glut.glutSolidCube(gl, 2.0f);
    }

    public void frameUpdate(CaveClient cave) {
        // increase the rotate angle
        rotate += 1;
    }

    public void init(CaveClient cave, GLDrawable drawable) {
        GL  gl = drawable.getGL();

        // setup GL defaults
        gl.glShadeModel(GL.GL_SMOOTH);
        gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
        gl.glClearDepth(1.0f);
        gl.glEnable(GL.GL_DEPTH_TEST);
        gl.glDepthFunc(GL.GL_LEQUAL);
    }
}
```