NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

COMPARISON OF VEGA™ AND JAVA3D™ IN A VIRTUAL ENVIRONMENT ENCLOSURE

by

Brian K. Christianson
Andrew J. Kimsey

March 2000

Thesis Advisor: Michael Capps
Thesis Co-Advisor: Michael Zyda

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE
Comparison of Vega™ and Java3D™ in a Virtual Environment Enclosure

5. FUNDING NUMBERS

6. AUTHOR(S)
Brian K. Christianson and Andrew J. Kimsey

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Large enclosures offer a myriad of possibilities for virtual environments and can dramatically improve presence for a number of applications. Scene graphs are accepted as the logical and optimized way to generate and render applications, however most scene graphs are proprietary or platform specific. Open source scene graphs are emerging that are easily used and cross-platform.

This thesis describes the physical construction of a large sized Multiple Angle Automatic Virtual Environment (MAAVE) and the programming of visual simulations using Vega, a powerful commercially available software package, and Java3D, an open source scene graph. The two simulations are networked walkthrough virtual environments using the same geometry.

After the MAAVE was built, the two applications were tested on multiple platforms with frame rate being the main measure of performance. Initial expectations were that Vega would be faster, but the ease and speed of development of each application was unknown. Results showed that the Vega application was 10 to 30 times faster on sgi hardware and 4 to 20 times faster on a standard PC. The Java3D application required one third of the development time and was easier to program. Overall, we conclude that Vega is the better development platform for multi-channel walkthrough applications.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Virtual Environment, Visual Simulation, Scene Graph, Networking, CAVE, MAAVE	
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

COMPARISON OF VEGA™ AND JAVA3D™ IN A VIRTUAL ENVIRONMENT ENCLOSURE

Brian K. Christianson
Lieutenant Commander, United States Navy
B.A., University of Washington, 1988

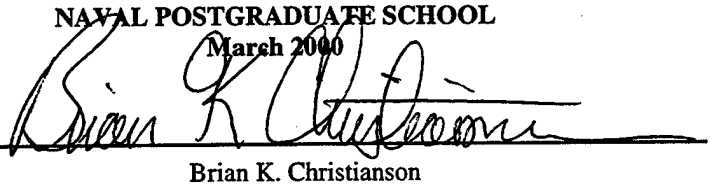
Andrew J. Kimsey
Lieutenant, United States Navy
B.S., United States Naval Academy

Submitted in partial fulfillment of the
requirements for the degree of

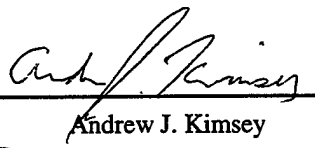
MASTER OF SCIENCE IN MODELING VIRTUAL ENVIRONMENTS AND SIMULATION

from the

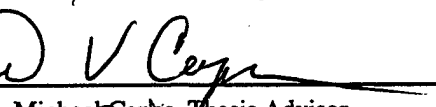
NAVAL POSTGRADUATE SCHOOL
March 2000

Author: 

Brian K. Christianson

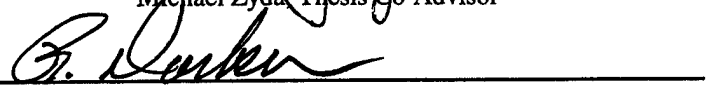
Author: 

Andrew J. Kimsey

Approved by: 

Michael Capps, Thesis Advisor

Michael Zyda, Thesis Co-Advisor



Rudy Darken, Academic Associate
Modeling Virtual Environments and Simulation Academic Group

Michael Zyda, Chair
Modeling, Virtual Environments, and Simulation Academic Group

ABSTRACT

Large enclosures offer a myriad of possibilities for virtual environments and can dramatically improve presence for a number of applications. Scene graphs are accepted as the logical and optimized way to generate and render applications, however most scene graphs are proprietary or platform specific. Open source scene graphs are emerging that are easily used and cross-platform.

This thesis describes the physical construction of a large sized Multiple Angle Automatic Virtual Environment (MAAVE) and the programming of visual simulations using Vega, a powerful commercially available software package, and Java3D, an open source scene graph. The two simulations are networked walkthrough virtual environments using the same geometry.

After the MAAVE was built, the two applications were tested on multiple platforms with frame rate being the main measure of performance. Initial expectations were that Vega would be faster, but the ease and speed of development of each application was unknown. Results showed that the Vega application was 10 to 30 times faster on sgi hardware and 4 to 20 times faster on a standard PC. The Java3D application required one third of the development time and was easier to program. Overall, we conclude that Vega is the better development platform for multi-channel walkthrough applications.

TABLE OF CONTENTS

LIST OF FIGURES

List of Acronyms

ADF	-	Application Definition File
AOIM	-	Area Of Interest Management
API	-	Application Programming Interface
CAVE	-	CAVE Automatic Virtual Environment
DIS	-	Distributed Interactive Simulation
ESPDU	-	Entity State PDU
GUI	-	Graphical User Interface
HMD	-	Head Mounted Display
IEEE	-	Institute of Electrical and Electronics Engineers
JVM	-	Java Virtual Machine
LCD	-	Liquid Crystal Display
LOD	-	Levels of Detail
LOS	-	Line Of Sight
MAAVE	-	Multiple Angled Automatic Virtual Environment
NPS	-	Naval Postgraduate School
PDU	-	Protocol Data Unit
SIGGRAPH	-	Special Interest Group for GRAPHics
VEE	-	Virtual Environment Enclosure
VRML	-	Virtual Reality Modeling Language
XML	-	eXtensible Modeling Language

List of Trademarks

Vega, VegaNT, LynX, Creator, Multigen are trademarks of Multigen-Paradigm, Inc.

Java, Java3D, "Write Once, Run Anywhere" are trademarks of Sun Microsystems, Inc.

Performer, sgi, Irix, Infinite Reality are trademarks of sgi.

Polhemus, Fastrak, Long Ranger are trademarks of Polhemus.

CAVE, RAVE are trademarks of Fakespace Systems, Inc.

Bose and Acoustimas are trademarks of Bose Corporation.

Athlon is a trademark of AMD Corporation.

Pentium is a trademark of Intel Corporation.

WindowsNT is a trademark of Microsoft Corporation.

I. INTRODUCTION

A. MOTIVATION

Large-sized Virtual Environment Enclosures (VEE) are increasingly used for solving networked visualization problems both in industry and scientific research. Commercially available systems offer excellent visual performance but are prohibitively priced for small budgeted research facilities. Most VEEs use proprietary software which incurs additional costs for procurement and lifetime support. Compatibility becomes an issue when networking several remotely located VEEs together to allow collaborative problem solving. Advances in electronic display capability, as well as exponential increases in computing speed, have made construction of a low cost alternative VEE feasible. In order to accomplish meaningful research, this alternative VEE must still retain all the crucial visualization elements of commercial systems.

> A VEE is any physically large structure that one or more users occupy while participating in a virtual environment. Users are often allowed to move around inside a VEE and have the scene update to the new orientation. Some type of three-dimensional glasses is required for users to view objects that are presented in three dimensions.

Software choices for operating this alternative VEE must balance providing a compelling virtual experience that can be shared by multiple users with cross platform compatibility and low cost. Additionally, the potential for future expansion must be considered and parallel software solutions can be viable depending on the desired results for a given environment. Beyond visualization, a large scale VEE can take advantage of other human sensory perception inputs. Spatial sound has been proven to increase the quality of immersion in virtual environments and

can be used to supplement visual displays or provide cueing for non-visual events. Providing the user a non-intrusive means for controlling their immersive experience in, and locomotion through, a virtual environment can enhance the believability of the simulation. One last consideration for creating a large scale VEE is that more than just a few users would be able to view the virtual environment simultaneously. This requires either a massive space to hold the VEE, or better yet a VEE that has the ability to set the screens at multiple angles.

B. BACKGROUND

Large scale VEEs have been around for less than 10 years, although the concept is much older. The *CAVE*™ (CAVE Automatic Virtual Environment) was the first major VEE to be publicly introduced and all follow-on systems benefited greatly from the groundbreaking advances that it incorporated [Cruz-Neira93]. First appearing in 1992, the original *CAVE* had three ten foot by ten foot walls with 90 degree corners between them. The virtual environment images were projected onto the back of the translucent screens with a fourth projection onto the floor between these three walls. Since the introduction of the *CAVE*, many large scale VEEs, like the C2, CABIN and C6, have been created. Along with the latest versions of the *CAVE*, these systems have many improvements over the original design to include more walls, better spatialized sound and less occlusion of the viewing area. All of these systems are expensive ($500,000+) and beyond the financial capabilities of most learning institutions. Early in 1999, the NAVE (NAVE Automatic Virtual Environment) was created for $60,000, giving a good example of a low cost large scale VEE.

The crucial element for any VEE is the displaying of the virtual environment through computer software. Scene graphs have emerged as the preferred data structure for manipulating and displaying complicated graphical models on computer systems. Highly specialized scene

graph software like Vega™ is costly and can only be operated on certain hardware platforms, but offer current state of the art in performance and flexibility. A cross platform scene graph that is growing in use is Java3D™. In addition to being built on Java and able to run on many computer operating systems, Java3D has the benefit of being open source which means it can be downloaded from the Internet at no cost. Java3D suffers a speed performance penalty since it has to run on top of the Java Virtual Machine (JVM) in order to interface with the hardware.

> The JVM is a software interface to a computer's hardware and is platform specific. The JVM can interpret any standard Java code. This means Java code can be written and compiled once and run on any machine having a JVM.

Networked virtual environments allow research collaboration, problem visualization, and multi-entity visual simulations. A decision had to be made on what type of networking solution would produce the most accurate visual graphics results, yet have minimal impact on the speed with which the graphics are updated or refreshed. Refresh rate has been shown as the single most critical factor in preventing simulator sickness and other aberrations that were manifest in early immersive systems. Distributed Interactive Simulation (DIS) allows each site to have its own terrain model and representations for each entity type [IEEE98]. Although this may cause some degree of uniqueness in the graphical representations by each network participant, it allows each system to use software and geometry that is optimized for its own platform, thereby having the least impact on refresh rate. It also allows each computer to maintain the entities' states, eliminating the need for a central server. DIS is highly enumerated and has become a well-accepted standard that enables a user to participate in a large number of networked virtual environments. Other options considered were the new High Level Architecture (HLA) and the network toolkit Bamboo. HLA was discarded because it is only available in C++, the source

code is unavailable, it runs from a central server, and it is very hard to implement. Bamboo was rejected because it was not developed enough to support this thesis.

DIS is an Institute of Electrical and Electronics Engineers (IEEE) standard for creating a data structure for sending physical parameters over a computer network. It is composed of a series of data fields in a particular order to ensure wide-scale interoperability.

C. DEVELOPMENT OF THESIS

Physical location constraints were the single most restrictive element in the creation of our large scale VEE. Room location also presented unique computer display problems because the room where the VEE is located is physically distant from the best graphics computer in the building. The thesis work progressed in the following parallel areas:

- Physical room construction and modification.
- Computer hardware identification and setup.
- Computer software selection, modification and implementation.
- Evaluation of software in VEE

The choice for the first graphics model to run in the VEE was a detailed layout of Herrmann Hall, which was formerly known as the Hotel Del Monte and is currently the heart of the Naval Post Graduate School campus. This model was created in MultiGen and saved in the Flight graphics format.

Since Vega and Java3D (through third party software) can directly read Flight graphics files, the Herrmann Hall model was a natural choice and saved the time of having to create or modify a compelling visual model for demonstrations and development. Unfortunately both

Vega and Java3D needed extensive modifications to create the illusion of walking through Herrmann Hall.

Physical construction provided several unique challenges to include using less expensive materials and not violating building codes. This meant using wood frames and allowing for their warpage, finding readily available acoustic material and attaching it to all hard reflective surfaces and several other alternative materials from what a commercially constructed VEE uses.

Purchasing a top end multiple channel graphics computer would easily keep this project out of the realm of financial plausibility. This meant having to choose alternate methods of delivering visual output to the projectors and the proper projector for the images we needed to display. Two methods were finally decided upon, one using a personal computer with three graphics cards and the other is using an older multiple graphics channel Silicon Graphics computer.

D. SUMMARY OF CHAPTERS

This thesis is organized into the following chapters:

- Chapter II: Detailed Background. Discusses the evolution of large scale Virtual Environment Enclosures, the evolution of software scene graphs that are used to provide graphics for the VEEs and a brief history of networked visual simulations.

- Chapter III: Physical construction of a Large Scale Virtual Environment Enclosure. Discusses in detail the construction of the Multiple Angle Automatic Virtual Environment (MAAVE) and how this VEE is connected in a networked virtual environment.

- Chapter IV: Vega Implementation. Describes the use of Vega and modifications that were needed to have the software perform in the desired manner inside the MAAVE.

- Chapter V: Java3D Implementation. Describes the implementation of Java3D software for use in the MAAVE.

- Chapter VI: Results and Conclusions. Discusses differences in programming complexity, cost, and performance between the Vega and Java3D applications for both PC and Silicon Graphics workstations in the MAAVE.

- Chapter VII: Future Work. Discusses ideas as to future work that could be completed in each area.

II. PREVIOUS WORK

Many evolving technologies have matured to the point where it possible for large-sized virtual environment enclosures to be created. These fundamental elements include: computer hardware that is capable of high speed computations for both orientation and drawing to a screen, projectors that are capable of delivering high resolution digital graphics, software that allows efficient representation of the virtual environment, and computer networking technology that allows for efficient communication between simulations on dissimilar computers at remote locations. This chapter looks at some of the previous work in VEEs, scene graphs, and networked visual simulations.

A. PREVIOUS VIRTUAL ENVIRONMENT ENCLOSURES

We begin with a look at other VEE systems including the approaches taken, the software, and hardware that was used and benefits and limitations of these systems.

1. The *CAVE*

The *CAVE* is the recursive acronym given to the first Virtual Environment Enclosure (VEE) that was created in 1991. It stands for *Cave Automatic Virtual Environment* and is based on the simile of the cave found in Plato's Republic. It was in this cave that a person would stand and peer into the dark, only seeing the flickering shadows from the fire light and let the mind race with imagination from the images formed on the walls.

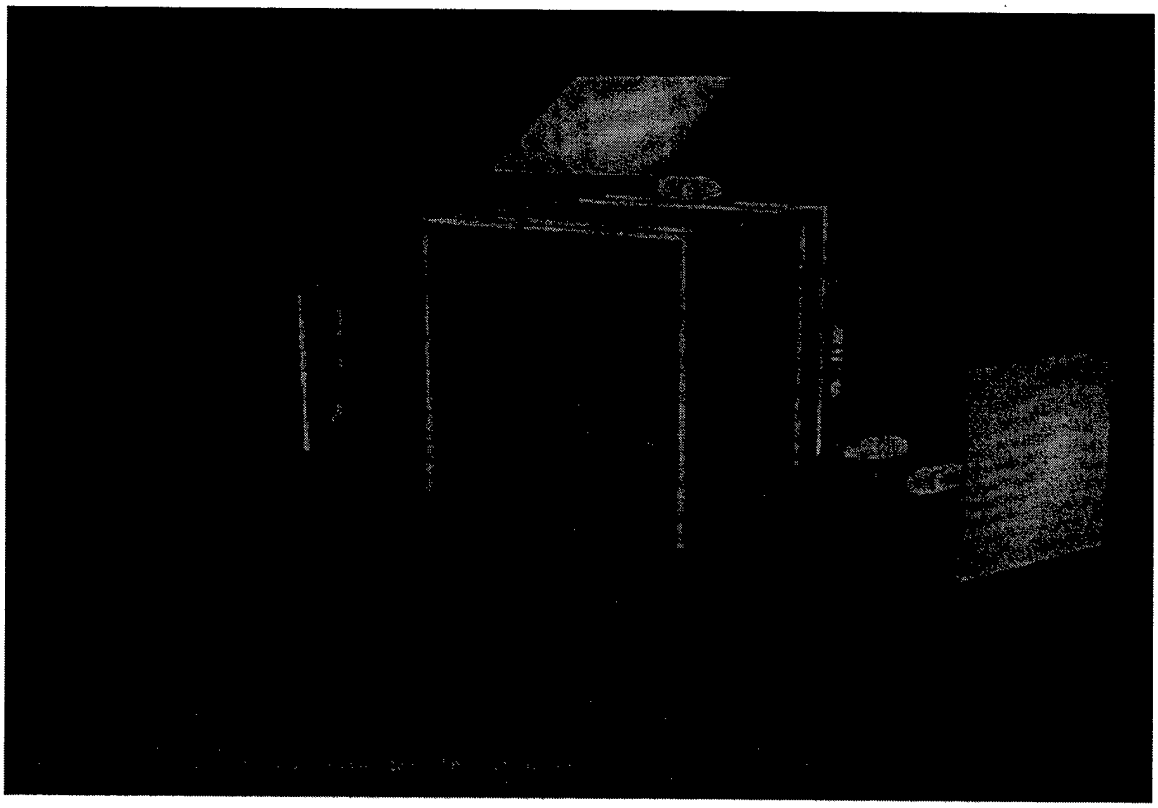


Figure 2.1. Illustration of CAVE created at Univ. of Ill. at Chicago. [CAVE00]

Created in the Electronic Visualization Laboratory at the University of Illinois at Chicago by Dr. Carolina Cruz-Neira as part of her research [Cruz-Neira93], CAVE has become the name given to all similar VEE structures. Motivation for creation of the first CAVE was rooted in furthering scientific visualization and was an entrant in the SIGGRAPH '92 Showcase effort. The goal of the Showcase was to create a life-size VEE that minimized user attachments and encumbrances while delivering high-resolution three-dimensional computer graphics that could create the illusion of reality and allow for a one to many presentation of material. The Showcase wanted to get away from the limitations of Head-Mounted Displays (HMD) to provide safer and less fatiguing sessions for multiple users.

Due to the physical limitations of the space where initial research was conducted, the first CAVE was only seven feet wide, deep and high. The necessity of folding the projection images

was made readily apparent given the size of the room. This is accomplished by having the projector shine onto one or more mirrored surfaces prior to being cast full size on the projection wall/floor. The size of the mirrors are smaller the further they are from the projection surface. An additional benefit from having the image bounced off of several mirrors is that the projectors no longer had to be pointed directly at the screen and could now be placed very close to the CAVE walls.

The reason that a cube shape was selected was the ease of calculations of the images on the projection planes. Human visual acuity is based on the linear distance from the object given that all of the light, contrast and hue factors are held constant for a specific object. This dictates that a sphere would be the ultimate projection surface for humans to participate in a virtual environment. Unfortunately computer graphics are designed to calculate images in a series of planar views. To emulate a spherical representation using planar image fields would require an immense number of these planes stacked together and then having concentrically larger circular disks removed from the center of the plane starting at the tangent point of the projected image onto the spherical screen. This also would require overlapping images from different projectors to be exactly aligned so the stereo image effect is not destroyed. It was for these two main reasons, plus the cost of having round projection surfaces made, that a cube was used to approximate a spherical projection surface. This choice also has the advantage of being the polyhedron that is the quickest for a computer to generate an image of. By having the cube be of large size, it allows for multiple simultaneous users who could walk around objects that were located between them and the plane of the projection screen surfaces.

The walls were rear projection screens to prevent the users from occluding the projected images since interposition of objects, also referred to as occlusion, is one of the most significant measures of object depth in the field of view for the human eye [Wickens, et. al.98, p. 96]. The walls were actually one continuous piece of material, with the corners formed by running an

eighth inch steel cable under tension from the floor to above the walls. Although these wires created only a small break in the continuity of the projection surface (estimated to be only one image pixel wide), they were large enough to destroy the stereo effect generated by the projectors if the object that is being viewed crosses either of the wires. Since the floor was not transparent in the first CAVE, an image was projected onto it. This floor image significantly enhanced a users experience in the CAVE. As all of the projectors were located behind the wall screens, the mirror for the floor projection to bounce off of was placed behind the head of the user. This caused the users to cast a shadow onto the floor in front of themselves. Follow on models of the CAVE therefore had the floor projector located above and behind the user with the mirror in front of them so any shadows would be cast behind the user out of sight. Most of the supporting structure was created out of wood to minimize the interference for the magnetic tracking equipment, giving better range for tracking with fewer anomalies to be corrected for by the software.

Stereo images were created for the user by using shutter glasses and having alternating images flashed upon the projection surfaces. Shutter glasses work by having an independent electronic shutter for each eye that can prevent light from going through the lens. The timing of the shutter action is controlled by sending an infrared signal to the glasses from the computer that is generating or synchronizing the refreshing of the images on the screens. When one eye is allowed to see light, the computer draws only half of the horizontal lines of resolution from the perspective of that eye's physical location in the CAVE. The computer then sends the signal to the glasses to switch shutters and at the same time removes the first eyes image from the screen. The computer then translates the same image that the first eye observed to the proper perspective location for the second eye and draws the same image. To prevent the human eye from detecting this shutter flicker the images need to be shown at a rate above the fusion frequency for humans. Fusion frequency is the point at which a series of images stops flickering and is seen as a coherent

image. This frequency is typically in the region of 40 cycles per second [Vince92, p. 138]. The CAVE delivered images at a rate of 60 cycles per second per eye. A magnetic tracker was attached in the center of the shutter glasses at five and one half inches above the eyes, allowing the computer to know where the eyes were located, but not what they were looking at.

Two major reasons were cited for the very low rate of uncomfortable simulator sickness reported. The first reason was the high refresh rate for the projected images, and the second was that for the first time the projection surfaces did not move with the users, like HMD or boom mounted viewers. Another possible contributing factor was that users could see their own bodies while in this environment, including both arms and feet.

The physical environment that the CAVE was placed in was also critical to its successful employment. The most harmful environmental effect was outside light sources that would bleed out the images on the projection surfaces. This meant that all light had to be controlled in the room where it was first constructed and necessitated building an enclosure for setting up the CAVE when demonstrating away from a controlled room. The acoustical properties of the space where a CAVE is located are also very important if using spatialized sound. The sound system for the first CAVE suffered from sound reflections off both its enclosures and the projection screens themselves. The first CAVE made no attempt at introducing haptic feedback, airflow and temperature control, vibration control, or 'smell-o-vision' to quote a popular term for introducing controlled scents into a virtual environment.

A major strength of the first CAVE was the ability to freeze position and request a more detailed image from a file server located remotely. The user would navigate through the virtual environment using their control wand, and upon reaching an object that needed to be investigated in greater detail they could freeze the motion and request a more detailed drawing of the object(s). The user would still be able to walk around the object, but the environment would no longer move. Doing this type of operation was possible with HMD's, but doing so was very

disorienting to the user and the user could not move around the object. The boom-based systems also suffer from not being able to view the object from other than the angle at which the motion was frozen.

The University of Illinois at Chicago teamed up with Pyramid Systems and Fakespace Incorporated to commercially produce the *CAVE*, and it is being widely used throughout the world [CAVE00]. CAVEs are used for problem solving, rapid prototyping, collaborative designing from remote locations, connectivity with supercomputers and also with other virtual reality devices. Examples of these are automobile design where engineers can sit in the newly designed drivers seat and see if they can easily and comfortably reach all of the switches, knobs and peddles. Major users of the *CAVE* system include: General Motors Corporation, Caterpillar Corporation, the National Aeronautics and Space Administrations, Argonne National Laboratories and the Defense Advanced Research Projects Agency. These commercialized versions use a specialized CAVE library, which takes in the inputs from the tracking devices and computes the correct transformations for the images to be displayed. The space required to set up a *CAVE* is 35 feet in width, 25 feet in depth and 13 feet in height for the standard ten foot by ten foot walls. [Fakespace00]

2. The C2

After leaving the University of Illinois at Chicago, Dr. Carolina Cruz-Neira went to Iowa State University to continue and refine her initial work. The result was the C2 that went into use in 1996 [C200]. The C2 has many technical improvements over the original *CAVE* system.

The most significant improvement is the way the corners of the C2 are formed. To eliminate the occlusion problem from her earlier work, a new method was needed to create a seamless corner. Working with the Engineering department at Iowa State University they came

up with a clamp system that has a fine edge and pinches the screen material into an exact 90-degree corner. Also the three screens are separate, vice continuous, with triangular spacers that the ends of the screen material wrap around so the clamps have a solid surface to press against. By doing this and adjusting the projectors exactly it is possible to have a continuous image in the corners that retain the binocular stereo effect since there is no occlusion. This truly makes the entire surface a useful projection screen and virtual environment creators no longer have to worry about keeping the images to a smaller size so that they fit easily onto a single panel of the display.

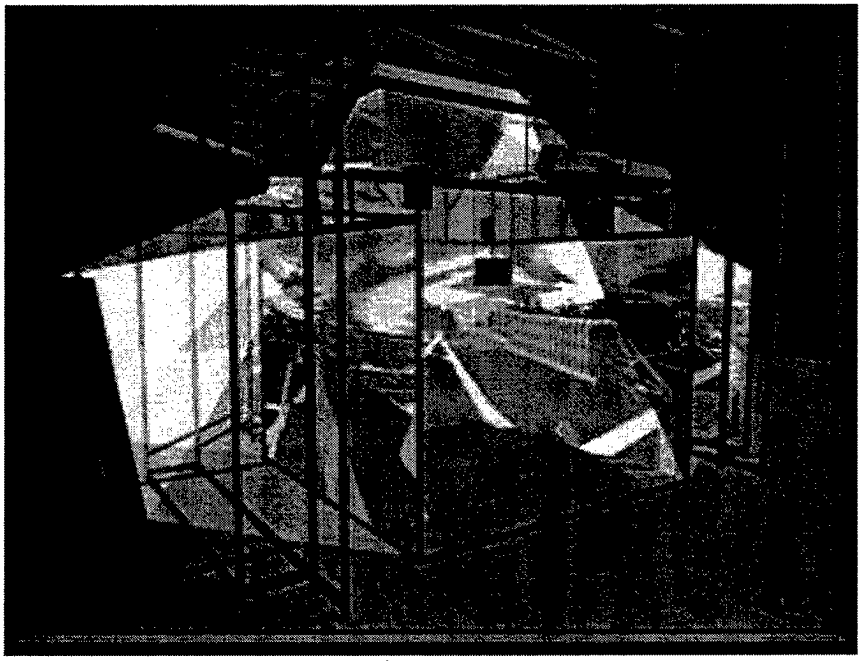


Figure 2.2. Illustration of C2 created at Iowa State Univ. [C200]

The other major innovation was in the construction of the frame that supports the entire structure. The system is called UniStrut and is made from metal tubing. All of the projectors, screens, speakers, and both infrared and magnetic tracking devices are attached to this structure. The magnetic tracking devices are suspended low enough that the metal frame is not an interference source. This design also helps with portability and construction of the C2. Disappointed with the acoustical performance of her earlier work, the C2 has a fully spatialized

four speaker stereo sound system with plans to increase to an eight speaker system. Using folded optics the C2 requires a space 28 feet wide by 21 feet deep and 15 feet high to be set up in with twelve foot by twelve foot walls.

Current work at Iowa State University is on a six-sided VEE called the C6. To house a structure of this size requires at least a three story high room or warehouse. The floor has to be made of a substance that will produce binocular stereo images, yet support the weight of several people. The biggest problem is how to get people in and out of the enclosure yet have no seems that would destroy the three dimensional images. The solution that Iowa State is going to use is to have the entire rear wall pivot at one edge and then be clamped once it is closed. The mirrors and projectors are not going to move so having the wall always go back to the same position is the only problem that must be overcome. This is not the first CAVE attempted that has users standing on a rear projection surface.

3. The CABIN

Created at the University of Tokyo, the Computer Aided Booth for Image Navigation (CABIN) first went into use late in 1996. This was a major undertaking made possible only through corporate support from around the world. The CABIN is located in a warehouse dedicated solely for its own usage and has three walls, a ceiling and a floor that are all rear projection screens. The floor is made of tempered glass and has no intermediate supports in the center. All of the seams are minute so binocular stereo images are possible in any direction except through the open back. The CABIN has a steel frame to support its own weight as the glass is extremely heavy.

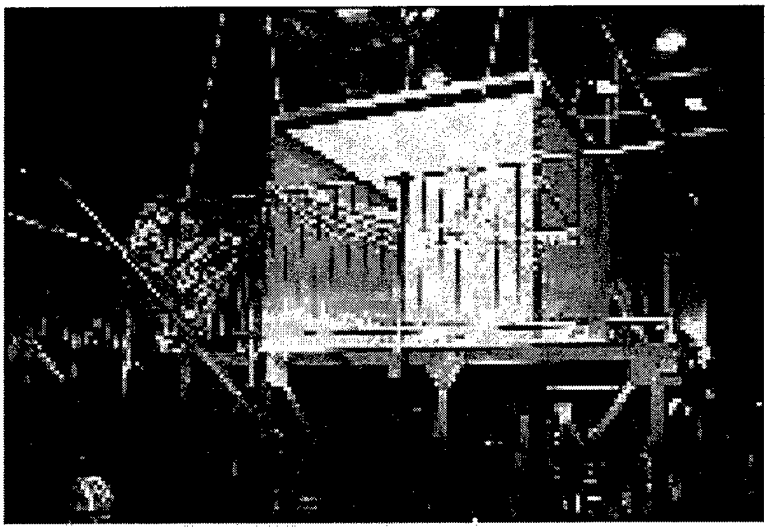


Figure 2.3. Illustration of the CABIN created at the Univ. of Tokyo. [CABIN00]

4. The NAVE

Created by the Georgia Institute of Technology Virtual Environments Group, the NAVE Automatic Virtual Environment was an effort to create a smaller scale low cost CAVE. Depending upon the options, a commercially produced *CAVE* can cost millions of dollars. The goal of the NAVE was to create a large scale VEE for around $60,000.00, showing that it was possible for smaller institutions to have access to such useful devices without business partners or major financial hardship.

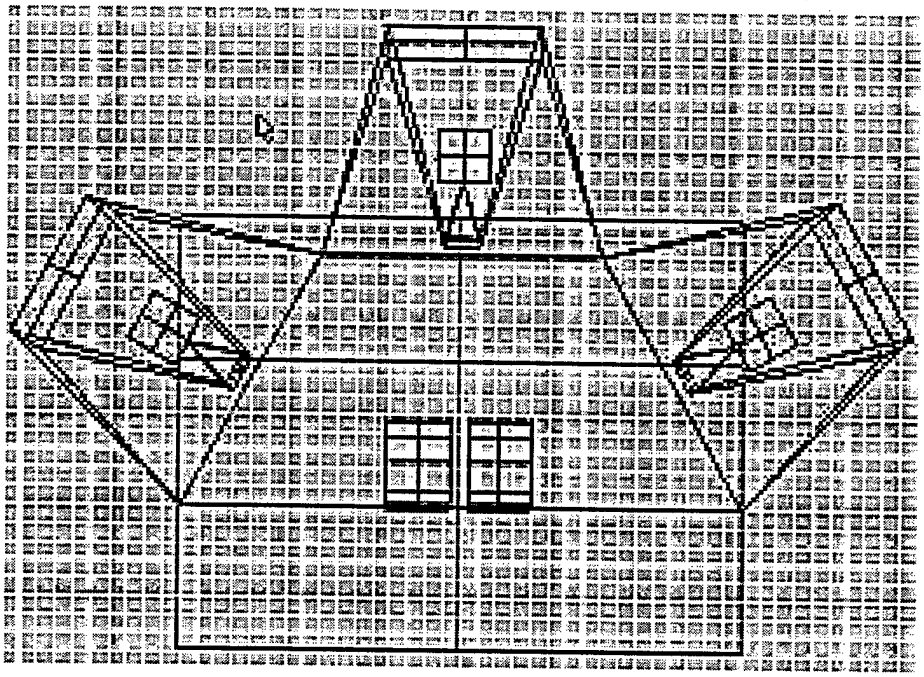


Figure 2.4. Overhead blueprint of NAVE. [NAVE00]

The NAVE is a three-screen display without any images projected onto the floor or ceiling. It is designed for two users seated in a rumble seat at the focal point of the screens. The screens are seven feet wide and have a 120-degree angle between them. This angle was chosen to better approximate a spherical screen than 90-degree corners. This choice of angles also made a single projection onto the floor incredibly difficult to achieve. The screens are rear projection rigid Plexiglas that are held at both the top and bottom. The edges are butted up against each other and were hand fitted to minimize the gap between the screens. The entire NAVE sits on a raised platform.

Conventional CAVEs allow the users to walk freely in the area between the walls. In the NAVE users are confined to sit during the entire duration of the virtual experience. This has both benefits and detractions. Benefits include not having to track where the users eyes are located since they are always seated in the same approximate location. This reduces the number of calculations needed for determining the image plane to be displayed on each screen. Also, with

the center of focus in the middle of the two users, it prevents disorienting motions when the main user and secondary users move in different directions inside of traditional CAVEs. As for motion control, with the users sitting in chairs that feel similar to bucket seats found in automobiles, a smooth driving or flying motion is very believable. When a person is standing and moving forward, a walking motion is most natural. For a VEE to incorporate this type of motion would require the projected images to be moved up and down several inches for every step, and increase the magnitude of the oscillations if the gait changes to a high speed motion such as jogging or running. A three-degree of freedom joystick is located between the two attached seats for movement inputs.

Major disadvantages of having the users seated include only two users can be truly immersed at any one time and close inspection of images from different angles is very difficult. With only two users at any one time in a device that occupies a 24 by 20 foot room, the NAVE is not much better in a cost per user sense. One of the major visualization advantages gained by having a large scale VEE is the ability to look at images between the user and projection screen plane from multiple views by having the user walk around and look over and under the object. By restricting the users to a seat they can no longer take advantage of moving around objects that lie between the screen and themselves. However, for demonstration purposes the rumble seat can be removed to allow more observers into the NAVE.

To keep overall costs down, a decision was made to employ a passive stereo image generation system. This is done by a projector that generates every other horizontal line of the image in one polarization and the other set of horizontal lines are generated using a polarization that is 90 degrees out of phase from the first. This allows users to wear inexpensive glasses that are very similar to sunglasses with each lens polarized to see only the desired half of the lines of resolution. These glasses cost less then two dollars per pair, vice the over 100 dollars per pair for the active shutter glasses. A disadvantage of the polarized glasses is users must maintain their

head orientation constant with respect to the screens. This means the line drawn from pupil to pupil must remain parallel to the floor which still allows users to rotate their heads from side to side or up and down, but not about the axis that extends out from the nose.

The NAVE takes full advantage of most of the body's sensory input mechanisms that are often ignored by the commercially available VEEs. These are sound, vibration, auxiliary lighting and air movement. The sound system is fully spatialized surround with a subwoofer that generates low frequency vibrations throughout the entire room. The rumble seat houses two shakers and the elevated platform that the screens and seat are atop have six shakers mounted underneath to simulate seat of the pants vibrations from moving over the ground. Small strobe lights are set out of direct view to add flashes to the background ambience that can be used to enhance the effect of lightning or bright flashes from weapon discharges. Small fans and heating elements are used to change the airflow and temperatures for the users. Using the fans without heat gives the sensation of motion and that of moisture in the air such as a rainstorm, while the heat can be used to enhance a desert scene or intensify an inferno.

The NAVE was created on the same paradigm as all the previous CAVEs, namely of fixed preplanned rigidity for the/ entire structure. The angles between the projection screens are designed to not be changed after construction has been finished. This can limit the overall usefulness of the VEE and restricts physiological evaluation measurements on how VEE screen angles affect the users sense of reality from the virtual scene presented before them.

5. The *RAVE*™

Introduced late in 1999, the *RAVE* has introduced a flexibility that was missing in large scale VEEs. Created by Fakespace/Pyramid Systems, *RAVE* is an acronym for Reconfigurable Automatic Virtual Environment [Fakespace00]. This has been accomplished by producing the

screen, frame, projectors and mirrors into modules that occupy a 10 by 12 foot area each. They are rear projection and come in two varieties, a center module and a side module. Side modules contain one projector and mirror, while the center module has two projectors and mirrors plus a detached ten by ten foot floor projection surface. The second mirror extends out above the rear projection screen so that the image appears on the floor. Both of the modules are 12 feet deep and have air cushioned coasters for easy movement once they are fully assembled in the space where they will be used.

The possibilities for configuration are almost limitless. The initial system consists of two side and one center module so any shape can be formed from a straight 30 foot Power Wall to a standard 10 foot cube. With a fourth module added, a five-sided CAVE can quickly and easily be configured. One drawback is that when the walls are not in a 90-degree orientation with each other, some portions of the floor inside the chord line from free end to free end do not have a projection on it. To resolve this problem would require all modules to be of the center type and software would have to control the shape of the image projected onto the floor for every possible angle configuration. Although this is possible, the additional cost makes it prohibitive. If a larger full floor CAVE is needed simply purchase an additional center section for a 10 by 20 foot four walled CAVE or two complete RAVEs for a 10 by 20 foot five walled CAVE. [Fakespace00]

B. SCENE GRAPHS

In order to maximize the usefulness and believability of virtual environments, the refresh rate the image is displayed at needs to approach the fusion frequency. Many factors enter into the refresh rate that are elements of virtual environment design and will be discussed later in this paper. The biggest and most important factor is the actual rendering and display of the image itself. Much of this factor involves culling, or clipping, the database to objects in the viewing

frustum, which is the area of the scene displayed on the monitor. The best organization of the graphical database for the culling process and hence rendering speed has become the scene graph.

1. Early Efforts

Early computer graphics algorithms used polygons, parametric surface patches, or procedures to model surfaces. These methods all were trying to improve minor areas of the rendering process to give an overall greater improvement to scene generation. They also increased the level of complexity of the surfaces, adding more curved surface algorithms to achieve a higher level of realism.

The polygonal method of scene generation built surfaces using polygons and then culled them with either a depth-first or depth-last sorting. The depth-first algorithm sorts the databases by how close polygons are to the viewpoint. The renderer then writes the polygons to a buffer, giving those closer to the viewpoint higher visible priority. As the image is drawn on the screen, those polygons with higher priority are rendered first. If the raster comes to a pixel that has already been output, it skips it and goes to the next. The depth-last algorithm sorts the polygons first by their vertical edges and then by their horizontal distances. By saving the depth sort for last, this process attempts to minimize the database to be sorted.

Parametric surface generation algorithms offer improvement for shading and for drawing curved edges. They use surface patches, such as Bezier patches or Non Uniform Rational B-Splines (NURBS), to draw smooth models. The tradeoff is that the algorithms are no longer linear and therefore much more mathematically complex. The best way to minimize the mathematical slowdown is to recursively divide the surface patches until they are pixel sized and then draw them. If an object closer to the viewpoint already uses the pixel, the patch is not rendered.

The procedural method uses geometric primitives, such as triangles, to build larger objects. This allows rendering similar to polygonal methods, but the geometric primitives need be compared only when they overlap one another.

2. Clark's Hierarchal Geometric Models

Clark introduced a revolutionary new algorithm that used a very structured graph format to store the graphical database. Previous structures attempted to store objects by defining where they were positioned or how they clipped objects to the viewing frustum. Clark combined these techniques and extended them to provide methods and structures to revolutionize graphics programming.

The first improvement Clark offered was a useful way to vary the amount of detail in a scene. This is done by the use of multiple Levels Of Detail (LOD). When an object is close to the viewer, it is displayed as a high-resolution model. When it reaches a set distance from the viewer, the renderer switches to a less complex lower-resolution model. This allows more detail on foreground objects while keeping the overall complexity stable. Clark's scene graph allowed for these multiple LODs by providing switching methods based on distance from the viewpoint. [Clark76, pp 550-551]

Another improvement offered by Clark was a better clipping algorithm. This algorithm did a logarithmic recursive search of the scene graph to clip objects to the viewing frustum. It identified what nodes were relevant by comparing them to a simple volume to determine if they were viewable. If the node was not in the volume, anything below that node in the graph was skipped in the test.

The next improvement was defining a "graphical working set" that identified what parts of the scene should exist in memory. Most complex environments are too large to store the entire

database in memory. The graphical working set defined what was in the current frame and what could be in the next frame. By keeping only this portion of the database in memory, a type of memory management occurred which is similar to paging and improves the frame rate.

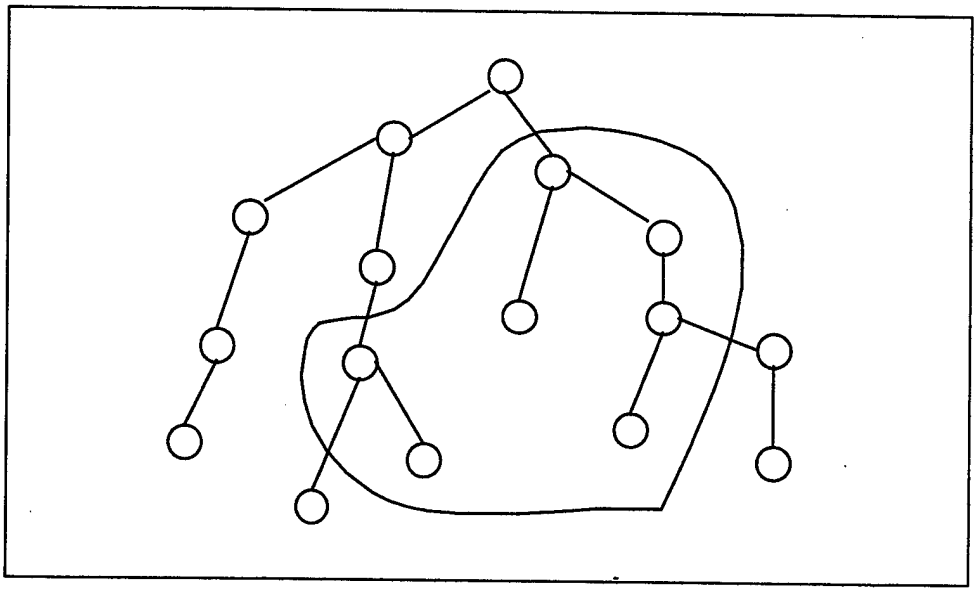


Figure 2.5. Sample graphical working set. Anything not in the contour is ignored for rendering.

The final improvement that Clark offered was a recursive descent, visible surface algorithm. This algorithm organized the database by bounding volumes. If the bounding volumes of objects overlap then one or more is visually occluded signaling it and its descendants need not be considered for rendering. The recursive part occurs since this is done for every node in the graph and its descendants. This is also done for the frustum culling, minimizing the amount of geometry that must be clipped. The sorting thus completed provides an extremely streamlined and easily navigable structure for visual environments: the scene graph.

3. Current Scene Graphs

One of the first commercially successful scene graphs was Silicon Graphics, Inc.'s (currently called sgi™) Performer™. It was developed to help users of sgi computers to realize the full potential of their machines. Performer was built on a multi-threaded model (app/cull/draw threads) that could take advantage of multi-processor sgi's and further enhance the capabilities of graphics applications.

Performer consists of two libraries, `libpf` and `libpr`, that contain the C programming language classes to perform display algorithms similar to those proposed by Clark. The `libpr` library contains the high speed rendering algorithms, while the `libpf` library contains methods that provide enhanced support for visual simulations.

The `libpr` library is a low-level library that provides many methods and objects to generate high performance graphics. The high performance geometry rendering is accomplished using `pfGeoSet` objects that bypass CPU bottlenecks in their rendering. Display lists, as instantiated in `pfdispList` objects, provide for better organization of geometric primitives. Finally, the `libpr` library also contains all the math, intersection and memory management methods needed.

Conversely, the `libpf` library is a higher level library that makes calls to the `libpr` library so that the user does not need to directly access it. The `libpf` library mainly provides multiprocessor support, rendering pipelines, and scene structures to store the visual database. The rendering pipelines also allow multichannel rendering to facilitate multiscreen displays such as CAVEs. The scene structure contains all the nodes, LODs, and traversal functions needed for compelling visual simulations. [Rohlf/Helman94]

A commercially available visual simulation development toolkit called Vega has been built upon Performer. Vega abstracts Performer one more level by making all of the scene graph

calls for the developer. It has a Graphical User Interface (GUI) that allows the user to quickly build an environment in much less time than programming in Performer requires. Conversely, Vega allows the advanced user to build specialized implementations that Vega will then execute with a scene file. Vega will be discussed in more detail in Chapter 4.

Although Performer is extremely powerful and arguably the best scene graph available, it is available only for Irix™ platforms. However, much of the computing world has shifted away from traditional Unix platforms to less expensive alternatives. This shift requires that a virtual environment be cross-platform to expose it to the widest set of potential users. Java is developing as the language of choice for cross-platform software developers with its "compile once and run everywhere" philosophy. Sun Microsystems has concurrently developed Java3D, a cross-platform scene graph companion to Java, for rendering three dimensional graphics using Java.

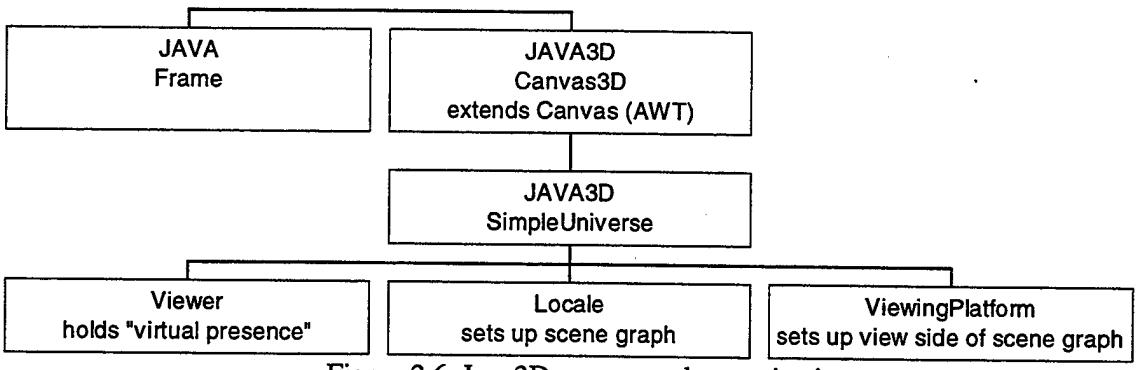


Figure 2.6. Java3D scene graph organization.

Java3D uses the same object-orientated approach as Java. A Java object called `SimpleUniverse` holds the information for the scene. The `SimpleUniverse` has a `Viewer`, `ViewingPlatform`, and `Locale` associated with it. The `Viewer` controls what the user looks like both to himself and others. The `ViewingPlatform` determines what the user sees. The `Locale` holds the geometry information for the graphical database. Java3D has advanced culling and scene graph traversal algorithms built in to ensure the rendering process takes as little time as possible. Ease of construction plus built in algorithms make Java3D a

viable alternative for cross-platform applications. Java3D will be discussed more in depth in Chapter 5.

C. VISUAL SIMULATIONS

Visual simulations are the underlying graphics protocols for virtual environments. Fairly new, they are just now becoming more popular as computer technology advances. What once required a high-end Silicon Graphics machine can now be run on a desktop PC. This section will examine the history of two of the major visual simulations. These simulations were both networked, allowing hundreds of users to interact, and provided the inspiration and example for many developers to follow.

1. SIMNET

SIMNET (SIMulator NETworking) was developed by the Department of Defense to provide inexpensive networked military simulators for training small units. Heterogeneous forces would be able to train together to practice combined arms techniques. It was designed to provide the majority of a soldier's training but still required the soldier to complete some field training [Singhal/Zyda99, p. 20]. Although it was not intended to replace all of a soldier's field training, SIMNET allowed soldiers to simulate large, expensive, and hazardous scenarios.

SIMNET used an architecture where each entity was an object that caused events to occur. Each object was responsible for determining its own position, initiating events to affect other players, and updating its state to reflect events affecting it. Each object sent out network packets that either reported its state vector, announced an event, or acted as a heartbeat to confirm

its presence in SIMNET. By having each entity be an object, SIMNET enabled the entities to be autonomous, thereby eliminating the need for a client/server architecture and its vulnerability to single-point failures. Each entity also implemented dead reckoning routines to control the display of other players between packets. Dead reckoning allowed the overall number of packets sent out by an entity to be smaller and reduced entity jumping due to packet latency. SIMNET also allowed fully destructible terrain, however any terrain feature that could be destroyed had to be an object with its attendant network packets. [Singhal/Zyda99, p. 21-25]

The network architecture used by SIMNET was very attractive to visual simulation developers. Unfortunately, the architecture was not well documented. After SIMNET had been running a few years, an attempt was made to formalize the SIMNET networking architecture. This resulted in the first DIS standard. DIS provided a protocol that allowed anyone on any platform to network with other people as long as the developer implemented the DIS standard. This allowed academic and commercial ventures to participate in SIMNET and other visual simulation research.

The inheritance of DIS from SIMNET is evidenced by the same autonomous object and event based protocols. DIS uses a Protocol Data unit (PDU) that has data fields allowing both object and event information. An IEEE DIS standard officially [IEEE98] defines 27 PDUs, however most implementations utilize only four: entity state, weapons fire, collision, and detonation. This is due to two factors, one being that some PDUs were introduced to the standard to provide special features for their developers, and that implementing more than those four would very easily lead to network congestion. Like SIMNET, each entity is responsible for issuing the PDUs for the events it instigates. Each entity is also responsible for updating its state based on PDUs it receives. Once again, DIS is autonomous and thus does not need a client/server architecture. Because of its wide acceptance, cross-platform capabilities, and serverless

architecture, DIS was chosen as the networking interface for the two simulations built for this thesis.

2. NPSNET

While DIS was in its infancy, the Naval Postgraduate School was asked by the Department of Defense if simulations could be built that would connect to and interact with SIMNET. The Naval Postgraduate School (NPS) had been developing visual simulation software to run on low-end sgi workstations. By the time DOD asked them to help with SIMNET, NPS had built a missile simulator (FOG-M), a vehicle simulator (VEH), and the Moving Platform Simulator (MPS), which was a networked virtual environment that could support over five players. [Singhal/Zyda99, p. 38]

The original NPSNET came from taking the MPS simulation and improving it to read SIMNET terrain files. Although it was a follow on effort, NPS-Stealth, that was finally able to interact with SIMNET, the original NPSNET received widespread attention from both government and academic institutions. The first NPSNET system had no dead reckoning; instead each entity sent a state packet every frame. NPSNET-II and III built on the original NPSNET and had better graphics and could read larger terrain databases. NPSNET-IV was developed to use Performer after it was released by sgi. NPSNET-IV also used DIS as its network protocol and implemented dead reckoning. In its debut at SIGGRAPH 93 it was able to network up to 60 players.

Since its initial development, NPSNET-IV has become the template for a successful networked virtual environment. It can interact with any other simulation that uses DIS and has been improved by various thesis students to add human articulation, special effects animation, and has been the test bed for many other efforts to improve immersion in virtual environments.

NPSNET-IV was able to access the Multicast Backbone (Mbone) of the Internet to interact with widely dispersed sites. Future development of NPSNET-IV has been stopped in favor of NPSNET-V, which is currently under development. [Singhal/Zyda99, p. 40-42]

Conclusion

This background work has provided the tools to make a large-sized VEE, connected by computer networking to other remote computers, both economical and relatively quick to construct. Of course these low cost solutions will not have all of the capabilities of the large commercial systems, but multitudes of useful research can be accomplished in them.

III. PHYSICAL CONSTRUCTION OF MAAVE

This section will cover the planning and physical construction phases of the MAAVE. The major constraints were the rooms available on campus, allocated budget and physical location of computer hardware. Two doctoral candidates started the initial planning for the MAAVE by ordering the rear projection screens and Liquid Crystal Display (LCD) projectors, but the construction of a large-sized VEE fit well with the software project that was the basis of this thesis.

A. PLANNING THE LAYOUT OF THE MAAVE

The size of the image, given distance to the screen, delivered by the projectors coupled with the floor space of the room available to house the MAAVE determined the maximum size of the screens that could be used for the first MAAVE. The physical dimensions and obstructions of the room are displayed in Fig. 3.1. The usable entrances to the room, Spanagel Hall room 240, are from the main hallway or the room 242 since the door to room 238 is blocked from the other side. To maximize the floor space for the enclosure, the entrance was decided to be from room 242, keeping the door to the main hallway as an emergency exit.

The projectors are the VRex 2210 and have an advertised maximum width of projection of seven feet one inch and height of five feet six inches with a screen twelve feet eight inches from the projector. A planning tool was created from graph paper that represented the projection cone of the 2210 to allow for an estimate of the minimum sizes of the two mirrors and the maximum size of the projection screens. Since the tool was made from graph paper it could be folded twice to represent the bounces off of the mirrors. The height and width of the mirrors

were derived from the above dimensions using the 1024 by 768 pixel aspect ratio that the projector displays. The screen size was selected as seven feet wide by six feet tall, while the secondary bounce mirror size of five feet wide by four feet tall and primary bounce mirror of 18 inches wide and 15 inches tall were selected to allow for adjusting the placements of the mirror.

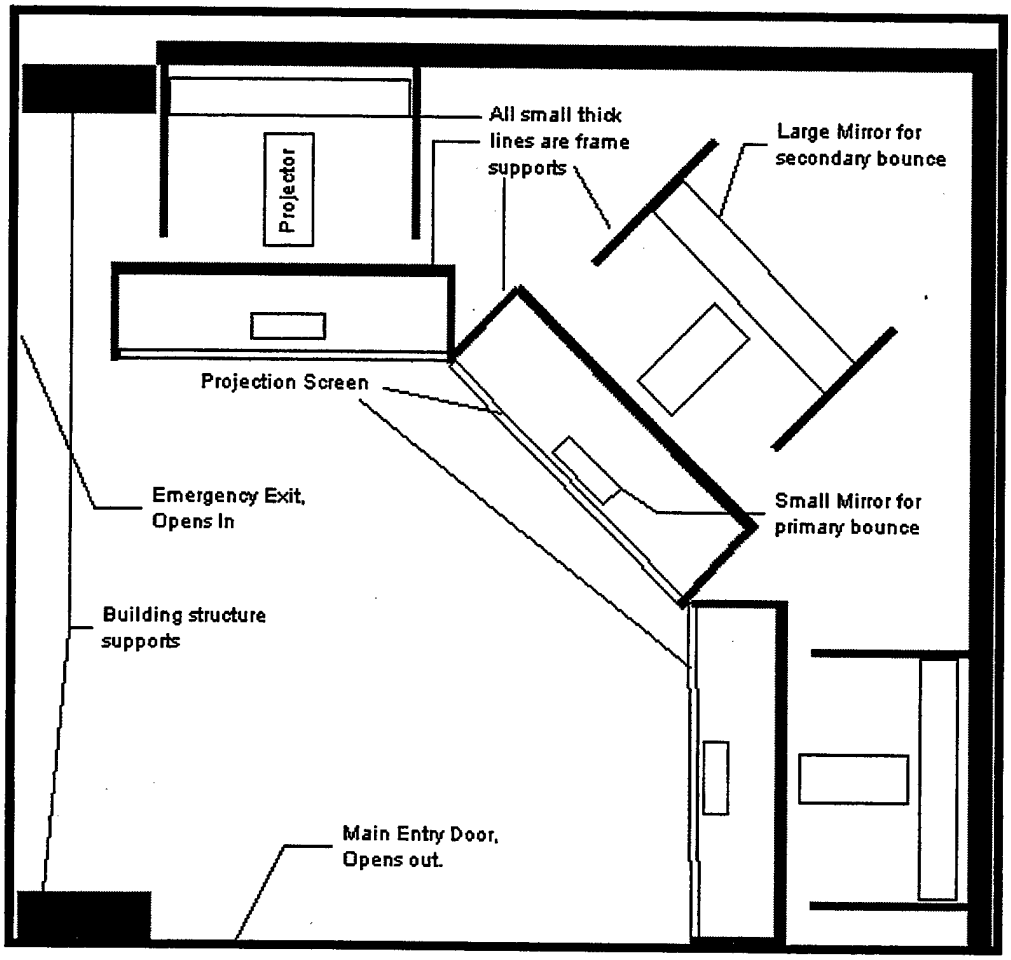


Figure 3.1. Floor layout of Spanagel Room 240

Once the room had been selected and the size and shape of the MAAVE were decided upon, setting the characteristic properties of the room and creating stands to hold the screens and mirrors needed to be done. Factors that weighed in these decisions were: desired acoustical properties of the room, controlling all sources of light entering the room, not obstructing the light path from the projector to the screen, setting the number of observers and their height of eye,

composition of the floor and whether or not to build a raised platform, and how to get computer graphics to the projectors.

Since the only other known VRex based VEE, the NAVE (see section II.A.4), used a raised platform, and the design of the screen holders depended on this decision, this was the first to be made. Since a large volume of visitors would be passing through the MAAVE, keeping everyone on the floor seemed to be the most cost effective and lifecycle prudent. The floor had hard tiles, which would reflect both light and sound, so dark industrial grade carpeting was installed. To keep costs down the floor was not made into a single level plane. A level floor would have made fine tuning the images and joining the screens easier, but since the MAAVE was created in components each individual piece had to be made level relative to the center screen.

Two of the walls in the room were smooth finished concrete; one was almost entirely windowed and the last was completed only two-thirds of the way to the ceiling. Although the large windows had venetian blinds, these still allowed far too much light to enter the room and had to be darkened. Curtains were installed which helped dampen echoes, but the light that leaked over and under them was still too disruptive. The solution was to take the packaging the screens were shipped in and attach them to the window frames. The short wall was completed to the ceiling by the school facilities department, which resulted in a light controlled room with a terrible echoing problem and highly reflective white walls. Acoustic ceiling tiles were attached to the three non-windowed walls. A watered down black paint was used to darken these tiles without filling the holes that provide sound dampening.

B. GRAPHICS GENERATION HARDWARE

Connecting computer graphics to the projectors was a major concern since the type of computer or computers to power the MAAVE were not decided upon until less than two months from the scheduled completion date. The choices were between using the five-year-old Infinite Reality™ sgi computer in the graphics research laboratory, a single Intergraph dual Pentium™ II personal computer (PC) or buying three Athlon™/Pentium III PCs and networking them for synchronization of display.

The Infinite Reality solution had the following benefits:

- Best graphics rendering speed for complex visual models in the computer science department.
- Multiple display output option already installed.
- Native platform for Vega software.

Unfortunately, the age of the computer was only one of the following detriments for its utilization:

- All signals needed to be amplified and sent 150 feet down the hallway to room 240 at a total cost of around $2,700.00.
- Only one RM6 graphics board was installed and it did not have sufficient memory to store all of the textures for the Herrmann Hall model.
- A second RM6 graphics board would cost around $6,000.00.
- A remote terminal would need to be set up in room 242 to access the Infinite Reality computer.

- In the current configuration the Herrmann Hall walkthrough program ran as slow as ten frames per second and the second graphics card would not result in a significant increase of the speed for drawing.

Although this computer was designated to drive the graphics for the MAAVE at the beginning of the project, the hardest decision was not to upgrade and use this computer.

The Intergraph PC solution was to work in conjunction with the Infinite Reality solution. The decision not to upgrade Infinite Reality computer left this PC as the only graphics source for the MAAVE that could be implemented before the conclusion of this thesis. Benefits of this PC are:

- Three Wildcat graphics cards are the standard output.
- Dual Pentium II 400MHz processors, which benefit Vega.
- Uses WindowsNT™, which benefits Java3D.
- Contains 512 Megabytes of RAM.
- Very portable so configuration with the projectors is easy.

Detriments for this Intergraph PC include:

- Age is over two years old.
- Graphics are delivered over a PCI bus vice AGP bus (the controller for the graphics uses the AGP bus and each of the Wildcats only utilize PCI slots).
- Uses the VegaNT™ software, which is not as fast as Vega on an sgi platform.

The decision to not upgrade and use the Infinite Reality computer was based on the cost of nearly $10,000.00 that would not boost its performance compared to that which could be obtained from three new PCs networked together. Since this decision came late in the construction phase and the computers would not arrive before this thesis was concluded, further details of this option can be found in Chapter VII.

High traffic volume and minimizing costs dictated the MAAVE would be used by standing in front of the screens or having a row of chairs in the front with a second row of observers standing behind them. A consensus guess on the average height of eye for these uses was just below five feet. This height was designated for the middle of the screens. This also meant the center pivot of the large second bounce mirrors would be located below this height due to the upward angled output from the VRex projectors. The actual construction of the screen and mirror frames is covered in the next section.

C. CONSTRUCTION OF SCREEN AND MIRROR FRAMES

With 2 x 4 wood selected as the construction material, standard housing construction designs were used for the wall like screen stand. The difficulty was in how to support the top of the screen from behind without blocking any of the reflected light that would occupy the entire width of the screen. With the screen material six feet high by seven feet wide the base support under the screen was chosen to be 21 inches. Screen thickness is ¼ inch, so a ¼ inch wide by ½ inch deep groove was routered into the top of the 2 x 4, located ½ inch from the front edge of the support. Figure 3.2 shows that the screen sits in this groove in a seven foot long 2 x 4 while the bottom 2 x 4 is not as long. This was done to allow the horizontal supports to extend backward from the front of the screen without adding to the width of the screen frame.

To save weight and help prevent bowing the screen, a 2 x 2 was used to hold the top of the screen. Again this board was seven feet long and had a groove routered into it of the same dimensions as the board that holds the bottom of the screen. To add counter weight for balance a seven foot wide by eight foot tall frame was made to hold the top 2 x 2 board in place. This frame is attached the screen support by the horizontal support from the front screen and a diagonal support to maintain the frame in a vertical position. From the top horizontal bar three 2

x 2s extend forward to support the 2 x 2 that holds the top of the screen. This configuration leaves the entire back open, except the 2 x 4s that go around the perimeter of the screen. These boards do not interfere with the projected image since the image is not full size when it passes through the plane of the back frame. Since wood will always expand and contract with changes in temperature and humidity, plywood triangles were nailed onto all joints where maintaining the angle of the joint was critical.

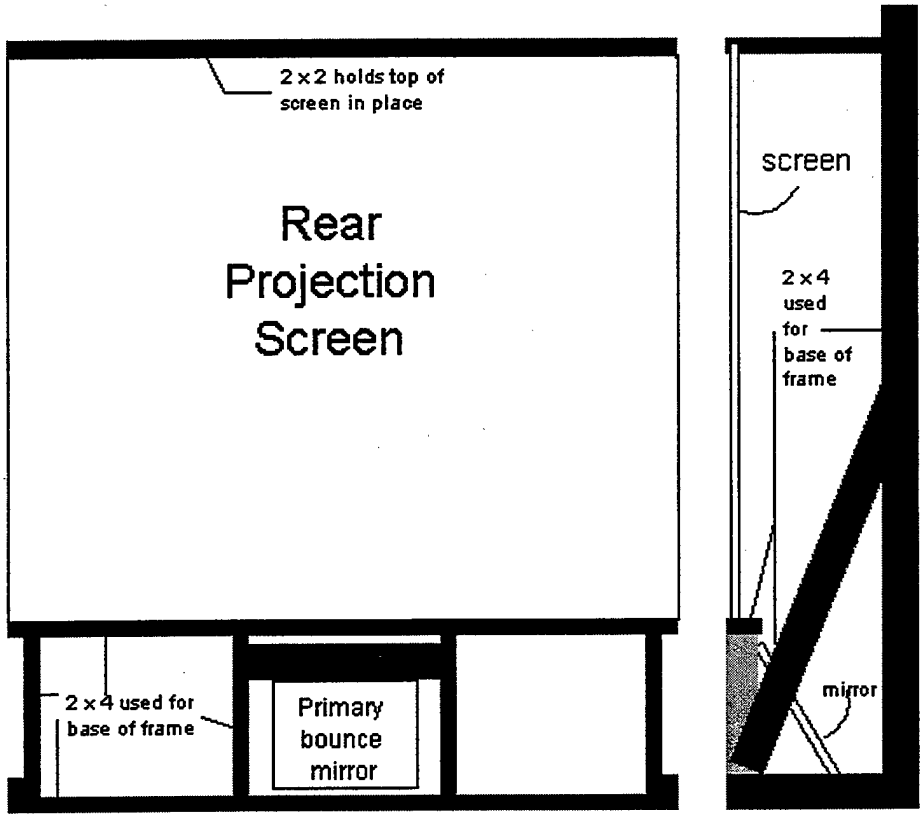


Figure 3.2. Front and side views of screen frames.

The finishing touches to the screen frames were to cut a 45 degree corner from the back of the screen grooves to the front edge of the frames. This same cut was done on the top screen holder and also the boards at the bottom of the frame. This exposes the edges of the screens to allow them to meet flush against each other and also allows for the screens to be set at multiple angles from 135 degrees and smaller. Flat metal brackets were used to hold the top and bottom

boards with grooves beside each other so the screens would touch. The center screen also had two pieces of veneer attached to the outer sides of the frame to prevent the images from the side projectors from bleeding over onto the center screen and vice versa. The edges on the screens were very roughly cut by the manufacturer and did not form neat joints. All four joining screen edges had to be ground down by hand tools to minimize the gaps along the seams. This process was very meticulous and difficult since large tools such as planers or router tables could not be used. The gaps could only be reduced to approximately one millimeter. Clear silicon caulking was used to fill the joint where the screens met and helps to minimize the seam by allowing light from both projectors to commingle in the seam.

The primary bounce mirror is attached to the center of the back of the screen frame and is allowed to pivot towards the projector so the image can be bounced upwards to the secondary bounce mirror. The primary bounce mirror is held in a three sided frame consisting of 1-1/2 inch wide by 1/8 inch thick angled aluminum. A hinge is attached between each of the free ends and the screen frame, and these allow pivoting of the primary bounce mirror. Slotted track guides (commonly used to hold open the lids on cedar or similar type chests) are attached to the bottom of both the mirror frame and the screen frame to allow setting and holding of particular angles using the tensioning screws.

The large secondary mirrors are heavy (approximately 70 lbs.) and need to be suspended above the ground. They also must be able to tilt to allow tuning of the images. The five foot wide by four foot high mirrors are set on top of a 2 x 4 frame. The mirror is loosely attached to the front of this frame by the same aluminum angle metal used for the smaller mirrors. Cardboard is used as the gasket material and to keep the mirror from being warped by the wooden frame. When the mirror is in position it actually hangs on top of the metal since it is trying to fall down and away from the frame. It is hoped that over time this will keep the optics truer than other VEEs that have used wood to hold mirrors. ½ inch diameter bolts were placed through the

center of the sides of the wooden mirror frames to support pivoting. This makes the side with the mirror always seek to go down to a near horizontal position and provides a means for setting the mirror angle.

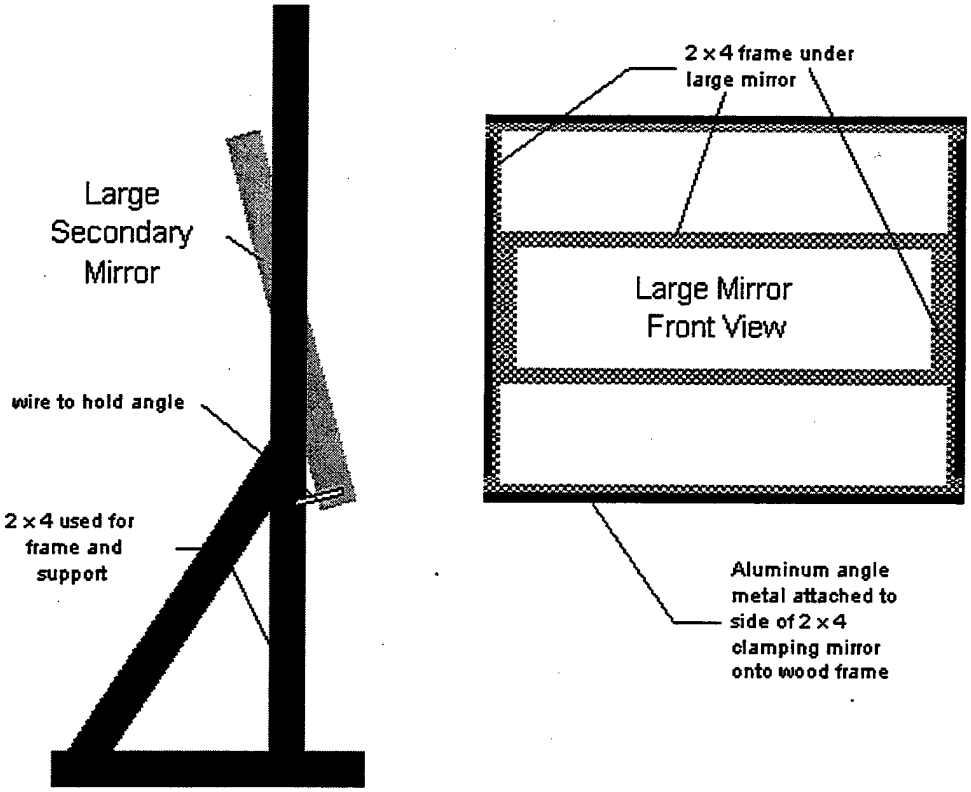


Figure 3.3. Secondary mirror frame and stand.

The frame that suspends the mirror is made from 2 x 4s and is similar to the back support for the main screen. As can be seen in Fig. 3.3, the horizontal support has to be extended forward (instead of being centered) since the walls limit the placement of the mirror stands. At first the pivot bolt was placed at the height of the center of the screens. The calculation of this initial height neglected to compensate for the image from the projector being skewed in the upward direction. Consequently pivots of the mirrors were lowered to approximately four inches below the center height of the projection screens. Two wires, one on either side, maintain the second

bounce mirror's angle by extending from the mirror's wooden backing to the suspension frame and taking advantage of the tipping motion mentioned earlier.

D. TUNING THE IMAGES ON THE SCREENS

The last phase of construction was matching the image to the screen. The composition of the screen frames allows for the image on the screens to be slightly wider than the screen. This is crucial to using software to help align the multiple screens. Also, the projectors themselves have the ability to slew the projected image up, down, left or right. Prior to attempting image alignment, all of the projectors were set to have the images centered to enable use of the slewing as needed.

In order to keep a squared image on the screen, the light must travel the exact same distances from the projector to all places on the screen (as demonstrated in Fig. 3.4.) This was easiest to achieve by keeping the projector perpendicular to the screen and parallel to the floor. The upward angle from the projector caused the primary bounce mirror to be raised from its initial position just as it caused the large mirror to be lowered. In the initial configuration, the image on the screen did not have parallel vertical lines until the bottom of the image was two feet up from the bottom of the screen. Reanalysis of the angles for the mirrors led to the appropriate corrections in height for each mirror. The small mirrors were raised first, but could not be placed high enough without blocking the image on the screen. This meant the second bounce mirrors had to be lowered. Attaching the primary bounce mirror to the screen frame ensured the horizontal plane of this mirror would stay parallel to the horizontal plane of the screen. To position the second bounce mirrors so that its horizontal plane would be parallel to the screen, simply keeping both sides of the mirror equidistant from the screen would suffice.

The projectors were placed so the size of the images produced was two inches wider than the screens width. This allows for adjusting the image in software for each screen to try and obtain an exact pixel match from one screen to the next. Also the edges of the images tend to have minor waviness from either imperfections in the screens and mirrors, or torques applied to the screens and mirrors. Two of the three screens stand almost perfectly vertical, while one of the screens bows backwards. To compensate for this bowing the top of the top support board is rotated towards the back helping to counter the bowing in the top of the screen. A 1/8 inch diameter rod had to be used to push out the bow in the lower half of the screen. This rod was notched in the top so only half of it touches the back of the screen, making it invisible from the front side of the screen.

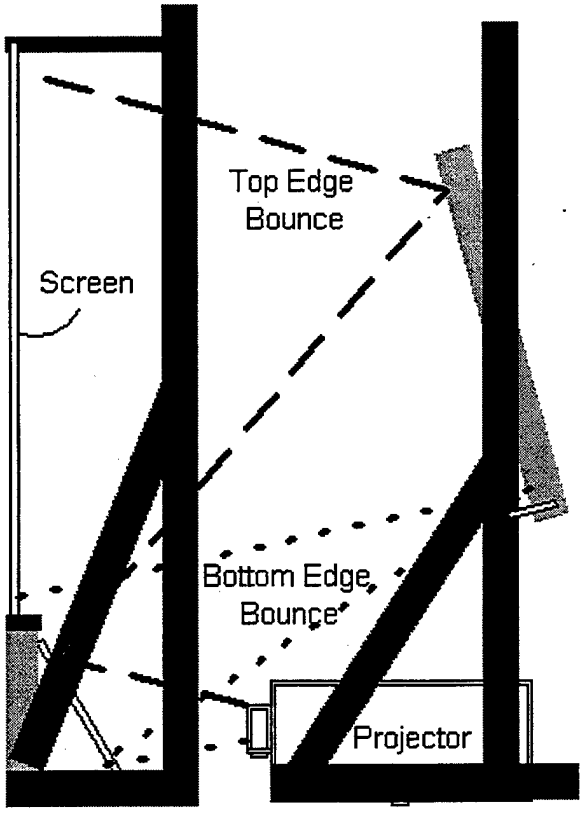


Figure 3.4. Bounce pattern from projector to screen.

The image on the right hand screen was set first due to its physical location constraints. Next followed the center screen and lastly the left screen image so each would align with the previous. Once the images were squared with respect to the screen, the actual heights of the images were adjusted by using the slew of the projectors. Having the image appear as continuous was done by slightly overlapping the fields of view in software so that the duplicated pixels were projected onto the veneer between the screens.

After setting the computer in room 242 and running video cables to the VRex and then back to the monitors, a ghosting problem was discovered. The sensitivity of the LCD projectors further manifested itself when new video cables were installed so the left and center projectors only had two segments of cable going into the projectors. The image on the monitors was clear, but the projectors still had difficulty display a proper image, especially in stereo mode. The solution was moving the computer into the room so all projectors only needed a single cable.

The finishing touches were to sew six black sheets together to use as a canopy for the MAAVE. This canopy extends from the top support boards to the walls opposite each screen and are attached by snaps to allow for easy removal to access the speakers and other equipment that might be installed above the screens. The center of the canopy is supported from the suspended overhead fluorescent lights that are in the room. Black bunting material is used to cover the bottom frames under the screens and the side access to the far left of the left screen. This thicker material still allows sound from the speakers inside the screen frames to pass through, but helps block residual light from the projector bulbs.

E. HERRMANN HALL MODEL CONSTRUCTION

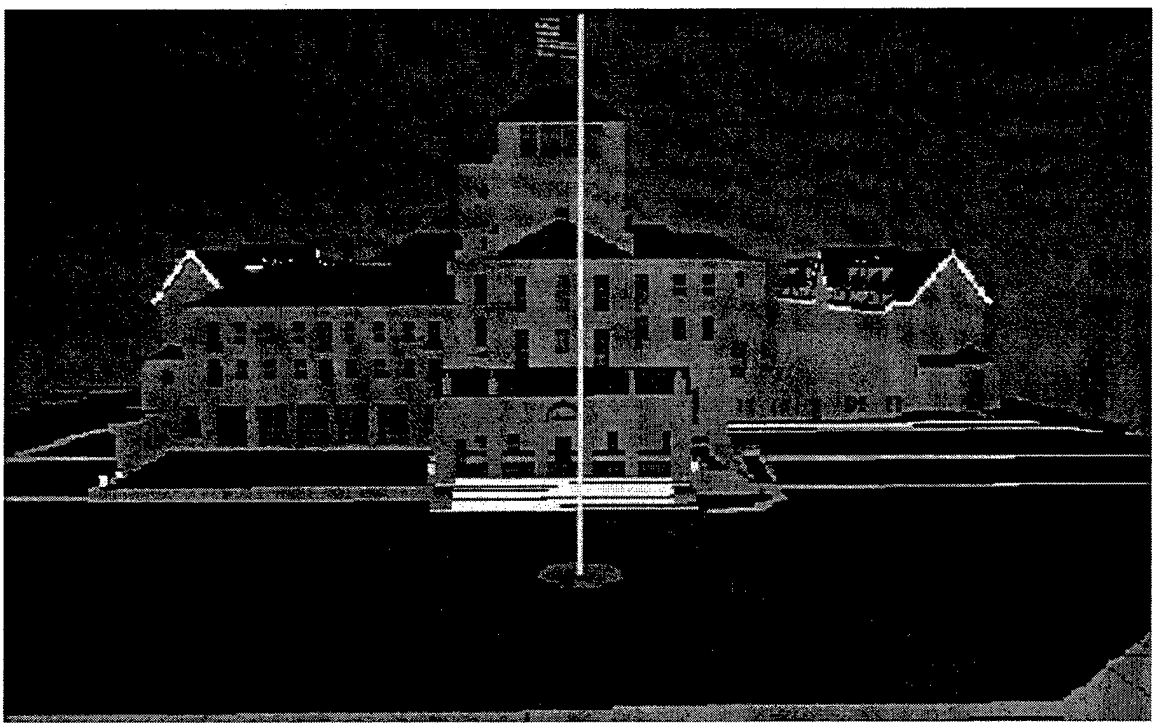


Figure 3.5. Front view of Herrmann Hall model.

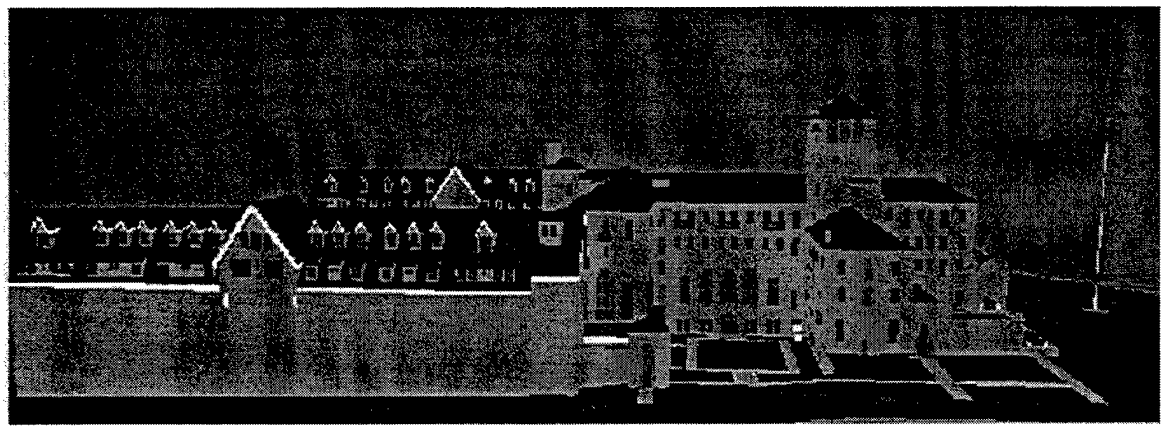


Figure 3.6. Side view of Herrmann Hall model.

The model used in this simulation is a model of the NPS administration building, Herrmann Hall (Fig. 3.5 and 3.6.) John Locke built the model for the NPSNET-IV networked

virtual environment using Multigen™ Creator™. The model is approximately 10,000 polygons with 92 different textures. This model was chosen primarily because of its detail and ready availability. The other main reason this model was chosen was because the file format could easily be loaded into both Vega and Java3D.

This model did not utilize any optimization techniques during its construction and therefore poses a significant challenge to graphics hardware and software. All textures are separate image files, and all of the polygons are separate shapes that are not part of meshes. Textures vary in size from 512 by 512 pixels down to 16 by 16 pixels, with most under 64 by 64 pixels. By having separate textures, the software must load each texture into the texture hardware each time it is referenced, causing a multitude of graphics state changes every time a frame is drawn. Not having continuous shapes, like walls, as polygon meshes causes a large increase in the number of vertex coordinates that must be sent through the graphics pipeline. Additionally, no spatial separations were delineated in the model that could facilitate culling algorithms.

IV. VEGA IMPLEMENTATION

Vega and VegaNT, by Paradigm Simulation Inc., were chosen for the commercial software since their underlying scene graph implementation is Performer or Performer-like. Performer has been recognized as one of the quickest and most efficient scene graphs in wide scale use. The introduction at the beginning of the Vega LynX™ User's Guide is so succinct it will be repeated verbatim here. "Vega is a software environment for virtual reality and real-time simulation applications. By combining advanced simulation functionality with easy-to-use tools, Vega provides a means of constructing sophisticated applications quickly and easily. Vega supports rapid prototyping of complex visual simulations. LynX is the graphical user interface for defining and previewing Vega applications. These Vega applications are programs that you create using the Vega development environment or using a basic Vega executable provided with all Vega packages." [Paradigm: LynX98, p. 11]

A. RUNNING VEGA USING THE LYNX INTERFACE

As stated in the chapter introduction, LynX is the GUI used for the rapid creation of complex visual simulations that run in real-time. The GUI, Fig. 4.1, shows up as a standard Window with icons attached on one of the sides, the panel view which shows the fields for each of the icons, and the toolbar and menus across the top. The end result of using LynX is the creation of an Application Definition File (ADF) that holds all of the unique settings chosen for a particular simulation that can be run with compatible Vega executable files. An ADF is not needed if a programmer puts the multitude of settings required for a simulation in the executable file, but then modification can be tricky and time consuming.

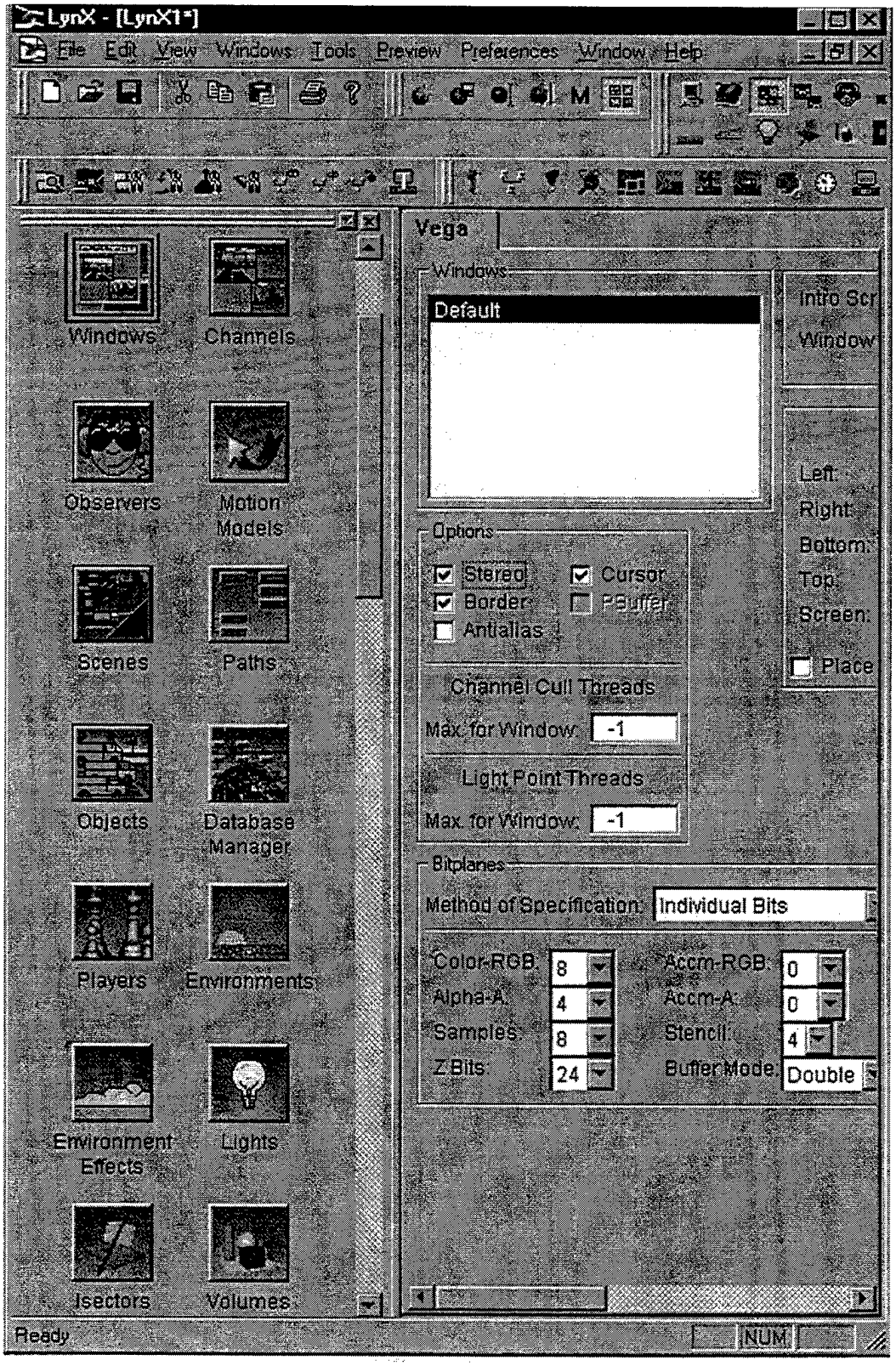


Figure 4.1. LynX Graphical User Interface screen capture.

1. Using LynX

Starting LynX is very easy once Vega is installed. LynX will launch with the default ADF open that can be immediately run with the default Vega executable. Either with a quick tutorial from an experienced user or with the help of the LynX User's guide, this start-up process is very short. Then learning the intricacies of the relationships between data fields begins. One exceptional advantage of using the Active Preview is the ability to change some of the ADF values in the LynX GUI while the simulation is running and witness the results immediately.

The LynX interface is presented identically in both the Irix and WindowsNT versions, which makes the platform used for creation of a simulation irrelevant. Excellent electronic documentation is provided with software. All of the LynX panels are accessible even if the licenses to run some of the options have not been purchased. This can help in determining what additional licenses are needed to create the desired simulation.

A very important thing about Vega is that it cannot be purchased for use, rather it is licensed to be used during a given time interval. Every executable function class checks for its specific license or it will not operate. The proper license information must be made available to the executable code during run-time and other applications must not have checked out more licenses than the license server has registered. Licenses are purchased both by specific type and by the number of users that will be running Vega programs simultaneously.

The modules that Vega and LynX are subdivided in to are as follows: Vega – the main elements that come with the basic license, AudioWorks2, Large Area Data Base Management, Special Effects, Vega Marine – maritime specific options, DIS – for networked simulations, Symbology, Vega Audio, and Vega Class Recorder. The rest of the LynX section will be spent on the important relationships that are easily accessed through this GUI.

2. LynX menu items and critical simulation relationships

This will not be an attempt to fully explain how to use LynX, but will highlight those areas that are necessary to fully understand the level of complexity that is removed from the user by this powerful software package.

a. Windows Panel

For Vega the Window is the defined rendering pipeline. It is one of the required elements since it is near the root of the Vega scene graph. The window defines the total maximum display area for the given rendering pipeline. For example, the MAAVE has three screens each with a display of 1024 x 768 pixels. The window that is used when running on a single graphics card machine is 3072 x 768 pixels while three windows are run on the system with three graphics cards, each with the resolution of 1024 x 768.

This panel also allows the quick setting of stereo images, placing a border around the window, enabling anti-aliasing, and defining how the mouse interacts with the window. For advanced graphics applications, the depths of the buffers are also set through this menu.

b. Channels Panel

A channel is the actual view into a virtual environment by defining the plane onto which the computer must place the current image. A channel occupies some portion of a window and multiple channels can exist for a particular window. Since two unique hardware options were used to produce computer graphics in the MAAVE, the flexibility of Vega was demonstrated by

the easy changes to the channels and windows panels to produce the most optimized rendering pipeline for each system. The conventional sgi computer with a multiple screen display option is most efficient with one large window and three channels, but the Intergraph computer is more efficient with three channels that each have only one window. The latter is true only since the specially designed Intergraph computer has three separate graphics pipelines via three dedicated graphics rendering cards. Positioning the channels is quickly done by entering the ratio positions of each corner with respect to its window.

Setting the viewing volume for each channel is probably the most critical element for multiple screen displays. Most single screen displays use a symmetric viewing frustum that best emulates how human vision operates. By filling in either the desired horizontal or vertical field of view and setting the near and far clipping planes, the generated symmetric frustum is perfect for the center channel of the three-screen display. This keeps both the near and far clipping planes perpendicular to the line of sight of the viewer. Since the Herrmann Hall model contains several narrow passages, the near clipping plane had to be set to one-tenth of a meter so that walls on the sides of the users position did not get clipped at the far edges of the side screens. Both the left and right displays could use a symmetric frustum if they were at 90 degree angles with the center screen and the user stood at the center of the focus of all three screens throughout the simulation. Since the MAAVE has 135-degree angles between screens, an asymmetric viewing frustum is needed for both of the side screens.

The asymmetric viewing frustum provides for the near clipping plane not being perpendicular to the users line of sight. Six values are needed to define this frustum since the one side touching the center screen will be nearer to the user. This means the image plane is no longer equidistant from the viewer about the center of the viewing screen. This functionality was incorporated specifically for large multiple projection surfaces. Six degree of freedom skew control is available to fine tune the image projected by the asymmetric frustum. This allows

individual adjustment of the adjacent images in X-axis, Y-axis, Z-axis, Heading, Pitch and Roll using the right hand coordinate system. The powerful combination of these viewing transformations makes it possible for the images that each individual projector transmits to slightly overlap. Having a divider at the joint between the screens prevents image spillover and greatly reduces the time needed to get smooth images through the screen transitions.

c. Observers Panel

An observer is similar to a camera in most other visual graphics applications. It defines the geographic location that the window(s) and channel(s) can then display the environment for. An observer is a required element for the user to define in a simulation. The panel allows for the observer to be attached to objects moving in the scene, interact with collision isectors, be forced to look at a given point or object, and control sounds that are within the observers range. Observers can be attached directly to motion models or to specific players. The main focus of this panel is to control the flow of the simulation. If a change is made to any other panel, it can almost be guaranteed that some update is needed in the observer settings.

d. Motion Models Panel

Believable motion through a virtual environment is a crucial part of making the experience real for the users. There are currently nine motion models predefined for use in the Vega software. With the exception of the Spin motion model, which only allows the observer to circle and object, all of the motions are best suited to some type of vehicle in an outdoor environment. The scenegraph attachments for the motion models allow switching between any of

them by simply having a software key designated to set a new one to control the object or observer. Easy settings for all of the magic numbers associated with the motion model, like maximum speed, acceleration rate and initial/reset position, are done quickly on this panel. The type of input device plus the collision isectors are also defined here.

The most flexible and dynamic of the motion models is the Flight Simulator. It is very physics intensive and correct. Performance settings for flaps, brakes, landing gear and low speed flying characters are the major parameters. Lift coefficients and incidences are available for every set of control surfaces. Turning rates in each of the three rotational directions, as well as the dampening functions for each are also defined. This functionality is best suited for a militaristic type of real-time simulation.

e. Scenes and Objects Panels

The scenes is a simple panel, but it defines what the observer and motion models interact with. Any objects that will appear in a simulation must be added to the scene through this panel. Objects are set as either scenes or objects, which is important for the collision detection isectors. The objects panel is where all geometries are defined for the simulation. The filename name is set for models that are created using other software packages. The Open Flight file format is the only software loader that comes with Vega. Since Vega sits on top of Performer, if the current version of Performer has loaders for additional file formats, then more types of model creators can be used. If not, having to convert the file formats often results in the loss of fidelity and texturing on a model.

LynX allows for easy optimization of dataset models depending on their usage during the simulation. Flattening and Cleaning of the loaded model is normally selected to allow Vega to

optimize the way the model is displayed. If a model is large and will interact only in small portions at any one time with the user then partitioning the model can increase the overall speed in which the model is searched without the user having to decide how to best cut up a given model. Creating a display list for models will force all of the geometry into hardware, but it does not allow for changing of any of the characteristics of the model during run time. An initial and reset position and orientation for the object is also set from this menu.

One last critical item needs to be set on this menu, the isector bit mask. There are 28 bit mask values to uniquely represent objects in the simulation so that collision isectors can be assigned. This makes it incredibly easy to define complex interaction behaviors between multiple objects by selecting one or more of the following categories as the type of object that is being added to the scene graph; Terrain, Static Object, Dynamic Object, and Special Object. There are eight groupings for each category, or the object can belong to all eight at once. This quickly allows specialized behavior such as when dismounted infantry enter a forest their movement ability would not be hindered, but if an armored vehicle encounters the same terrain group its movements will be greatly affected.

f. Isectors Panel

Isector stands for Intersection Vector. It is a single directional line segment that tests against all loaded geometry models that have a bit mask value for which the isector was designated to detect. The primary use of isectors is for determining when two objects collide. Collisions between two moveable objects should normally exhibit some specialized behavior, while collision with the ground is a useful thing for vehicles to simulate moving over the terrain.

Currently seven isectors are operational and they include; Z, HAT, ZPR, TRIPOD, LOS, BUMP, and XYZPR. Z represents a single isector that is oriented in the vertical direction for detecting collisions primarily with terrain. HAT stands for Height Above Terrain and is normally used in flying simulations where a specialized Z-isector is moved with the object and adjusted to maintain above, below or at the designated relative altitude. ZPR is another specialized Z-isector that also returns the slope of the ground in both the Pitch and Roll directions. TRIPOD consists of three isectors and is primarily used to represent a driving vehicle. TRIPOD needs settings for the width between the two rear isectors and the length from the perpendicular bisection of the two rear isectors giving it the performance characteristics of a tricycle. LOS is a single isector that goes from a designated point along a Line Of Sight (LOS) and can be used to determine visibility or collision with other vertical objects. BUMP uses six isectors, one along each of the positive and negative X, Y and Z-axes. BUMP requires three parameters, height, width and length, and is used primarily as an educational example. XYZPR is used when the simulation is on a large enough scale that a rounded terrain is needed and performs like a ZPR.

g. Environments and Environment Effects Panels

These panels are mentioned since an outdoor scene can be quickly generated. It is easy for fog to be set, the color of the sky designated, light sources added, time of day, and several special effects. Fog can use the linear, exponential, exponential squared or a spline function to determine the effect. Overall visibility ranges for the environment can also be specified. Lights used include infinite, local, spot sun and moon. The sun and moon light are moderated based upon the time of day. Several types of clouds are available and quickly add believability to the simulation.

3. Limitations of LynX Implementations Alone

Although LynX delivers remarkable results in an incredibly short time, the overall desired results might not be achievable using this interface alone. The main thrust of the Vega LynX software package is outdoor, large-scale simulations. Although dismounted humans are able to interact with other objects in a simulation, there is no specific walking motion model. Also, the default behavior for isectors is to return that part of an object that is the highest in absolute elevation, which is desirable for outdoors, but detrimental for maneuvering inside complex, multi-level structures. Another drawback is the cost for each additional software license to expand the capabilities of the simulation beyond just the basic application. For example, to use the DIS networking module requires additional licenses for each runtime application.

B. CREATING SPECIALIZED VEGA EXECUTABLE CODE

The afore mentioned limitations of LynX prevent the creation of a first person, walk through simulation. Fortunately Vega comes with a detailed programmer's guide. The prerequisites for using this guide are knowledge of scene graphs, especially Performer, and detailed knowledge of C or C++ programming languages. The software documentation provided with Vega has several code examples that are incredibly useful in understanding how all of the Vega data objects interact. The rest of this section describes how the specialized executable code

was created from the provided examples for a first person, walking motion model capable of realistically navigating through a complex multiple layered building.

1. Creating the Walking Motion Model

After experimenting with all of the motion models available through LynX, nothing could realistically replicate the motion of a human walking. The code sample provided in chapter 9 of the Vega Programmer's Guide [Paradigm: Vega98, p. 270] was the starting point for creating the needed executable code to go with the values set through the LynX GUI. This creates a simple motion model that uses a three-button mouse to give motion along all three of the dimensional axes.

A new user-defined motion model is created, but only if no motion model is selected in the ADF designated as the input for the simulation. This does not prevent the user from switching to the predesignated motion models during runtime, as long as the user delineates how the input devices are used to manipulate the various motion models. As stated earlier an observer is needed to have a view on the simulated environment and motion models can only be attached to observers or players. Since this is a first person simulation the observer is attached directly to the motion model. An input device is then attached to the motion model that can be switched during run time with keyboard toggles or location switches. Lastly the motion model code must be called at least once per run time cycle to update the location and orientation of the observer.

By registering the motion model code as a motion model callback, the Vega software inserts the code into an optimized position within the scene graph. After all of the geometry has been loaded and the rest of the data structures have been initialized, the program enters an infinite loop that contains two Vega calls that cause the graphics to update and be sent to the screen. It is

in between each of these function calls that all of the callbacks are executed. Callbacks are similar to function calls and care must be taken to ensure they do not impede the processing speed of the overall program by inserting them at the wrong times.

Since only a mouse or flight control joystick is used to control the motion model in this simulation, and both are fast reading, serial input devices, the devices can be used synchronously during simulation. If a device such as the Ascenscion Flock of Birds were used, a callback could not directly read the input device data without slowing the overall simulation. The reading of the input device data is accomplished each cycle in the motion model callback for this simulation and then the observers position is determined so the graphical rendering process can begin.

The callback itself is broken into five sections: data initialization and a switch statement that goes to one of four events based upon when the callback is invoked. During initialization the motion model, the input device, and the input devices' initial positioning (such as "joystick centered" or "speed slider at zero velocity") are obtained for local usage. Each of the inputs from the input device needs to have a local value to store the input data for motion model algorithm use and a Vega position vector to read the input device in Vega database units. An instance of the motion model data structure (a C struct) is created from the user defined struct model which holds the following information: X, Y, and Z-axis coordinates in database units, heading in degrees from North using the right hand rule, pitch and roll in degrees, a current velocity, a Vega position vector, a time stamp holder and five results from isector tests.

```
/* struct definition */
typedef struct  {
        float x, y, z;              /* x, y, z position in database units */
        float h, p, r;              /* heading in degrees (from north) */
        float velocity;
        vgPosition *pos;
        double now;
        int     onMyNose, onMyLeft, onMyRight, inBack, atRail;

} motion_model_data;
```

Figure 4.2. C Structure definition for motion model.

The first time a motion model callback is executed during a simulation, whether at the start of the simulation or switched to at a later time during a simulation, the Vega motion model initialization (VGMOT_INIT_EVENT) event is called. It is at this time that the input device is first read and the starting position of the motion model is set along with the velocity, simulation time and isector test results. The next event is a motion model reset (VGMOT_RESET_EVENT) event. This code is only executed when a reset event is triggered by either the user or the run time code and allows for changes to all of the struct data. As with all good code an exit (VGMOT_EXIT_EVENT) event is also included to clean up all newly assigned memory when this motion model is no longer needed.

By far the most important and frequently used event is the update (VGMOT_UPDATE_EVENT) event. Update is called any time the init, reset or exit events are not called. The first order of business is to obtain the X and Y-axis locations of the mouse cursor or the joystick handle. Next, depending on which device is currently in use, the buttons, toggles or sliders are read. If the stop velocity button or toggle is not pushed and none of the isector values indicating a collision status with an object are set, then any updates to velocity are calculated. For mouse input, one-tenth of a meter per second is added for a left click and is subtracted for a right click. Clicking the middle mouse button sets velocity to zero. When using

a joystick the slider is used to directly set the velocity unless the stop button is depressed. As with mouse cursor positioning, a dead spot of no velocity is set at the middle of the sliders range of motion with pushing the slider forward increasing velocity and pulling the slider rearward causing a negative velocity. This motion model is based upon the laws of inertia in that once a velocity is set it will remain until acted upon by some external force such as user input or collision.

Heading is then calculated based upon current heading minus the current value of the input devices X-axis multiplied with a scaling constant and the change in time since the last update. The scaling constant is used to set the maximum rate at which the motion model can pivot. When using the mouse, the X-axis values go from negative one at the far left of the display window to positive one at the far right. A dead spot of five percent has been set in the middle of the screen so that a user does not have to find the exact center pixel of the window to have heading remain constant. This dead spot is not necessary for input from a joystick provided that the center position is properly calibrated. The X-axis is read directly from the deflection from the center position to the left or right and uses the same values as the mouse.

Elevation pitch angle is determined in much the same manner as heading, except the Y-axis values for the input devices are used. For the mouse, positioning the cursor above the center of the screen above the five- percent center dead zone will cause the pitch angle to increase and below the center dead zone will cause a pitch angle decrease. The joystick uses a flight system style of control such that pulling back on the stick causes the pitch angle to increase and pushing forward causes it to decrease. With all of these values a new X and Y-axis position for the motion model can be determined. The X-axis value is calculated by subtracting a Vega sine value based on the current heading multiplied with the new velocity and change in time since the last update event from the current heading. The Y-axis value is calculated by adding a Vega

cosine value based on the current pitch angle multiplied with the new velocity and change in time since the last update event from the current pitch angle.

The challenge now is to determine the correct Z-axis database value given the new X and Y-axis values. This is accomplished through attaching a Z-isector to the motion model and having it check against all classes of terrain objects for a collision that would be possible if a person were walking. An example of this is not allowing the motion model to step up greater than one and a half meters in a single step. The isector extends three meters down and two meters up from the position of the users' feet, if the user had an avatar. This line segment is adjusted every cycle to remain centered at this position on the motion model based upon the last Z-axis value. With this new value all terrain objects are tested for intersection against the segment. It is possible for more than one collision to occur and care was taken to return the best value given the current Z-axis value. One sensitive area for proper functioning of this motion model's Z-isector in Herrmann Hall was when ascending stairs were encountered because the polygon representing the current floor extended under the stairs. Since this was the only location that this anomaly occurred, whenever a same level and a one stair height step up value were encountered simultaneously, the one step up value was returned by the Z-isector.

The final piece of making a realistic, first person, walking motion model was inclusion of collisions with walls, railings and other physical obstructions. This was accomplished by using five LOS isectors. Three of the isectors are for forward motion with one extending on the current heading and the other two spaced 20 degrees horizontally to either side. Twenty degree offsets were determined by trial and error in the Herrmann Hall model to allow access into all of the passage ways, including the narrowest, while not allowing the extreme right and left sides of the display to see through walls that were clipped by the near viewing plane. The fourth isector extends directly opposite the current heading to provide some collision detection while moving backwards. The two side isectors were not used in the backward direction since a person can not

see through the back of their head and clipping of polygons should not be detected. All of these isectors are positioned 1.05 meters above the feet of the motion model. Again, trial and error set this value. All four of these isectors extend 0.8 meters from the center of the motion model. Several of the stairwells are very steep, which causes a collision to be detected against higher stairs while ascending, or the underside of a higher flight of stairs while descending. There are many low railings in the Herrmann Hall model that prevent people from falling to lower levels such as in stairwells or around the mezzanine overlooking the main reception area. To prevent the motion model from walking through these, a fifth isector was added that extends in the current heading 0.3 meters and is located 0.6 meters above the feet of the motion model. Figure 4.3 shows the orientation of the five isectors.

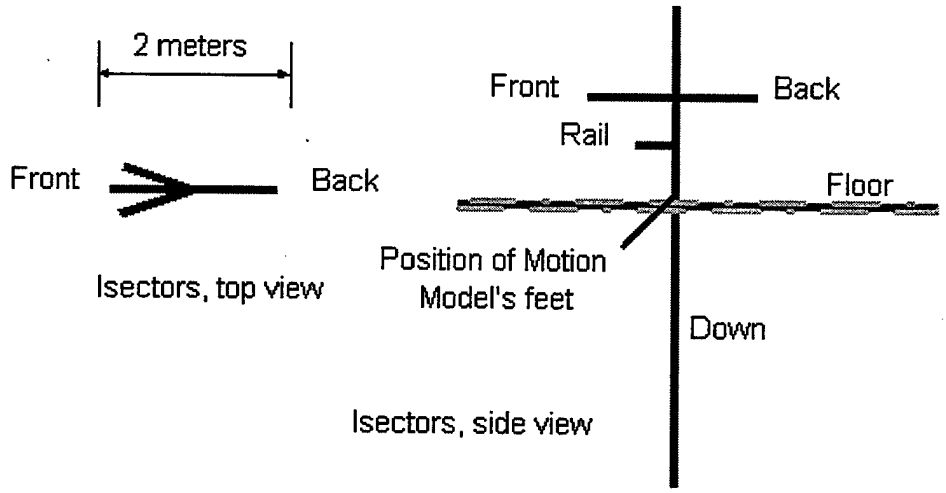


Figure 4.3. Diagram of isector orientation.

All five of the LOS isectors are used to set flags that inhibit motion in the direction of the detected collision. For example, if the user is walking backwards and is about to hit a wall, once the isector registers a collision with the terrain, velocity is set to zero. Another example would be if the user is walking around a pillar in the forward direction and turns quickly so that the rear

isector detects a collision with the pillar. Since velocity has a positive value the flag is ignored and soon the observer will move away from the collision condition.

2. Networking the Scene Graph

As stated in chapter I of this document, DIS was decided upon to convey user simulation state information to remote users. The Vega/LynX software has an excellent, fully defined DIS section, but it requires an additional per-seat license fee. Since Vega is written in the C++ programming language and NPS has been using the DIS standard for several years, a decision was made to incorporate the DIS libraries created by John Locke, an NPS employee, into the Vega executable code.

DIS PDUs are read asynchronously and stored in a circular buffer. A function call is made every cycle through the run time infinite loop that reads new packets in the buffer. Currently only the Entity State PDUs (ESPDU) are kept by the networking code. Networking is achieved through a multicast address protocol. Each ESPDU contains an entity identification (ID) number that should be unique for every participant in the network.

This entity ID is then checked against an array of currently held entity IDs for a match. If a match is found then the position data is used to update the location of the remote participants avatar in the local simulation. If no match is found, a new entry is made in the entity ID array and the ESPDU is read to determine not only position and orientation, but also what type of geometric model to use as the avatar for this new user. Vega allows all geometry models to be loaded as a vgDataSet prior to usage. If the name of the avatar file does not exactly match the vgName given to the vgDataSet then the avatar will not appear until its geometric model can be obtained and converted into a vgDataSet object. A default avatar will be displayed until the

correct model can be shown. LynX makes the avatar loading process easy by simply creating an object in the objects panel and entering the file location where the code for the avatars geometry model exists. Since these avatars will move throughout the simulation it is important to designate them as dynamic objects and to select the clone or copy option for the data, so multiple instantiations can be made from one vgDataSet object.

A timing flag is used so that the next time the "update networked entities" function is called, the motion model position is encoded into an ESPDU multicast to all users on that address. The timing flag can be set to whatever frequency is necessary for a smooth simulation. This approach was taken to prevent too many or too few ESPDUs being sent by any one individual in the simulation. Along with the current location and orientation of each user, the current velocity is sent and used for dead reckoning each participant's location in between ESPDU receipts. Smoothing algorithms can be employed to keep avatar motions continuous if the extrapolated position is greatly different than the newly received ESPDU.

3. Limitations Of The Vega Walking Motion Model Code

The two most prevalent limitations have to deal with computer hardware. Vega was originally created to run using the Irix operating system resident on sgi computers and Performer. Performer is the most optimized C++ scene graph, but only runs on sgi computers. VegaNT has been released to run in the WindowsNT operating system and had to bring a modified version of Performer with it to use a the scene graph.

Performer and a few Vega graphics calls are optimized and created specifically for sgi graphics hardware. Depending on the generation of sgi computer used, specially detailed Vega

graphics options can be selected. This creates a bias in favor of Vega when running software applications on an sgi computer.

The networking code is also substantially different between the two operating systems. The DIS library can be used in both cases, but the process for opening, reading, and closing the network connections is very much different. A disadvantage for the VegaNT code is having to convert the order of the bytes received from network order to the order used by WindowsNT 4.0 machines (Big Endian to Little Endian.)

C. GENERATING PASSIVE STEREO IMAGES

Vega supports stereo imaging, as did all of the computers used for developing the walking motion model. Great concern was raised over how to send the proper stereo signals to the LCD projectors for interlaced horizontal line display. The solution was purely hardware driven since the Intergraph computer supports generating interlaced stereo images. To view stereo images in the MAAVE, all that is needed is the passive stereo glasses and setting both the computer output and the projector output to interlaced stereo. To view stereo at the three monitor control station, active glasses are needed even though the monitors receive the interlaced signals repeated through the projectors.

One of the arguments against using the Infinite Reality computer is the stereo output format. This computer uses an over-under stereo generation technique that would require a special signal converter to present the projectors with a signal in interlaced format. Another possible solution would be to use a software bit-mask that first blanks the even horizontal lines and then recalculates the image for the other eye and applies an odd bit-mask. This software solution requires twice the number of graphics cycles to obtain stereo vision and would only be

useful on a computer that has a very high refresh rate. Future computers used to power the graphics in the MAAVE will use interlaced stereo display. For further details on stereo imaging refer to [McAllister93].

V. JAVA3D IMPLEMENTATION

The Java3D API was written to extend the Write Once, Run Anywhere™ capability of Java to the display of 3-dimensional graphics in a Web browser [Java3D00]. It has further grown to provide a viable programming alternative for the construction of visual simulations with built-in support for HMDs, multiple views, users' physical environments and collision detection. Java3D allows developers to build applications without needing to know the end user's final display environment [Sowrizal/Deering99]. This chapter explains the Java3D scene graph and describes how to build a networked visual simulation in Java.

A. JAVA3D OVERVIEW

The Java3D scene graph is organized into a hierarchy that is simple yet allows complex manipulation of the scene. As fig. 2.6 shows, the `Canvas3D` allows Java3D to interface with Java through the `Frame`. The `SimpleUniverse` attached to the `Canvas3D` is the real delineating point for Java3D. The `SimpleUniverse` holds all of the information in the scene graph and is the starting point for the rendering cycle. The `SimpleUniverse` is made up of three parts: the `Locale`, the `Viewer`, and the `ViewingPlatform`. Each of these parts will be discussed in turn.

1. Locale

The `Locale` is an object that holds all of the geometry information of the scene graph. It also contains the behaviors defined for the scene graph. Behaviors, discussed fully in section V.B, allow actions to be performed on scene graph objects based on Java or Java3D events. The

`Locale` is the reference for all geometry objects in the scene graph. The coordinate system in Java3D is referenced relative to the origin of the `Locale`. The `Locale` is the starting point for all of the paths for scene graph traversal. A `SceneGraphPath` object can be instantiated that will return the path from the `Locale` down the scene graph to a desired node. This is extremely useful for picking operations and collision detection. The `Locale` does not need to be explicitly defined if a `SimpleUniverse` is instantiated, as Java3D will then implement a default `Locale` for the scene graph. Multiple `Locales` can be defined but should be rendered in separate `Canvas3Ds`.

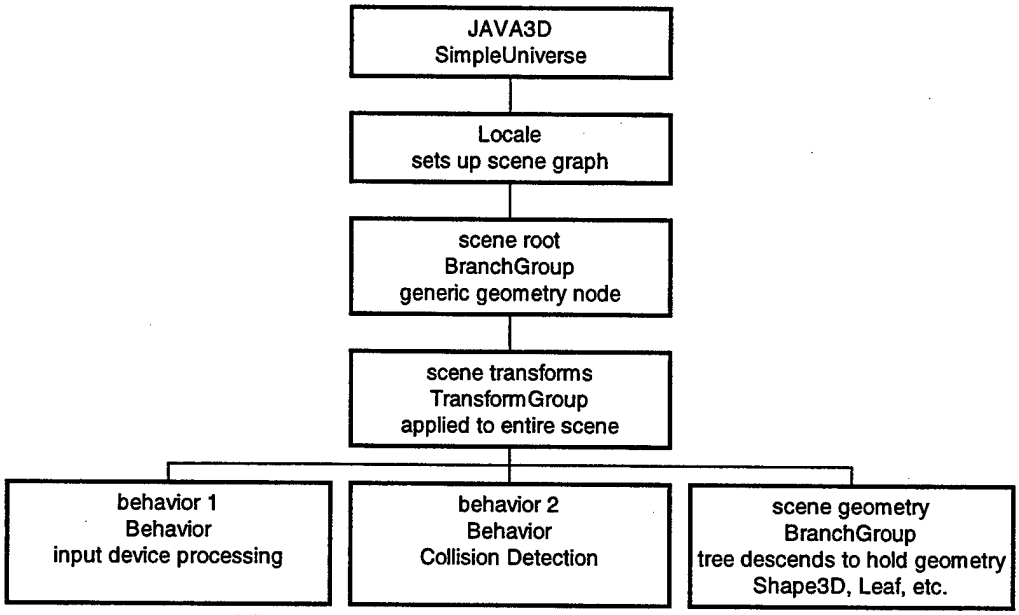


Figure 5.1. `Locale` implementation in Java3D.

2. Viewer

The Viewer in Java3D holds all of the presence information about the user. It contains information about the user's physical appearance to himself and his virtual appearance to others. On the physical side, the Viewer contains the Canvas3D object that the user is associated with. It also contains a PhysicalEnvironment object that describes the hardware performance of the

machine the user is using. A PhysicalBody object completes the physical appearance. The PhysicalBody object holds the user's preferences and represents how the user views himself in the application. A ViewerAvatar object provides the virtual appearance. The ViewerAvatar is the geometry the user wants everyone else to view him as. The Viewer also contains a reference to the current View object. The View is a Java3D object that holds all of the information about the current view in the current canvas, yet it exists outside of the scene graph. Finally, the Viewer also contains all of the sound information in the scene graph.

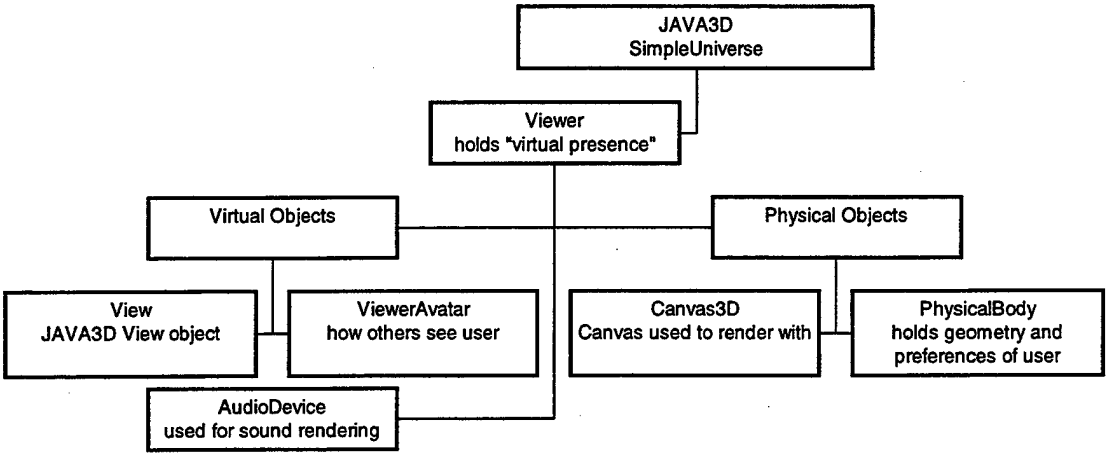


Figure 5.2. `Viewer` implementation in Java3D.

3. ViewingPlatform

The ViewingPlatform sets up the view in the scene graph. The ViewingPlatform contains a multifunction TransformGroup object that allows manipulation of the current view. The ViewingPlatform also has a reference to the current View, along with containers for any geometry associated with the ViewingPlatform. One of the most important parts of the ViewingPlatform is the ViewPlatform.

The ViewPlatform controls the scale, orientation, and position of the viewpoint. It is the ViewPlatform that provides the hooks into the View object for the ViewingPlatform. The

ViewPlatform allows the user to set the view attachment policy and the activation radius. The view attachment policy determines where the viewpoint is placed in relation to the geometry in the scene. The default policy has the virtual world viewpoint correspond to the physical world viewpoint, while another changes the virtual world viewpoint to be a constant height above the ground. This policy allows the programmer to vary this height based on the preferences in the PhysicalBody object.

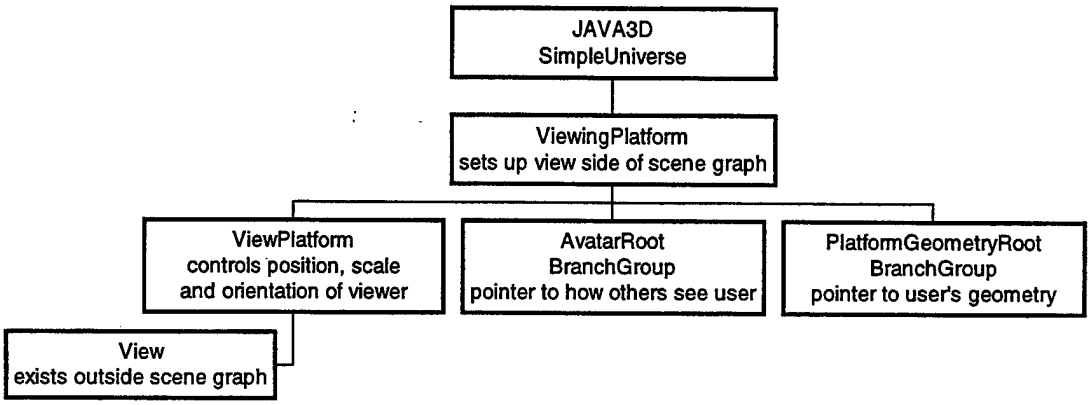


Figure 5.3. `ViewingPlatform` hierarchy.

B. BEHAVIORS

`Behaviors` are used in Java3D to describe how a node in the scene graph responds to a given manipulation. These manipulations range from simple set rotations to complex input devices affecting the location and orientation of the viewpoint. Two different applications of `Behaviors` were used in this simulation and will be discussed here: scene navigation and collision detection.

1. Scene Navigation

The scene navigation goal for this simulation was to have a mouse-driven walking style navigator similar to that of the Vega simulation. Java3D includes three default mouse behavior classes that allow the user to rotate, zoom, and pan around an object. This is similar to a VRML browser, where the rotation is about a fixed axis and the user cannot actually walk through the scene. David Nadeau from the San Diego Supercomputer Center developed a walking navigator class for the SIGGRAPH98 Conference [Sowizral, et. al.98] to provide a real scene navigator. This `WalkViewerBehavior` class sets the axis of rotation to the viewpoint and moves the axis as the viewpoint moves. This provides a realistic walking effect in the virtual simulation.

The `WalkViewerBehavior` object was instantiated with the current `Frame` object and the current `ViewPlatform` transform passed in as parameters. The `WalkViewerBehavior` class listens to the `Frame` object for mouse events. When a mouse button is pressed and the mouse is moved, the `WalkViewerBehavior` object changes the passed-in transform. Since this transform is a reference to the `ViewPlatform`'s transform, the viewpoint is changed to reflect the new orientation or position. In the `WalkViewerBehavior` class, the left mouse button provides forward and backward motion and rotation. The middle mouse button provides rotation without movement, while the right mouse button provides translation side-to-side and up and down without rotation. The end result is that the viewpoint moves through the scene, just as if the user were walking in the real world.

```
BoundingSphere bounds = new
  BoundingSphere(new Point3d(0.0,0.0,0.0), 100000.0);
TransformGroup viewXForm =
simpleUniverse.getViewingPlatform().getViewPlatformTransform();
WalkViewerBehavior mover = new
  WalkViewerBehavior(viewXForm, myFrame);
mainScene.addChild(mover);
mover.setSchedulingBounds(bounds);
mover.setEnable( true );
```

Figure 5.4. Addition of a navigation behavior to the scene graph.

2. Collision Detection

Collision detection allows the user to move through the scene without walking through walls or other obstacles. This provides a touch of realism to the simulation and makes for a more enjoyable experience. Java3D implements some events that will search the scene graph to determine collisions. These events trigger the activation of the collision behaviors. The collision behaviors are `WakeUpOnCollisionEntry`, `WakeUpOnCollisonExit`, and `WakeUpOnCollisionMovement`. The collision behaviors only indicate that a collision event condition has occurred. It is up to the programmer to define what happens as a result of the collision event. There are two problems with this implementation. The first is that the behaviors indicate that a collision occurred last frame, but it would be better to know that a collision will occur next frame. This would allow the software to anticipate and stop movement in the direction of collision, providing a more realistic experience. The second problem with the behaviors is that once a collision condition exists, no further collisions are detected until that condition is removed. This problem is illustrated in a simple example: a building walkthrough simulation that needs an entity to be in a collision condition with the floor to follow it but needs to be out of a collision condition to detect collisions with the walls. If the entity is in collision with the floor, a collision with a wall will go unnoticed, resulting in the user walking through the wall and degrading the

virtual experience. If the entity is out of collision with the floor, it cannot walk through walls but it also cannot negotiate staircases.

Collision detection was implemented in this simulation using a `CollsionDetector` class based on the demo `TickTockCollison` class included with Java3D. It uses the `WakeUpOnCollisionEntry` and `WakeUpOnCollisonExit` events to signal collision entry and exit. At this point in development nothing further occurs, however the triggering object can be found by getting the event's triggering path. This returns a `SceneGraphPath` object that will indicate what object caused the collision. Based on the object's orientation, an input to the `WalkViewerBehavior` object could stop its motion in the direction of the triggering object. Further improvements to the collision detection will be discussed in Chapter VII.

C. NETWORKING

A virtual simulation is fine until the user realizes that it would be nice to have company (either friendly or hostile) in the virtual world. Networking allows multiple users to interact with each other and the virtual world. Many different ways of networking were considered but the DIS standard won out because of its widespread use and programming ease.

1. DIS Implementation

The DIS standard was implemented in Java by Don McGregor at NPS as part of the Web3D DIS-Java-VRML working group. Although he did not implement the complete standard, he implemented the most widely used PDUs. He also implemented the enumerations needed to best utilize these PDUs. This simulation implemented only the ESPDU as it is the most common

and necessary one. The ESPDU contains the entity's EntityID, location, orientation, and velocity information.

The first step in implementing DIS is the network interface. This simulation uses the Multicast protocol, as it needs no central server and is very flexible. The `java.net` library is the network interface to the operating system and contains all of the networking algorithms. The program first opens a `MulticastSocket` with a port number and then calls the `joinGroup()` method, which joins the socket to the provided multicast group address.

```
try {
   multicastAddress = InetAddress.getByName(MULTICASTIP);
   multicastSock = new MulticastSocket(APP_PORT);
   multicastSock.setTimeToLive(1);
   multicastSock.joinGroup(multicastAddress);
}
catch(IOException ioe) {
   System.out.println("Establish multicast UDP port error: "
      + ioe.getMessage());
   return;
}
```

Figure 5.5. Instantiation of a Multicast socket using Java's built-in networking.

The DIS implementation is straightforward. A `BehaviorStreamBuffer` object starts the buffer process to send and receive the PDUs. To send a PDU, the simulation gets the value of the current `ViewPlatform` transform that is set by the `WalkViewerBehavior` object and extracts its x, y, and z location. This is then entered into an `EntityStatePDU` object, stamped with the program's `EntityID` and sent out using the `sendPDU()` method of the `BehaviorStreamBuffer` object.

The `BehaviorStreamBuffer` object collects all of the received PDUs into the buffer that can then be read out to a `Vector` object. Reading the buffer clears it for new PDUs. The program uses a Java `Hashtable` object to store known entities and their geometry. The `EntityID` is the key for the hashtable and the entity's geometry is the stored data. The

simulation checks each `EntityID` against the hashtable to determine if the entity is known. If the entity is known, the program updates the transform of its geometry with the information stored in the PDU. If the entity is unknown, the program creates new geometry for the entity and adds it to the hashtable and the scene graph.

For this simulation only the entity location field of the ESPDU was used. Since the entity velocities are not implemented, the program has no dead reckoning. Although this does not delay the rendering process, when updates are received the entity jumps to the new position. This results in less believability for the simulation.

2. Multi-threading

A single-threaded simulation would check the arrival buffer every frame, but this can slow the graphics display by making the rendering pipeline wait for the networking update to complete. This slowdown is increased if there are numerous entities in the environment. This slowdown can be prevented by multi-threading the application. By having all of the networking occur in separate threads, the rendering thread, which now only has to deal with geometry, can execute while the networking threads are blocked waiting for input. Since the geometry pipeline is usually the limiting factor for rendering speed, this allows for better frame rate and overall performance. Another reason for multi-threading is that on a Windows platform, the JVM is able to take advantage of a multi-processor machine. In this case, if a network thread blocks another thread can execute, allowing the overall process to keep running. This simulation uses three user separate threads for networking. The first initializes the `BehaviorStreamBuffer` object and regulates it. The second is a thread that collects incoming PDUs and updates the hashtable in the scene graph accordingly. The third thread sends an updated ESPDU onto the network.

D. LOADING THE MODEL

Java3D has a loader interface that provides for future expansion and implementation. Any class that implements this loader interface can be used to load a file of a given format and convert it to a Java3D Scene object. This Scene object can then be attached to an existing scene graph and integrated into a program. The only loader built by Sun and included with Java3D is a Lightwave loader. Other file format loaders are under construction, including VRML and XML. Shawn Kendall, from the Full Sail Real World Education in Orlando, Florida, built a FLT file format loader [Full Sail00]. This loader takes standard Multigen FLT files and converts the geometry into `Shape3D`s, which are then built into a `Scene` object. This `Scene` object then is attached to the scene graph as a `BranchGroup`. This branch of the scene graph is then compiled. Compiling allows Java3D to rearrange the scene graph so that it can optimize the rendering cycle. It optimizes by minimizing the number of graphics state changes that occur. State changes occur when colors, textures, and other appearance properties change. Each state change causes a complete halt to the rendering process. By minimizing the number of state changes, Java3D attempts to speed up the rendering process, subsequently increasing the frame rate.

```
try
{
    Scene []sceneBase = new Scene[4];
    sceneBase[0] = fltLoader.load( "herman/herman_ground.flt" );
    sceneBase[1] = fltLoader.load( "herman/herman_main.flt" );
    sceneBase[2] = fltLoader.load( "herman/herman_rear.flt" );
    sceneBase[3] = fltLoader.load( "herman/herman_wings.flt" );
    BranchGroup []scene = new BranchGroup[4];
    for(int i=0; i<scene.length; i++){
       scene[i] = sceneBase[i].getSceneGroup();
       BranchGroup me = new BranchGroup();
       me.addChild(scene[i]);
```

Figure 5.6. Loading the model into the scene graph.

VI. RESULTS AND CONCLUSIONS

One of the two main objectives of this thesis is the comparison of two different software systems for moving through a three dimensional virtual environment. The second is creation of a large-sized VEE to use for three dimensional visualization and experimentation. Both of these objectives were met and the results are presented here.

A. DIFFERENCES IN SOFTWARE IMPLEMENTATIONS

Two major differences in software implementation occurred due to usability considerations and development time constraints. Therefore, comparisons cannot be completely accurate, but the biggest factor, the geometry, was the same for both applications. The differences are implementation of the collision detection algorithms and networking.

Collision detection in the Java3D application was implemented using the built-in collision behavior nodes. This computational load resulted in such poor performance that collision detection was turned off for the experiments. Future modifications, discussed in Chapter Seven, should implement collision detection in a manner similar to that used in the Vega application. Without collision detection, the user must manually avoid walking through walls or floors.

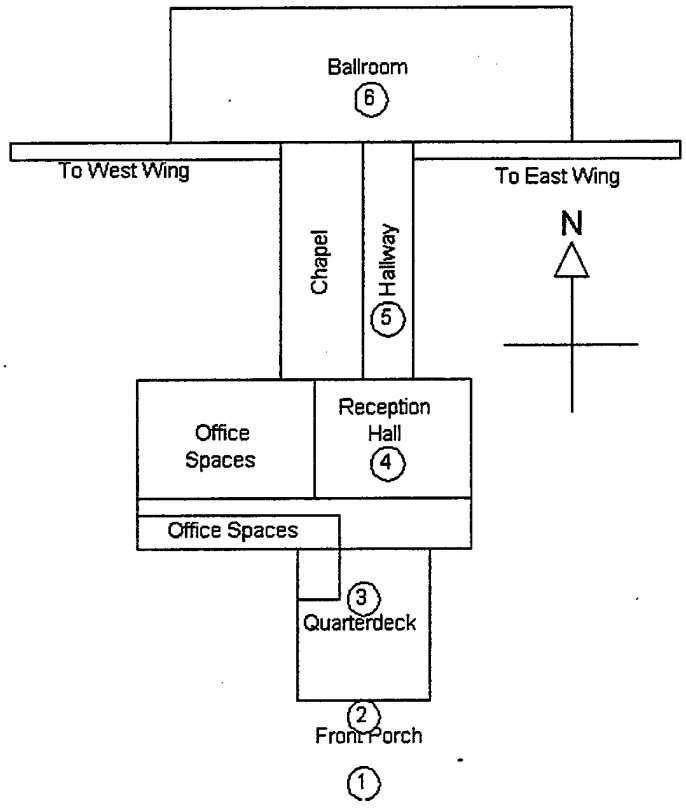
The Vega implementation on Irix had asynchronous networking during the frame rate testing. The VegaNT implementations did not have asynchronous networking operating properly, so frame rate measurements will be affected. Since the implementation is asynchronous, the impact on frame rate should be minor. Although networking was running in the Java3D

application, no packets were actually sent to the application; therefore no additional geometry was added to the scene to represent entity positions.

B. MEASUREMENT OF GRAPHICS RENDERING SPEED

The speed that a computer is able to render the graphics picture on the screen greatly determines the success of any VEE. As such, the primary measure for effectiveness of the two software packages tested is frame rate. Frame rate measures how many times per second the computer is able to redraw the image on the screen. When possible, multiple screen display modes were tested running each application. Single screen and reduced size single screen were also tested to gain better understanding of the graphics load placed on each computer. All frame rate measurements were taken from the locations denoted on Fig. 6.1. Readings were taken in multiple facing directions to determine the effect, if any, of culling portions of the Herrmann Hall model.

Location 1 is beside the flagpole; when looking South (location 1S) it does not have any geometry in view. Conversely, the new frustum at location 1 looking North (location 1N) encompasses the entire geometry, and culling could only affect the visually occluded portions of the model. The other locations were chosen because of significant changes in frame rate on one or more of the tested hardware systems. A brief description of, and the results from, each system are delineated in the rest of this section. Some surprising results were obtained, and the following sections include hypothesized explanations. Unfortunately, we were unable to provide explanations in all cases.



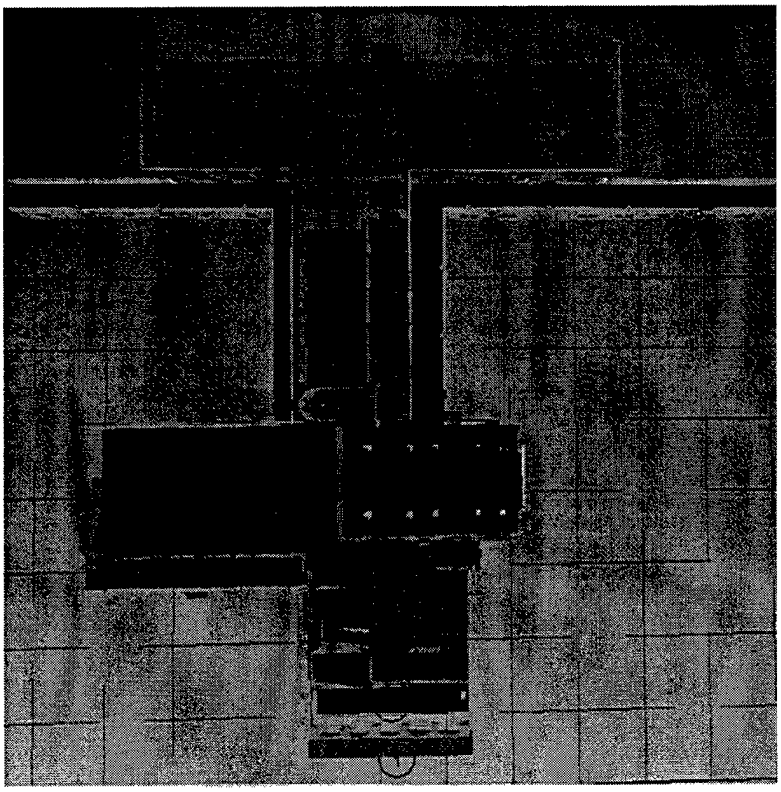


Figure 6.1. Main floor diagram showing frame rate testing points.

1. Frame Rates Obtained Using sgi Infinite Reality Computer

Hardware configuration for the Onyx Infinite Reality Computer:

- Four 194 MHz MIPS R10000(IP25) processors running Irix version 6.5
- One RM6 graphics board with 64 MB of texture/video memory
- 128 MB system RAM

The first computer tested for frame rate was the five-year-old Infinite Reality computer in the computer graphics laboratory. The results are listed in Fig. 6.2. Several interesting trends can be found in the numbers. The most significant are the almost unchanging frame rates for Java3D

Location Direction	Vega 1024x768	Java3D 1024x768	Vega 3x1024x768	Java3D 3x1024x768
1N	15	0.8	13	0.8
1S	29	15	24	16
2N	16	0.9	12	0.9
2S	24	1.4	22	1.5
3N	18	1	12	1
3E	23	1.4	18	1.3
3S	23	1.4	18	1.4
3W	22	1.3	15	1.2
4N	20	1.2	12	1
4E	25	1.4	20	1.3
4S	22	1.3	18	1.3
4W	21	1.3	15	1.2
5N	21	1.3	12	1.3
5E	22	1.3	15	1.3
5S	20	1.1	13	1.1
5W	20	1.3	10	1.2
6N	22	1.4	20	1.4
6E	22	1.3	20	1.3
6S	20	1	15	1
6W	22	1.3	15	1.3

Figure 6.2. Frame rate results on Infinite Reality Computer (frames per second).

regardless of number of pixels drawn to the screen, and the moderate steady speed for Vega.

Speculation as to why the Java3D frame rate was so steady, and so slow, is difficult to make without more detailed testing. When the number of pixels to fill on the screen increased, the frame rate did not significantly decrease. One possible candidate is the implementation of the JVM by sgi which will be further explored in the next section. The Java3D code was not compiled on this computer and it executed without fail- demonstrating the portability and speed of development of Java3D.

Frame rate using Vega was lower than initial expectations, yet followed a predictable pattern. When the number of pixels to fill on the screen increased, the frame rate decreased. From these numbers it is hard to determine whether the graphics hardware is the limiting factor, or if the additional geometry culling is the limiting factor. Given the age of the hardware, and the frame rates at location 1 looking south, it appears the graphics hardware is running at maximum capacity and below the desired goal of 30 Hz. The most notable numbers are the three screen display numbers that never drop below 10 Hz, even when looking at the most complicated parts of the model for culling.

2. Frame Rates Obtained Using sgi 320 Visual Personal Computer

Hardware configuration for the sgi 320 Personal Computer:

- Two 450 MHz Pentium III processors running WindowsNT 4.0
- Cobalt graphics board that shares system RAM
- 256 MB system RAM with 48 MB assigned as graphics memory

The second computer tested was an sgi 320 Visual PC. Expectations were for Vega's frame times to be faster than they were on the Infinite Reality since the computer still uses some of the technology that Vega was originally optimized to run on and is much newer. Expectations

for Java3D were to have an increase in performance over the Infinite Reality since WindowsNT was the operating system. The results were very surprising and are included in Fig. 6.3.

Location Direction	Vega NT 640x480	Java3D 640x480	Vega NT 1280x1024	Java3D 1280x1024
1N	17	0.7	8	0.7
1S	87	9.5	43	8.5
2N	17	0.9	10	0.8
2S	87	1.5	43	1.5
3N	21	1.1	12	1.1
3E	43	1.5	21	1.5
3S	43	1.5	30	1.4
3W	29	1.5	17	1.5
4N	29	1.4	14	1.3
4E	43	1.4	29	1.5
4S	43	1.4	21	1.5
4W	29	1.5	17	1.5
5N	29	1.6	17	1.5
5E	43	1.4	21	1.4
5S	22	1.6	14	1.4
5W	29	1.5	17	1.4
6N	43	1.4	29	1.4
6E	43	1.4	21	1.4
6S	21	1.1	12	1
6W	43	1.4	17	1.5

Figure 6.3. Frame rate results on sgi Visual PC (frames per second).

The differences in Vega and VegaNT frame rates range from in favor of the Infinite Reality computer to slightly in favor of VegaNT when using full screen displays. Test position 1 looking South shows that this newer computer has faster overall system speed, but when facing North from position 1 with full geometry in view, the Infinite Reality suffered much less of a slowdown and almost doubled the frame rate of the newer Visual PC. The Infinite Reality always outperformed the Visual PC whenever the majority of the model was in the viewing frustum.

Multiple screen output was not available on the Visual PC, but based on the full screen numbers the older Infinite Reality would greatly outperform the Visual PC.

Possible explanations for the performance differences might be the fundamental architectural differences between Vega and VegaNT themselves. As mentioned in chapter IV, Vega uses the Performer scene graph. Performer is only available on sgi platforms that run the Irix operating system and directly accesses the specialized sgi graphics hardware. To make VegaNT function properly, Paradigm Simulations had to develop a Performer-like scene graph capable of running on WindowsNT. This leaves VegaNT graphics rendering at a disadvantage.

Frame rate numbers for Java3D on the Visual PC were not statistically different than those obtained from the Infinite Reality computer when compared to the differences experienced by the Vega applications. The number of pixels being displayed, on either sgi computer, had no effect on Java3D's frame rate, which strongly indicates that pixel-fill rate was not the limiting factor for frame rate.

One possible explanation for the almost identical frame rates on both sgi computers when running Java3D is the JVM implementation used by sgi hardware. This position is further supported in the next section where Java3D and VegaNT display similar behaviors given the graphics load capability.

3. Frame Rates Attained Using Intergraph PC

Hardware configuration for the Intergraph Personal Computer:

- Two 400 MHz Pentium II processors running WindowsNT 4.0
- Three Wildcat 6000 PCI graphics cards 16 MB each under AGP controller
- 512 MB system RAM

The computer actually used in the MAAVE was the last computer tested since it was being used for a field experiment almost until the completion of the MAAVE. After the surprising results from the Visual PC running Java3D, actual performance numbers for this computer were highly anticipated and are shown in Fig. 6.4.

Location Direction	Vega NT 640x480	Java3D 640x480	Vega NT 1024x768	Java3D 1024x768	Vega NT 3x1024x768	Java3D 3x1024x768
1N	21	3.8	12.2	3.5	8.5	0.4
1S	85	70	85	65	28	60
2N	21	3.8	8.5	3.3	4.7	0.4
2S	42	68	42	47	17	3.8
3N	21	3.8	10.6	3.5	4.5	0.5
3E	44	24	21	14	6.3	1.6
3S	36	24	28	14	7.8	2.2
3W	29	7	17	3.7	5	1.1
4N	26	4	14.2	3.6	5.3	0.6
4E	44	11	28	6	7.1	1.8
4S	29	23	17	3.8	7.7	1.3
4W	29	13	17	3.8	5.3	1.4
5N	29	6	17	3.6	4.8	1.1
5E	44	7	21	4	6.1	1.1
5S	29	7	14.2	3.6	5.3	0.6
5W	29	7	14.2	3.7	4.3	0.9
6N	44	10	28	8	10.6	2.6
6E	44	8	23	7	8.5	1.2
6S	21	3.8	10.6	3.3	5.7	0.5
6W	34	7	19	5.2	6.5	1

Figure 6.4. Frame rate results on Intergraph PC (frames per second).

VegaNT frame rate numbers were almost identical to those obtained from the Visual PC given the same number of pixels. The slight advantage for the Visual PC could easily be related to the difference in processor speeds. Again, frame rate responded appropriately to changes in the rendering window size with slower rates for larger windows.

This was the first comparison of multiple screen VegaNT to multiple screen Vega. Based on these numbers the Infinite Reality computer would be the better choice for powering the MAAVE. As mentioned in chapter III, the cost of replacing the Infinite Reality, or even upgrading it, would far exceed the cost of purchasing three PCs with the latest processors and AGP graphics cards. The frame rate performance differential was not deemed significant enough to justify the expense. Both the Visual PC and the Intergraph single screen numbers are close enough to the goal frame rate of 30 Hz when running VegaNT to infer that three networked PCs would be capable of providing the minimum desired frame rates.

For the first time, Java3D performed as expected. Since frame rate finally was dependent on number of pixels rendered, a true gauge for the speed differential between Java3D and VegaNT could be obtained. The three screen numbers for Java3D are somewhat suspect since a single canvas was only stretched across the entire environment. The actual field of view, and therefore culling performance, was not changed as pixel format changed. VegaNT actually had a 170-degree field of view for the culling routine to traverse. This may possibly explain location 1 South in the three screen view, in which Java3D ran faster than VegaNT. Figures 6.5 and 6.6 show the single screen performance of Vega and Java3D on the three platforms.

When Java3D was only displaying on a single monitor, it was just fast enough to be useable for maneuvering through the Herrmann Hall model. With the three-screen display, maneuvering became impossible due to the infrequent updates and collision-less motion.

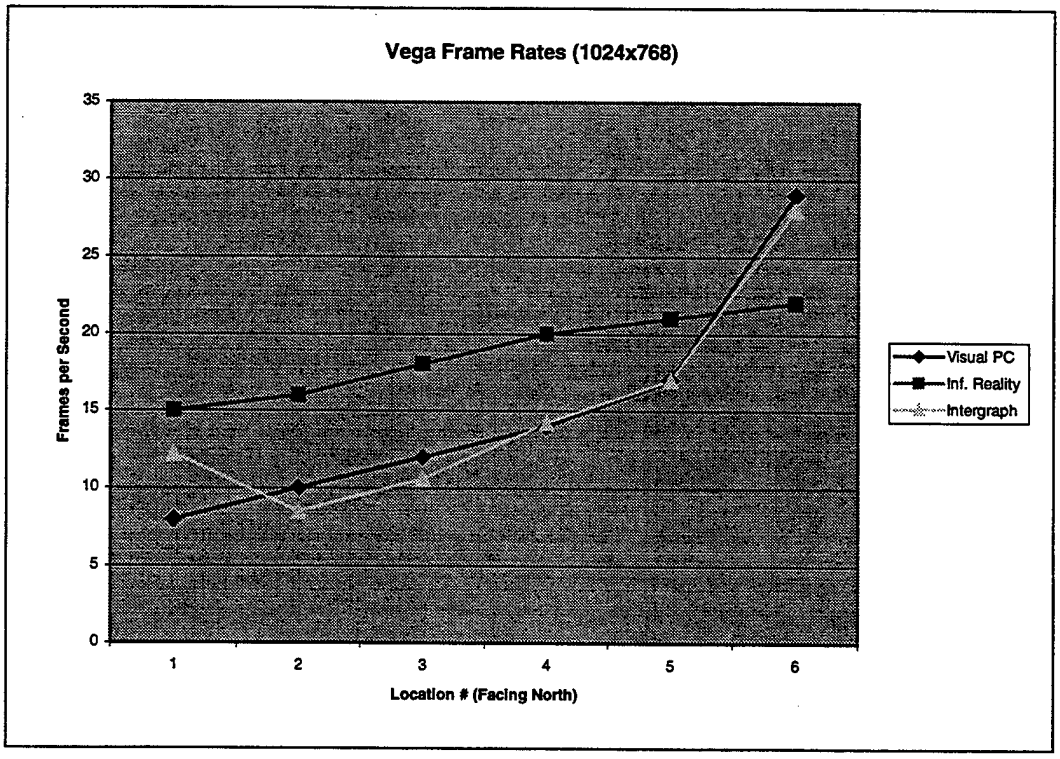


Figure 6.5. Performance graph of Vega facing North.

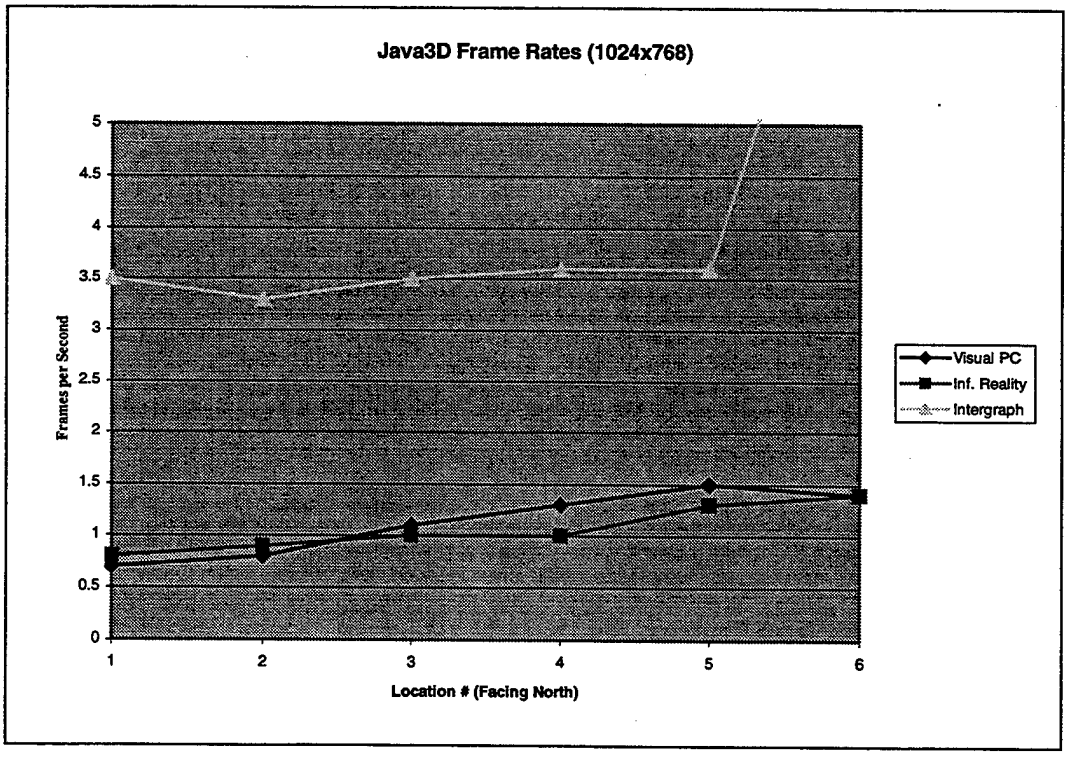


Figure 6.6. Performance graph of Java3D facing North.

C. IMPLEMENTATION COMPARISONS

The level of programming difficulty was marked and deserves comment. The largest difference is the portability issue. Java3D was reconfirmed to be cross platform compatible; once the code was compiled on one machine it functioned perfectly on all other machines. Vega and VegaNT had to be recompiled on every platform tested. The architectures on the individual machines were different enough that modifications to the C++ code were needed to obtain fully functional operations. Major coding differences were encountered using Vega between the Irix and WindowsNT implementations, especially with the multicast networking code.

Programming effort was also significantly different. The Java3D implementations for most of the Vega behaviors required between one half and one third the programming time. This ratio was even greater for the implementation of networking code since the specific socket connections needed to be hard coded for each variation in platform using C++, vice only once in the Java3D application.

Cost comparisons are hard to estimate since the only other comparable VEE is the NAVE. The total cost for the NAVE was reported at $60,000.00. Figure 6.5 shows the total cost of the MAAVE to also be approximately $60,000. The majority of these costs are for the projection hardware and screens, with the computer coming in second and all other construction materials a distant third.

Price list as built with educational discount and no sales tax:

3 VRex 2210 projectors	$38,600 total
3 Vrex rear projection screens	$3,000 total
Miscellaneous materials (wood, mirrors, etc.)	$1,400 total
Intergraph GT1 with 3 Wildcat 4000 graphics cards	$17,000 total
Total hardware:	**$60,000**

Vega licenses*:

Vega for Irix	~$15,000 each
educational price	~$7,500 each
VegaNT	~$8,000 each
educational price	~$4,000 each
DIS module	~$9,500 each
educational price	~$6,200 each

*Prices are approximate as of 1999. Call Multigen-Paradigm, Inc. for current price list.

Figure 6.7. Price list for MAAVE.

D. OVERALL RECOMMENDATIONS

The overall requirement of the virtual environment must be carefully chosen to determine which software implementation is appropriate. In all tested configurations, Vega greatly outperformed Java3D using the frame rate metric. If a complex computer model is used and smoothness of motion is crucial, Vega is by far the better choice. If, on the other hand, motion is not a priority (or a much smaller model is utilized) Java3D can be used. Since Java3D is freely distributed open source, it is certainly more cost effective than Vega. Due to the performance of Vega, the overall recommendation is to use Vega for interactive walkthrough simulations.

VII. FUTURE WORK

This thesis met its intended goals, however there is room for improvement in several areas to provide a better overall virtual environment experience. All three major areas of this thesis, the MAAVE, the Vega application, and the Java3D application, will be examined in turn to provide some ideas for that area's improvement.

A. MAAVE IMPROVEMENTS

Although the MAAVE provides a usable virtual experience, there are a few areas that can be improved. The graphics rendering capability is the area where improvements would most likely enhance user experience. A more optimized model of Herrmann hall would also ease the graphics issue. Other additions, such as spatialized sound and user position tracking ability, will also result in improvement.

1. Networked PC Configuration

The MAAVE was built using a three graphics card Intergraph machine to run the three graphic displays. Although the Intergraph provides an acceptable frame rate for some programs, Chapter Six showed that it is still slow for the Herrmann Hall model. Another low-cost solution for the graphics is to use three high-end PCs (one to run each projector) and a fourth mid-range PC to provide frame synchronization. This way, each computer is only responsible for one screen, which will dramatically increase the frame rate. This should allow the Vega application's frame rate to approach the human fusion frequency and make the Java3D application much more immersive.

The difficulty ofthis configuration is the view synchronization process. This involves writing software to send a timing signal to the three PCs to tell them to display the next frame. Each graphics PC must maintain its own separate graphics and entity states. This results in increased network traffic, which may in turn result in a lower number of usable entities. However, the gains from the increased graphics capability will likely outweigh the issues of increased network congestion.

The four graphics PCs would cost approximately $2,000 each. This configuration would lower the overall MAAVE price to approximately $51,000.

2. Spatialized Sound

Spatialized sound systems enable directional sound cueing, either by using multiple speakers or through phase modulation. The addition of spatialized sound to a visual simulation provides a more immersive and believable virtual experience.

The MAAVE currently has no speaker system to provide specialized sound. Two options were considered but were not implemented due to time restraints. The first option was to use an eight-speaker system with four front and four rear channels. These speakers were available but are large and bulky and would require heavy ceiling anchors to mount them. The second option was to buy a Bose™ Acoustimas™ surround system. This system has eight or ten cube speakers with two subwoofers. The appeal of this design is the small size of the speakers and ease of installation. Because of these benefits, the Acoustimas system would likely be a better choice for adding specialized sound to the MAAVE. Currently, headphones from a PC are available for a single user in the MAAVE.

3. Magnetic Position Tracking

Another planned improvement for the MAAVE is the addition of a position tracking capability. At NPS, a Polhemus Long Ranger™ is available that will likely be moved into the MAAVE. The Long Ranger is a magnetic tracker consisting of an 18" globe with three orthogonal coils comprising the transmitter. Combined with a Polhemus Fastrak™ system, the magnetic tracker has a range up to 30 feet with only 4 ms latency at 120 Hz [Polhemus00]. As this system is already available, it could easily be installed in the MAAVE. The system would need to map the magnetic eddies in the room to account for the speaker magnets and ferrous iron pipes in the room.

4. Better-Organized Model

An improvement to the Herrmann Hall model, although not MAAVE specific, would improve the performance of the Vega and Java3D applications. The model could be improved by better organization and use of surface meshes. Another improvement would be creating large texture tiles from the 96 separate textures, thereby reducing the graphics state transitions. This would give better rendering times, increase the frame rate and make a more compelling visual simulation.

The Herrmann Hall model is incomplete, and further construction of the database would greatly improve its realism. Additions would include moveable doors, an operating elevator, water fountains, room interiors, and additional hallway details such as fireplaces. Also, most windows are only one sided, so collision detection only works against that one side.

B. VEGA IMPROVEMENTS

VegaNT 3.3 proved very useful, but it still contains some software glitches that are intended to be corrected in release 3.5. With newer computer hardware the Herrmann Hall walkthrough program will function smoothly and can be used for navigation experimentation.

1. Spatialized Sound

Vega supports spatialized sound as part of the standard core application. Implementation is via area of influence spheres. When the user is within a sphere, they can hear the sound source. Sphere sizes are based on sound levels and attenuation. Sounds such as footsteps that indicate the user's speed and floor material would be a great enhancement. Of course, since Herrmann Hall is a large masonry structure, the footsteps would require some type of echoing for utmost realism.

2. Enhancing Network Capabilities

Current networking implements only the ESPDU and does not allow for the dynamic downloading of geometry models. Increasing this capability would enhance the usability of the MAAVE. These capabilities were not added due to time constraints, but their implementation is straightforward.

3. Additional Control Devices

A three-button mouse and PC joystick with slider are the only input devices currently able to control the movement of the walking motion model. Once the magnetic position tracking capability has been added to the MAAVE, additional code must be created to enable these devices. Inserting this code should be quick and simple once the device port path-name is known, since Vega and VegaNT natively support most positional input devices and tracking systems.

4. Creating User Avatar

Adding a humanoid avatar for the user would make the simulation feel more physically compelling. This needs to include arms that reach forward to open doors, a body that moves when walking, and a sinusoidal walking bounce to the view that is tied into speed of motion. Once an avatar has been created, it can be tied to the motion model. Then, by taking advantage of Vega's ability to tether away from motion models, the user could have either a first or third person view of the scene.

5. Implementing Levels of Detail

Vega contains LOD functionality that could help reduce graphics display times for distant complex models. This would be most useful for other avatars inhabiting Herrmann Hall, but could be used for the exterior of the building if Herrmann Hall were inserted into some larger terrain database.

C. JAVA3D IMPROVEMENTS

Given the frame rate findings, the Java3D application is not very usable. This is partly due to the overhead involved with running on top of the JVM, but also because of the application design. There are three major areas that could be improved in the application: collision detection, multi-channel support, and improved networking.

1. Collision Detection

The current application implements collision detection by using the built-in Java3D `WakeupOn` events. A better way to do collision detection is to use Java3D's picking algorithms. They allow a user to use `PickRay` objects that send a line out from the viewpoint in a desired direction. The `PickRay` object returns the closest object along its path. A straightforward distance comparison indicates if a collision occurs. This provides collision prediction, giving a better immersive experience than collision notification. The picking tools also include `PickSegment` objects, which act similar to `PickRay` objects but do not extend indefinitely. `PickSegments` emanating from the user's position can be used for accurate terrain following without accidental motion between building floors.

2. Multi-channel Support

Multi-channel support would make the application better suited for the MAAVE. It can be implemented in the current version of Java3D, but the next version of Java3D allegedly has

better built-in support for integrated multiple views. For, the three PC solution, multiple views would allow each of the three networked PCs to render only its channel and not need to know what the other PCs are rendering. As currently implemented, with one view, each PC would have to render the entire scene but display only one third of it. With multi-channel capability, the geometry would be drawn on separate Canvas3D objects, reducing the required rendering time. Multiple views will also be necessary to implement stereo images in Java3D.

3. Improved Networking

The application currently uses only the position fields of the ESPDU. Follow-on work should implement all of the state vector fields, including orientation and linear and angular velocities. This would facilitate the implementation of dead reckoning and reduce the PDU transmission requirements. The other three major PDUs, collision, detonation, and firing, should also be implemented.

Another way to reduce the network traffic would be to add Area of Interest Management (AOIM) algorithms. AOIM frees the user from knowing the entire entity database by allowing them access only to entities that they are interested in, i.e. entities that are either in the same geographic area or are of a certain type. Chapter Seven of [Singhal/Zyda99] provides a thorough overview of AOIM techniques and their benefits.

4. Java3D Performance Issues

The surprising results in Chapter Six raise issues about Java3D's performance that bear further examination. For instance, there was no difference in frame rate from the Visual PC to the Infinite Reality machine. Additionally, there was no significant change from small screen to

large or multi-screen on either platform. Common sense would dictate that increased resolution results in lower frame rates, however this was not the case. One possibility may be that the architecture of the Visual PC is not the standard IBM architecture that the JVM was written for. All of these issues need to be explored further.

Another issue with Java3D is that every 30 to 40 frames the application freezes for a varying length of time (2 – 5 seconds) and then resumes at the previous frame rate. This has been consistently observed in most Java3D applications. The phenomenon seems to be the result of an interruption to the Java3D rendering thread. This could be caused by a myriad of reasons including garbage collection, networking, or memory paging.

LIST OF REFERENCES

[C200] C2 web page at *http://www.vrac.iastate.edu/about/selabBuilt/index.html*

[CABIN00] CABIN web page at *http://www.ihl.t.u-tokyo.ac.jp/Projects/CABIN/*

[CAVE00] CAVE web page at *http://www.evl.uic.edu/EVL/VR/systems.shtml*

[Clark76] Clark, James H., Hierarchal Geometric Models for Visible Surface Algorithms, *Communications of the ACM* Vol.19 Number 10 (October 1976), 547 - 554.

[Cruz-Niera, et. al.93] Cruz-Niera, Carolina, Sandin, Daniel J., and DeFanti, Thomas A., Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE, *Proceedings of SIGGRAPH 93, Computer Graphics Proceedings, Annual Conference Series* (August 1993), 135-142.

[Fakespace00] Fakespace Systems, Inc. product page at *http://www.fakespacesystems.com/products.html*

[Full Sail00] Full Sail web page at *http://www.fullsail.com/index1.html*

[IEEE98] Institute of Electrical and Electronics Engineers (1278.1a-1998), Standard for Distributed Interactive Simulation – Applied Protocols, Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1998.

[Java3D00] Java3D API Specification page at *http://java.sun.com/products/java-media/3D/collateral/j3d_api/j3d_api_1.html*

[McAllister93] McAllister, David, Stereo Computer Graphics and Other True 3D Technologies, Princeton University Press, Princeton, NJ, 1993.

[Multigen00] Multigen-Pardigm product page at *http://www.multigen.com/products/prodindex.htm*

[NAVE00] NAVE web page at *http://www.cc.gatech.edu/gvu/people/jarrell.pair/NAVE/index.html*

[NPSNET-IV00] NPSNET-IV web page at *http://www.npsnet.nps.navy.mil/npsnet0/software.html*

[Paradigm: LynX98] Paradigm Simulation Inc., LynX User's Guide, Paradigm Simulation, Inc., 14900 Landmark Blvd., Dallas, Texas 75240, email: support@paradigmsim.com, 1998.

[Paradigm: Vega98] Paradigm Simulation Inc., Vega Programmer's Guide, Paradigm Simulation, Inc., 14900 Landmark Blvd., Dallas, Texas 75240, email: support@paradigmsim.com, 1998.

[Polhemus00] Polhemus web page at *http://www.polhemus.com/ourprod.htm*

[Rohlf/Helman94] Rohlf, John and Helman, James, "{IRIS} Performer: {A} High Performance Multiprocessing Toolkit for Real--{T}ime 3{D} Graphics, *Proceedings of SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series* (July 1994), 381-395.

[Singhal/Zyda99] Singhal, Sandeep and Zyda, Michael, Networked Virtual Environments, ACM Press, New York, NY, 1999.

[Sowizral/Deering99] Sowizral, Henry A. and Deering, Michael F., The Java3D API and Virtual Reality, *IEEE Computer Graphics and Applications* (May/June 1999).

[Sowizral, et. al.98] Sowizral, H., Nadaeu, D., Bailey, M., Deering, M. "Introduction to Programming with Java3D," ACM SIGGRAPH 98, Course #37 Course Notes, July 1998, in form of a printed book and on the course CD-ROM.

[Vince92] Vince, John, 3-D Computer Animation, Addison-Wesley Publishing Company, Inc., New York, NY, 1992.

[Wickens, et. al.98] Wickens, Christopher, Gordon, Sallie, and Liu, Yili, An Introduction to Human Factors Engineering, Addison-Wesley Educational Publishers, Inc., 1998.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Capt. Steve Chapman, USN 1
   N6M
   2000 Navy Pentagon
   Room 4C445
   Washington, DC 20350-2000

4. George Phillips 1
   CNO, N6M1
   2000 Navy Pentagon
   Room 4C445
   Washington, DC 20350-2000

5. Dr. Michael Zyda, CodeCS/Zk 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

6. Research Assistant Professor Michael Capps, CodeCS/Cm 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

7. LCDR Brian K. Christianson 1
   313 Metz Rd.
   Seaside, CA 93955

8. LT Andrew J. Kimsey 1
   2715 Nth. 5th St.
   Kalamazoo, MI 49009

9. Jaron Lanier 1
   Advanced Network Services
   200 Business Park Drive
   Armonk, NY 10504

10. Don Brutzman, Code UW/Br 1
Naval Postgraduate School
Monterey, CA 93940-5000