VR Juggler: A Virtual Platform for Virtual Reality Application Development

Allen Douglas Bierbaum

Major Professor: Carolina Cruz-Neira
Iowa State University

Virtual reality technology has begun to emerge from research labs. People are beginning to make use of it in mainstream work environments. However, there is still a lack of well-designed virtual reality application development environments. This thesis describes VR Juggler, a virtual platform for the creation and execution of immersive applications, which provides a virtual reality system-independent operating environment. The thesis focuses on the approach taken to specify, design, and implement VR Juggler and the benefits derived from this approach.

VR Juggler: A virtual platform for virtual reality application development

by

Allen Douglas Bierbaum

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTERS OF SCIENCE

Major: Computer Engineering
Major Professor: Carolina Cruz-Neira

Iowa State University
Ames, Iowa
2000

Graduate College
Iowa State University

This is to certify that the Mater's thesis of

Allen Douglas Bierbaum

Has met the thesis requirements of Iowa State University

_____
Major Professor

_____
For the Major Program

_____
For the Graduate College

TABLE OF CONTENTS

TABLE OF CONTENTS ..... **III**

LIST OF FIGURES ..... **VIII**

ACKNOWLEDGMENTS ..... **IX**

ABSTRACT ..... **X**

CHAPTER 1 INTRODUCTION **1**
- **Research problem** **1**
- **Statement of purpose** **3**
- **Scope of research** **3**

CHAPTER 2 BACKGROUND **6**
- **The promise of virtual reality** **6**
- **What is virtual reality?** **7**
  - Characteristics of VR 8
- **What is a VR System?** **9**
- **Examples of VR systems** **11**
  - Desktop VR 11
  - HMD systems 13
  - Single screen immersive projection displays 14
  - Multi-screen immersive projection displays 16
- **What is a VR development environment?** **17**
- **What makes up a VR application?** **18**

CHAPTER 3 VR DEVELOPMENT ENVIRONMENT REQUIREMENTS **20**
- **Primary needs** **20**
  - Performance 20
  - Extensibility 21
  - Flexibility 21
  - Simplicity 22
  - Robustness 22
- **Performance** **23**
  - Low latency 23
  - High frame rate 24
  - Support for hardware 25
  - Performance monitoring 26
- **Extensibility** **27**
  - Hardware abstraction 27
  - Simple extension 27
  - Do not require application changes 28

Flexibility **28**
Scalability 28
Cross-platform 29
Run-time changes 29
Support use of other application toolkits 32
Do not be overly restrictive 32
Simplicity **33**
Short learning curve 33
Rapid prototyping using simulation 33
Robustness **34**
Failure protection 34
Maintainability and correctness 34

CHAPTER 4 CURRENT DEVELOPMENT ENVIRONMENTS **36**
Iris Performer **36**
Summary 36
Availability 36
Platform 36
Supported VR hardware 36
Description 37
Strengths 39
Limitations 39
Alice **40**
Summary 40
Availability 40
Platform 40
Supported VR hardware 40
Description 40
Strengths 42
Limitations 42
CAVE Library **42**
Summary 42
Availability 43
Platform 43
Supported VR hardware 43
Description 43
Limitations 44
Strengths 44
Avango **45**
Summary 45
Availability 45
Platform 45
Supported VR hardware 45
Description 45
Strengths 47
Limitations 47
Lightning **47**
Summary 47

Source 47
Platform 48
Supported VR hardware 48
Description 48
Strengths 49
Limitations 49
MR Toolkit 49
Summary 49
Availability 49
Platform 49
Supported VR hardware 50
Description 50
Strengths 52
Limitations 52
World Toolkit (WTK) 52
Summary 52
Availability 52
Platform 53
Supported VR hardware 53
Description 53
Strengths 56
Limitations 56
Analysis of previous work 56
Performance is of utmost importance 57
Rapid prototyping makes development easier 57
Do not tie the environment to a specific graphics API 58
Environments need wide range of robust open device drivers 58
Monolithic architectures present problems 59

CHAPTER 5 THE ARCHITECTURE OF VR JUGGLER 60
VR Juggler microkernel core 60
Mediator 62
Kernel portability 63
Consequences 63
VR Juggler virtual platform 63
Virtual platform API 64
Architecture and OS independence 65
Device abstraction 65
Operating environment 66
Allow for use of multiple graphics APIs 67

CHAPTER 6 IMPLEMENTATION OF APPLICATIONS 68
Application object 68
Base application interfaces 69
No main() – Don't call me, I'll call you. 70
Benefits of application object 71
How to write an application 73
Derive from base class interfaces 73

Define drawing methods 73
Define processing methods 74
Get input from system 75
How does everything get started? 76

CHAPTER 7 DETAILED DESIGN OF VR JUGGLER 77
Microkernel 77
Mediator 78
Kernel portability 79
Configuration information 79
Internal Managers 80
Input manager 80
Environment manager 82
Display manager 82
External managers 83
Draw manager 83
Other external managers 83
Application 83
Multi-threading 84
System interaction 85

CHAPTER 8 DISCUSSION 87
Implementation methods 87
Challenges in VR Juggler development 87
Iterating based on applications 88
Iterations 88
Run-time reconfiguration 89
Extend/refine application interface 91
Multi-user extensions 93
Find and eliminate performance problems 93
How well did it meet the design goals 94
Virtual platform 94
Hardware abstraction 94
Run-time flexibility 94
Performance tuning 95
Cross-platform 95
Extensible 95
Problems encountered 95
Learning curve problem 95
Cross-platform programming 96
Java virtual machines 96

CHAPTER 9 CONCLUSIONS 98
Contributions to field 98
Flexible standard for building long lived applications 98
Virtual platform 99
Open source system 99
Reconfigurable system 100

CHAPTER 10 FUTURE WORK 101
Component system 101
VR operating system 101
VR tools 101

CHAPTER 11 BIBLIOGRAPHY 102

LIST OF FIGURES

Number *Page*
Figure 1: General overview of a VR system 9
Figure 2: Desktop VR system 11
Figure 3: HMD base VR system 13
Figure 4: Single projection screen VR system 14
Figure 5: Multi-screen projection VR system 16
Figure 6: VR application dependencies 19
Figure 7: System frame rates 24
Figure 8: Typical Alice script 41
Figure 10: Microkernel architecture 61
Figure 11: Application/VP interface 64
Figure 12: Application object 68
Figure 13: Application class hierarchy 69
Figure 14: Kernel frame 70
Figure 15: User app base classes 72
Figure 16: Sample application object 72
Figure 17: Input interface 75
Figure 18: Kernel startup 76
Figure 19: Microkernel architecture 77
Figure 20: Window chunk 80
Figure 21: Input device hierarchy 80
Figure 22: Device store 81
Figure 23: Display Manager 82
Figure 24: Kernel startup and execution 86
Figure 25: Reconfiguration classes 89
Figure 26: Dependency checking classes 90

ACKNOWLEDGMENTS

I would like to thank the people who have contributed to this research. First, I give many thanks to Chris, Patrick, Andy, and Kevin for their work on VR Juggler. Without them, I would have never been able to complete this research. Their comradery, insight, and tireless efforts have been invaluable.

I would also like to thank the many people who have been there for me during my education. I would like to thank my parents for always believing in me and supporting me even when I was unsure. I thank Terry and Kevin for their friendship and understanding during my research. I thank Chris and Scott for reminding me that there is much more to college than just studying all the time.

I would especially like to thank Lora. She was my friend and confidant during this time. She was always there for me when I needed to get away and just take a break with my best friend. I know that she had to make sacrifices so that I could complete this research. I thank her for all that she has given up so that I could accomplish this. I look forward to spending the rest of my life with her and making up for the lost time.

Lastly, I would like to thank God for blessing me with the abilities that have allowed me to complete my education and for walking with me through life's trials and tribulations.

ABSTRACT

Virtual reality technology has begun to emerge from research labs. People are beginning to make use of it in mainstream work environments. However, there is still a lack of well-designed virtual reality application development environments. This thesis describes VR Juggler, a virtual platform for the creation and execution of immersive applications, which provides a virtual reality system-independent operating environment. The thesis focuses on the approach taken to specify, design, and implement VR Juggler and the benefits derived from this approach.

CHAPTER 1 INTRODUCTION

Research problem

VR is a mature field that is being used by researchers in many disciplines to gain new insight into problems from many domains [1]. Recent advances in virtual reality (VR) enabling technologies have led to very innovative VR systems that integrate a wide variety of hardware and software elements.

These systems enable the development of advanced scientific and engineering applications in a wide range of disciplines.

Unfortunately, these systems place strong demands on application developers. Application developers are expected to not only have expertise in the problem domain, but must also have expertise in the development of sophisticated software systems. VR software systems utilize features such as multiple processes, message passing, shared memory, dynamic memory management, and a variety of process synchronization techniques. Developers must also be concerned with very low-level issues like device drivers for particular VR I/O devices, or techniques to generate multiple stereoscopic views. In many cases, the software developers do not have this background or the time to devote to these problems. Instead, they would like to focus on their specific problem domain and simply make use of standard VR software routines.

Until recently, the VR field has not been mature enough to develop a common development platform that is open for everyone to use and extend. It has been difficult for VR application development environments to keep pace with the rapid rate at which new technologies are appearing. Early VR software environments were tied to specific technologies or to particular requirements of individual applications [2]. Many of the software system were not designed to be long-term software platforms for VR applications. Although this method was effective when VR systems were built as proofs of concept, it is limiting the growth and usage of VR as an enabling tool for other disciplines.

There have been some notable attempts at creating standards [3][4], but most of them either focus on specific uses and requirements or are monolithic packages that offer little flexibility to developers. These software tools have enabled many generations of VR applications, but they still suffer from several key problems that a VR development must address. Some systems make use of hardware specific features thus tying the users to specific hardware architectures. Other systems

restrict developers to only using limited set of software tools in their applications. Many of the tools require that the application be changed when support for new devices and other technologies is added. The majority of the tools do not allow easy extension to add support for new technologies.

The lack of a common VR development platform has created a situation for developers where they must either make use of a single framework for all applications or make use of a wide range of disparate frameworks. If they chose to use a single framework, then they must use it for all applications even when the framework does not support the type of application well. If they chose to use many separate frameworks, then they must learn several incompatible programming interfaces and in most cases they cannot share code between separate projects. Neither of these options is optimal for the developers. Instead, they need a single common tool that they can use with many types of applications and allows the sharing of code between applications.

We believe that to enable the widespread use of VR technology a common VR application development interface needs to be engineered. This development interface should hide the specific details of the underlying technologies, providing a *virtual platform* to the application designer. A virtual platform enables researchers to concentrate their development efforts on the content of the application -that is on issues related to the visualization and manipulation of the data of the problem domain, and not on the details of complex programming issues for the immersive system being used. Furthermore, because VR technology continues to evolve, a virtual platform facilitates the scaling of existing applications to newer systems without affecting the core of specific applications. A virtual platform should also provide a functional environment to developers, so they can create and run an application independently from the resources available.

The VR community needs a standard platform in order to solve current problems and provide a basis for further progress. A virtual platform for VR development can alleviate the problems that developers currently face by providing a standard framework that incorporates the best of current VR techniques in an open system that everyone can build upon. The next steps that VR research must take deal with how to write applications that make use of common components and how to make VR more usable in corporate production environments. In order to solve these problems, the VR community needs to standardize its software technologies. Just as Microsoft Windows helped to provide a standardized platform for desktop PC's, VR needs a standardized platform.

Statement of purpose

This research addresses the lack of a standard VR development environment that is designed to be extensible, maintainable, and freely available to the VR community. This research introduces VR Juggler, a standard platform for VR application development.

VR technology has progressed to a state where is has become feasible and desirable to create a standard development environment that is open and extensible. The purpose of this research is to design and begin development of such an environment, VR Juggler. The research started by analyzing the specific functional and non-functional requirements of VR applications and developers. Next, using these requirements as a guide, we analyzed existing VR development environments to see what lessons can be learned from current tools. After drawing upon these experiences, a team of developers analyzed the requirements and created an initial design for a new development environment, VR Juggler. Once a baseline implementation of the design was completed, we began to progressively refine the requirements and design based upon case studies of applications that were developed with VR Juggler. The information gained from application developers was used to further shape the development environment.

Scope of research

This research presented here was completed in several stages, which are:

1. Define and categorize VR specific needs and requirements

To achieve the goal of producing a standard VR development platform, we started by carefully analyzing the requirements of VR applications, development environments, and developers. Specifically we answered the question, "what requirements are there for VR software, and how can these requirements be organized in a manageable way?" Once the requirements of a VR development environment are defined, they can be used to evaluate current and potential VR development environments in an objective manner. The requirements also help to focus the research into areas where current environments are weak.

2. Analyze existing VR development environments

This research explores and extends the current state of VR software. To this end, an evaluation was performed of the major development environments that are currently in use making note of specific strengths and weaknesses of the environments. From this information, it was possible to find many useful features and designs to incorporate into a new VR development environment as well as determine what things to avoid.

3. Analysis and design of VR Juggler based upon VR requirements

After completing the review of current tools, it was clear that there is a need for a standard development environment. Because such a development environment would need to build upon and bring together many mature software techniques, we created a set of functional and non-functional requirements based upon the analysis of current environment's completed in step two and combined this with an understanding of the needs of VR environments gained from step one. From this list of requirements, we performed an initial analysis of the system to further refine the requirements. This iterative process continued until we had the analysis model that dealt with the majority of the architecturally significant requirements. Based on this model, an initial design of VR Juggler was created.

4. **Initial implementation of VR Juggler core**

Once we had the initial design, we implemented the basic core of VR Juggler's architecture. During the implementation of the core, the system requirements were further elaborated and refinements were made to the design. When the core was completed, it was now possible to incremental add and test new system components. This also allowed many developers to work on the system simultaneously by working on separate components of the system.

5. Iterative refinement of the design

After completing an initial implementation of the design, the design went through many refinements. We created applications with the development environment, analyzed how well the design of the framework held up, and looked for ways we could further improve the design. Design improvement did not consist solely of adding new features and fixing bugs, it also included refactoring the current system to make it simpler or more flexible. We spent a large amount of research time in this stage of development because the iterative refinement process presented us with many opportunities to gain new insights and evaluate new ideas.

6. Test with everyday application development

As part of the refinement process, many users tested the architecture in everyday use. Developers tested the VR Juggler design in two VR courses at Iowa State University, by creating numerous projects at VRAC, and by many developers outside of VRAC. This testing provided valuable feedback leading to changes in the system and further refinement in step five.

These research stages are presented in this thesis as follows:

- Chapter 2 covers background material and definitions for VR.

- Chapter 3 satisfies stage one of the research by outlining the needs of a VR development environment.
- Chapter 4 satisfies stage two of the research by presenting an overview an analysis of existing VR development environments.
- Chapter 5 satisfies stage three of the research by describing the analysis and design of VR Juggler.
- Chapters 6 and 7 satisfy stage four by describing the implementation of VR Juggler.
- Chapter 8 satisfies stages five and six by discussion the iterations that VR Juggler underwent and the application testing that was used to guide the refinement.

CHAPTER 2 BACKGROUND

The promise of virtual reality

VR holds many promises for the future of human computer interaction by simplify the way in which humans and computers interact. It also has the potential to open new avenues of interaction that are not currently possible.

In future VR environments, it may be possible to test a car design without physically producing a car. A VR simulation of the will allow for the same interactions that a person would normally use in a real car. The computer will also simulate the design in order to test out performance and other vehicle capabilities.

As another example of the promise of VR, consider that in the future it will be possible to use VR to take virtual tours of distant locations. From the comfort of your own home, you will be able to enter a virtual world where you can see and feel everything that you could if you were really at the place you are virtually visiting.

VR could also greatly change the way in which people conduct business. Imagine for example that you want to meet with several other people spread across the world. You could enter a virtual meeting room where each person has a virtual embodiment that represents them in the environment. You can see people and interact with them as if they were in the environment with you. It may become as natural to meet and interact in a virtual environment, as it currently is to meet them in real life.

VR shows many promises for the future, but there are still technological challenges that researchers and developers must face in order to fulfill these promises. The research presented in this thesis deals with the software technology problems that VR presents. Before presenting the research, there are several concepts that need to be described to have an understanding of what VR is truly about.

What is virtual reality?

Rory defines a Virtual Environment[1] system as:

> Systems capable of producing an interactive immersive multisensory 3-D synthetic environment; it uses position-tracking and real-time update of visual, auditory, and other displays (e.g., tactile) in response to the user's motions to give the users a sense of being "in" the environment, and it could be either a single or multi-user system. [5,p. xx]

This definitions shows that there are several common features that contribute to a virtual reality environment:

VR applications must be ***interactive***. In an interactive system, the input from the user controls the system. Virtual reality uses interactivity to guide application behavior and enable the user to directly modify the virtual environment. This level of interaction engages the user in a way that may seem more natural because users have a feeling of connection to the application -- the environment is directly responding to their stimuli.

VR applications provide a sense of ***immersion***. Immersion has three distinct aspects. An immersive application must be perceptually immersive, it must provide a sense of presence, and it must provide a sense of engagement. Many people use the term immersive to be all encompassing of all three of these aspects of immersion, but it is actually possible to discuss each individually.

For a VR application to be immersive, it must be perceptually immersive by providing "the presentation of sensory cues that convey perceptually to users that they're surrounded by the computer-generated environment." [5] This means that the VR application is providing all-encompassing sensory input to the user.

Providing the user with a sense that they are "in" the application is providing a sense of ***presence***. This sense of being "inside" the applications is referred to as an ego-centered frame of reference.

The final element of immersion is ***engagement***. Engagement is the degree to which the user has a sense they are deeply involved in the environment.

An immersive environment can be very convincing. In some cases it can actually convince the user that they are a part of the running application. When the user enters this state they have entered into a "suspension of disbelief" where they willing accept the virtual environment as real.

---

[1] The term "virtual environment" is used by Rory instead of "virtual reality" because of the common misconceptions of the term "virtual reality"

VR is ***multi-sensory***. A VR system makes use of multiple human sensory systems to present the virtual environment (VE) to the user. These senses may include: visual, auditory, haptic, smell, and taste.

Multi-sensory presentation increases the level of immersion. By involving more senses in the experience, the virtual environment can provide a higher degree of engagement and a greater sense of presence because the user is presented with a more complete representation of the world.

VR is ***synthetic***. This means that the computer system synthesizes the environment at run-time. The environment is not a pre-recorded presentation that is simply presented to the user; the environment is actually created at the time of presentation.

VR uses multi-modal interaction. VR applications make use of several methods of input simultaneously. By allowing multiple methods of input, a VR application can allow for interaction that is more natural then may be possible in a non-VR application.

Characteristics of VR

VR applications differ from conventional interactive applications because of several characteristics specific to VR environments. These characteristics form the basis for the special needs and concerns of VR systems and VR development environments.

- **VR applications are "user centered"**
  This premise can be summed up in a single statement, “it is the user that matters.” The perceived experience by the user is the overriding concern of the VR application developer. If the users perceives the application in an adverse way, then it does not matter how correct the algorithms are, the application is faulty. Because this single characteristic is so important, nearly every decision in VR application development is based on this single characteristic.
- **There is an integration of many physical components with differing interfaces and performance characteristics.**
  VR applications make use of many devices and system components. Each of them behaves differently and requires different levels of system interaction. In addition, each component has differences in performance, ease of use, robustness, and programming interface. This conglomeration of components leads to a complex and potentially fragile system.
- **Complexity is an intrinsic part of any VR system.**
  VR systems, the hardware and software needed for all VR applications, are inherently complex. The complexity is intrinsic because VR brings together a wide range of distinct hardware components combined with an abundance of advanced software tools and algorithms. Bringing all these separate technologies together in one system is a required aspect of VR.

These characteristics uniquely distinguish VR applications from traditional interactive computer applications because they place many additional requirements on the software that are not normally required. In VR applications, there are stringent requirements on human computer interaction. In traditional applications, users can tolerate latencies and uneven response time from the application; if the application does not respond predictably, it is just an annoyance. In a VR application, it could cause physical effects [5,p. 80].

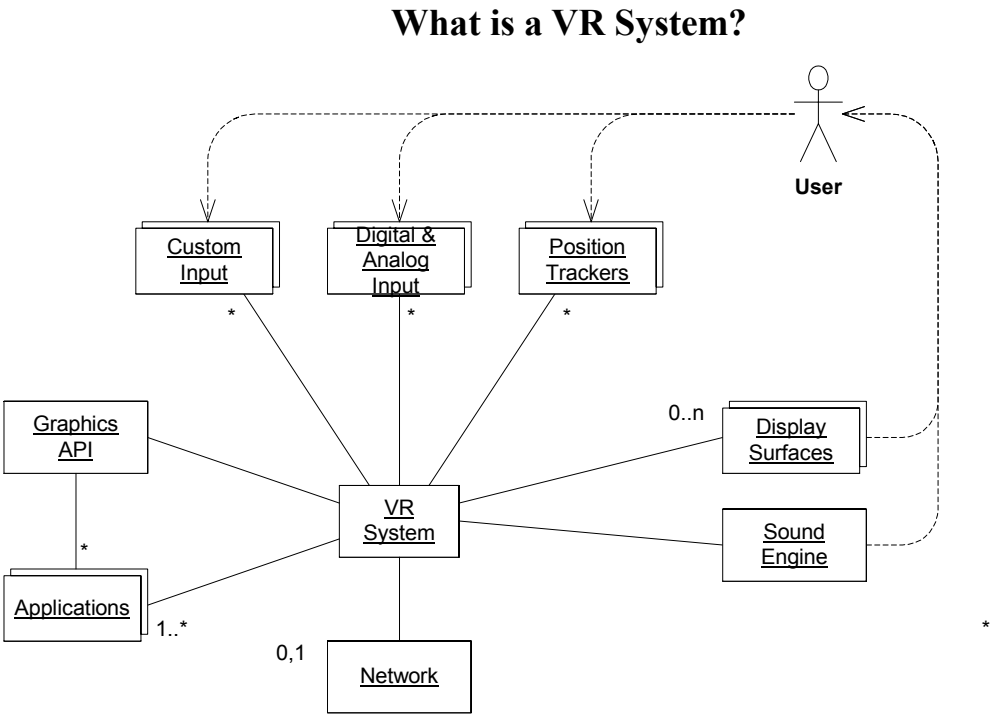

Figure 1: General overview of a VR system

A VR system is the combination of the hardware and software that enables developers to create VR applications that present a virtual environment to users. The hardware components of a VR system receive input from user-controlled devices and convey multi-sensory output to create the illusion of a virtual world. The software component of a VR system manages the hardware that makes up VR system. This software is not necessarily responsible for actually creating the virtual world. Instead, a separate piece of software (the VR application) creates the virtual world by making use of the VR software system. The specifics of VR application software are covered in the next section.

The first duty of the VR hardware system hardware is to receive input from the user or from external input sources. The VR system receives input from tracking systems, gloves, digital input

devices, and a wide variety of other devices. Each of these input types is explained below in more detail.

Positional input devices provide information about the location of the user in space. These tracking systems are composed of a device called a tracker and a base unit. The tracker can be attached to the user, to some thing worn by the user, or to other devices used by the user. The base unit remains stationary so the tracking software can use it as a reference point for calculating the position of the tracker(s). Tracking systems may also include advanced software drivers that filter the positional information or make use of predictive algorithms to approximate the position of the tracker in the future.

Positional information is needed to allow for immersive and interactive applications. The positional information is used to synthesize the environment for the users current location. Positional information is also used to detect what virtual objects the user is attempting to interact with.

Because humans interact most directly with their environment using their hands, it is only natural that they would like to do so in a virtual environment as well. Glove devices provide the system with information about the current arrangement of a user's hand. From this information, software is used to calculate the current position of all the hand's digits.

The hand information can be used directly to provide visual feedback or perform collision detection. In most cases, the hand information is given to a piece of gesture recognition software. This software allows the system to recognize certain preset "gestures" that the user may use to interact with an application. A VR application can use these gestures to control the application.

Many current VR devices use experimental interfaces. Because of this, VR systems allow for general signal input in the form of analog or digital values. This can allow for the use of devices that are in development. This type of input is also commonly used with devices that have buttons or dials on them.

In addition to these types of input devices, there are others including: speech recognition, bio-signals, locomotive input, and more. Because there is an every increasing amount of input available, a VR system must be able to handle many types of input at once.

The second duty of VR hardware systems is to provide multi-sensory output to the user. To give the user feedback about the virtual environment, VR applications employ a wide range of output technologies the most common of which are used for visual output. Visual presentation devices include projection-based systems, HMD's, and CRTs. In addition to visual feedback, many VR applications also provide auditory feedback using localized sounds. Some VR applications also make

use of tactile and haptic feedback to enhance the virtual environment. In the future, there may be output devices for the remaining senses as well.

VR software systems must provide access to all these types of input and output technologies to successfully create a virtual environment. Other types of applications outside the realm of VR can also make use of a large number of technologies, but VR is different in that even simple VR applications require the use of many technologies. In addition, VR applications need to make use of a gamut of software technologies to not only manage the VR system but also to create and present information to users. The integration of all these technologies makes VR applications not only powerful, but also complex.

There are several common classifications of VR hardware systems in use today. The next section gives a brief description of each of these classifications and describes the complexities associated with each system. Notice that even the simplest of these classifications makes use of a complex set of systems.

Examples of VR systems

Desktop VR

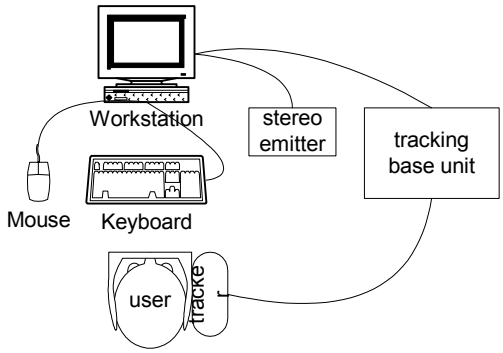

Figure 2: Desktop VR system

Desktop VR is the most basic type of VR system. Desktop VR systems, commonly known as "fish-tank VR", are a natural extension of the traditional desktop computer metaphor. In a desktop VR system, a traditional graphics workstation is used with head tracking and various other input devices. A single user is tracked and their view is shown on a desktop monitor using either a stereoscopic or monoscopic view.

Even in this simplest of VR systems, there are many software complexities. The software system has to get tracking information and integrate that positional information into the running application. If the system is running a stereoscopic view, then it must also make sure the stereo view is synchronized with either active glasses or with whatever passive display method is employed.

Benefits

- **Relatively inexpensive**: Desktop VR systems are composed of hardware that is part of commodity computer systems. Because of this, most of the components are inexpensive.

- **Easy to setup and reproduce**: Most computer users are already familiar with installing devices for use with a desktop computer. Since desktop VR systems only add a few devices to a normal desktop computer, this makes it easy for users setup and run such a system reliably.

- **High resolution:** Desktop monitors commonly have higher resolution graphics then other types of VR displays. This extra resolution allows for the user of applications with fine graphic details such a small text.

- **Allows for multiple users:** It is possible for multiple users to simultaneously view a desktop VR display, although only one of them may be tracked at any given time.

Shortcomings

- **Limited sensorial immersion:** The display on a desktop VR system only covers a small area of the user field of view (FOV). This limits the amount of visual immersion that is possible on such a system.

- **Frame violations at edge of screen:** Stereoscopically displayed objects are displayed incorrectly at the edges of the desktop monitor because one eye receives a view of the object, while the other eye does not.

HMD systems

An HMD device places a pair of screens directly in front of the user's eyes. A helmet worn by the user supports the displays and contains all the display hardware needed to run the displays. The displays cover the user's field of view, effectively isolating them from their surroundings unless the HMD has a passive display in which case the virtual display is combined with the real world.

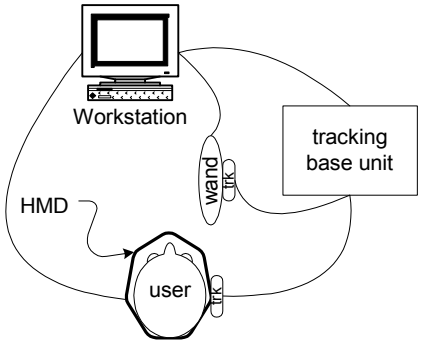

Figure 3: HMD base VR system

The software system for an HMD based VR system has all of the software complexities that a desktop VR system does, but adds an additional layer of complexity. In an HMD system, latency become much more of an issue than it is in a desktop based system. In an HMD system, low frame rate or high lag can begin to cause cybersickness. This was not as much of an issue in the desktop system because the user does not have their entire field of view covered as they do in an HMD system. Another complexity encountered with HMD systems is that they require the software system to keep stereo frames synchronized correctly. Since the HMD has a screen for each eye, the software needs to correctly generate the views for each eye and route that image correctly to the display hardware.

Benefits

- **Complete visual immersion:** By placing a screen in front of each eye, HMD displays cover the entire FOV for each eye.

- **No stereoscopic frame violations:** Since there is no projection surface with an edge (such as a monitor or a screen), there are not stereoscopic frame violations.

- **Easy to setup and maintain:** Because these systems place most of the complex hardware within a single device (the HMD), they can be much easier to maintain than more complex VR hardware systems.

Shortcoming

- **Invasive:** An HMD has weight and inertia. This can make such systems very invasive to new users. It can also lead to physical strain and discomfort after extended use.

- **Isolation:** An HMD separates the user from the real world.

- **Single user only:** Only one user may use an HMD VR system at once. This can make it difficult to discuss the environment with others.

Single screen immersive projection displays

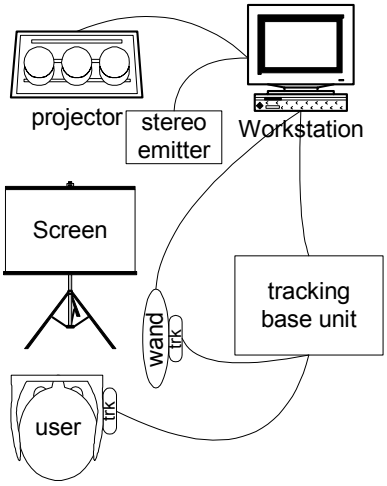

Figure 4: Single projection screen VR system

In a single screen projection VR system is the user is tracked and the visual representation of the environment is displayed on a single screen with which the user interacts. The user interacts with There are many variation of this type of system such as desks and wall based displays. This type of system can allow for either single user or group interaction in a VR environment.

A single screen projection system has many of the same software requirements of a desktop VR system, but adds several key elements that can greatly increate software complexity. First, because of the larger field of view covered by the large projection screen, lag and latency become much more of an issue much as they do in an HMD based system. In a projection based system, low frame rate and system lag can cause cybersickness for the user. Another added complexity of a single screen projection system is that the generation of stereoscopic frames can be more difficult than it is on a desktop system.

Benefits

- **Larger FOV than desktop:** Projection screens have a larger FOV than desktop base VR systems, leading to a more immersive environment.

- **Can render objects at correct scale:** Because the projection surface is large, the objects that are rendered can be represented at full scale to give the user a better understanding of size.

- **Non-invasive:** Projection based systems are much less invasive than HMD systems. This makes it much easier for new users to use the environment and also allows for longer usage.

Shortcomings

- **Frame violations at border of screen and from user's body:** Projection based VR systems suffer from two forms of stereoscopic frame violation. This first type is border violation, and is identical to the type of violation that occurs when using desktop VR systems. The second type of violation occurs when a virtual object is between a part of the users body and one of their eyes, the hand occludes the object giving incorrect depth information

- **Restricted range of movement:** The user is restricted to the area in front of the projection surface.

Multi-screen immersive projection displays

In a multi-screen projection VR system a single user (possibly multiple users) is tracked within a system. The system has multiple adjoining walls, each of which has images projected onto it. These walls display the visual representation of the environment to the user. Most systems present stereoscopic images to the user.

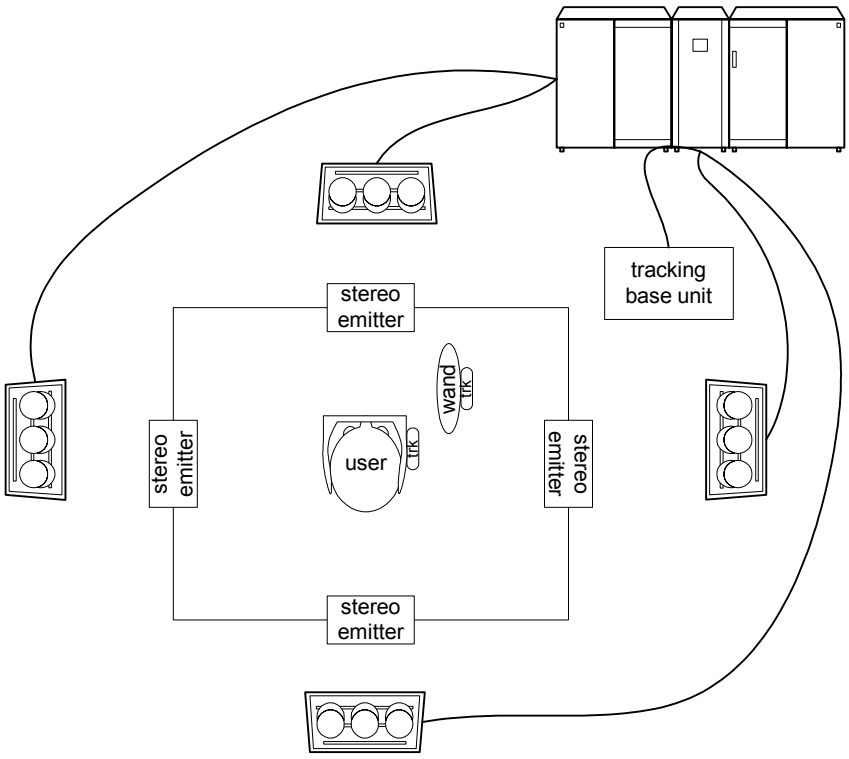

Figure 5: Multi-screen projection VR system

The rendered image on each wall may come from a single graphics engine in one machine, multiple graphics engines in one machine, or multiple graphics engines from multiple machines. Because the images may be coming from separate image generators, the images must be synchronized so that changes in the virtual environment being displayed are updated on each display simultaneously.

Benefits

- **Large FOV:** This type of VR system can cover the entire visual field.

- **Non-invasive display:** Just as with the single screen projection environment, multi-screen projection environments are less invasive then other VR hardware systems.

- **Allows multiple users:** Multiple users can share the environment although it is common to only track one of the users.

Shortcomings

- **Space usage:** This type of environment requires large amount of space to setup.

- **Occlusion violation:** When an object is between the user's hand and eye, the hand occludes the object giving incorrect depth information.

- **Calibration:** The system requires precise adjustment and calibration of walls and projectors

All of the outlines VR systems are very different from the hardware technology point of view, but to a user application need to be independent of the hardware systems. A VR software system needs to present a single common view that does not depend upon the specific hardware being used.

What is a VR development environment?

A VR development environment provides developers with the software framework, libraries, and run-time needed to develop and execute VR applications.

A VR development environment provides is a common base on which to write applications. It abstracts hardware and software complexities in the system thereby allowing users to write applications without having to know every detail of the system. The development environment allows users to concentrate on the developing the applications that use the environment instead of applications that manage the environment. In this way, a development environment simplifies the software development process and helps to decrease production time.

The development environment provides the common application base by defining a domain specific software architecture specialized for VR application development. This software architecture includes components for management of input devices, presentation of the environment, and processing any simulation that is part of the application. Many software architectures also extend this architecture to include components that manage thread allocation, resource management, and networking. The rest of this section will examine the components of the software architecture in more detail.

The software architecture specifies how all components of the system interact. Because the application is simply another system component, the architecture also defines the way in which applications interact with the system. The architecture specifies the structure of an application, the order of system events, and the allowable communication methods between the application and the system. Since the framework strictly controls system interactions, it is able to define reproducible behaviors in the system and applications.

Development environments also provide common routines that developers do on have to use in order to successfully create an application, but are merely provided to make application development easier. The routines may provide solutions to common needs or to problems that are difficult to correctly solve. These types of routines prove especially helpful for new users because they reduce the amount of code that must be written.

For example, many development environments provide routines to provide basic navigation in a virtual world. These routines encapsulate the mathematics and user interaction that are necessary to allow an application user to move about in the virtual world presented by the VR application.

Any development environment must also provide for application execution and debugging. The development environment not only specifies how a user creates, it also specifies how to execute an application. There are varieties of ways to execute that an application could be executed in a VR development environment. The application may just be a “normal” binary application that links against a library of VR routines, a script that is evaluated by the VR system to create the environment, or a component that is loaded into a running system. Whatever the method, the VR development environment provides the tools needed to correctly execute the application.

Additionally, a development environment should provide debugging assistance to developers. Debugging abilities can greatly increase developer productivity by minimize the amount of time spent finding bugs [6]. Common examples of features that can ease debugging are: system message logging, traceable code, code assertions to check for bad parameters or system state,

What makes up a VR application?

A VR application is a program that uses VR as an enabling technology to solve a practical problem. This means that the application constructs an immersive virtual world where the user makes use of natural interaction methods to control the application. By definition, this means that the application must be interactive, immersive, multi-sensory, and synthetic.

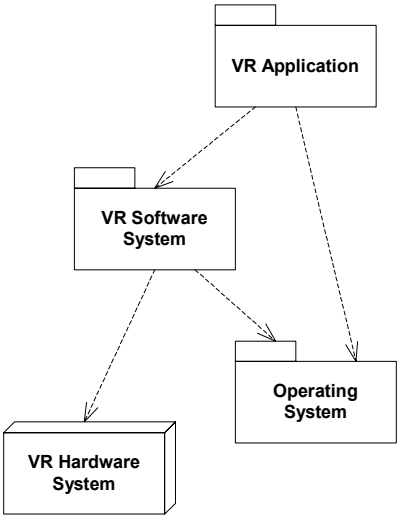

Figure 6: VR application dependencies

VR applications are built on top of the VR software system, which in turn controls the VR hardware system that is being used (See Figure 6). By building the application using the VR software system instead of directly accessing the VR hardware system, applications can run on many different VR hardware systems without requiring changes to the application code.

VR applications require real-time performance. The term real-time as it is used with VR applications has a loose meaning. It can mean that from the user's standpoint, events are perceived as occurring simultaneously or just meaning that that the system is time-critical. Real-time in the context of VR is not the same as in the context hard real-time systems [,pg. 1625]. This means that computations are still correct if they are not met within a given time constraint, but the system makes a best attempt at completing the computation within the given time frame. This does not mean that there are not some hard time constraints in a VR system. It just means that there is a range of valid times. It is a fuzzy real-time system.

Real-time performance is needed to meet the needs of the user. If the system is not updating the environment quickly enough, then it is impossible for the application to operate in an interactive manner. Performance problems in VR applications can also cause the user to experience "cybersickness." This occurs when the system is not updating perceptual output fast enough for the user.

It is worth noting, that a VR application is not a straight translation of a desktop application. VR applications need to provide an immersive environment where the user interacts naturally with the information presented. Most desktop applications are not written in a way that allows them to be easily transferred to this form.

CHAPTER 3 VR DEVELOPMENT ENVIRONMENT REQUIREMENTS

A VR development environment must address several specific needs in order to successfully create VR applications. This section divides these requirements into five broad categories: performance, extensibility, flexibility, simplicity, and robustness. This chapter first gives an overview of each of these general categories then proceeds to enumerate and describe many specific requirements in each category.

Primary needs

Performance

Performance is the key requirement of any VR system. VR applications are "user centered", therefore the physical comfort and experience of the user is of vital importance. As covered in the previous chapter, the experience of the user relies upon presenting an interactive and engaging environment. If the performance of the system is too low, the interactivity of the system becomes erratic and can lead to disengagement from the application that significantly degrades the experience of the user. Poor performance is not merely an inconvenience for the end user; performance problems can cause serious physical side effects including disorientation and motion sickness [7].

Because of these potential problems, VR software requires the utmost in performance [8]. Effective immersive environments must maintain a high visual frame rate (15hz or better) and maximize the responsiveness of the system to user inputs [, pg. 695]. To achieve the best performance, VR systems should take advantage of all available resources on a system, such as processors and special graphics hardware. In addition, the development system itself should have as little application overhead as possible.

Current VR software systems have been successful at achieving good performance. Unfortunately, many of these systems do so while neglecting several fundamental needs of a software system such as: reusability, extensibility, flexibility, portability, and robustness. In some cases, system developers sacrifice these needs in an attempt to increase performance by tying the software system as closely to the hardware as possible. Another reason that current systems may not implement these features is that it is much more difficult to design a system that supports these features. Since the primary focus of VR research to date has been hardware systems and not software

systems, these types of features have not received the attention that they deserve in a VR software system. We believe that a high-performance VR software system does not need to sacrifice any of these features in order to maintain high performance. In addition, we believe that these features are vitally important for creating a long lasting standard VR software system.

The next sections discuss some of the software architectural needs that are commonly overlooked.

Extensibility

Extensibility in a VR development environment allows user applications to survive technological changes of the future. Extensibility refers to the ability to add new features and extensions to a current software system. Extensibility is required because the hardware and software tools used for VR development change rapidly. Researchers are constantly creating new VR hardware devices that must be supported by development environments. The development environment should not require a programmer to re-write their application every time support for a new VR hardware device is added.

If a development environment does not allow easy extension, then it becomes difficult for users to write applications that can survive into the future. To avoid rewriting applications for new hardware, application developers need the ability to write an application once and rely upon the VR development environment to support future hardware advances. Although it would be adequate to simply require users to re-compile to get support for new hardware, it is better if the users are not required to even re-compile. In order to avoid the need for re-compilation, a development environment must support dynamic extension.

Flexibility

Extensibility of the software architecture is not enough. The software architecture must also be flexible enough to adapt to new requirements. Flexibility here refers to the ability of the system to adapt to the shifting configurations and changing requirements of a VR system. For example, the development environment must support multiple operating systems in addition to supporting many types of graphics software and hardware.

Development environments should not require developers to rewrite an application for every type of VR system. Instead, the software should adjust itself to the local VR system and facilitate the execution of the user's application. If the environment cannot adapt to new configurations, applications will be limited in the scope of their usefulness.

In addition, the design of the system itself should not lock developers into writing only one given type of application. For example, the development environment must make it just as easy to write a

passive architectural walk-through application as it is to create an interactive scientific visualization application. This requires a development environment that is not only flexible about what hardware it is running on, but is also flexible about what type of applications and toolkits are running within the software system.

Simplicity

Although a VR system is inherently complex, a VR development system does not have to be. The complexity of VR systems has unnecessarily led to the expectation and acceptance of corresponding complexity in development environments. This software complexity limits the ability of on-technical users to develop VR applications.

As more people begin using VR, system designers need to simplify development environments to allow for widespread application development by non-technical users. VR allows users from many fields to gain insight into their problems, but these users are not necessarily expert software developers. Because users may not have software expertise, a VR development environment should be as simple and easy to use as possible. They should not have to worry about the complexities of VR systems, but should instead be able to spend time creating innovative applications.

Robustness

Before VR applications can completely escape the domain of research, applications will be required to run reliably. Many current VR development environments were developed in research labs that have contributed many innovations and continue to do so. The problem is that in a research lab a "good" program may only be required to be partially stable; crashing one out of five times is commonly considered acceptable. It is research after all, so if the application crashes occasionally, that is to be expected. Outside the research lab, users are not so forgiving.

VR is beginning to enter the mainstream of corporate users. These users will make use of VR applications in their production environments, and will not settle for down time or sporadic behavior in an application. VR development environments need to consider this in order satisfy the rigid demands of corporate users.

The next pages will look at each of these broad requirements of VR development environments in more detail.

Performance

Low latency

Latency is defined as the total delay time between a user action and the system response [,p. 695]. Latency can come from the data rate of input devices; the time spent processing input, running applications simulation, and rendering output; the time required for multiprocessor synchronization; the refresh rate of display devices; and cumulative transmission times [5,p.69]. Delays in the system introduce the lag that causes latency in a VR environment.

High system latency adversely affects engagement and presence in applications because it causes cur conflicts for the user. System lag causes cue conflicts because although the user may have made a change to the system state, the system may not have updated to that change yet. A commonly observed cue conflict occurs when the user moves their head but the tracker data has latency such that the image generator does not update the user's view to the new position fast enough to fool the visual senses of the user. This effect can be very disorienting for the user and at best causes them to disengage from the application.

Latency can also cause users to experience an uncomfortable side effect called cybersickness. Cybersickness consists of motion-sickness-like symptoms during the use of a virtual environment and residual effect afterwards [,p. 805]. According to Rory, the effects may include nausea, disorientation, stomach awareness, fatigue, and headache. Motion sickness can also have after effects including postural instability, weakness, fatigue, and visual problems.

Several theories exist about the cause(s) of cybersickness. The most commonly accepted theory attributes the phenomenon to cue conflicts such as visual cues without vestibular cues. Cue conflicts can be caused by purposely creating an environment that behaves in a way that is contrary to real-world behavior, or more commonly, the cue conflicts are caused by lags within the VR system being used.

A development environment must reduce system latency in order for the system to be usable.

High frame rate

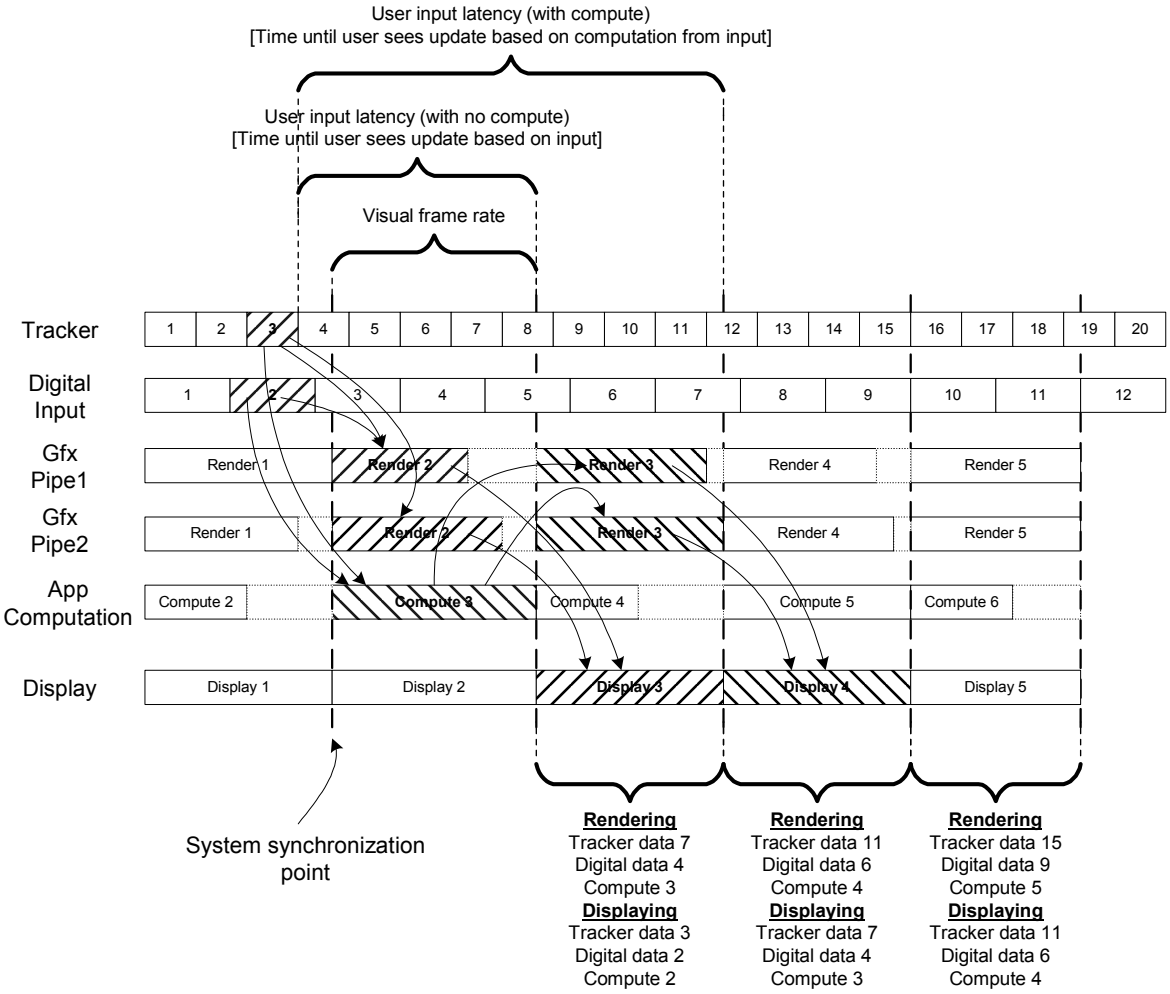

Figure 7: System frame rates

There are many potential areas in a VR software system where latency can be introduces. In order to reduce latency, the sources of the latency must be thoroughly understood. Figure 7 shows a diagram that outlines the sources of latency is the system. The diagram shows the path of a single set of sampled input data. The effects of the data are traced through the system

The diagram shows the number update "frames" of several system components. The input devices are running asynchronously to the rest of the system which is coordinating through the use of synchronization points during processing. The chart highlights the tracker data that is valid at the first synchronization point (at the end of display frame 1). The system then uses this input data to start the rendering and computation of the next interval. When this interval completes, the user can see a rendered display (display frame 3) that is has used the original tracker readings, but the display does

not have any updates that are dependent upon computing a new application state from the input. Display updates of this type are not available until display frame 4.

There are several ways to measure the update rate of the application. Three of the most commonly used methods are shown at the top of the chart. The first of these is the visual frame rate. This is simply a measure of how long it takes for the application to render the graphics of a single frame. The second measurement is the input latency. This measures the time from an input device update until the environment outputs new sensory information to the user based upon this input. The third measurement outlined is the latency when there is a computation that has to be performed upon the input data. Each of these measurements must be taken into account when measuring the performance of a VR application.

The amount of interactivity and engagement in an application depends on the response time of the application. If the application responds quickly to user input, then the user has a feeling that they are directly influencing the application. If the application responds slowly to the user's input, then the user loses the feeling of interactivity and instead they start using non-interactive interaction methods. Within a VR software system, these response times are measured in terms of frame rates and latencies.

Because interactivity is so important to system success, developers constantly strive to reduce system lag thus increasing perceived interactivity. The simplest way to detect lag in a system is to observe low rendering frame rate. Because of this, applications developers spend much time trying to increase the visual frame rate. Visual frame rate is not the only factor to decreasing lag though. There are many other factors that influence the user-loop frame rate. A development environment should help the developer to tune the system for a high frame rate.

Support for hardware

Modern hardware systems have many special features that a VR application can use to dramatically increase performance. However, explicitly making use of custom hardware can make applications hardware specific. If possible, a VR development environment needs to transparently take advantage of any special abilities of the hardware. If it is not possible to transparently use the features, the development environment should still allow the developer to use a direct interface on their own even if it will make the application platform specific.

VR development environments differ widely in the interfaces provided for creating an application. Some provide a very high-level view, where users create applications with custom scripting languages and graphical tools, and the system itself takes on most of the responsibility of

simulation, geometry, and interaction. Other interfaces float just above the hardware level, using well-known graphics APIs and programming languages to ensure the greatest performance. Often, the higher-level tools will enable faster development with a shallower learning curve. The other side of the argument is, "If you want something done right, do it yourself." The more one of these systems is willing to do by itself, the more likely it is that it will do something unwanted, or do it in a way that is not optimal for a particular application. The key to balancing this trade off is to make application development as easy as possible, but all the developer enough flexibility to use whatever optimizations they need.

Performance monitoring

Because performance is so critical to the success of a VR application, development environments need to provide a way to collect performance analysis information. In a VR application, there are many potential areas for performance problems; the tracking system may be running to slowly, the device updates may be taking to much processing overhead, the graphics may be taking longer than normal to render, the simulation code may be using too much processing time. Performance monitoring allows developers and end users at production sites to quickly zero in on the source of performance problems.

Developers can use the collected performance information to find bottlenecks in their application code. According to Barry Boehm, applications spend 80% of their time in 20% of the code [9]. This may actually be an understatement; Donald Knuth has found that less than 4% of a program accounts for 50% of run-time, and others have found that 90% of code accounts for 10% or run-time. This author has found the 90/10 rule to be an accurate estimate. In any case, the key to good application performance is finding the 10% that is performing badly and optimizing it. In a multi-threaded system, it becomes even more important to have support from the development environment for performance monitoring because many times a performance problem can be related to synchronization issues in a system. For example, the user may be executing simulation code while the rest of the system is waiting for the user thread. This application may be able to increase performance by moving the code to a parallel section of the application. Standard profiling techniques have difficulty showing this type of performance problem because the tools do not have the knowledge of the software system that the development environment does.

The performance information can also be used to find and alleviate performance problems with a specific VR system's configuration. Often small changes in configuration options can have a dramatic impact on the performance of applications in a given VR system. For example, it may be

possible to tweak the configuration of graphics windows in a way that achieves a higher visual frame rate. This type of configuration tuning can increase the performance of all applications that run in the local VR system.

Extensibility

Hardware abstraction

In order to be usable, the VR development environment must provide support for the physical hardware devices in the local VR system, but almost as vital is how well the toolkit abstracts away the details of the low-level hardware interfaces. Do devices of the same type share the same interface, or are there specific APIs for each one? This comes into play when replacing hardware. For example: If an application has been using tracking system A, but a newer, better tracking system B becomes available, will the application have to be modified to take advantage of it? Preferably, the environment will support this with a change in a script or configuration file, without requiring any application re-coding.

A well-designed hardware abstraction is very important. While a less generic interface might be better able to take advantage of a device's unusual features, the generic interface makes the application more portable and easier to upgrade, maintain, and understand. While major changes, such as replacing a joystick with a glove, might require rethinking the user interface, smaller changes, like switching one tracking system for another or changing between models of HMDs, should not require changes to the VR application itself.

Simple extension

The development environment should allow developers to easily extend the VR software system using simple programming interfaces. Users need to be able to extend the system when an application requires the use of customized interaction devices. For example, when creating a vehicle simulation application, it may be necessary to support a customized device that mimics the interaction methods of the vehicle being simulated. These devices commonly have non-standard hardware, which means the application developer needs to add a custom device driver to the VR system.

In addition to being easy to extend, developer should be able to add support for the custom devices without possessing expertise about the internals of the rest of the VR software system. Instead, they should only need to know about a small subset of the system that they can use to add device support in a straightforward manner. By allowing developers to easily extend the system, the development environment is able to change and adapt quickly to new advances in VR systems.

Do not require application changes

Extending the development environment's functionality should not require any changes or rebuilding of current applications; extensions should be transparent to current applications. Current applications should be able to take advantage of the new extensions simply by changing the configuration parameters of the system. The configuration system parameterizes the settings of the VR system that is being used. This also means that an application compiled for use at one location's VR system can be distributed to another location and run on the second VR system without requiring any changes to the application. Instead, the users only need to provide the configuration parameters for the new system.

An extendable parameterized VR system leads to applications that have longer lifetimes. I have personally experienced applications that have been running perfectly fine for years, but then the VR system changes and requires an update to the development system. This in turn requires all applications to be re-compiled. This is fine as long as the source code is available, someone knows how to compile it, and someone has the time, knowledge, and tools to do so. In any other case, this task can range from difficult to impossible. In a production setting, this problem is worsened by the fact that any downtime is costing the company. This problem can be avoided entirely by designing the VR development environment to allow extensions to be transparent to the applications.

Flexibility

Scalability

Scalability refers to the ability of a development environment to be able to run on a wide variety of VR systems. A VR development environment should provide the scalability to run applications efficiently on any type of VR system, be it a simulator on a desktop PC or a high-end VR system like a CAVE. Scalable systems have the benefit of allowing developers to write an application once and run it in any environment.

A scalable system also eases application development when VR system resources are scarce. Most sites only have one large scale VR system that all developers share. A scalable development environment allows applications to be development on a desktop PCs or small VR systems. This means that applications developers can write and debug their applications without requiring access to the high-end target VR system. This idea is explored further below when discussion rapid prototyping using a simulator.

Cross-platform

What happens if an application has twenty potential customers, but ten of them use Windows NT workstations the rest use Linux workstations, and the high-end VR system is running on Irix? Today's VR systems make use of a wide variety of system architectures. To be widely accepted, a successful VR development environment must offer support for not one, but many platforms.

A well-designed VR development should provide support from cross-platform development. It should hide platform-specific details well enough that porting an application between different platforms requires little or no change in the application. For toolkits that use their own data formats and scripting languages, it is often the case that no changes are necessary. On the other hand, toolkits where the developer writes his or her own code in a language like C++ are subject to all the usual headaches of porting source code.

Run-time changes

Most current virtual reality (VR) systems do not allow users to make run-time changes to modify initial settings. A user configures the system before running an application, and the configuration remains static for the duration of the application. For example, the initial settings specify how many projection surfaces to display and what type of tracking system to use. Each facet of the system is specific a priori. There is no way to modify these settings once the software system has started.

This is because, most VR control software relies upon having all configuration information when the application starts. If the user needs to change a system setting, they have to shutdown the running application, change the configuration parameters, and then restart the application with the new parameters. They may have to repeat these steps many times to get the system into the correct configuration.

Requiring static configurations limit the ability of a VR system to adapt and change to new requirements. A flexible reconfiguration system can provide benefits in many areas that make the system more flexible and robust. A brief overview of a few of these areas follows.

Setup

Run-time reconfiguration can prove invaluable when setting up and configuring a new system. Reconfiguration allows users to change device configuration parameters while an application is running. For example, users can use this ability to interactively tweak tracker settings at run-time. A person setting up a system can run a calibration program that draws a coordinate axes at the position of a tracker. By using this visual feedback, a user can interactively test whether the offset and

rotation parameters for the current tracking system have been configured correctly. This type of interactive testing is helpful in determining correct system settings.

Reconfiguration also permits display setting to change at run-time. This allows users to configure new projection surfaces while running test applications. For example, it is possible to interactively change the display settings of a projection environment at run-time.

First, the user specifies an initial system configuration that may include information such as the tracking system used and any other information that they believe to be correct. This initial configuration also includes speculated settings for the display surfaces in the environment. The user then starts a test application in order to try the settings. Once the application is running, the user can then interactively change the settings of the active display surfaces, and even add new displays or remove current ones. They can change parameters such as the size, location, and a variety of other projection parameters, all while observing the result of the changes.

Reconfigurable VR software systems also enable the use of advanced projection systems where the physical settings of the projection surface itself are changing at run-time. Examples of this type of system include desk-based systems with a movable projection surface and large scale CAVE-like devices that allow users to move the walls. Run-time reconfiguration allows users to reconfigure the desk projection surface while an application is running; or if a driver is available, the running system can actively monitor the desk's current settings and automatically update the projection parameters to reflect any changes.

Software and hardware testing

A reconfiguration VR system can also be very helpful when testing new software and hardware.

Reconfiguration reduces the turn around time for testing multiple configurations while developing applications. Many times VR applications take a lengthy amount of time to load due either to loading large models or the time required to start input devices. When debugging an application, there is no reason to bring the application up in a full VR environment every time the program starts. Instead, a developer can start the application in a simulated environment to quickly test the program. If the application is working in the simulated environment, then instead of restarting it with new configuration information, run-time reconfiguration allows a user to simply change the configuration so that it is running in the full VR system.

Developers can make use of this flexibility to test an application using many different VR system configurations. For example, it may be helpful to start the application first in a simulator, reconfigure it so that it is running with an HMD, and then reconfigure it again so that it is using a

CAVE or some other large-scale projection environment. Applications can be tested in all of these environments without ever halting execution.

Performance tuning

As touched upon above, performance tuning is very important in a VR software system. Run-time changes can be used to tune performance of VR applications. By analyzing the performance of a running application, and adjusting the current system configuration based upon these measurements, users can change a VR system configuration to achieve better performance. For example, device drivers could be moved to lightly loaded systems or have parameters changed in such a way that the driver requires less system resources. As another example, consider advance graphics architectures where it is possible to change the parameters of the graphics hardware. Users of run-time reconfiguration can exploit these abilities in order to find more optimal settings for the VR system's graphics hardware.

Application adaptations

Reconfiguration is not limited to only the VR system. VR applications can also take advantage of the abilities afforded by such a system. Since the VR system provides the infrastructure, it becomes much easier to write applications that are reconfigurable as well. Applications can allow for their parameters to be configured using the reconfiguration system.

Runtime reconfiguration be used to change application specific parameters such as models loaded or interaction methods used. For example, an application can be written where the user can remotely change the model that is being viewed in the VR environment and the navigation method that is being used. Because the run-time reconfiguration system is being used to make these application changes, they can come from any entity be it a remote controlling interface, another application that is running, or the local application it self.

In a VR system that allows for multiple simultaneous users in a single environment, run-time reconfiguration allows user to exchange control. An example would be an application that was not written with multiple users in mind. As such, the application would only expect to be controlled by one input device. Using run-time reconfiguration, the two users can exchange which user's interaction device is active in the system.

This same idea can be used to enable multiple tracked users to use an environment that only has support for a single tracked user at one time. In such an environment, the run-time reconfiguration system can be used to choose which tracked user is "active" and thus controlling the environment at any given time.

Support use of other application toolkits

A VR development environment should assist the user in creating the best VR application possible. As such, the development environment should allow the user to create the application using whatever tools are best suited for the problem domain of the application. If the user wants to create a scientific visualization application, then the development environment should allow them to use VTK or OpenDX [10][11]. If the developer wants to create a visual simulation application, the environment should allow the user to make use of the advanced features of Iris Performer. The environment should not restrict the developer by requiring the use of only one tool for all jobs because there will always be limits to what a single tool can do.

Several current VR development environments include support for graphics and/or simulation in the core of the software system. For example, a system may include a scene graph that is specific to the development environment. This works well when the user's application is suited for using the type of tools included in the development environment. The user can write the application using the integrated libraries and be assured that it will work on all platforms and with all hardware supported by the development environment. This can greatly eases the software development burden on the developer. However, when a user wants to create an application that is not supported well by the integrated tools, the development environment becomes restrictive.

What is needed is a VR development environment that easily supports a wide variety of other toolkits. The application developer can then choose which tool works best for the job and use it. The power and ease of use that comes from integrated tools does not need to be lost. It is still possible to create modules that have strong ties with many common toolkits in such a way that they work as well as a completely integrated tool would. However, by decoupling the development environment from a specific tool, it can be used in a wider variety of applications.

Do not be overly restrictive

A VR development environment should have no restrictions that prevent a skilled user from implementing an advanced solution. Developers should never hit a wall where the development environment restricts them from creating an application that works the way they have envisioned it. For example, many development environments that allow the use of OpenGL, do so by using draw callbacks that are called once per OpenGL context by the software system. In order to keep the applications simple and to prevent new users from writing non-portable code, many of these systems do not explicitly get the user access to the current context id. An advanced user may need to use this context id to interact with OpenGL directly. The software system should not prevent the application

from getting to this information. If it did, then it would be impossible for the user to interact directly with OpenGL.

While simplicity is valuable, the software should not be so restrictive as to prevent the implementation of advanced techniques. The environment should not require the use of an overly-restrictive program structure, nor should it place an impenetrable barrier between the developer and the computer system – there should be a way to go outside of the environment, and access the operating system or hardware directly, when that is required.

Simplicity

Short learning curve

A VR development environment makes use of many complex software concepts including: multi-processor programming, components, run-time loading, and run-time reconfiguration. A new developer should not have to know how to use these concepts to write simple applications that solve their problems. VR systems are complex, but application development does not need to be.

A VR development environment should provide a small and simple interface that makes basic functionality available. By using a simple interface, the environment does not require users to understand the entire. Instead, the developer only needs to understand the basics of a small portion of the system. To further ease learning, the development environment can provide sample applications and re-usable application components that developers can use to rapidly create new applications.

By simplifying the development environment, VR application development becomes more accessible to a wide number of users.

Rapid prototyping using simulation

Since most development groups only have access to a few VR systems, it is important to be able to run an application and interact with it without requiring access to the entire VR system. If developers have to wait for VR hardware, they will waste time waiting for their turn to use the equipment. Additionally, while debugging applications it is overly burdensome to use a full VR system. Many times, it is much faster and easier to debug applications on desktop machines. On a desktop machine, there is no worry about devices loading, projectors synchronizing, or other people needing to use the system. Instead, a developer can focus on writing the application and fixing any bugs that pop up.

To allow developers to quickly prototype an application, a development environment should include a simulator environment that accurately imitates a full VR system. This generally involves

drawing the display in a window on a monitor and using the keyboard and mouse to simulate head tracking and any other input devices. A correct simulator also must replicate the underlying system behavior of a full VR system. This includes accurately reproducing the program conditions that will occur in the full environment such as: multi-processing, shared memory, etc. The simulator must also model restrictions that the user will encounter in the real world VR system such as: restrictions on device ranges, collision with projection screens and walls, etc. The more accurate the simulator, the less time the developer has to spend making the application work in the physical VR system.

Robustness

Failure protection

Application developers need a robust platform to run VR applications. When running on a modern operating systems, a single process failing does not bring down the entire system. A VR system should behave in much the same way. A single component failure should not result in the entire VR system crashing. With a system as large and complex as a VR system, components are bound to fail or break. A tracker may be unstable, a cable may be loose, or the driver software may just be buggy. However, just because a single component has problems does not mean that it should affect the entire system.

The key to protecting the system from a single component failure is to keep the components of a system separated so that the interface between components can shield the components from failures. The interfaces can then provide a layer of protection between all the components. For example, the system may give a “smart” handle to an input device. The application uses this handle to reference a device instead of using a direct reference. In this way, the system can protect the resource by placing failure detection logic in the handle such that the handle will never return a reference to an invalid device. The application itself can protect itself in this way by using an application harness to separate the application from the VR software system.

Maintainability and correctness

VR systems are complex and as such, VR development environments are large and complex software systems. As with any large software system maintenance and correctness become important issues. How can we design a VR development environment so that is maintainable into the future? How do developers test for correctness in a large VR development environment?

Maintenance presents a problem in a large system because of inter dependencies within the software system. In a complex system, it is very difficult to completely understand every part of the

system. If system components are highly intertwined, changes in one component could affect the correctness of another component. These dependencies are often very difficult to predict and take into account.

Maintenance is simplified by breaking the system up into many separate self-contained code modules. Each of these modules presents a single interface that can be accessed by other modules in the system. By breaking the system up into small chunks, it is possible for system designers to make changes within a module without affecting code outside the module.

Testing correctness in a large system is important because you need some level of confidence that the system will work. A common way to manage this is to separate the major system components into separate entities that can be individually tested and verified. The idea is that if a system is made up of many components and each of these components individual works correctly then, the combined system will work correctly (assuming the linking code works correctly). To test such a system, the components should be tested at many levels of granularity. For example in an object-oriented system you may want to test at object, module, and sub-system levels.

CHAPTER 4 CURRENT DEVELOPMENT ENVIRONMENTS

The second stage of this research involved analyzing current VR development environments to build upon what has been learned in previous research.

This chapter gives an overview of many software toolkits that are currently used to write VR applications. For each package, a summary that describes the basic operations is given and is followed by a list of strengths and weaknesses of the software. These strengths and weaknesses are based on evaluating the software tool using the requirements outlined in the previous chapter as guidelines.

NOTE: These evaluations are based upon information that was available at the beginning of this research. As such, some of the information may have changed since then. In such cases that changes are known to have occurred, the corrections are made or a note is given in the write up.

Iris Performer

Summary

Iris Performer is a high performance graphics API for Silicon Graphics Inc. (SGI) machines. It is targeted at the real-time visual simulation market, but can be used to create very high performance VR applications. If you need to get every ounce of graphics performance out of an SGI machine, then Iris Performer is definitely worth consideration as a renderer or as the basis for a custom VR solution.

Availability

Performer is available from SGI.

Platform

Performer is available for SGI machines only.[2]

Supported VR hardware

Performer has no direct support for input devices other than mouse and keyboard. It also has no support for non-visual sensory output such as audio.

[2] Since the time of this report, Iris Performer has been release for Intel based Linux systems.

Description

First, it should be stated that Performer is not designed to be a VR development environment. Performer is targeted at creating high performance visual simulation applications. Performer is a C/C++ based graphics library produced by the advanced graphics division at SGI. As such, Performer is designed to maximize the performance of graphics applications on high-end SGI machines. It enables SGI developers to achieve peak performance with little or no intervention on the part of the user for many simple applications. Because of the ability of Performer to achieve peak performance on SGI machines, it is commonly used as the basis for custom VR libraries. For example, it is used as the primary rendering engine by Avocado and Lightning (see below for more information on these packages).

Performer is a scene graph based API. The scene graph holds a complete representation of all objects in the virtual world. This means that all geometric data within the scene is constructed from node objects. Performer provides a wide array of node objects that applications use to create a scene description. Connecting these nodes in a directed acyclic graph[3] forms a scene graph. Performer has basic scene graph nodes such as transformation nodes and geometry nodes, but it also supports nodes that allow for behavior that is more complex. Performer has nodes to support level of detail (LOD), animations, and morphing, to name a few.

To bring geometric data into an application, Performer provides a large number of database loaders that allow users to import scene models. Performer provides loaders for more than thirty database formats. The loaders are dynamically loaded as needed to convert the given file format into Performer's internal scene graph structure. Once in the internal scene graph, developers can manipulate all aspects of the geometric data through Performer's scene graph API.

Performer gives developers full control over scene graphs and the geometry contained within them. It is possible to manipulate data down to the vertex and polygon level. Performer also allows the addition of user-defined callback functions to the scene graph. This can be used to place custom rendering routines written in OpenGL into the scene graph. The combination of geometry nodes and callback functions allows developers to create any type of graphic effects that are necessary.

Performer has a high-performance rendering engine at its core, which has been engineered to achieve peak graphics performance for the entire line of SGI graphics architectures. Performer uses several strategies to optimize rendering performance. First, most rendering loops are specially tuned to send graphics commands to the rendering hardware in an optimal way. These routines are hand

[3] A data structure reflecting nodes and their relations to each other

tuned to ensure high throughput. In addition, state management routines track the current state of the renderer in order to minimize the number of costly state changes the graphics hardware is required to perform.

A major benefit of Performer for VR users is its ability to handle multiprocessing automatically. Performer uses a pipelined multiprocessing model to execute applications. The main rendering pipeline consists of an application stage, a cull stage, and a draw stage. The application stage updates the scene graph and normally executes any user code. The cull stage determines which parts of the scene are visible. Then the draw stage renders only the geometry that passed through the cull stage. Applications can have multiple rendering pipelines directed at multiple graphics hardware pipelines. Performer automatically multi-processes all stages of the rendering pipelines or the user can give Performer varying degrees of direction in choosing how to allocate system resources. A user can direct Performer to use a general method of process allocation, or a take direct control of allocating processor resources to the application.

In addition to using multiprocessing in the rendering pipeline, Performer also provides additional asynchronous processes. Performer provides an intersection process that can be used for collision detection, a compute process that can be used for general computations, and a database (dbase) process for handling database paging. All of these multiprocessing details are transparent to the user because Performer internally handles issues such as synchronization, data exclusion, and coherence.

Performer also has the ability to maintain a consistent frame rate while scene content and complexity are varying by making use of a number of special hardware and software features. Performer reduces the amount of geometry drawn by culling parts of the scene that are not visible. Performer also uses LOD nodes in the scene graph to choose between varying complexities of models to render. This allows less complex versions of an object to be rendered when the viewer is beyond certain thresholds. Both of these methods decrease the amount of geometry that needs to be sent to the graphics hardware. Another tool that Performer can use is dynamic video resolution (DVR). DVR is a feature of some advanced SGI graphics architectures that allows the system to dynamically change the size of the rendering area in the frame buffer. This area is then scaled to fit the graphics window the user sees. By using DVR, Performer applications can decrease their fill requirements.

Performer has window management routines that allow developers to use the advanced windowing capabilities of the SGI hardware. Performer allows multiple graphics pipelines, multiple windows per pipeline, multiple display channels per window, and dynamic video resolution. These

features allow programs to use all the capabilities of the underlying hardware. These features are a key ability needed when working on VR systems such as a CAVE.

Performer includes the ability to collect statistics on all parts of a Performer application. This data can then be used to find application bottlenecks and to tune the application for better performance. For example, exact timings of all pipeline stages can be monitored to determine which section of an application is taking the most time. Performer also tracks many draw statistics that can help developers tune applications. Performer tracks parameters such as the number of graphic state changes, the number of transformations, the number of triangles rendered, the size of triangle strips, and more.

Imagine for instance that an application has a frame rate of 24 Hz, but the target frame rate is 48 Hz. The user believes that it should be possible to maintain the higher frame rate, but does not know what is causing the slow down. By looking at Performer's statistics, it is possible to determine which pipeline stage is causing the slow down. After determining which stage is slow, it is then possible to find out if user code or internal Performer code is using the extra time. Performer statistics allow developers to quickly zero in on the area of an application that is lagging behind. Due to the real-time constraints of VR applications, capabilities like these are needed to maintain the high frame rates required for VR applications.

Strengths

- Performance: In nearly all cases, Performer will result in a very high performance graphics application.
- File Loaders: Performer can load many popular file formats. The loaders preserve the model hierarchy to allow users to manipulate the scene data.
- Visual Simulation Features: Performer has many visual simulation features that are invaluable for developing VR applications with a visual simulation focus.

Limitations

- Not designed for VR: Performer is not a VR development environment, but can be used as the basis for very powerful custom solutions.
- Not Cross platform: Performer only runs on SGI machines.[2]

- VR Display Devices: Performer has no direct support for VR display devices. Application developers have to write the routines for computing viewing frustums, etc.

- VR Input Devices: Performer has no support for VR input devices. Users must write device drivers for input devices.

Alice

Summary

Alice is a rapid prototyping system for creating interactive computer graphics applications. Alice is designed as a tool to allow people without technical backgrounds to create interactive 3D applications.

Availability

Alice is freely available at http://www.cs.virginia.edu/~alice/

In order to have support for VR devices, an internal developer version is necessary.[4]

Platform

Alice requires Microsoft Windows 95/98/NT/2000 running Direct X with Direct3D.

Supported VR hardware

The freely available Windows version of Alice only uses the mouse and keyboard. Internal versions support HMDs, gloves, and other VR devices.[4]

Description

The Alice system is designed to enable rapid development and prototyping of interactive graphics applications. VR software development usually consists of many "what if" questions. "What if we scale the model?", "What if we rotate faster?", "What if we move through the environment using this new path?" These are all examples of questions that normally require re-coding and re-compiling. Rapid prototyping systems such as Alice allow for all the "what if"s to be quickly evaluated. By making slight changes in the script, an endless number of ideas and options

[4] Since the time of this report, there has been some information posted on the Alice web site talking about using Alice with some VR devices. The code given is not supported and only supports a limited number of devices, some of which are custom to the Alice development laboratory.

can be tried in a very small amount of time. Rapid prototyping can greatly cut the development time of VR applications.

In addition to rapid development, Alice is designed to provide non-technical users with the ability to write VR programs. This means that the development interface and language must be simple and easy to learn and understand. With this in mind, the developers of Alice chose Python as the language for writing Alice scripts. Python is a high-level, interpreted, object-oriented language. It allows novice users to write Alice scripts easily.

```
FishBoat.Move(Forward, 2)
Camera.PointAt(Shark, EachFrame)
Shark.Move(Forward, Speed = 0.5)

GoSharkAnim = DoInOrder (
Shark.PointAt(FishBoat),
Shark.Move(Up, 0.25, Duration =1),
Shark.Turn(Up, 45, Duration=0.5),
…
Ground.SetColor(Red, Duration = 3),
Shark.Turn(Right, Speed=.05)
```

Figure 8: Typical Alice script

As can be seen in the example script (see Figure 8), the scripting language is very readable. Just by looking at the script, it is possible to understand what it does. By using an easy to read and understand scripting language, Alice maintains a very short learning curve. Combining an easily comprehendible scripting language with a simple graphical user interface (GUI) development environment, Alice makes it possible for novices to easily write working scripts. By making the script easy for non-experts to use, Alice brings technology to those whom would not ordinarily be able to use it.

Alice organizes the world as a hierarchical collection of objects. An interesting feature of the Alice hierarchy is that parent/child relationships can change at run-time and it is easy to switch between multiple coordinate systems. Any object can be referred to based on another object's local coordinate system. The ability to switch coordinate systems gives the application developer a large amount of power as object transformations can be specified relative to any other objects in the hierarchy. For example, in Alice it is possible to say "Translate object 'Tree Leaf' relative to object 'Ground'" instead of having to base the translation off the hierarchy local to the 'Tree Leaf' geometry node.

In order to maintain high frame rates, the Alice system decouples simulation from rendering. Alice separates the application's computation process from the application's rendering process. The first process computes the simulation's state, and the second process maintains the geometric data and renders it from the current view position. The separation allows the rendering process to execute as fast as possible because it does not have to wait for the simulation's calculations to complete. It should be noted that this separation of processing is completely transparent to the programmer. The programmer writes a single-threaded sequential application, and the Alice system takes care of the multi-processing details.

Strengths

- Rapid Prototyping: Alice is designed from the ground up with rapid prototyping in mind. It succeeds at making rapid prototyping easy and powerful. The interpreted scripting language makes it possible to easily test many scenarios very quickly.

- Easy to learn: Alice targets non-technical users. Because of this, the product is very simple to learn and use. The scripting language (Python) is simple yet powerful. The GUI development environment is clear and easy to use as well.

Limitations

- VR Devices: Creation of VR applications requires an internal developer version that includes support for VR devices

- Application Limitations: Places limitations on the types of VR applications that users can develop. Alice allows the rapid construction of very simple applications. However, it is very difficult to create complex applications that are more typical of the types of VR applications that industry currently uses.

CAVE Library

Summary

The CAVE Library was originally created by Carolina Cruz-Neira at the University of Illinois at Chicago's Electronic Visualization Laboratory (EVL) [3][17]. It provides a low-level API for creating VR applications for projection-based systems.

Availability

The CAVE Library is now commercially available from VRCO. For information, consult their home page at http://www.vrco.com/ .

Platform

The CAVE Library is only available for SGI computers.[5]

Supported VR hardware

The CAVE Library was initially designed to support the CAVE, a multiple-screen VR projection system. Support has been added for desk-based projection systems and HMDs.

The CAVE Library provides support for a wide variety of devices. It also supports the trackd API.

Description

The CAVE Library is a set of function calls for writing VR applications in C or C++. It is a low-level library – it handles setup and initialization of processes, and provides access to various input devices. It does not include higher-level features like collision detection or built-in support for object behaviors. The standard version of the library makes use of OpenGL for graphics rendering.

A running CAVE Library application is composed of several processes for handling devices, networking, and so on. Most importantly, the system creates a display process for each physical display. The CAVE Library allows the display processes to be split up between two machines, a master and a slave.

The major part of an application is a set of callback functions written by the developer. For example, the developer can define a frame callback that is called by one of the graphics processes immediately before rendering each frame. This can be used for querying user input and updating program data. After this, each display process calls a display callback. The library sets up the viewing parameters for the display and user head position, so the callback is usually just a set of OpenGL drawing commands to render the scene. After the library calls the display callback, it synchronizes the display processes for each screen and then swaps the display buffers. Several other callback functions can be defined for display or application initialization.

The CAVE Library has support for networking applications built into it. Three callback functions are defined explicitly for networking purposes, being called upon the addition or removal of

[5] Since the time of this analysis, vrco has released a beta version of the CAVE library that runs on Linux based systems.

a user and on receipt of data sent by a remote CAVE Library application. The CAVE Library automatically transmits user and tracker information between all connected CAVEs, but the application is responsible for transmitting whatever other information needs to be shared by the application (a function exists to transmit data to the remote applications). Note that the set of remote machines that the application can send to is specified in the CAVE Library's configuration file, and cannot be changed once the application has begun.

In addition to the standard OpenGL version of the library, versions of the CAVE Library are available with support for SGI's Iris Performer and Inventor software. Additionally, a version of the library designed for display and interaction with VRML models has been announced.

Limitations

- Cross-platform Support: The CAVE Library is not a completely cross-platform solution. It is limited to SGI systems, and is heavily oriented toward projection systems such as the CAVE.

- Distributed Applications: Support for load-balanced distributed applications is limited, as all input devices must be connected to the master machine, and the slave is only used for rendering.

- Shared Memory Issues: The CAVE library forces the user to deal with shared memory issues in order to create even non-distributed applications that will run on multiple screens.

- Extensibility: The CAVE library was not designed as a long-term solution for VR development. As such, its APIs are often difficult to extend in backwards-compatible ways.

Strengths

- Acceptance: The CAVE library has been in use for many years and has gained a wide base of users and acceptance within the VR community.

Avango

Summary

Avango is a VR development environment created at GMD (German National Research Center for Information Technology) [18]. It is based on Iris Performer and therefore only runs on SGI platforms. Avango greatly extends Iris Performer's scene graph objects to allow for multi-sensory VR application development. It has a scripting language (Scheme) that allows for rapid prototyping of applications.

Availability

Avango is available to research institutes for non-commercial usage for a small license fee.

Platform

Avango is available only on SGI machines.

Supported VR hardware

Avango supports the CyberStage CAVE, the Responsive Workbench (RWB), and user workstations. (Other devices may also be supported)

Description

Avango is a VR software system developed by GMD to be a framework for their VR applications. The main goal of the library is to integrate the wide variety of VR devices used at GMD and to be highly extensible. The system is also designed to allow rapid prototyping for quick development and testing of applications. In addition, Avango supports the development of distributed applications.

Avango's scene graph structure is based on Iris Performer (see description of Iris Performer Section 3.1). As a result of using Performer, Avango can only run on SGI machines. In order to fully represent a virtual world, Avango must extend Performer's scene graph structure. Performer defines only the visual characteristics of the environment. Avango extends (more accurately, sub-classes) Performer's scene graph nodes to create the Avango scene graph objects. These new objects have added features that enable Avango's advanced capabilities. Not every Avango node has to be based on a corresponding Performer node. For example, since Performer has no support for sound, Avango extends the scene graph to allow sound nodes.

Avango uses an object-oriented scene graph structure to represent the virtual world. The scene graph is a directed acyclic graph, a data structure reflecting nodes and their relations to each other.

Everything in the world is represented as node objects whose state is manipulated in order to change the virtual world. The representation is a complete representation, meaning that all the possible sensory outputs are represented in the same scene graph. This is important because it means that not only are the visual aspects of the environment represented, but also the auditory and tactile. In order to present the environment to the user, each sensory channel has a separate renderer that traverses the scene graph.

Every Avango object encapsulates its internal state in fields. Avango defines a uniform public interface to access field data, so that all objects can be manipulated in a common way. This allows the implementation of a scripting interface, persistence, distribution, and run-time loading of new objects.

Avango fields can be connected to one another creating data flow networks, that is if field A is connected from field B, field A will receive field B's value whenever field B changes. The ability to interconnect fields can remove much of the VR application's programming burden. The data flow network allows nodes to have their attributes "linked" to the attributes of other nodes and objects in the system. This ability allows Avango to define very complex behaviors very easily through connected networks of objects.

In addition to nodes, Avango provides two other types of objects: sensors and services. Sensors contain field data but are not derived from Performer classes. Sensors are used to import and export data between Avango and the rest of the system. They are not visible to any sensory channel, so therefore they are not part of the scene graph. Sensors can be used for objects such as display windows, device data, etc. Avango also provides service objects. Service objects provide an API to system features. They can be used to implement things like device drivers. Sensor objects can use device service objects to get device data. This device data is then maintained in the sensor object's fields where it may be referenced by nodes in the scene graph.

An Avango application can be seen as a collection of node groups that encapsulate some specific behavior. The node groups can be looked at as tools that the application developers have at their disposal. In addition to groups of nodes, Avango can be extended with entirely new nodes, sensors, and services to create new tools. Some examples of tools that have been developed for Avango are explosion nodes, video texture nodes, pick nodes, dragger nodes, and intersection services. It is easy to create new groups of nodes to create new tools.

All relevant parts of the Avango system are mapped to a scripting language, Scheme. This allows Avango applications to be interpreted at run-time. The scripting language also eliminates the

need to recompile an application when changes are needed. This greatly speeds the development process by allowing rapid prototyping of applications. New algorithms can be tried immediately.

Avango supports distributed environments by transparently distributing all scene graph nodes. This is done by sharing the nodes' field data between the different client applications viewing the shared environment. Object creation and deletion is also shared transparently between all browsers. This allows applications to be developed where many users can navigate through a single shared virtual environment. In many VR libraries, writing applications like this can be difficult if not impossible. However, because of the way Avango uses a single scene graph to store everything, the library makes distributing environments relatively simple. User interaction could be handled by maintaining scene graph nodes that correspond to each user in the environment.

Strengths

- Scripting: The inclusion of a scripting language allows rapid prototyping of applications.
- Fields: Data flow network allows very powerful applications to be easily created. In addition, the ability to have sensors as part of the data flow network greatly simplifies input processing
- Extensibility: The node, sensor, and service objects are very easy to extend. Because every object has a uniform API, once a new class is created it is very easy to begin using it within the system.

Limitations

- Cross Platform: Because Avango is based on Iris Performer, it only runs on SGI platforms

Lightning

Summary

Lightning is an object-oriented system for creating VR environments that is designed to support development with multiple programming languages [19][20].

Source

Lightning is under development at Fraunhofer Institute for Industrial Engineering.

Platform

Lightning is currently implemented for Silicon Graphics computers.

Supported VR hardware

For an immersive experience, Lightning supports projection screens, the BOOM, and several head-mounted displays. Tracking support includes 2D mouse, BOOM tracking, the BG Systems Flybox, the Division Flying-Joystick, the Polhemus, Inc., Fastrak, and Ascension Technologies Flock of Birds.

Description

The part of Lightning that developers actually interact with is an object pool. The objects in this pool are of various general kinds – sensor objects for trackers and other inputs, simulation objects that control visual or audio output, behavior objects to control interactions, and so on. The developer writes an application by creating a set of these objects, sometimes extending objects to create new ones.

Different objects can be written in different languages, and then these diverse objects can be combined in a single application. For example, a behavior object could be written in Scheme, and communicate with a tracker object written in C++. Most of the work by the Lightning developers so far has been in Tcl.

The run-time system for a Lightning application is a set of managers that control the various objects and perform the duties of the VR system. For example, a Device Manager controls all the sensor objects.

Output is controlled by various Render Managers – "render," in this case, used in a very general sense. For example, the Audio Render Manager renders audio objects. A Visual Render Manager exists based on SGI's Performer software. The system is designed so that it should be possible to create a new Visual Render Manager based on another graphics API, though this has not yet been implemented.

One interesting feature of the Lightning application structure lies in its dynamic support for multiprocessing. The objects in the object pool are formed into a directed graph. For example, a particular sensor object feeds data into a behavior object, which in turn controls several visual objects. Lightning includes a Link Manager which attempts to divide the processing for all objects into multiple processes while preserving the order of operations that affect one another. This is done without any special effort on the part of the developer.

Strengths

- Multiple Language Support – Since Lightning is designed to allow modules written in different languages to work together, developers can use whichever supported language that they know best, or that best supports the concepts they are trying to code.

Limitations

- No Distributed Application Support – Despite the Lightning developers' interest in making an effective system for multiprocessing environments, their reference papers fail to mention any support for distributed operation. It appears that all the processes of a Lightning application must execute on the same computer.

MR Toolkit

Summary

MR (Minimal Reality) Toolkit is a toolkit in the classic sense – that is, it is a library of functions called from within an application. Its design emphasizes the decoupling of simulation and computation processes from the display and interaction processes. Several higher-level tools have been built on top of it for object creation and behavior scripting; some of these are also discussed [21] [22] [23][24][25].

Availability

The MR Toolkit is a creation of the University of Alberta's Computer Graphics Research Group. Licenses are available at no cost to academic and research institutions.

Platform

Version 1.5 of MR Toolkit is available for numerous UNIX systems, including those from Hewlett Packard, SGI, and IBM. Parts of the Toolkit have been ported to Sun and DEC Alpha-based machines. The developers' stated plans are to make version 2.0 available for SGI and HP-UX machines only. Windows users may also be interested in the newly-released MRObjects, a new C++ based development environment.

Supported VR hardware

MR Toolkit supports numerous VR-specific devices. A variety of popular trackers from Ascension Technologies and Polhemus are supported, as well as several space balls and 3D mice. A Motif-based tracker simulator is also included. Other supported input devices include the VPL DataGlove and the Virtual Technologies CyberGlove.

For output, MR Toolkit supports many different HMD devices, such as the VPL EyePhone 1, Virtual Research Flight Helmet and EyeGen 3, the General Reality CyberEye, and Virtual I/O I.Glasses.

Description

A basic MR Toolkit application can be written in C, C++, or FORTRAN. Calls to the MR Toolkit API are made to configure the application and start various processes.

There are several different kinds of processes in an MR Toolkit application. There is one "master process", which controls all the others and performs all rendering done on the main machine. A "server process" is created for each I/O device, such as trackers or sound output devices. Simulation and other computation-intensive tasks are segregated into "computation processes." The goal of the MR Toolkit process design is to let these potentially time-consuming simulation processes run without interfering with the performance of the display processes. As a proof-of-concept, the MR Toolkit design team built a fluid dynamics simulation. Response to user input and head movement and graphical updates were kept to a very acceptable 20 Hz, even though the simulation process could only update the fluid data twice per second.

MR Toolkit has some built-in support for distributed processing. A slave process can be created on another machine to perform additional rendering. For example, the left eye image for an HMD could be rendered by the master process running on the main machine, while the right eye image is rendered by the slave process on another workstation. Server processes (and their associated hardware) can also be distributed across machines. TCP/IP is used for communication and synchronization between the master process and the servers, but the MR Toolkit API hides this detail from the application writer.

A program using MR Toolkit is divided into two parts: the *configuration* section and the *computation* section. The configuration section initializes MR and declares slave and computation processes and shared data items. Shared data is used for inter-process communication; the shared data items can be of any non-pointer C data type. Finally, the program makes procedure calls to specify and start up the devices to be used.

The computation section of the program comes next. The main part of this section is the interaction loop for the master process. In this loop, the master process checks any computation processes for new output and examines the data from each input device. Any new commands for the computation process are issued (for example, if the user changes something in the environment). Finally, the master process and any slaves draw new images for the user.

MR Toolkit currently supports several graphics systems, including PHIGS and OpenGL. However, the developers' stated plans are that version 2.0 of MR Toolkit, when released, will support only OpenGL and Pex5. An application skeleton for interfacing with SGI's Performer software also exists, and simple VR viewers for some 3D file formats have been written.

By itself, MR Toolkit is a fairly low level tool; it does not have built-in support for collision detection or multi-user applications, for example, and using it requires writing source code in C++ or FORTRAN. However, MR Toolkit's designers meant it to be a tool on which more powerful development systems could be built, and several projects have already been written to enhance its capabilities.

The *MR Toolkit Peer Package* provides the ability to connect two or more MR applications at remote sites using the User Datagram Protocol (UDP). The master processes for each application can exchange device data, as well as application-specific information defined by the developer.

The *Object Modeling Language* (OML) is a procedural language designed for defining object geometry and behavior, including animations and interactions. An object in OML includes geometry and material information for drawing an object and defines behaviors that can be called in response to events. The OML parser's geometry engine can perform collision detection and object culling.

JDCAD+ [24] is a solid modeling tool which can output OML code. A 3D tracker can be used to create, distort, and chain together primitive shapes. JDCAD+ includes a key frame animation facility, letting the user create OML animations without writing any code.

The *Environment Manager* (EM) [25] is an MR Toolkit application written in C which allows a developer to create a virtual environment from a collection of OML data and a text-based description file without writing and compiling new code. It supports advanced multi-user abilities using the Peer Package's networking capabilities. EM provides a very high-level way to create VR applications on top of MR Toolkit.

The University of Alberta recently released a related system, MRObjects, an initial version of which is available for Windows 95 and NT. MRObjects is an object-oriented framework for building VR and other 3D applications in C++. It is also designed to support multi-user environments and

content distribution through the web. As of April 1998, this was only a preliminary release, and in particular was still lacking stereoscopic graphics support and other important features. While it looks promising, it is still too early to recommend for use in a production setting.

Strengths

- Flexibility: Dealing with the base MR Toolkit and one of the supported graphics libraries gives the developer a very close-to-the-hardware environment for creating applications. The packages built on top of MR Toolkit allow the easy and fast creation of VR applications. MR Toolkit has proven itself to be a useful package on which more advanced authoring tools can be built.

- Performance Measurement: MR Toolkit includes built-in support for performance measurement. Timing support in the toolkit includes the ability to attach time stamps to key points in the application and to quantify the time spent in inter-process communications.

Limitations

- Low-end Basic System: Most of the limitations of MR Toolkit are simply because of features omitted in favor of the low-level approach of the basic Toolkit, and are remedied by using one of the higher-end tools like the Environment Manager.

- Support for Projection Systems: While MR Toolkit supports a wide variety of hardware, its hardware support lists make no direct references to supporting projection-based VR. The emphasis of MR Toolkit's developers seems to have been very much on HMDs for display devices.

World Toolkit (WTK)

Summary

WTK is a standard VR library with a large user community. It can be used to write many types of VR applications. Although other products may have better implementations of specific features needed for VR, WTK is one of the few packages that has an answer for the entire gamut of needs [4].

Availability

WTK is a commercial VR development environment available from EAI/Sense8 Corporation.

Platform

WTK is a cross-platform environment. It is available on many platforms, including SGI, Intel, Sun, HP, DEC, PowerPC, and Evans and Sutherland.

Supported VR hardware

WTK supports a large range of devices. A full up-to-date listing is available at their web site [26].

Description

WTK is a VR library written in C (C++ wrappers are available). To create a virtual world, the developer must write C/C++ code that uses the WTK API. WTK manages the details of reading sensor input, rendering scene geometry, and loading databases. An application developer only needs to worry about manipulating the simulation and changing the WTK scene graph based on user inputs.

The WTK library is based on object-orient concepts even though it is written in C and has no inheritance or dynamic binding. WTK functions are ordered into 20 classes. These classes include: Universe (manages all other objects), Geometries, Nodes, Viewpoints, Windows, Lights, Sensors, Paths, and Motion Links. WTK provides functions for collision detection, dynamic geometry, object behavior, and loading geometry.

WTK geometry is based on a scene graph hierarchy. The scene graph specifies how the application is rendered and allows for performance optimization. The scene graph allows features such as object culling, level of detail (LOD) switching, and object grouping, to name just a few.

WTK provides functions that allow loading of many database formats into WTK. It includes loaders for many popular data file formats. WTK also allows the user to edit the scene graph by hand if that level of control is needed. Developers can create geometry on the vertex and polygon levels or they can use primitives that WTK provides such as spheres, cones, cylinders, and 3D text. It is of note that when WTK loads in a data file all geometry is put into one node. The data file is not converted into the internal scene graph structure. This means that WTK does not allow the user to manipulate the geometry within their files once loaded. A developer can only manipulate a single node that holds all the geometry.

WTK provides cross-platform support for 3D and stereo sound. The sound API provides the ability for 3D spatialization, Doppler shifts, volume and roll-off, and other effects.

The basis for all WTK simulations is the Universe. The Universe contains all objects that appear in the simulation. It is possible to have multiple scene graphs in an application, but it is only possible

to have one Universe in an application. When new objects are created, the WTK simulation manager automatically manages them.

The core of a WTK application is the simulation loop. Once the simulation loop is started, every part of the simulation occurs in the Universe. The simulation loop looks like this:

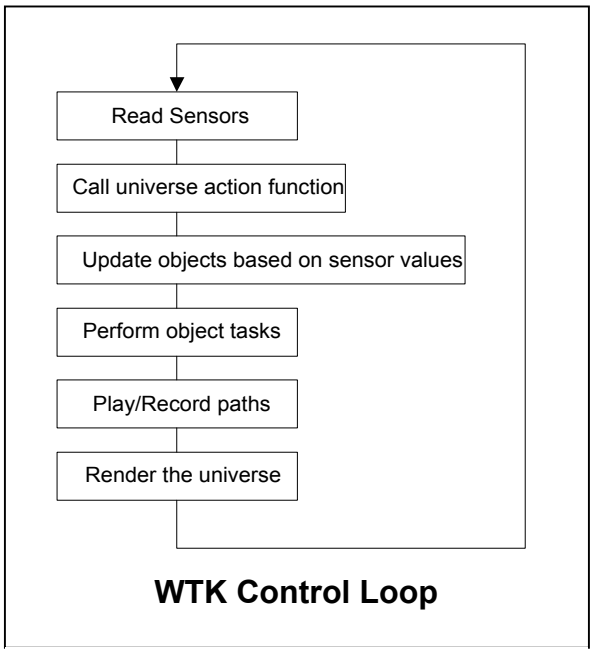

Figure 9: Worldtoolkit main loop

NOTE: The order of the simulation loop can be changed via a WTK function call.

The universe action function is a user-defined function that is called each time through the simulation loop. This function is where the application can execute the simulation and change the virtual environment accordingly. Examples of things that can be done include: changing geometry properties, manipulating objects, detecting collision, or stopping the simulation.

WTK sensor objects return position and orientation data from the real world. WTK allows sensors to control the motion of other objects in the simulation. WTK has two major categories of sensors it can deal with: relative and absolute. Relative sensors report only changes in position and rotation. Absolute sensors report values that correspond to a specific position and orientation.

WTK allows users to treat these two categories of sensors identically by using a common interface. The simulation loop takes care of updating all sensor data and dealing with what category of data the sensor is returning.

The WTK interface allows sensor pointers to be used nearly interchangeably in an application. But when creating a new sensor object, the type of sensor being used must be specified in the function call. This means that when given a sensor pointer, retrieving data from a sensor is identical regardless of the type of sensor. However, in order to get a sensor pointer, the user must specify the type of device that they would like to use. If the user wants to use a different type of sensor, the application code has to be changed and re-compiled. This leads to applications that are not completely portable across differing VR hardware. It is worth noting that this problem can be avoided if the user writes code to control sensors using a configuration file for the application.

WTK supports the creation of paths. A WTK path is a list of position and orientation values. These paths can be used to control viewpoints or transform objects in the scene graph. WTK provides the ability to record, save, load, and play paths. There is also support for smoothing rough paths using interpolation.

WTK support motion links that connect a source of position and orientation data with some target that is transformed based on the information from the source. The source of a motion link can be a sensor or a path. Valid targets include viewpoints, transform nodes, node paths, or a movable node. It is also possible to add constraints to motion links in order to restrict the degrees of freedom.

In an attempt to extend the object-oriented feel of WTK and to allow for multi-user simulations, WTK Release 8 includes a new Object/Property/Event architecture. This architecture has three key capabilities: all objects have properties that are accessed through a common interface, property changes trigger an event that can be handled by an event handler, and properties can be shared across multi-user simulations using World2World. The new interface also allows users to add user-defined properties to objects. These properties can then be treated exactly like all other properties in the system.

The new Object/Property/Event architecture can help simplify many common programming tasks. By using the event-based structure, data updates can be propagated through the system. For example, if a sensor value is modified, the event-handler can automatically modify any objects that rely upon that sensor value. This event-based architecture can greatly simplify programming burden.

In order to provide for multi-user distributed environments using WTK, Sense8 provides a product called World2World. World2World is a server for WTK that distributes the property information about each object. Because World2World distributes object properties, it will only work with applications that use the Object/Property/Event architecture of WTK.

World2World works by allowing client applications to connect to the World2World server. The server then acts as the central distributor of data updates for the simulation. The server controls the system by distributing the properties of objects that are specified as shared. Whenever a shared object has a property changed, an event is automatically generated that will distribute that data to the World2World server and from there on to the other clients.

Strengths

- Well Established: WTK is a well-established development environment with a large user base.
- Cross Platform: WTK has solid cross platform support.
- Multi-Pipe Support: The SGI version supports multi-pipe applications. This allows WTK to control CAVEs and similar devices.
- Device Drivers: WTK has a vast library of device drivers. WTK supports nearly every device on the market.

Limitations

- Device Abstraction: Application code has to be changed and recompiled to change the type of sensors it uses.
- Performance: WTK does not perform as well as some other VR libraries, most notably the libraries based upon Iris Performer.

Analysis of previous work

This overview provides a great deal of insight into current VR development environments. Unfortunately, none of the current VR development environments seems to provide for all the requirements specified Chapter 3. Although many of the systems work very well and provide innovative solutions to many of the requirements, each system has weak points as well. There is simply no tool that solves all the problems outlined in Chapter 3.

This analysis has revealed that there is a need for a standard VR development environment that is open and extendable. The current systems do provide a reference for what has and has not worked in the past. It is possible to learn a great deal from these tools about how to create a VR development

environment that will solve the problems the VR community faces today and hopefully be able to work well into the future. The remainder of this section highlights some of the key insights gained from the analysis of these packages. Notice that some of these insights reflect the requirements outlined in Chapter 3.

Performance is of utmost importance

The importance of application performance cannot be reiterated to often. It is the key for a successful VR development environment. No matter how good the development environment, if it performs poorly it will not be well received and will not be able to create good applications. VR developers know that performance is critical, and because of this, they will frequently use development environments that fail to satisfy other requirements solely so that they can get every bit of performance from their VR systems. This needs to be taken into account when developing a VR development environment because it shows just how important performance is for user acceptance.

Rapid prototyping makes development easier

When developing a VR application, user need to frequently test many aspects of the application. Application developers generally develop applications in many small incremental steps. With VR applications, this incremental style of development seems to be even more prevalent; this is mostly likely due to the visual aspect of the application. Developers may make a change to the graphics and need to see if it worked, or they may make a slight change to the user interface and need to try the new functionality. Because they are making many changes, they need instant feedback to test the new changes. In many cases, the users do not have the time or the need to use a full VR system. Since they are only making incremental changes, they frequently only have to test a small portion of the application, they do not need access to complete system. They simply need a simulated VR environment that accurate models a full-scale VR system and allows them to develop applications quickly and easily.

Alice and the CAVE library are examples of how current VR systems provide for rapid prototyping. By using an interpreted language (Python[27]), Alice is able to provide instantaneous feedback to developers as soon as a single line of code it changed. Alice provides an entire environment for rapid prototyping of running applications that goes far beyond any of the other packages. This is an example of what application development should be like in the future.

The CAVE library on the other hand provides a simulated VR environment that runs on desktop machines. This allows developers to rapidly test their applications by running their applications in a "simulator" that emulates the basic VR input devices. One failure of the CAVE library simulator is

that it is does not provide a completely accurate simulation of the VR software system that is running in a large-scale VR system. Because of this, it is possible to write applications that perform perfectly in the simulator, but do not run at all in a full-scale environment.

Do not tie the environment to a specific graphics API

If the development environment is tied to a specific graphics API or to a custom graphics API, then it does not allows users to make use of the best tools for the job. Several of the environments evaluated make use of a specific graphics APIs as the base for the system. By doing so, they limit the development environment greatly.

The first limitation is that the development environment can only run on platforms supported by that graphics API. This is a very commonly seen problem in development environments that are based on Iris Performer. Since Iris Performer only supports the SGI and Linux platforms, these development environments will not run on other platforms.

A second limitation is that developers are required to use only the given graphics API. This limits users because not all applications can be easily written in all graphics APIs. For example, it is much simpler to write a scientific visualization application using the Visualization Toolkit (VTK)[10] than it is to use a scene graph API.

Environments need wide range of robust open device drivers

A VR development environment is only as good as its device drivers. When using a VR development environment, there is nothing as frustrating as finding out that the software system does not support a piece of hardware that your local VR system relies upon. Whether it is a tracking system, an instrumented glove, or a custom device that only exists at your site, if it is not supported then there is no way to use it in a VR system. Device drivers are the foundation of any VR system.

Almost more frustrating that finding that your device is not supported is finding that the device driver is buggy and unstable. VR devices are notorious at being very unstable and a buggy driver only makes that worse. Successfully VR development environment provide solid drivers for VR devices and provide the source code for the drivers so that users can correct any bugs encountered.

Development environments with open source driver libraries help alleviate the problem of non-existent or buggy drivers. If the system has close sourced device drivers, then it is impossible for users to provide new drivers for un-supported or experimental devices. It is also equally impossible for users to fix bugs in drivers that they already have. However, if the development environment has an open sourced driver library, then users can add new drivers and fix old drivers as they need.

Monolithic architectures present problems

Monolithic architectures have trouble with flexibility and extensibility. While monolithic architectures can provide high-performance and simplicity, they have difficulty supporting many aspects of extensibility and flexibility. This is because many modularity of the system is key to extensibility and flexibility.

Monolithic architectures also tend to have many internal dependencies that can make maintenance difficult.

In the next sections, we discuss how the lessons learned from other systems have been built upon to create a new open extendable system called VR Juggler.

CHAPTER 5 THE ARCHITECTURE OF VR JUGGLER

The previous chapters have dealt exclusively with understanding the problems of VR development environments and analyzing the requirements of such a system. This chapter builds upon this analysis and outlines the design of VR Juggler. The next two chapters discuss the implementation details of this design.

The design phase of VR Juggler set out to achieve the following design goals:[28]

- Define major sub systems
- Specify major interfaces between sub-systems
- Design a well engineered architecture
- Establish a blueprint for implementation

In the first phase of the design of VR Juggler, we decomposed the system into its major sub-systems based upon the requirements laid out in the previous chapters. The first two of these major sub-systems are the microkernel and the virtual platform, and they are described in the remainder of this chapter.

VR Juggler microkernel core

Based upon the requirements that the system must be extensible and flexible, the first problem the VR Juggler team set about to solve was how to design the core of a system that could support the needs of such a dynamic system. The system would need to allow evolve as it grew and needed to support the ability to add new functionality and make changes to existing services without affecting the entire system. The design also needed to have ingrained support for its own modification and extension.

It was decided that a specialization of the microkernel architecture [29] would be the best solution to this problem.

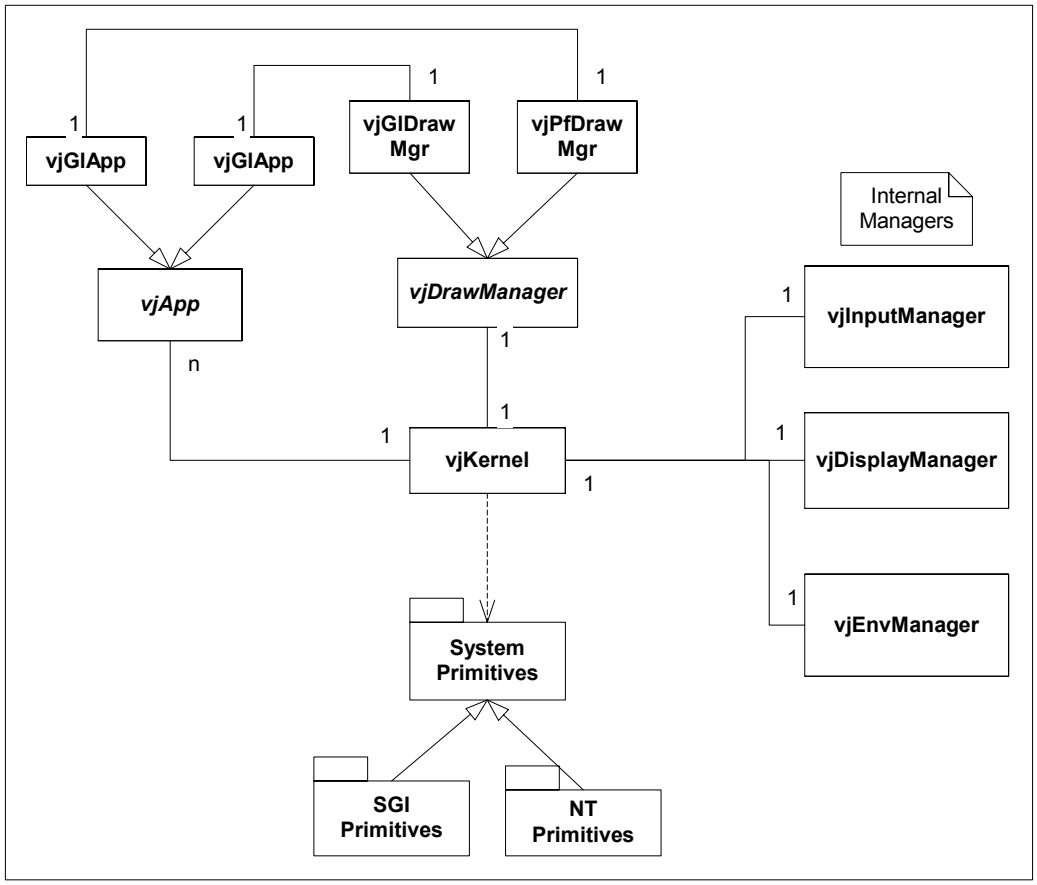

Figure 10: Microkernel architecture

The microkernel controls the entire run-time system and manages all communication within the system. The VR Juggler microkernel architecture (Figure 10) has a core kernel object that implements the central services needed for VR application development: input devices, display settings, and configuration information.

Internal managers implement core functionality that the kernel cannot easily handle. If a core service would unduly increase the size or complexity of the kernel, the kernel uses an internal manager to provide the service. Internal managers can also be used to group logic functionality together in a single sub-unit. By grouping common functionality together, it allows the kernel to manage the features as a single group. There are internal managers to handle input devices, display settings, configuration information, and to communicate with the external applications.

External managers provide an interface to the system that is specific to the application type. Client applications communicate with the VR Juggler system through the interfaces of the external managers and of the kernel. Currently the only external managers are the graphics API specific draw managers. These draw managers give applications a view specific to a graphics API. The system can

be easily extended to add other types of external managers such as sound systems. The Juggler external managers are primarily used to provide an interface to external software tools that applications need to share

The VR Juggler kernel is a modular architecture that allows managers to be added, removed, and reconfigured at run-time. The kernel only loads the modules that running applications currently require. This modularization of the kernel helps to prevent the system from becoming monolithic. It also give the kernel a high degree of robustness because the kernel can execute with any combination of modules.

The kernel has references to each active manager in the system. By changing the references at run-time, the kernel can alter the behavior of the system. When the references are null, then the kernel simply ignores that functionality. Although the kernel can execute without any other managers, it must be connected to managers in order to render a virtual environment.

Mediator

The kernel in VR Juggler acts as a mediator by encapsulating how all the other components in the system interact. The kernel frame controls how the managers interact with the system. It can do this because only the kernel knows about the managers; there are no direct dependencies between the managers themselves. The kernel can change the way the system frame executes by simply changing the timing of calls it makes to the managers. It can do this without disrupting their normal behavior.

Many of the managers are active objects [31] that are kept synchronized by the kernel. The kernel maintains control because all the managers require the kernel to signal them during the stages of their processing. Because of this, the main kernel thread can control the timing of all the other active objects in the system. The managers and application only get processing time when the kernel allocates it either by calling a method of the class or by signaling the active object's thread to continue processing.

Capturing the interaction between the managers decreases coupling since the managers only know about the kernel. This means that the kernel can change the way the managers interact with each other without requiring changes in the implementation of the managers. It also means that the managers can change independently of the interactions. This independence is an aid to development because of the flexibility it gives to the design. If new capabilities are needed, it is only necessary to add a new internal or external manager. Changes to one part of the system, such as the addition of a draw manager for a new graphics API, have no effect on the rest of kernel.

Kernel portability

The VR Juggler kernel is layered on top of a set of low-level primitives that ease porting and allow for performance tuning on each hardware platform. The primitives control process management, synchronization, and other hardware-dependent issues (Figure 10). Because these primitive classes account for the majority of hardware-specific implementation differences, they can greatly ease the porting of VR Juggler to other architectures. During the porting process, each low-level primitive is extended and optimized in order to achieve high performance on each system.

Consequences

A microkernel design has several important consequences to the design as discussed in [29]

Benefits

Portability: Porting the microkernel to a new platform only requires modifying the hardware depended components. In a microkernel design, the dependencies are captured in a small subset of the system.

Flexibility and Extensibility: "One of the biggest strengths of the Microkernel system is its flexibility and extensibility." To add new features to the kernel, a new internal manager is added. To add a new system interface or support a new external API, a new external manager is added to the system.

Costs

Complexity of design and implementation: "Developing a microkernel-based system is a non-trivial task."

VR Juggler virtual platform

The microkernel architecture of VR Juggler's design allows for a great deal of flexibility, but the VR Juggler team realized that the microkernel architecture could be used to get even greater benefits. Because of this, the VR Juggler team designed the VR Juggler virtual platform (JVP). The purpose of the JVP is to separate the hardware-dependent and hardware-independent software components of the VR software system. The virtual platform provides a simple operating environment for virtual reality application development. The virtual platform is independent of hardware architecture, operating system (OS), and available VR hardware. By using the JVP, a developer can write an application once on a local VR system and run it on any other VR system.

The basic JVP system (Figure 11) is composed of an application object, a draw manager, and the VR Juggler kernel. Each of these systems and their relationship will be explained in detail in the next two chapters.

Specifically, the JVP design has the following characteristics.

Virtual platform API

The interface between the application object and the JVP consists of the kernel interface that provides the hardware abstraction for the virtual platform, and the draw manager that provides the abstraction for the graphics API (Figure 11).

The JVP kernel interface provides all application accessible functionality except for graphics API specific features. The kernel itself is responsible for controlling all components in the VR Juggler system. Because the kernel controls all the other VR Juggler components, its interface provides the virtual platform API for the hardware-specific details of the environment. Because the kernel interface is the only way the application accesses the hardware, it is possible to change the implementation details of any component of VR Juggler as long as the kernel interface remains the same.

The draw manager provides the virtual platform interface for specific graphics APIs. The kernel does not depend upon any graphics API specific details; it captures all of these in the draw manager, which is an external manager of the VR Juggler kernel. Applications use the draw manager to access any API specific details that are needed.

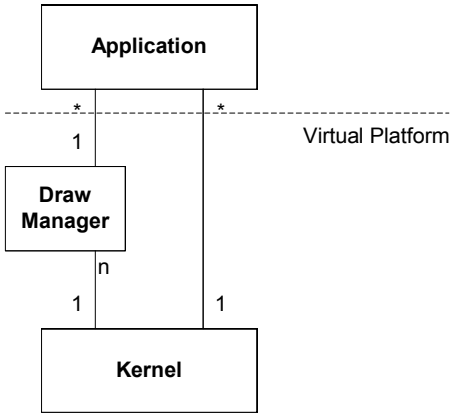

Figure 11: Application/VP interface

The virtual platform interface means that application code does not have to change when new system features are added or even when running on a different VR system, thus satisfying the code

change requirement from Chapter 3. The virtual platform in VR Juggler separates the application developer from the system details that can change. Since the virtual platform consists of the kernel and the draw manager, we have the freedom to change any details of the VR Juggler system as long as the interfaces to the draw manager and the kernel remain the same. As long as the interface looks and behaves the same, the application can never see or rely upon any details hidden by the virtual platform. This provides VR Juggler applications system independence. Once a VR Juggler application is written for one system, it can run with VR Juggler on any other system.

Architecture and OS independence

The JVP allows development of applications that are free from architecture and OS dependence leading to truly cross-platform applications. This allows applications written with the virtual platform to run on architecture that the JVP has been ported to. It also allows the virtual platform to be tuned for each local platform in order to run VR Juggler applications with high-performance.

Freedom from architecture and OS allow application development on any available hardware. It is possible to develop applications on low-end PC systems without sacrificing any functionality. The application will still run on high-end systems, but access to the high-end system and VR hardware is not required during development. Developing on low-end systems cuts costs and allows for easier application development.

Device abstraction

The JVP provides standard abstractions for many classes of VR devices. VR devices can be separated into several primary classes of input: positional, digital, boolean, analog, glove, and gesture. By using these common abstractions for device classes, the virtual software system hides the details of the actual devices in use. The application can make use of these device abstractions to get data from the devices.

The JVP eliminates the need for direct ties between the application and the hardware by separating the application and the device in use. Applications written using the virtual platform only need to use a device handle to an input device. A device handle has an associated device class and returns data of that class type. The input data can come from any available device that is of the needed class type or can simulate that class type.

The VR Juggler system defines base class interfaces for each device class in order to make all devices of a given class look the same to the developer. All input devices in VR Juggler implement a base interface for each class of VR device data it can return. For example, a simulated positional

device driver has the same interface as a magnetic tracker. Since VR Juggler only interacts with devices through this interface, application developers are not tied to specific hardware devices.

In addition to using the abstract interface for each class of input, VR Juggler applications refer to devices by using a handle to a named device. Whenever an application requests a device's current data, it uses a handle to the device. This level of indirection allows VR Juggler to change the physical device that the handle refers to without disturbing the running application. For example, a magnetic tracker being used by the application can be replaced with a simulated tracker without interrupting any running applications.

Operating environment

The JVP provides a simple operating environment for VR applications. This operating environment allows for multiple running applications and components. Each running application is an object under the control of the JVP that shares resources and processing time with other applications currently executing.

Benefits of an operating environment

The JVP operating environment allows smooth transitions among applications. Similar to a desktop environment, it is not necessary to restart the entire system each time you want to switch applications. You do not need to stop the running device drivers, or reconfigure the environment each time you want to switch to a new application. When a user wants to run a new application, that application is passed to the virtual platform, which then begins communicating with the new application.

An operating environment also results in less down time. Often in a VR environment, it takes longer to load the devices and to get the software system configured than it does to load the applications. It is also common to have VR devices that have difficulty starting up. Once a device is running with the virtual platform, it keeps running even when applications are added and other changes are made to the running environment.

The JVP allows environments to have many small highly specialized applications running that manage essential common issues. For example, on a desktop machine, there is not just one program that handles everything the user needs to do. There are many individual programs such as web browsers, e-mail reader, program launchers, chat clients, text editors, etc. To accomplish productive work, a user makes us of all these applications running at the same time within a single environment. VR Juggler allows this same type of multi-program productivity with VR applications in a virtual environment. There could be one tool that allows launching new applications, another tool may allow

the user to receive communications from people in other virtual environments, while yet another tool takes verbal notes about the data in running applications. There is a wide range of utility applications that can be helpful to users in a virtual environment .

Allowing multiple applications to run simultaneously within a VR environment opens many new possibilities. It is possible to have a suite of applications that run together or in turn to facilitate the manipulation of a common data source. For example, an architectural walk-through application could switch to a modeling program when the user wants to edit an element in the model. The JVP allows this application switch to happen transparently.

An operating environment also opens up the possibility of using plug-in applications to edit pieces of data imbedded in other applications. This is similar to the way word processors allow documents to contain imbedded information from other applications. While editing the document, users can use the other applications to edit that portion of data. By allowing multiple applications to run simultaneously, the virtual platform permits VR applications to edit data in much the same way.

Allow for use of multiple graphics APIs

In addition to freeing the application from hardware dependencies, a virtual platform must allow developers to use any graphics API they choose. This satisfies the requirements of not tying the environment to a single graphics API and also allowing the developer to use whatever tools are best suited for the job.

The JVP supports multiple graphics APIs by encapsulating all graphics API specific behavior in draw managers. Because the kernel represents only the part of the virtual platform that hides system details, we must add an additional interface to the virtual platform that is specific for each supported graphics API. Each draw manager controls the details of writing an application for its specific graphics API. The draw manager's interface represents an entirely different virtual platform interface that presents the application with an API-specific abstraction.

The kernel and the draw managers are used in parallel to create applications that are both independent of hardware and that make use of features of the graphics API used by the application. When we refer to the virtual platform in the rest of this writing, we will be referring to this combination of the kernel virtual platform interface and the draw manager virtual platform interface unless specifically noted.

CHAPTER 6 IMPLEMENTATION OF APPLICATIONS

This chapter builds upon the design outlined in the previous chapter by describing the application object sub-system and how application objects fit into the VR Juggler development environment.

The chapter starts by describing the rationale for making the application an object. Then it describes how an application object is defined and discusses some benefits of application objects. The chapter finishes with a description of how a user would create a VR Juggler application object.

Application object

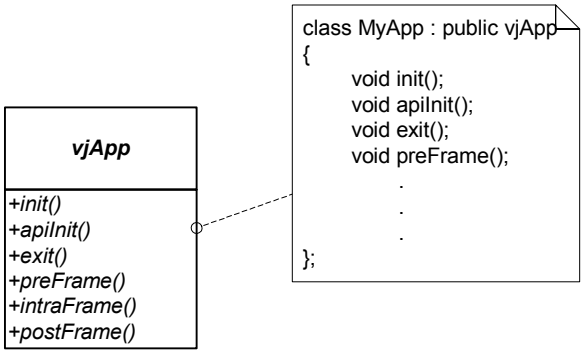

Figure 12: Application object

In VR Juggler, user applications are objects (see Figure 12). The VR Juggler system uses the application object to create the VR environment in which the user interacts. The application object implements interfaces needed by the virtual platform to create the virtual environment. (An interface is a collection of operations used to specify a service of a class or a component .) The kernel maintains control over the environment and calls the methods defined in the application interface. When the kernel calls the application's methods, it gives up control to the application object so the application can execute the code needed to create the virtual environment.

Base application interfaces

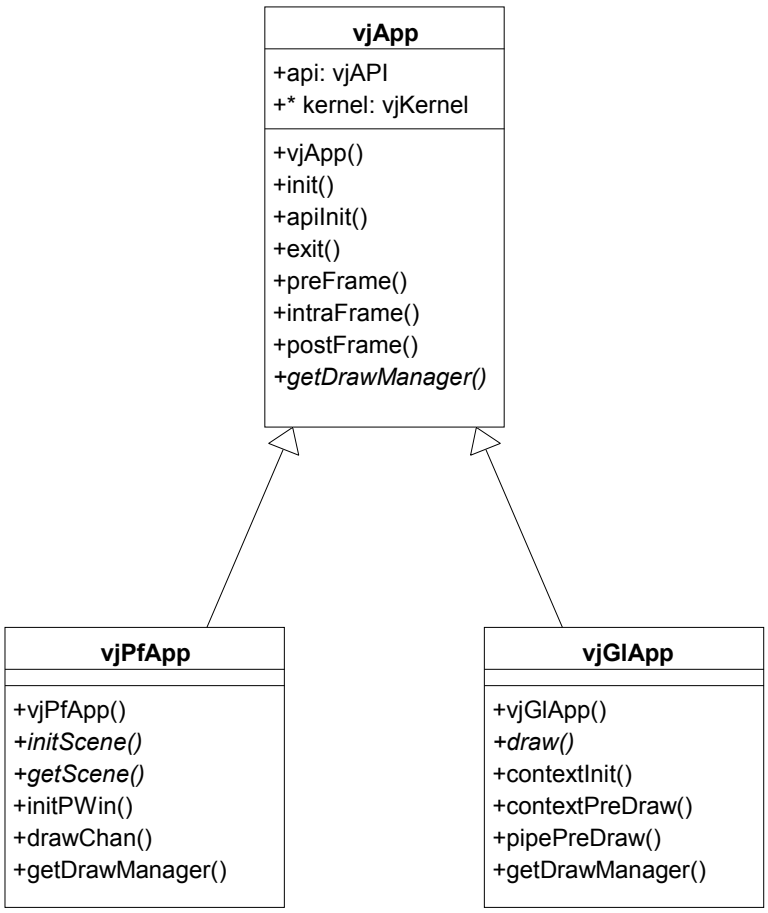

Figure 13: Application class hierarchy

The VR Juggler kernel and draw manager communicate with an application through pre-defined interfaces that all applications must implement (see Figure 13). When the kernel and draw manager need the application to do processing or to return information, they call the member functions of the interface. VR Juggler includes base classes for the interfaces needed for the kernel and draw manager interaction with the application. An application inherits from and extends these base classes in order to realize the required interfaces.

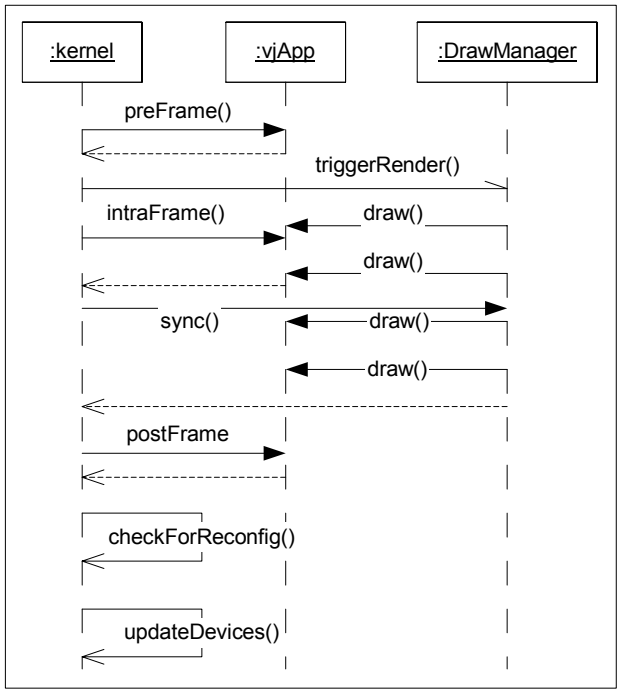

Figure 14: Kernel frame

The kernel calls each of the member functions based on a strictly scheduled frame of execution. During the frame of execution, the kernel calls the application methods and performs internal updates (see `updateDevices()` in Figure 14). Because the kernel has complete control over the frame, it can make changes at predefined "safe" times when the application is not doing any processing (see `checkForReconfig()` in Figure 14). During these "safe" times, the kernel can change the virtual platform configuration as long as the interface remains the same.

The frame of execution also serves as a framework for the application. The application can expect that when `preFrame()` is called, the devices have just been updated for this frame. Applications can rely upon the system being in well-defined stages of the frame when the kernel executes its methods.

No main() – Don't call me, I'll call you.

There is no `main()` function. Since VR Juggler applications are objects, developers do not write a `main()` function. Instead, developers create an application object that implement a set of pre-defined interfaces.

In common programs, the main function signals the point where the thread of control enters the application. After the main function is called, the application starts performing any application

processing necessary. When the OS starts the program, it gives the main function some processing time. Then after the process's quantum expires, the operating system switches to another process.

In VR Juggler, we accomplish the same functionality. The kernel is the scheduler, and it allocates processing time to an application by invoking the methods of the application object. The difference is that the kernel is stricter about when the application gets processing time because it has extra knowledge about how the application works.

Application objects can exist either linked in with the kernel startup code or alone as a dynamically loadable object. In this paper we will only talk about compiling application objects with the startup code to produce a standalone executable. However, it is possible to do some exciting things with dynamically loadable application objects which are beyond the scope of this paper.

Benefits of application object

VR Juggler uses application objects to allow for the flexibility of the system. The most common approach for VR application development is to have the application define the main function and have the application call library functions when needed. The library in this model only executes code when requested to do so by the application because the application is in control of the main thread of execution. In VR Juggler, we do not use this model because the system needs to maintain more control to have the flexibility needed to make changes to the virtual platform at run-time.

Flexibility works because the kernel is in control of each frame of execution. Because the kernel always knows the current state of the system, it can make changes at run-time. If the application was not an object, but was instead a program that was in control of the kernel, then the kernel would not have so much flexibility. The kernel would not be able to know what the application was doing and as a result, it would be possible for the application to rely on something in a way the kernel did not expect.

Since the kernel controls each execution frame, it is simpler for the system to change at run-time because the kernel knows when it is safe to make changes to the virtual platform. The VR Juggler system allows nearly every parameter to change at run time. It is possible to change applications, start new devices, reconfigure devices, and send reconfiguration information to the application object. The ability to modify the system's behavior at run-time is one of the major strengths of VR Juggler, and it is enabled because the application is an object with a standard public interface.

Application objects lead to a robust architecture as a result of low coupling and well defined inter-object dependencies. The application interface defines the only communication path between the application and the virtual platform. By restricting interactions to the interfaces of the kernel,

draw manager, and application, the system restricts object inter-dependencies to those few interfaces. This decreased coupling allows changes in the system to stay local. Changes to one object will not affect another unless the change involves a change of the interface of one of the objects. This leads to more robust and extensible code.

Because the application is simply an object, it is possible to dynamically load and unload applications at run-time. When the virtual platform starts up, it waits for an application to be passed to it. When the application is given to the VR Juggler kernel at run-time, the kernel performs a few initialization steps, and then executes the application.

Since applications use a distinct interface to communicate with the virtual platform, changes to the implementation of the virtual platform do not affect the application. This makes it simple to make significant changes to the implementation of the virtual platform without affecting any applications that currently run on the platform. These changes could include bug fixes, performance tuning, new device support, or any number of other changes.

```
class userOglApp : public vjGlApp
{
public:
    wandApp(vjKernel* kern)
      : vjGlApp(kern)
    {;}

    // -- Kernel interface -- //
    virtual void init();

    virtual void preFrame();
    virtual void intraFrame();
    virtual void postFrame();

    // -- OpenGL Mgr interface -- //
    virtual void draw();
    virtual void contextInit();
    virtual void contextPreDraw();

    //: Draw a box at the end of the wand
    void myDraw();
    void initGLState();
};
```

Figure 15: Sample application object

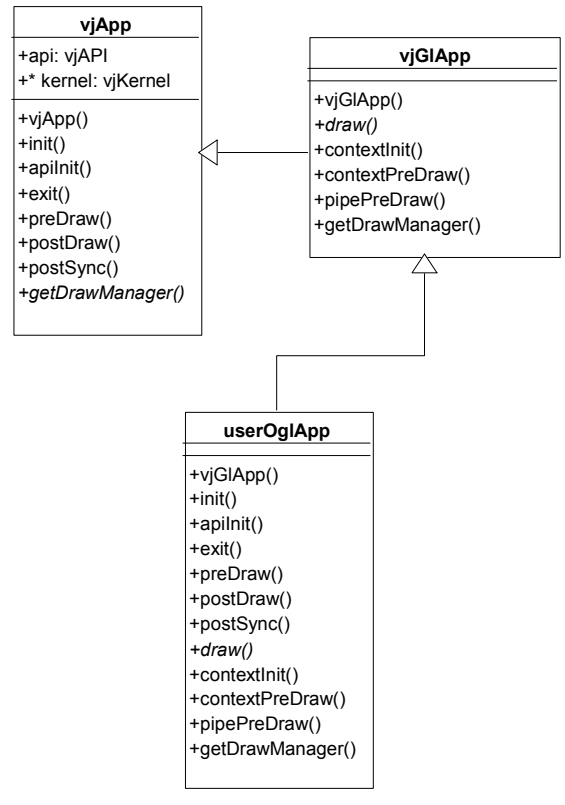

Figure 16: User app base classes

How to write an application

Now we will show how to create a simple application in VR Juggler. The example will be an OpenGL application designed to simulate an object at the position of a tracked wand. First, we will show how to define the application object. Next, we will describe the way to do drawing in the application and how the application accomplishes other processing. We will finish by showing how to get input from VR devices.

Please note that we use the example of an OpenGL application here, but VR Juggler has support for other graphics APIs. An OpenGL example was chosen because many people have experience with this graphics system, and the API specific interface is simple and easily understandable.

Derive from base class interfaces

The first step in defining an application object is to derive from the base classes that define the kernel and draw manager interfaces the application needs to implement. There is a base class for the interface that the kernel expects and a base class for each draw manager interface (see).

The kernel interface specifies methods for initialization, shutdown, and to give the application processing time.

The draw manager interface for this application specifies the functions that are necessary to render and OpenGL application. The interface has functions for drawing the scene and for initializing context-specific information.

The system expects all applications to implement this set of methods that the kernel uses to interact with the application.

Define drawing methods

The OpenGL draw manager handles all the details of the OpenGL rendering. It opens windows, creates contexts, and sets up viewing parameters. Before executing the application drawing method, the draw manager sets up the viewing system and the OpenGL context. The drawing functions only needs to execute the OpenGL drawing code necessary to draw the environment that the application represents.

OpenGL draw manager interface methods

The `apiInit()` member is called when the graphics API has been initialized. It is used to initialize any data members that cannot be initialized until the graphics API has been initialized. In OpenGL applications, there is no concept of initializing the API, so it is not normally used.

The draw manager calls the `draw()` function when the system needs to render a view of the virtual environment. This method is called with the correct OpenGL context for the current view window and with the model view and projection matrices setup to correctly render the current view. This method may be called multiple times per frame by the system, and may be called by several processes in parallel.

The `contextInit()` function is called when a new display is added to the system. When a new display is added, a new OpenGL context is created. This method is called by the system immediately after the new context has been created. This function can be used to initialize any context-specific information needed by the application, such as display lists and texture objects.

The `contextPreDraw()` function is called once per frame for each OpenGL context. It is used to generate or update any context-specific information at run-time. For example, if an application needs to add a new texture, this function can be used to generate the texture object.

Define processing methods

The base application interface functions define methods that are used for initialization and application computation (see `vjApp in` Figure 15). The kernel requires that all applications implement this basic interface so the kernel can allocate processing time to the applications. The kernel thread calls all kernel interface methods.

Base application interface methods

The `init()` member is called by the kernel to initialize any application data. When the kernel gets ready to startup a new application, it first calls the `init()` function to signal to the application that it is about to be executed.

The `preFrame()` member is called when the system is about to trigger drawing. This is the time to do any last minute updates of data based on input device status. It is best to avoid doing any time-consuming computation in this function because the time use in this function contributes to the overall device latency in the system. The devices will not be resampled before rendering begins.

The `intraFrame()` member is called after rendering has been triggered but before the rendering has necessarily finished. The code in this member will be executing in parallel with the rendering function. This is the function to put any processing that can be done for the next frame in order to make use of the time that is being used for drawing. By putting processing in this function, it can increase frame rates because the application can parallelize drawing and computation. Special care needs to be taken to ensure that the data is buffered to prevent it from changing while it is being used to render.

The `postFrame()` member is called after rendering has completed but before the trackers have been updated. This is a good place to do any data updates that are not dependent upon input devices, but cannot be overlapped with the rendering process.

Get input from system

VR Juggler maintains a set of named devices. Each of these devices is running in their own thread and is executing asynchronously from the rest of the system. Once each frame, the kernel updates a local copy of the input data for applications to use. The access devices, application make use of a device handle to a named device.

Every time an application receives device data, it must use this handle. As shown in Figure 17, the `mWand` data member is a device handle. It is initialized by passing the name of the device it references to the `init()` method. Then to use the device in an application, the `getData()` method is called which returns a reference to the type of data that the device supports. In this case, `mWand->getData()` returns a reference to a `vjMatrix` because the "VJWand" is a positional device.

Because all devices are accessed through device handles, VR Juggler has flexibility in how the devices in the system are handled and how the device data is returned. The device a handle refers to can be changed at run-time. This change can occur without the application ever knowing it took place.

```
class userOglApp : public vjGlApp
{
...

// Initialize devices
virtual void init()
{ mWand.init("VJWand"); }

//: Draw a box at the end of the wand
virtual void draw()
{
    // -- Draw box on wand --- //
    vjMatrix* wandMatrix;
    wandMatrix = mWand->GetData();

    glPushMatrix();
        glMultMatrixf(wandMatrix->getFloatPtr());
        drawCube();
    glPopMatrix();
}

public:
    vjPosInterface mWand; // the Wand
};
```

Figure 17: Input interface

How does everything get started?

```
...
// Start the kernel
vjKernel* kernel = vjKernel::instance();
kernel->start();
.
.
.
// Instantiate application.  Set application
wandApp* application = new wandApp(kernel);
kernel->setApplication(application);
...
```

Figure 18: Kernel startup

The VR Juggler system is started separately from the actually application. To load the system, a boot loader process instantiates the kernel and gives it a new thread to start running. The kernel then initializes the system and waits for an application to be handed to it or for configuration data is passed to it.

An application is given to the kernel at a later time through the kernel interface. Once the kernel has an application, it begins executing application methods within the kernel execution frame. The application can be given to the kernel as a dynamic object or as an object allocate in the kernel boot loader code. In the case that object is allocate in the loader, a common `main()` function can hold both the kernel start code and the code to give the kernel the application.

CHAPTER 7 DETAILED DESIGN OF VR JUGGLER

Now we will describe VR Juggler in more detail. We will start by discussing the microkernel architecture used in VR Juggler that was first introduces in Chapter 5 when describing the main sub-systems. That is followed with a description of the internal managers that maintain the input devices, store the display information, and interact with the external GUI control system used to reconfigure the architecture. Next, we describe the external draw manager and the application class that connects to it. We will conclude with a brief discussion of several features of VR Juggler and their implementation.

Microkernel

VR Juggler's virtual platform is based on a specialization of the microkernel architectural pattern.

The microkernel controls the entire run-time system and manages all communication within the system. The VR Juggler microkernel architecture (Figure 19) has a core kernel object that implements the central services needed for VR application development: input devices, display settings, and configuration information.

Internal managers implement core functionality that is not easily handled in the kernel object. If a core service would unduly increase the size or complexity of the kernel, the kernel uses an internal manager to provide the service. There are internal managers to handle input devices, display settings, configuration information, and to communicate with the external applications.

External managers provide an interface to the virtual platform that is specific to the application type. Client applications communicate with the VR Juggler system through the interfaces of the external managers and of the kernel. Currently the only external managers are the graphics API specific draw managers. The draw managers give applications a view specific to a graphics API.

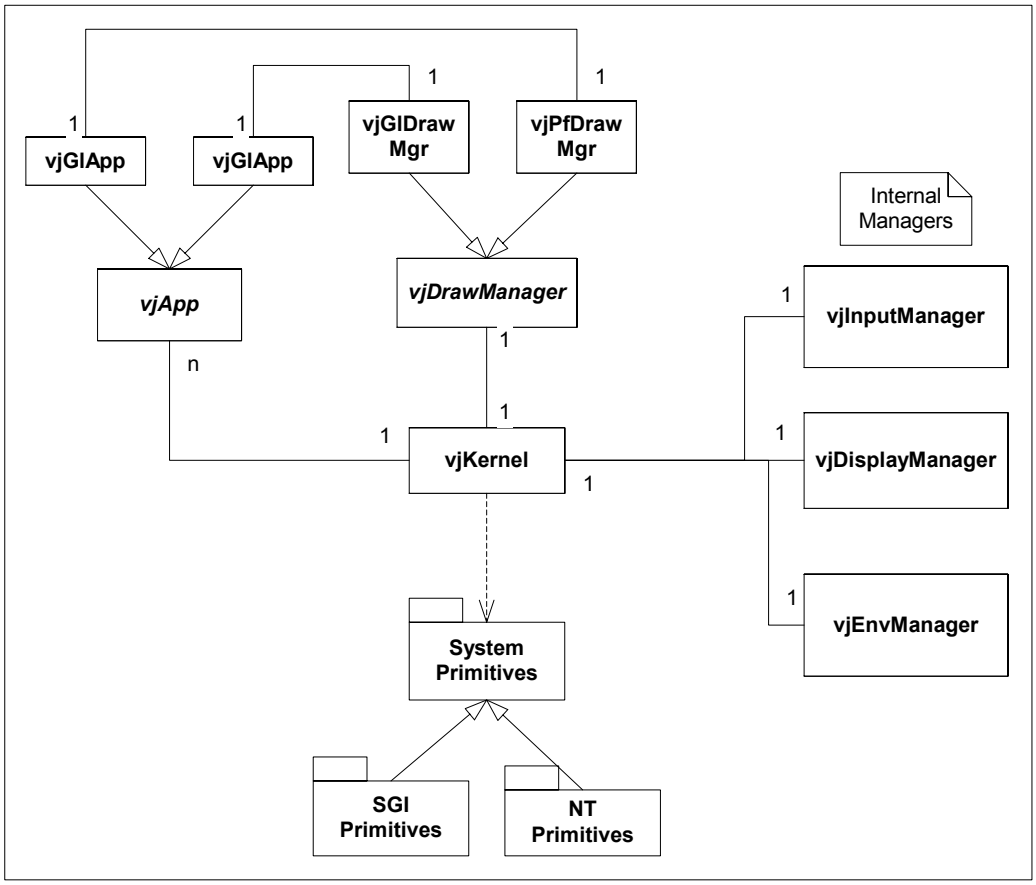

Figure 19: Microkernel architecture

VR Juggler allows managers to be added, removed, and reconfigured at run-time. The kernel has references to each active manager in the system. By changing the references at run-time, the kernel can alter the behavior of the virtual platform. When the references are null, then the kernel simply ignores that functionality. Although the kernel can execute without any other managers, it must be connected to managers in order to render a virtual environment.

Mediator

Many of the managers are active objects that are kept synchronized by the kernel. The kernel maintains control because all the managers require the kernel to signal them during the stages of their processing. Within the kernel frame, the kernel controls the timing of all the other active objects in the system. The managers and application only get processing time when the kernel allocates it either by calling a method of the class or by signaling the active object's thread to continue processing.

The kernel in VR Juggler acts as a mediator by encapsulating how all the other components in the system interact. The kernel frame controls how the managers interact with the system. There are

no direct dependencies between the managers. The kernel can change the way the system frame executes without changing the way the managers behave or relate to each other.

Capturing the interaction between the managers decreases coupling because it prevents managers from referring directly to each other. This means that the kernel can change the way the managers interact without requiring changes in the implementation of the managers. It also means that the managers can change independently of the interactions.

The individual managers are loosely coupled and very independent, because the kernel controls all interaction within the system. This independence is an aid to development because of the flexibility it gives to the design. If new capabilities are needed, it is only necessary to add a new internal or external manager. Changes to one part of the system, such as the addition of a draw manager for a new graphics API, have no effect on the rest of VR Juggler.

Kernel portability

The VR Juggler kernel is layered on top of a set of low-level primitives that ease porting and allow for performance tuning on each hardware platform. The primitives control process management, synchronization, and other hardware-dependent issues (Figure 19). Because these primitive classes account for the majority of hardware-specific implementation differences, they ease the porting of VR Juggler to other architectures. While porting, each low-level primitive is extended and optimized in order to achieve high performance on each system.

Configuration information

All configuration information is contained within small units called "chunks", and these chunks are divided into properties. Each property has a type and one or more values. A chunk contains all the configuration information for a particular part of the VR system or application. For example, the chunk given in Figure 20 defines configuration information for a window. It has three properties: name, size, and origin. The size property has two values of type int.

Chunk: Window		
Name	String	
Size	Int	Int
Origin	Int	Int

Figure 20: Window chunk

VR Juggler uses chunks to set configuration options for the components of the system. There are chunks for specifying configuration information for all types of information needed to set up a VR environment: display chunks, tracker chunks, C2 chunks, HMD chunks, and so on.

Configuration information is edited with a Java based GUI called vjControl. It used to edit configuration information and interface to the running VR Juggler kernel. It allows users to create and edit config files, change the configuration at run-time, start and stop devices, and view performance data.

Applications can also use the configuration system, adding their own chunks of data that can be loaded with the same interface and edited with the same graphical tools. This allows applications to use the same simple, graphical configuration tools as the VR Juggler environment itself.

Internal Managers

Input manager

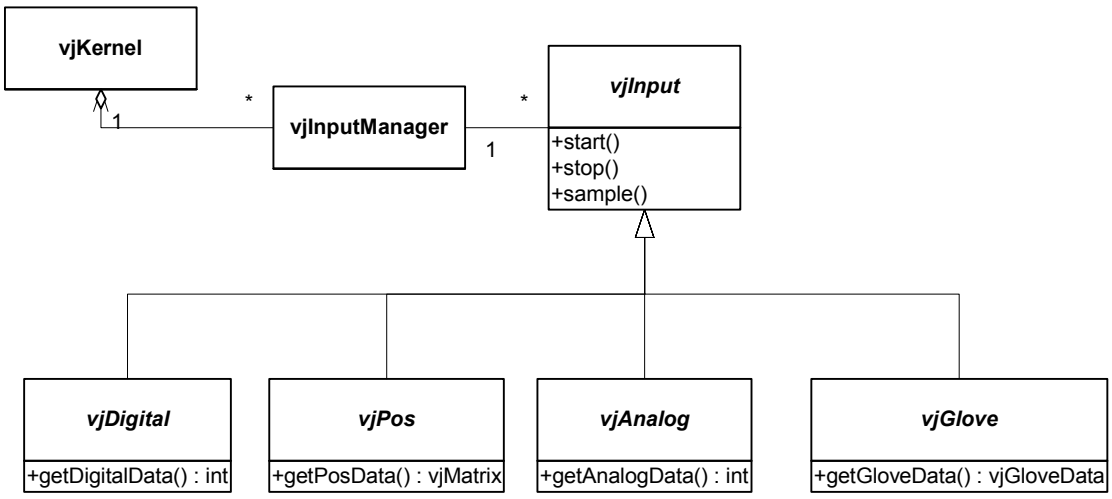

Figure 21: Input device hierarchy

The Input Manager controls all manner of input devices for the kernel. The input devices are divided into distinct categories, including position devices (such as trackers), digital devices (such as a wand or mouse button), analog devices (such as a steering wheel or joystick), and glove devices

(such as a CyberGlove™). VR Juggler defines a class hierarchy for the all of these types of input devices (see Figure 21). There is base class for all input devices that specifies the methods that must be implemented to start, stop, and update the device (see `vjInput` Figure 21). There is also a base class defined for each distinct category that specifies the generic interface that any device of that category must support. For example all positional devices must support a `getPosData()` member function that returns a position matrix.

Adding a device

To add support for new devices, developers create a new class for the specific device. The new class must be derived from the base classes of the input types that the new device can return (Figure 21). For example, a device that can return glove and positional data is derived from `vjGlove` and `vjPos`. The new class has to define member functions that implement the base input device interface as well as define the methods for returning the input type of each of the parent classes.

Device proxies

The application uses device proxies to interface with all devices. Before an application gets data from a device, it must first get a proxy to the device. The application requests the device by using the name that the device was given in the current configuration. The input manager looks up the requested device, and returns a proxy to that physical device.

Device proxies allow the application to be decoupled from the devices in use. The device referred to by a proxy can change at run-time. For example, a proxy may initially be linked to a hardware tracker in the system. If the user wants to change tracking systems, then the user points the proxy to a different tracker. The next frame, the application still uses the same proxy but the data returned is now coming from a different device. The application can not detect that a change occurred. The proxy abstraction allows the configuration of the virtual platform to change during execution without disturbing the application.

Device store

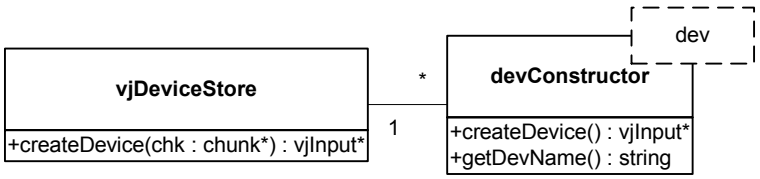

Figure 22: Device store

VR Juggler uses a device store [34] to allow the system to keep all device drivers separate from the main library and to dynamically load device drivers at run-time. The device store is a factory

object [30] that keeps track of device constructors and the associated device drivers that have registered with the system. A device constructor is a proxy that hides the exact type of a device driver and the method used to create new instances of the device. There is no coupling between the library and the device drivers, so the set of device drivers can vary independently.

The device store supports the addition of new devices at run-time or at link time by registering a new device constructor object with the device store. This allows a developer to add new device drivers without having to recompile an application.

When the input manager receives a configuration request to add a new device, it asks the store if it has a constructor that knows about the device name given. The store queries all the registered device constructors and if one of them knows how to instantiate a driver for the named device, then an instance of the new class is created and the configuration information is passed to it. A handle to the new device is then added to the input manager's list of active devices.

Environment manager

The environment manager holds information about the state of the system and allows communication of the state to external programs. VR Juggler's run-time control interface, vjControl, communicates with the environment manager via a network connection. The environment manager supplies data to the GUI and passes on instructions from the GUI to the kernel. From this interface, a user can view and dynamically control every aspect the running virtual platform.

Display manager

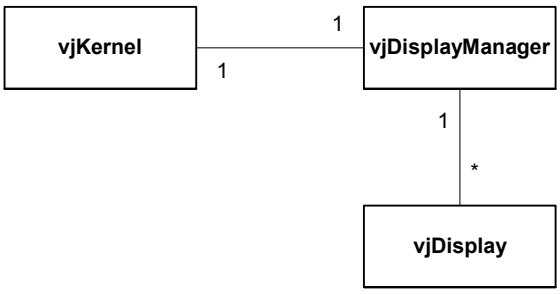

Figure 23: Display Manager

The display manager encapsulates all the information about the display windows' settings. This includes information such as size, location, graphics pipeline, and the viewing parameters being used. The display manager is also responsible for performing all viewing calculations for the windows it controls. The display manager is used to configure the draw managers in the system.

External managers

Draw manager

The draw manager executes client applications that need access to API-specific functionality. The draw manager defines a base class application interface that is customized for a specific graphics API. Since the draw manager is specific to an API, it allows for use of API specific features. For example, a scene graph API has a custom interface that queries the application for the scene graph to render. A direct mode rendering API, such as OpenGL, has an interface that defines a draw method to send all graphics rendering commands. Customizing the draw managers for specific APIs allows for maximum application performance because the application can make use of any advanced API features the developer desires.

Draw managers manage all the details specific to each API. They handle the details of configuring the settings of the API, setting up viewing parameters for each frame, and rendering the views. The draw manager also manages API-specific windowing and graphic context creation. Because all the graphics API specific details are captured in the draw managers, VR Juggler maintains portability to many graphics APIs.

Other external managers

Currently VR Juggler has no other external managers, but there are other possibilities for future external managers. These external managers function much like the draw manager by providing the applications with specific interfaces that the application needs to use. As long as the applications make use of the encapsulated features using the interface provided, VR Juggler will take care of the details of keeping the manager synchronized with the rest of the system.

Application

A VR Juggler application only has access to the kernel and the external draw manager associated with the graphics API of the application. The application queries the kernel for system state and to get access to any input proxies that are needed for the application.

VR Juggler treats the application as it does any other component plugged into the kernel. The application has a base interface that it must implement to support kernel interaction. The kernel keeps the application synchronized with the rest of the system by invoking application callbacks at pre-defined times during the frame of execution. Because the application plugs into the kernel, it can be removed or replaced at any time. This allow applications to be changed at any time while VR Juggler is active.

The base interface is extended through sub-classing to create application interfaces that are API specific (Figure 19). For instance, an Iris Performer application may have a function that returns the base scene node so the draw manager can render the graph in each channel. This allows the application to be specialized for the specific graphics API being used.

Multi-threading

On technique used extensively in the VR Juggler implementation is multi-threading. This section describes how multi-threading helps increase the performance of VR Juggler and also simplify the design of the system.

VR systems can make use of multiple threads to increase performance. In many instances, this is actually the only way to achieve high performance. VR applications have to deal with numerous input and output devices in addition to whatever simulation and processing is executed by the application. If each of these tasks is executed sequentially, then the system performance will suffer because the performance will be limited by the completion time of the slowest operation[35].

To avoid this problem, several current VR development environments decouple each of the tasks so that each operation can execute individually without incurring delay by waiting for other operations[21][36][3]. Each task can be viewed as separate component that executes relatively independently of the other components. The system core is responsible for making sure that the components are synchronized when needed. This allows the application to run as fast as possible because, the display may be able to run at 60hz while the tracking system runs at 50hz and a haptic controller runs at 500hz. If the devices were not in separate threads, they would have to run at least as slow as the slowest device, and would actually run slower because they would have to wait for all devices.

It is worth noting that multi-threading the VR software system increases performance even if there are not multiple processors available. In the case where there is a single CPU, multi-threading still increases performance because when one thread is blocked waiting for a system resource, another thread can execute the operations that it needs to.

The multi-threaded architecture also simplifies the architecture greatly. In a multi-threaded architecture is possible to have many small modules executing small amounts of specific code with a well defined function instead of having a small number of monolithic code segments that try to handle everything.

System interaction

In order to better explain interaction within the VR Juggler architecture, we will now describe how the system starts up and loads a single OpenGL application (see Figure 24). In order to simplify the diagrams we will not go into the details of configuration, we also leave out some method invocations, and we do not deal with multiple application objects.

The first step in starting the system is to initialize the kernel. This starts by giving the kernel a thread of control. This is done in a startup routines that instantiates the kernel object, and then activates it by executing the kernel method start() which creates a thread for the kernel. Once the kernel has been activated, the kernel then initializes each internal manager.

The next step in startup is to create and initialize an application. First, an application object must be instantiated. Then the application object is given to the kernel to start executing. The first thing the kernel does with an application is to create the draw manager that the application needs. Once this is done, the kernel then executes the applications initialization methods and signals the draw manager to do the same.

The final phase of a clean startup is to start the execution frame loop. When the kernel does not have an application object, it does not execute any parts of the execution frame that deal with application objects.

The first step in the execution frame is to call the preFrame() method of the application to tell it that a frame is starting. Next the kernel triggers the draw manager to render. While the draw manager is rendering, the kernel calls the application intraFrame() method to allow for parallel processing of rendering and application processing. Next, the kernel synchronizes with the draw manager by invoking the blocking sync() method which will not return until rendering has been completed. The kernel then calls the application postSync() method to notify the application that the frame is done. Next, the kernel check for any reconfiguration requests. If there are requests, then the kernel reconfigures the virtual platform before continuing. The final step in the execution frame is to update all device data and compute the drawing projections for the next frame.

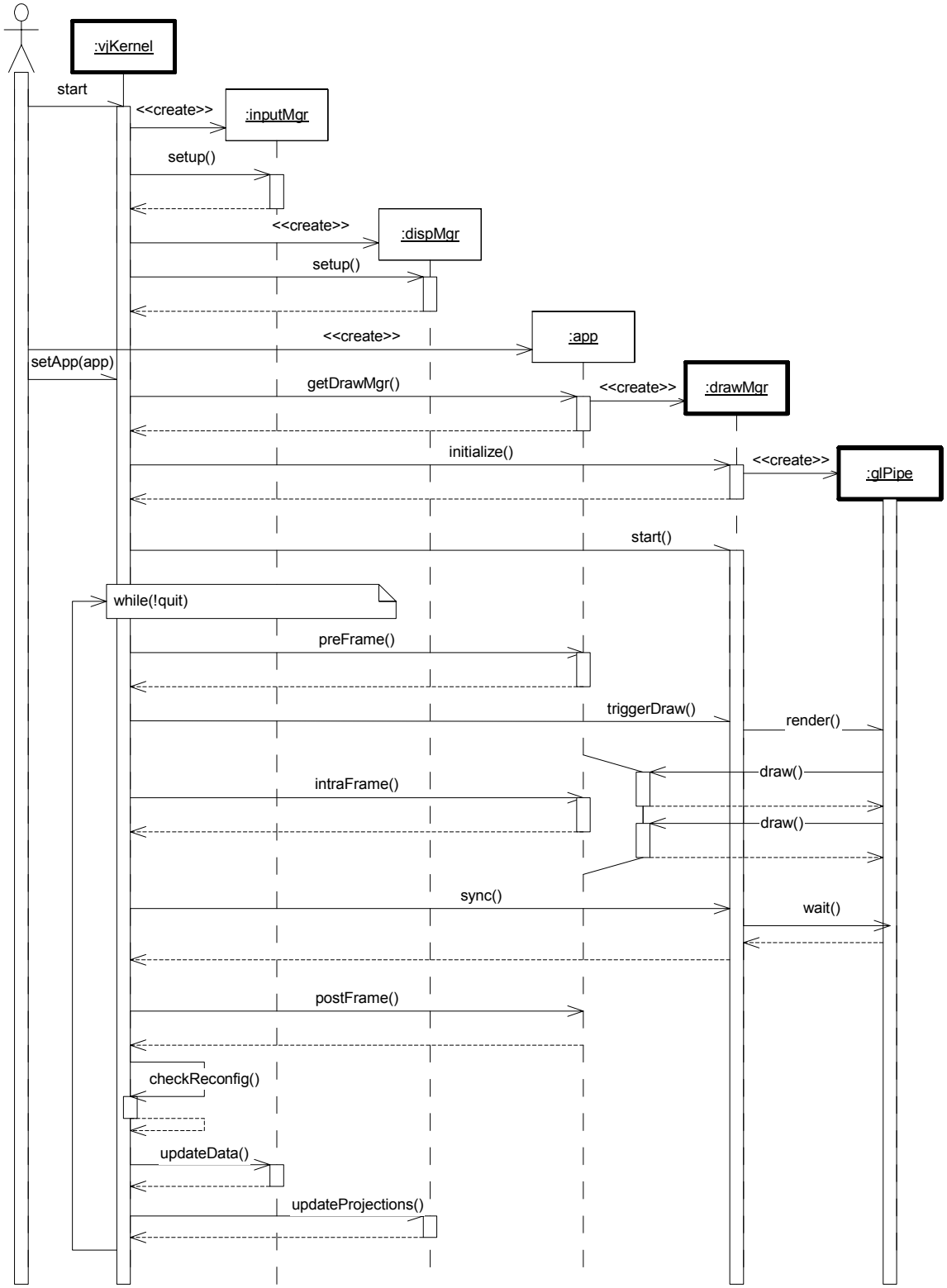

Figure 24: Kernel startup and execution

CHAPTER 8 DISCUSSION

Implementation methods

Iterative development

As touched upon previously, VR Juggler was developed using an iterative design method. Iterative design methods focus on building up the functionality of a system one step at a time. First, a small working system is created to start development from. Progressively the system moves towards its final state by adding new feature in small increments.

Iterative development has several advantages for development of a large project such as VR Juggler. The first benefit is that at any given time it is easy to evaluate the state of the system. After a developer makes a simple change, the system can still run and the change can be tested. Another advantage is that the system can be re-evaluated and changed as development proceeds. Unlike non-iterative development where the entire project has to be completed before the project can be evaluated, developers can refine their designs as they go. If suddenly they realize that the system will not work in the way they thought or if they find that there is a better way of doing things, the system design can be corrected. This ability to find problems and deal with them early greatly reduces risks involved in software development.

Challenges in VR Juggler development

There were several challenges faced during the development of VR Juggler that were both expected an unexpected.

Cross-platform primitives

One of the primary challenges what in achieving cross-platform capabilities. When the VR Juggler project was begun, there was no library such as the NSPR [37] that would allow for easy cross-platform system primitives. Because of this, the VR Juggler development team had to create its own primitives library.

Creating the primitives library was a challenging task for many reasons. One of the major challenges was deciding what the interface should look like. This was difficult because we did not have any of the Juggler system written, so we did not know for sure what abilities we would need in the primitive system. Another problem was that we had to learn how to create the primitives on every

platform. Unfortunately, we found that the only thing that was consistent across the platforms was the lack of good documentation for low-level synchronization primitives and thread management.

Flexibility vs. performance

A major challenge of the project (that is still ongoing), is the process of balancing flexibility and compatibility with performance. For example on the SGI system, there are ways to optimize the synchronization of frame buffer swaps between hardware pipes. Because this sort of functionality has an SGI specific implementation, it does not fit well into the general framework. In cases like this, we had to iteratively refine the frameworks to create our own generalization for this behavior.

Another area where we have had to fight system specific code is in user applications. Many current developers have unfortunately become accustomed to using system specific features in their applications. Because we want VR Juggler applications to be portable, we have restricted the access to these types of system features. It is not impossible to use these features, but it is discouraged. Where possible, we have created abstractions to allow the user to make use of generalized versions of the behavior in a way that can be ported to different system.

Iterating based on applications

One area that we leveraged heavily in the development of VR Juggler was the development of test bed applications. In order to evaluate the system at each stage, we would implement both test and full applications to evaluate the features (or lack there of) in VR Juggler. In later stages of development, this became the primary way by which to introduce features into the VR Juggler API.

We would create an application and evaluate how easily the application could be developed. When we found things that could not be done in the application, we would examine the cause of the problem. Then from that information we would progressively refine the VR Juggler design until the development team was satisfied that the library had the features necessary for applications of that type.

The next section describes some of the iterations that test-bed applications influenced.

Iterations

After the initial design and implementation of VR Juggler, the development team continued to iteratively refine the architecture in order to extend its abilities, correct flaws in the initial design, and to fix bugs.

This section gives an overview of several of these refinements. The section starts with a large section describing the extensive run-time reconfiguration system that was added to VR Juggler. It

then moves on to discuss several ways in which the developers had to refine the application object interface and how multi-user abilities were added to the system. The section finishes by talking about refinements dealing with system performance.

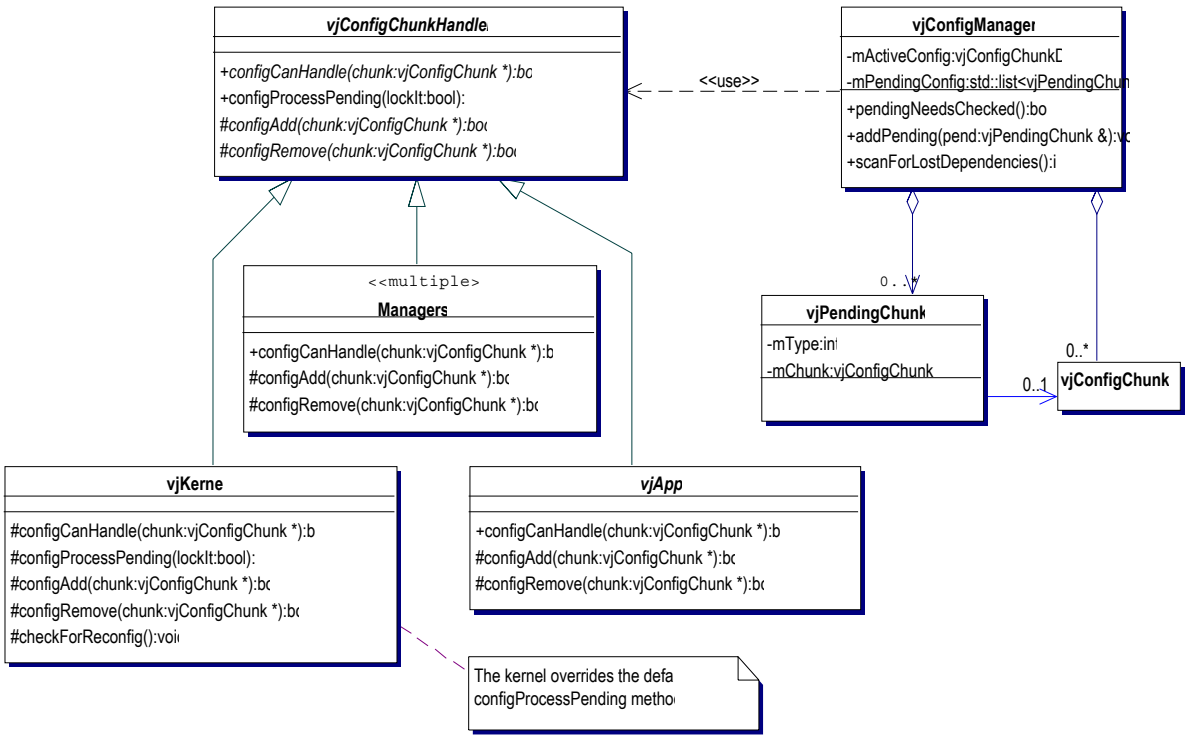

Figure 25: Reconfiguration classes

Run-time reconfiguration

One of the largest refinements was to add support for run-time reconfiguration. VR Juggler was designed with run-time reconfiguration in mind, and we actually started implementation an initial version in its original coding. However, during implementation it became quickly apparent that the initial design would not work as well as it needed to. Because of this, the initial version of VR Juggler did not have support for run-time reconfiguration. Instead, we differed the implementation of run-time configuration until later when the design could be re-worked.

The next sections describe how run-time reconfiguration is implemented in VR Juggler based on the refinement performed in this iteration.

Pending configuration queue

Reconfiguration is implemented using a pending config queue that is contained in the config manager (see `vjConfigManager` in Figure 25). As with all other managers, the kernel controls this manager. The system initially starts with only the kernel and several system managers instantiated. When the kernel receives a new config chunk, it passes it to the config manager which adds the config chunk to a pending reconfiguration queue.

Each frame, managers and any other interested component in the system check the reconfiguration queue for new entries. If there are new entries, and the querying component knows how to use the new entry, then the component processes the entry. It is the responsibility of the configuring component to alert the kernel to the change in configuration.

This method of processing the configuration information is based on the chain of responsibility pattern [30]. It differs only in that the handlers in VR Juggler do not have a successor chain. Instead, the kernel keeps a list of all possible handlers in the system. The system then traverses this list to find a valid handler for a given reconfiguration config chunk. It is also possible to directly query the config manager if an object would rather not have the kernel manage its reconfiguration.

Class interfaces

VR Juggler provides a common framework for processing configuration entries in the config queue. A class can make use of this framework by inheriting from the `vjConfigChunkHandler` class (see Figure 25). Once the class has these abilities, it is only necessary to provide custom implementations for the functions in the framework.

Advances Abilities

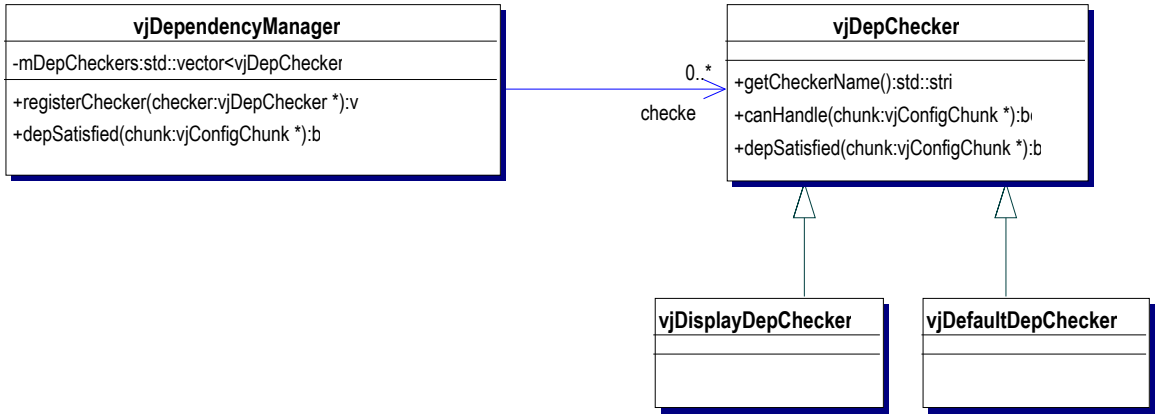

Figure 26: Dependency checking classes

As implemented in VR Juggler, the reconfiguration system has several advantages beyond basic reconfiguration.

VR Juggler allows for smart dependency checking between system components. Dependency checking refers to the process the system goes through when it runs a check to see if all the system resources required by a new configuration request are available before it allows that request to be processed. These checks are implemented in the `vjDependencyManager` using customized dependency checker classes derived from the `vjDepChecker` class (see Figure 26). Developers can make these dependency-checking classes as simple or complex as necessary for a given dependency test.

VR Juggler also allows for smart unloading of system components. This is needed because in a reconfigurable system it is possible to remove a component that another component relies upon to function correctly. Smart unloading allows dependent components to reconfigure themselves dynamically to modify their dependency if possible. If the dependencies cannot be satisfied by modification, then the component is unloaded and placed back into the pending queue until the dependencies are satisfied.

The reconfiguration system is currently still being refined. In addition, portions of the VR Juggler system are still being redesigned to have better support for that reconfiguration abilities that this system affords them.

Extend/refine application interface

One major area of refinement has been the application object interface. Currently, VR Juggler has undergone at least three major revisions of this interface. Some of the changes are due to adding new features and abilities that were not originally included. Other changes have been made in attempt to simplify the interface and make it more understandable to users. This section will touch on just a few of the revisions that have been made.

OpenGL applications

Context ID: Originally, VR Juggler did not give the user direct access to a context id. Instead, users had to use context specific data templates that would provide them with unique copies of their data for each context. VR Juggler was designed in this way in order to simplify application development and to keep developers from writing applications that could be dependent upon context information.

This design works for nearly every application that has been written for VR Juggler, but there have been a few application where getting a context id has proven to be very valuable. Because of

this, a context specific identifier and a method to query it were added to VR Juggler. Using this interface is not encouraged, but in order to support advanced users it needed to be added to the system.

Additional context callbacks: The original design of VR Juggler only provided one context based callback function: `contextInit()`. This was fine for many of the original application that were written, but it soon became apparent that `contextInit()` only provides part of the support needed. It provides a callback for initialization upon context creation, but VR Juggler did not have support for cleanup upon context destruction. To support cleanup, `contextClose()` was added to the `vjGlApp` interface.

VR Juggler also did not have support for writing context-based code that was not tied to creation (or destruction). Because of this, a `contextPreDraw()` function was added to the `vjGlApp` interface. This callback is called upon entry to each context during each drawing frame. It can be used for dynamic context based code, such as dynamic allocation of OpenGL display lists.

Performer applications

Pre-fork initialization: When a Performer-based application starts up, the Performer system is first initialized, and then it forks off several processes. In the initial design of the draw manager for Performer, we allowed users to initialize data before Performer initializes its run-time system and after it forks off processes. This provided all the functionality necessary for simple applications, but after writing several more complex applications, it was discovered that some applications need to do processing after the system had been initialized, but before the system forks off it's processes. To support this functionality, we added `PreForkInit()` to the interface of the Performer application object.

Frame buffer configuration: When a Juggler Performer-based application creates display windows, it tries to request specific common graphics attributes as a default. Once again, this worked for simple applications, but once we wrote several complex applications, we realized that the system needed to provide a way for user applications to customize the graphics attributes requested from Performer. We added the `getFrameBufferAttrs()` to the interface of the application object to so user applications can give the run-time system specific attributes that it requires. Once again, this was a simple change, but one that was very important to the system.

Multi-user extensions

After the initial implementation of VR Juggler was complete, we were given the task of adding support for multiple simultaneous users tracking in a single VR system. When we initially designed VR Juggler, we had not considered this specific requirement. Nevertheless, we needed to quickly add support for this new feature to the system.

It turned out that we where able to implement it very quickly due in large part to the modular structure of the system. Because the architecture of VR Juggler has no direct ties between the managers, there is no requirement that the software system render views for only one user. All that was needed to render the view of multiple users was to associate the position of a user with the window rendering their view. We added a user object to the system that contains information about the position of the user in the environment. Once user objects were added it was a simple process to add the code that would allow a user to be associated with a given window and would use that user information to render the corresponding view.

In addition, since VR Juggler has no limit to the number of trackers it can support, it also has no limit to the number of tracked users it can render views for.

Find and eliminate performance problems

A recurring theme in our refinement work has been finding and eliminating performance bottlenecks. The attitude we have taken in developing VR Juggler is to keep performance in mind during design, but to place our first priority on design correctness and an elegant architecture. Once we have a design in place that works correctly, we begin analyzing the performance. In most cases, we have had to make not changes because we took performance into consideration and there were no problems. In some cases though, we have had to track down performance problems.

How we solve the performance problem varies depending upon the cause. If it is a single routine that is performing badly, then we try to solve it by using a better performing algorithm. If it is an architectural issue, then we take the information learned from the problem and refactor the design until we achieve the desired performance.

This method has actually proven invaluable because it allows the system designers to constantly refine the system design to increase performance.

How well did it meet the design goals

Virtual platform

The concept of a virtual platform facilitates and simplifies the effort of application development in complex VR systems. It provides a unified working environment that supports development and execution of applications, independently of the underlying technology. A virtual platform guarantees the longevity of applications, and allows application developers to keep up with the technology advances without having to invest time and resources in modifying applications to support the new technologies.

Although there is a popular, and all-to-often well-founded, belief that object-oriented abstraction introduces severe penalties in program performance, we believe that object-oriented design approach for VR Juggler as a virtual platform provides the best avenue to achieve its goals. We have placed a great deal of effort on optimization of our abstraction levels to minimize the performance impact. Currently, performance evaluations of the different components are under way; early results are showing that the overall VR Juggler performance is within acceptable response times.

Hardware abstraction

Displays are abstracted to allow support for any VR device. The display manager uses a generic surface description that allows for any projection surface. Currently, we use VR Juggler with projection-based system such as the C2 or CAVE. It also has support for HMDs. By using a generic display description, we can configure an application to run on nearly any VR display device.

For some types of display devices, it is necessary to change the way the application behaves. Currently we are investigating ways to allow application to change behavior depending on the type of the device the display is set for.

Input is abstracted through proxies. The proxies provide the application with a uniform interface to all devices. The application developer never directly interacts with the physical devices, or the specific input classes that control them. New devices can quickly be added by deriving a new class to manage the new device. Once this class exists, applications can immediately begin to take advantage of it.

Run-time flexibility

The proxy system gives VR Juggler much of its run-time flexibility. The physical device classes can be moved around, removed, restarted, or replaced without affecting the application. The proxies themselves remain the same, even when the underlying devices are changed.

VjControl offers an easy-to-use interface for reconfiguring, restarting, and replacing devices and displays at run-time. This allows users to interactively reconfigure a VR system while an application is running. The ability to reconfigure at run-time increases the robustness of applications because it allows devices to fail without taking down the entire application.

Performance tuning

VR Juggler includes built-in performance monitoring capabilities. These include the ability to accumulate data about time spent by various processes, performance data for the underlying graphics hardware (as available), and measure tracker latency (the time between generation of tracker data and the display of data generated from the tracker data). VjControl can display the performance data at run-time or the performance data can be captured and analyzed later.

The environment manager allows users to dynamically reconfigure the system at run-time in an attempt to optimize performance. When the user changes the VR system configuration, the performance effects will be immediately visible with the performance monitor.

Cross-platform

VR Juggler is portable to all major platforms use for VR development. To maintain portability, all system specific needs (such as threads, shared memory, and synchronization) are encapsulated by abstract classes. The library only uses the abstract, uniform interface. This allows easy porting of the library to other platforms by replacing the system-specific classes derived from the abstract bases. Currently the library has support for SGI, Linux, and Windows NT.

Extensible

The library allows extension without impacting the rest of the system or applications that have been previously written. Adding new devices of an already supported general type (such as new position inputs, or new displays) is simple and transparent to applications. This is because the library uses generic base class interfaces to interface with all objects in the system.

Problems encountered

Learning curve problem

One problem that we have encountered is that the learning curve for VR Juggler seems to be much steeper than we would like. Users seem to initially have trouble grasping some of the basic concepts of VR Juggler such as how proxies work and how to configure the system. Once the user gets beyond these hurdles, they generally advance very rapidly in their understanding of VR Juggler.

A related problem is that the majority of the users do not seem to actively pursue the process of learning how to make use of VR Juggler's abilities. For example, the run-time reconfiguration system gives developers and users a new level of flexibility when choosing how to configure and make modifications to executing applications. However, because many of these concepts are new to the developers, they do not fully understand or take advantage of the abilities.

We are currently investigating ways to ease this with more complete and readable documentation and tutorials. In addition, we are working to develop simpler user interfaces for the system.

Cross-platform programming

Cross-platform programming presented many interesting and sometimes frustrating challenges. The development team faced the task of designing an abstraction that would take into account not only the abilities, but also the limitations of all the target platforms. This was because when writing performance critical applications, the system must not be restricted to the least common denominator of the systems that it is to be used upon. Instead of limiting the system to the constraints of each system, the development team had to come up with innovating solutions that would take advantage of each system's advantages while working around it's weaknesses.

We also had to deal with the more mundane cross-platform challenges derived from differences in compilers, system file locations, and programming language support. At the beginning of the project, these problems severely restricted the number of platforms that we developed on. The main reason for this was that we did not have a good mechanism in place to discover the state of the current development system and modify the building of VR Juggler to that system's configuration. Fortunately, we were able to create a build system for the project that allowed us to work through and in many cases eliminate all of these problems.

Java virtual machines

When originally designing VR Juggler, we made the decision that the support tools for the system would be written in Java. This was because Java provided a cross-platform development system with a reasonably complete GUI system. Because the support tools needed to run on many platforms and did not have to be high performance, Java was a logical choice although we could just have easily chosen Python, or any other interpreted language.

The problem we discovered during development was that not all Java virtual machines (JVMs) are created equal. The mantra of Java "compile once, run anywhere" is nice, but we discovered the reality to be much different due to problems and quirks of each specific JVM. Several JVMs performed extremely badly, and others had GUI errors that prevented users from using our tools. It

proved to be very frustrating to the development team, because the problems were beyond our control. The code was correct, the but JVM's just executed it incorrectly.

It is worth noting that as of this writing, this situation with the JVMs is starting to look more positive. Currently, none of the JVMs on our primary development environments presents GUI problems although the performance is still a problem with some JVMs.

CHAPTER 9 CONCLUSIONS

Contributions to field

VR Juggler has made several major contributions to the VR research community. It has contributed the concept of a virtual platform for VR development along with a concrete implementation of such a system. This system provides a flexible standard upon which developers can build long lasting applications. This research has also opened up VR research to all researchers and developers by contributing the VR Juggler virtual platform to the open source software community. Additionally, VR Juggler has shown how reconfigurable systems can be useful for VR developers and end users.

Flexible standard for building long lived applications

VR Juggler's architecture has been design to be long lasting. The design is long lived because it allows current components to be refined and replaced as needed without having dire consequences for the rest of the system. In addition, because VR Juggler is not reliant on any single API, it does not rely upon any other tool's survival for its long-term use. Another key area of survivability is VR Juggler's capacity to be easily extended with additional capabilities.

The VR Juggler architecture enables easy integration of new system components to add new functionality and fulfill new requirements. This ability is due to the microkernel architecture of VR Juggler. It allows developers to easily support emerging technologies and techniques. Previously developed applications are able to transparently take advantage of these extensions without any changes.

VR Juggler provides a flexible standard that researchers can use to experiment with new ideas and innovations. The architectural flexibility of VR Juggler enables VR researchers to extend and augment the VR Juggler virtual platform to use it as a test bed for experimentation and further development. They are not required to write an entire system from scratch, they only need to work with the section that they are interested in exploring.

Virtual platform

Provides a single generic programming target that works everywhere

The VR Juggler virtual platform introduced in this research provides a single generic programming target for VR development. The virtual platform encapsulates all the services and capabilities of the VR system. This allows developers to write an application for the generic system presented by the virtual platform, and have the application transparently work on any system that the virtual platform can run on top of.

Provides hardware abstraction

The virtual platform abstracts the VR hardware system. The application does not have to rely upon any specific functionality of the hardware system in use. For example, if the VR hardware system being used has custom methods to synchronizing graphics displays, the application does not need to deal with these issues. The application interacts only with the virtual platform. It is the responsibility of the virtual platform to deal with the system specific issues.

VR Juggler device proxies allow user applications to depend on the type of input data instead of the hardware that the data is coming from. The virtual platform provides devices proxies for generic types of input data (positional, digital, analog, etc). Since the application does not have ties to the actual hardware providing the input data, the input hardware can be changed without affecting the application code or executable. This prevents applications from ever being locked to specific hardware systems.

Cross-platform

The virtual platform also provides system primitives so applications are free from platform specific dependencies. The system primitives allow the applications to be cross-platform.

Open source system

Levels the playing field

Because VR Juggler is open source, it levels the playing field for people interested in making use of VR. Most current VR systems require license fees that can keep them out of the hands of the majority of people. No longer do researchers that are unwilling to pay license fees have to implement an entire system from scratch. VR Juggler changes this by providing a highly capable system that anyone can use. VR Juggler brings high-end VR development environments to anyone that wants it.

Peer review

An open source software system facilitates peer review and continued research. Peer review is a fundamental tenant of research. It allows fellow researchers to examine the research and attempt to

reproduce the results. Because VR Juggler is an open source system, other VR researchers and examine, test, and refine the system in any way that they wish.

This system can also be helpful for the future of peer reviewed VR research because it provides an open platform that future research and be based upon. If a researcher creates a new VR interaction method that they would like to allow others to use, they can implement it on top of VR Juggler and be assured that any one who would like can make use of the method.

Extension and growth

It would be foolish of the VR Juggler development team to think that we have solved all VR software development problems. With this in mind, we decided to make the project open source to allow other developers to further extend and refine the system.

It is our hope that other people will contribute to VR Juggler and help it grow to become the most widely used VR development software available.

Allows many researchers to build upon the platform to create and explore new ideas

User controlled software destiny

Open source software allows users to control their own software destiny. If there is a bug in the software and they can find no one else to fix it, they have the code so they can fix it themselves. It they need support for a new device, they can add it. Users do not have to wait for any controlling group or company to fix a problem or extend the system. Users can modify the system in any way necessary, and they are assured that they can do this for as long as they wish.

Reconfigurable system

VR Juggler has introduced the benefits of a run-time reconfigurable system to the VR community. As discussed previously, run-time reconfiguration can assist in setup, testing, performance tuning, and allowing application adaptations.

Run-time reconfiguration can also be used to monitor and tune VR system performance. It also supports the ability to switch between applications at run-time or even run multiple simultaneous applications.

CHAPTER 10 FUTURE WORK

There are several areas that still need future work and investigation.

Component system

Before the VR Juggler system can become fully modular and extendable, it must make use of a component system. We are currently investigating the use of a component-based approach for the low-level components of VR Juggler. Publicly available tools, such as Bamboo [38] are good candidates to provide the component-based infrastructure for VR Juggler.

VR operating system

A very exciting use of VR Juggler in the future will be the creation of a VR operating environment where many VR applications execute within the same environment. This could form the basis of a true operating system for virtual reality.

VR tools

We currently have development groups working on many tools that can be used with VR Juggler including networking tools for distributed environments, user interface libraries for application development, and general application structures for rapid application development.

BIBLIOGRAPHY

[1] J. Vince, *Virtual Reality Systems*, Addison Wesley, Reading, Massachusetts, 1995,

[2] C. Cruz-Neira, "A Look Behind the Scenes of Virtual Reality Applications," *ACM Symposium on Virtual Reality Software and Technology*, Hong-Kong, 1996, pp. 161-162

[3] C. Cruz-Neira ,*Virtual Reality Based on Multiple Projection Screens: The CAVE and Its Applications to Computational Science and Engineering*, PhD. Dissertation, University of Illinois at Chicago, 1995.

[4] "WorldTookKit Release 8: Technical Overview," www.sense8.com (current Jun 3, 2000).

[5] R. Stuart, *The Design of Virtual Environments*, 1996,

[6] S. Maguire, *Writing Solid Code*, Microsoft Press, Redmond, 1993,

[7] R. Kalawsky, *The Science of Virtual Reality and Virtual Environments*, Addison-Wesley, 1993,

[8] S. Singhal and M. Zyda, *Networked Virtual Environments*, ACM Press, New York, 1999,

[9] S.C. McConnell, *Code Complete*, Microsoft Press, Redmond, 1993,

[10] "VTK homepage," http://www.kitware.com/vtk.html (current May 30, 2000).

[11] "OpenDX homepage," www.opendx.org

[12] "Iris Performer Homepage," http://www.sgi.com/software/performer (current Oct 1, 1999).

[13] "Iris Performer Getting Started Guide," http://www.sgi.com/software/performer/developer.html (current Oct 1, 1999).

[14] J. Rohlf and J. Helman, "Iris Performer: A High-Performance Multiprocessing Toolkit for Real-Time 3D Graphics," *SIGGRAPH*, ACM Press, New York, 1994, pp. 381-394

[15] R. Pausch and e. al., "A Brief Architectural Overview of Alice, a Rapic Prototyping System for Virtual Reality," *IEEE Computer Graphics and Applications* 1995,

[16] "Alice Homepage," http://www.alice.org (current Oct. 1, 1999).

[17] C. Cruz-Neira et al. , "Surround-Screen Projections-Based Virtual Reality: The Design and Implementation of the CAVE," *ACM SIGGRAPH*, 1993, pp. 135-142

[18] "Avango Homepage," http://imk.gmd.de/docs/ww/ve/projects/proj1_2.mhtml (current Oct 21, 1999).

[19] R. Blach et al. , "A Highly Flexible Virtual Reality System," http://vr.iao.fhg.de/ccvr/publications/main-en.htm (current May 1, 2000).

[20] "Lightning Homepage," http://vr.iao.fhg.de/vr/projects/Lightning/OVERVIEW-en.html (current May 1, 2000).

[21] C. Shaw et al. , "Decoupled Simulation in Virtual Reality with the MR Toolkit," *ACM Transactions on Information Systems* Vol. 11, No. 3, 1993, pp. 287-317,

[22] C. Shaw et al. , "The Decoupled Simulation Model for Virtual Reality Systems," *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, 1992, pp. 321-328

[23] C. Shaw and M. Green, "The MR Toolkit Peers Package and Experiment," *IEEE Virtual Reality Annual International Symposium*, 1993, pp. 463-469

[24] S. Halliday and M. Green, "A Geometric Modeling and Animation System for Virtual Reality," *Virtual Reality Software and Technology*, 1994, pp. 71-84

[25] Q. Wang et al. , "EM: An Environment Manager for Building Networked Virtual Environments," *IEEE Virtual Reality Annual International Symposium*, 1995,

[26] "Sense8 Homepage," http://www.sense8.com (current Jun 3, 2000).

[27] "Python homepage," http://www.python.org (current May 20, 2000).

[28] I. Jacobson et al. , *The Unified Software Development Process*, Addison Wesley, Reading, Massachusetts, 1999,

[29] F. Buschmann et al. , *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996,

[30] E. Gamma et al. , *Design Patterns*, Addison-Wesley, 1995,

[31] R.G. Lavender and D.C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," *Proc.Pattern Languages of Programs,* 1995,

[32] T. Imai et al. , "The Virtual Mail System," *IEEE VR 99*, Houston TX, 1999, p. 78

[33] J. Rumbaugh et al. , *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999,

[34] J. Beveridge "Self-Registering Objects in C++," *Dr. Dobbs Journal* vol. 23, no. 8, pp. 38-45, 1998

[35] S. Bryson, "Approaches to the Successful Design and Implementation of VR Applications," *SIGGRAPH 94*, no. Course: Developing A, 1994, pp. 9.1-9.11

[36] R. Pausch et al. , "DIVER: A Software Architecture for Building Virtual Environments