Software Tools for Virtual Reality Application Development

SIGGRAPH '98 Course 14
Applied Virtual Reality

Allen Bierbaum and Christopher Just
(allenb, cjust@icemt.iastate.edu)

Iowa Center for Emerging Manufacturing Technology
Iowa State University

Abstract

With growing interest in Virtual Reality (VR) there has been a dramatic increase in the number of development environments for VR[1]. This paper presents a discussion of features to look for when choosing a development environment for virtual reality applications. These features include the software's capabilities, such as graphics and VR hardware support, the interface it provides for developers, and other items such as support for performance monitoring. It then gives a brief overview of several popular VR development environments from academic and commercial sources, discussing how well they address the needs of VR developers. The paper concludes with an introduction to VR Juggler, a development environment currently in development at Iowa State University. This includes a discussion of how Juggler's developers approached each of the desired features of VR development environments presented in the first section of the paper.

1. Introduction

As interest in Virtual Reality technology has increased, so has the number of tools available to the developers of virtual worlds. Some of these are libraries and toolkits, while others are application frameworks, and still others are full development environments, integrating every aspect of the creation of a VR application – modeling, coding, and execution – into a single package.

Each of these development systems has a unique set of features and, since there are not yet any standards for VR software, their own interfaces to those features. Each supports a particular set of hardware, giving developers and end users a particular level of abstraction. Each takes a different approach to handling the complex interactions of operating systems, networks, input devices, displays, audio outputs,

[1] Note: CAVE, ImmersADesk, PowerWall, Flock of Birds, SpacePad, CyberGlove, DataGlove, Direct3D, Windows NT, Windows 95, BOOM, PINCH gloves, and Iris Performer are all registered trademarks of their respective companies.

and haptic devices. Each has its own development interface, defining how to create graphical objects and control their behavior, how to interact with the environment, how to query trackers and other input devices, and so on. Needless to say, every one of these choices can affect the performance of applications.

In short, there are a lot of variables to consider when choosing a VR development environment. Sometimes the differences are subtle. Sometimes the pros and cons look like apples and oranges – difficult to quantify and compare. The intent of this paper is to illuminate and clarify the problem. First, we suggest what features to look for when evaluating VR development systems, the obvious and obscure points that will most influence a development effort. This is followed by short discussions of the most common development systems available from commercial and academic sources. Finally, we discuss VR Juggler, a system currently being developed by the Iowa Center for Emerging Manufacturing Technologies (ICEMT) at Iowa State University, and compare it to other currently available offerings.

1.1 Virtual Reality – Yet Another Definition

"Virtual Reality" means many different (and often contradictory) things to many people, from a low end of Quicktime VR [1] and web-based VRML [2] models to a high end of fully immersive environments like CAVEs [7] and head-mounted displays (HMDs). While they all have their uses, we need a narrower definition in order to make a coherent statement. For the purposes of this document, "VR" will refer only to systems capable of producing an immersive environment involving head-mounted or projection displays with support for head and possibly body tracking.

2. What to Look for in a VR Development Environment

Before deciding which development system is best suited to a particular application, the developer needs to know what questions to ask. This includes knowing what capabilities are required to implement the application and how to tell which systems meet those requirements. In this section we raise a number of questions and in many cases present what we believe to be the best answers. Keep the questions in mind when we discuss the various architectures – you will see that each environment answers the questions in different ways.

2.1 Primary needs

There are three primary requirements for a VR development system:

Performance

Effective immersive environments need a high frame rate (15 Hz or better) and low latency. Poor performance is not merely an inconvenience for the end user; it can cause unpleasant side effects including disorientation and motion sickness [3]. Therefore, a VR system should be able to take advantage of all available resources on a system, such as processors and special graphics hardware. The development system itself should have as little overhead as possible, letting the hardware run at its maximum efficiency.

Flexibility

The development environment should be able to adapt to many hardware and software configurations. If the environment cannot adapt to new configurations, applications will be limited in the scope of their usefulness. A developer should not be required to rewrite an application for every new configuration. In addition, the design of the system itself should not limit the sort of applications that can be developed with it. Developers should never hit a wall where the development environment restricts them from creating an application that works the way they have envisioned it.

Ease of Use

The development system should be easy to configure and to learn. The Application Programming Interfaces (APIs) and/or languages used to create applications should be cleanly designed and should hide as much of the system's underlying complexity as possible.

An ideal development system would score perfect marks in each category. In reality, they often conflict. An easy-to-use system's interface may limit the developer's options, sacrificing flexibility. A very flexible system might be difficult to optimize for performance because of the number of choices presented to the developer. Each of the systems we will present has chosen a different balance of these elements.

These general principles show up in many specific ways. We suggest that the following fifteen specific attributes should be considered when comparing VR development systems.

2.2 Capabilities of the Environment

Cross-Platform Development

What happens if an application has twenty potential customers, but ten of them use UNIX workstations and the rest prefer Windows NT systems? Many toolkits are available on two or more platforms, so it is important to consider present and future porting of applications when choosing the base for a development effort.

Well-designed toolkits should hide platform-specific details well enough that porting an application between different platforms with the same toolkit 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. A well designed VR development environment should help make this porting simple.

Support for VR Hardware

What kind of special VR hardware is the application designed to use? Is it simple enough to use a joystick or wand for user input, or will a glove make it easier to control? Is the finished product going to be used with projection systems, or HMDs, or both? Many groups already have large investments in

hardware, so it is important to make sure that hardware is supported by the development system. It is also possible to work the other way around – design the application first, then buy the hardware best suited for it. In addition to immediate needs, future plans should be considered – will a glove or some kind of haptic output device be available in the foreseeable future? If so, perhaps they should be added to the list of hardware devices that must be supported.

Hardware Abstraction

Support for required hardware is mandatory, but almost as vital is how well the toolkit abstracts away the details of the 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 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.

Locally Distributed Applications

Locally distributed applications attempt to increase performance by dividing the workload between several computers on the same network. For example, two workstations might be used to generate the two images used by an HMD. Some of the toolkits we present here have built-in support for distributing applications, and some do not. For those that do, the burden on the developer to create a working distributed application varies; it might be completely transparent, or it might require special consideration of what information needs to be shared and when.

Distribution has several advantages in addition to increasing the application's frame rate, such as increasing the number of input devices or display channels available to an application, or allowing additional computers to be used for simulations and other computationally intensive tasks. Our advice is that distribution is simply too useful an ability to ignore, unless it is absolutely certain that an application will be used only on single machine setups.

Distributed Environments

With application distribution, we think about connecting multiple machines at a single location. With distributed environments, we expand that to the idea of connecting machines – and users – at remote sites across a network. Development systems with this ability open up the possibility of bringing people together to collaborate in a virtual world. One example of such a system is

dVISE's dv/Review [10] component, designed to allow multiple users to meet together to explore and review product designs. Several toolkits offer capabilities for this sort of networking, though the level of support varies. Key issues include controlling interactions with multiple users and dealing with variable network latency.

How useful this feature actually is depends on the application and the environment. For a centralized organization, it may be superfluous. For a multinational corporation, it could be a valuable tool, reducing travel and communications costs.

Rapid Prototyping

Since most development groups will only have access to one or two full VR setups, it is important to be able to run the application and interact with it without using all the special equipment. Otherwise, the developers will waste a lot of time waiting for their turn on the equipment.

To allow developers to quickly prototype an application, the development system should include an environment that simulates the functionality of the actual VR hardware. This generally involves drawing the display in a window on a monitor and using the keyboard and mouse to simulate head tracking and other input devices. While support for this sort of low-hardware setup is fairly universal, the cumbersomeness of the input simulation and the closeness with which it models the actual hardware varies. Simulators for complex input devices, such as gloves or body-tracking suits, may not support the full range of inputs of the actual devices.

Several other factors are important for rapid development, including the level of support for object behaviors and the existence of easy-to-use scripting languages to control the virtual world. These are discussed in more detail in section 2.3.

Run-time Flexibility

It is often desirable to be able to change hardware configurations on the fly, without restarting the application itself. Sometimes devices fail and need to be restarted or replaced, and sometimes the initial configuration is not what the user expected. Some aspects of configuration may require trial and error, such as finding the optimal distribution of processes and resources across several machines. Sometimes it is convenient to be able to switch rapidly between two devices, as perhaps between two tracking systems, to see which performs better.

VR applications often take considerable time to start up: processes must be created and synchronized, hardware initialized, databases loaded, and so on. Restarting an application every time a small change is required or a device needs to be restarted can consume a great deal of time. For example, there is no overwhelming reason why a developer cannot move straight from viewing an application in a workstation simulator to viewing it on an HMD, without restarting it. All that is required is support from the development system: an API for replacing devices at runtime, and an abstraction that keeps the application from noticing the foundations shifting beneath it.

That said, support for this sort of dynamic reconfiguration is very rare today. The vast majority of development systems still require applications to be restarted whenever the configuration changes.

2.3 Development interfaces, tools, and languages

High-level and Low-level Interfaces

Each of the VR development environments we discuss gives the developer a different interface for creating applications. Some provide a very high-level view, where applications can be created with custom scripting languages and graphical tools, and the system itself takes on most of the responsibility of calculations, geometry, and interaction. Others float just above the hardware level, using well-known graphics APIs and programming languages to ensure the greatest performance and flexibility. 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. Therefore we see a continuing need for these lower-level environments, as well as their more featureful counterparts. The key, of course, is knowing which is right for a given project.

Graphics Interfaces

VR development environments differ radically in how they deal with graphics. Some environments emphasize loading models generated in a variety of external modeling and Computer Aided Design (CAD) packages, while others give developers a particular API for model creation. Some allow the user to manipulate models at the polygon level, while others only allow models to be loaded and viewed. Some support scene graph architectures, while others require all graphic routines such as rendering and collision detection to be written by hand.

Which kind of graphics interface is best depends upon the focus of the application. If it only needs to load the model of a car and allow a user to move around the environment to view the model, then the development environment needs to support loading models in the CAD format with which the car was produced. If the application is a scientific simulation, the developer may need to have access to the polygon level to render data for each frame. If the application needs a level of graphics interface that your VR environment does not support, obtaining the effect you need can be very painful or even impossible.

It is also important to consider what graphics API the library supports. Does it have its own custom API for defining graphics elements, or does it use a common API such as OpenGL or Direct3D that can be used in other, possibly non-VR, projects? Popular graphics APIs have several advantages. There tends to be a great deal of documentation about them from diverse sources, and it may be possible to find programmers who are already familiar with them. Furthermore, non-VR programs that use OpenGL (for example) might be easily rebuilt into VR applications – if the environment supports that API.

Interaction

How does the environment handle the details of user and program interaction? In some environments, the developer creates event handlers to deal with changes in the environment. The event could be anything from "User grabbed this object" to "These two objects just collided" to simply "New head tracker data is available." Alternately, the developer might have to write code that polls the current state of input devices and decides if the state triggers any interactions with the VR world.

Typically, "high-level" environments with their own modeling languages, or that use CAD or similar sources for objects, will have a greater range of interaction capabilities. Such environments usually have their own collision detection routines, and may even have built-in support for object physics (so that dropped items fall to or even bounce on the virtual floor, and so forth). Environments that give the developer the power to create her own graphics in OpenGL or another API will often give her much of the responsibility for handling user and object interactions.

APIs and Languages

Another thing to consider when choosing a VR development environment is how the developers will actually create applications. Will they use custom GUIs, specialized scripting languages, or an existing language such as C++, Java, or Scheme?

Custom languages often have special features or structures that make them particularly well suited for their tasks; they might be particularly aware of the data types and operations necessary for creating virtual worlds. These languages might have long-term advantages, and may aid rapid prototyping, but there can be short-term drawbacks. After all, every new language has its own (possibly steep) learning curve and its own training requirements. Also, some developers dislike learning new languages and would rather stick with what they know well and are comfortable with. Custom languages also have the disadvantage of requiring rewriting of any code that has already been written for other projects. If you are writing a flight simulator and you have thousands of lines of code in FORTRAN that control the aircraft simulation, you want to make sure that your development environment will allow you to use your current code.

If the environment uses an existing language, the learning curve is usually not as difficult. There can be other problems when the compiler becomes part of the VR development environment. For example, C and C++ code can have portability problems between platforms. With custom scripting languages, these problems are shifted away from the application developer and onto the VR environment's developers.

2.4 Other Factors

Extensibility

A VR software library should be easily extendable to new devices, new platforms, and new VR interfaces. The field of VR is constantly changing. A VR software library must be able to adapt easily to changes in order to keep from becoming obsolete. When the library is extended it should not require any changes to current applications.

Minimal Limitations

While simplicity is valuable, the software should not restrict advanced users from doing as they please. The environment should not require 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. Ideally, there should be no restrictions that a skilled user cannot circumvent.

Performance Monitoring

A VR software system should be able to collect performance data. This information is necessary in order to optimize software and hardware configurations, and to find bottlenecks. Given the collected performance data, the user may be able to reconfigure the system or change the application to maximize performance.

Commercial Versus Research Sources

Some of the systems this paper covers are commercial products, while others have come out of research environments. Both sources have their advantages.

Most of the research-based systems are available at no cost, which has a definite financial appeal. It may be possible to request changes or new features for the environment, or even to collaborate on new features. In many cases, the source code is also available, letting application developers take full control of the system.

On the other hand, commercial systems can be more comfortable to deal with. They tend to be more stable and portable, and to have been tested in a greater variety of environments. Commercial systems are usually better documented. Also, professional technical support staff can be invaluable.

3. Current VR Software

3.1 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.

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 like sound.

Description

First of all, 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 by Avocado, Lightning, and the CAVE Library.

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 can be used to build an application. Connecting these nodes in a directed acyclic graph, a data structure reflecting nodes and their relations to each other, forms a scene graph. Performer has basic scene graph nodes such as transformation nodes, but it also supports nodes that are 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 to allow users to import their data. 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 callback functions to the scene graph. This can be used to place custom rendering routines written in OpenGL into the scene graph to create special rendering effects. The combination of geometry nodes and callback functions allows developers to create any type of graphic effects that are necessary. Performer places no graphics limitations on developers.

Performer has a high performance rendering engine at its core, that has been engineered to get maximum graphics performance for the entire line of SGI graphics architectures. Performer uses several strategies to optimize rendering performance. First of all, most rendering loops are specially tuned to send graphics commands to the rendering hardware in an optimal way. These routines are hand tuned to ensure maximum 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 multiprocesses 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 multiprocessing the rendering pipeline, Performer also provides additional asynchronous stages. It provides an intersection stage that can be used for collision detection, a compute stage that can be used for general computations, and a database (dbase) stage 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 provides developers with the tools to create real-time applications. Performer is based on SGI's REACT system, which allows developers to fine tune process priorities and processor scheduling for time critical applications. Performer uses this fine control over CPU scheduling and process priority to allow real-time applications to be created. User applications written in Performer can also take advantage of REACT to ensure that user processes get the resources needed to guarantee real-time performance.

Performer also has the ability to maintain a consistent frame rate while scene content and complexity are varying. Performer maintains consistent frame rates 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 maximum 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. It is believed that it should be possible to maintain the higher frame rate, but the user does not know what is causing the slow down. By looking at Performer 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 it is user code or internal Performer code that is taking 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 maximum frame rate for VR applications.

Strengths

- **Performance**: Performer is designed to achieve maximum graphics performance on SGI systems.

- **File Loaders**: Performer can load many popular file formats. The loaders preserve the model hierarchy so as to allow users to manipulate the scene data.

- **Visual Simulation Features**: Performer has features such as fog, billboarded geometry, LOD, light points, terrain definition, and clip-texturing that are invaluable for developing visual simulation VR application.

Limitations

- **Not designed for VR**: Performer is not a stand-alone VR development environment. It does not provide support for VR hardware such as CAVEs, HMDs, and tracking systems.

- **Not Cross platform**: Performer only runs on SGI machines.

- **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.

- **Distributed Applications and Environments**: Performer has no networking support because it is not designed to support connected applications. Because of this, it has no support for application distribution.

References

Iris Performer Homepage. http://www.sgi.com/Technology/Performer

Iris Performer Getting Started Guide, IRIS Insight. SGI online manual

[4] J. Rohlf and J. Helman, "IRIS Performer: A High-Performance Multiprocessing Toolkit for Real-Time 3D Graphics," Proc. Siggraph 94, ACM Press, New York, 1994, pp. 381-394

3.2 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 VR 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.

Platform

The publicly distributed version of Alice is available for Microsoft Windows products.

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.

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, many ideas and options 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)
)
```

The typical Alice script looks something like this:

As can be seen in the example script, 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, it is possible for novices to easily write scripts that will work with Alice. By making the script easy for non-experts to use, Alice brings technology to those whom would ordinarily not use it.

Alice organizes the world as a hierarchical collection of objects. The parent/child relationships within the hierarchy can be changed easily at run-time. Alice gives developers the ability to easily switch between object 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.

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. The development environment is clear and makes developing almost enjoyable.

- **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**: Alice places limitations on the types of VR applications that can be developed. It would be difficult to create applications that deal with large amounts of data and require updating the geometry each frame. It would also be difficult to create any application that needs to have complete control over the geometry at the polygon and vertex level. This means, for example, that Alice may not be well suited to creating scientific applications.

References

[5] R. Pausch, et al. "A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality". May, 1995. IEEE Computer Graphics and Applications

Alice homepage. http://www.alice.org

3.3 Avocado

Summary

Avocado is a VR development environment created at GMD (German National Research Center for Information Technology). It is based on Iris Performer and therefore only runs on SGI platforms. Avocado 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

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

Platform

Avocado is available only on SGI machines.

Supported VR Hardware

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

Description

Avocado 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 extendable. The system is also designed to allow rapid prototyping for quick development and testing of applications. In addition, Avocado supports the development of distributed applications.

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

Avocado 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 Avocado object encapsulates its internal state in fields. Avocado 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.

Avocado 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 Avocado to define very complex behaviors very easily through connected networks of objects.

In addition to nodes, Avocado 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 Avocado 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. Avocado 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 Avocado 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, Avocado can be extended with entirely new nodes, sensors, and services to create new tools. Some examples of tools that have been developed for Avocado 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 Avocado system are mapped to a scripting language, Scheme. This allows Avocado 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.

Avocado 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. But because of the way Avocado 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 for very powerful applications to be easily created. 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 Avocado is based on Iris Performer, it only runs on SGI platforms

- **Scripting**: The choice of Scheme as a scripting language can be a hindrance because Scheme is not a well known language. In addition, functional languages can be difficult to learn and understand quickly.

References

[6] "Virtual Spaces: VR Projection System Technologies and Applications", Tutorial Notes, Eurographics '97, Budapest 1997, 75 pages. http://viswiz.gmd.de/~eckel/publications/eckel197c/eckel97c.html

GMD homepage. http://viswiz.gmd.de/

3.4 CAVE Library

Summary

The CAVE Library was designed by Carolina Cruz-Neira at the University of Illinois at Chicago's Electronic Visualization Laboratory (EVL) as part of her Ph.D. thesis[7]. It provides a fairly low-level API for creating VR applications for projection-based systems.

Availability

The CAVE Library is now commercially available from Pyramid Systems, Inc. For information, consult their home page at http://www.pyramidsystems.com/ .

Platform

The CAVE Library is only available for SGI computers.

Supported VR Hardware

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

Supported tracking devices include the Ascension Technologies, Inc. Flock of Birds, Spacepad, and MotionStar, and the Logitech tracker and 3D mouse.

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 is based on OpenGL for graphics rendering.

A running CAVE Library application is composed of several processes for handling devices, networking, and so on. Most importantly, a display process is created 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 which 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, a display callback is called by each display process. 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 scenery. After the display callback is called, the library 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 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 must be shared (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 Performer and Inventor software. Additionally, a version of the library designed for display and interaction with VRML models has been announced.

Strengths

- **Hardware Independence:** The CAVE Library shields the developer from most of the details of VR hardware.

- **Scalability:** The CAVE Library includes a simulator mode for rapid development of applications and running them without any of the special hardware. It is also capable of both distributed and non-distributed operation.

- **Distributed Applications:** The display processes for CAVE Library applications can be distributed across multiple machines, to take advantage of extra processing power and additional graphics pipelines.

Limitations

- **Cross-platform Support:** The CAVE Library is not a cross-platform solution, being 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.

- **Ease of use for Advanced Applications:** The CAVE library sometimes forces the user to be aware of and deal with the details of multiple processes and 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.

References

Pyramid Systems home page: http://www.pyramidsystems.com/ .

Electronic Visualization Laboratory: http://www.evl.uic.edu/EVL/index.html .

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

[8] C. Cruz-Neira, T. DeFanti, and D. Sandin. "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE". ACM SIGGRAPH '93 Conference Proceedings. August 1993. Pp. 135-142

3.5 dVISE

Summary

dVise and its components provide a high-level environment for developing VR applications. It emphasizes importing CAD data to create virtual representations of products for design and evaluation.

Source

The dVISE family of products is available commercially from Division, Inc. Additional information on the product line is available at their web site, http://www.division.com .

Platform

Division aims its software at a wide range of capabilities, from 2D and 3D desktop systems to fully-immersive environments. The software itself is available for SGI, Sun Microsystems, and Hewlett Packard UNIX workstations, as well as Windows NT-based PCs.

Supported VR Hardware

At the desktop level, dVISE supports a 2D viewing and navigation interface through a Netscape Navigator plugin called dv/WebFly. For a more interactive desktop or large-screen experience, 3D viewing is supported with stereo shutter glasses, using a spaceball or mouse for navigation. dVISE also supports fully immersive environments using equipment such as head-mounted displays, the CAVE, the Fakespace BOOM, and the Immersive Workbench.

dVISE is heavily multithreaded, and supports both distributed applications and networking between applications running at distant sites.

Description

dVISE's emphasis is on Interactive Product Simulation (IPS), and this is the key influence on its APIs and the underlying dVS run-time environment. The goal of IPS is to create a virtual representation of some product – a car, an engine, or any other sort of machine or object – and allow the user to interact with it. Division suggests uses such as prototyping, conducting online design reviews, or even showcasing finished products to consumers (for which the web-based viewing tool is particularly useful).

dVISE presents the creator of a virtual environment with a set of very high-level interfaces. In particular, it uses a simple scripting language to shield the developer from needing to do actual programming to create objects (called "assemblies") and define their interactions.

The geometry of assemblies is most often created using CAD tools. This lets the developer use whichever tool he chooses, or import data directly from the product design team's work.

The dVISE system can import and translate many such programs' file formats, including Autodesk's dxf, VRML, Inventor, MultiGen's flt, Wavefront's obj, STEP, and Pro/Engineer's native format. The conversion tools also allow the developer to add color, texture, and lighting information to the geometry.

Object definitions and interactions are stored in ASCII text files written in a simple event-based scripting language. The definition of an assembly can include a reference to its geometry description, information about its position and orientation, definitions of child assemblies, and instructions for handling events. The dVS system includes modules to handle object physics and collision detection, and these can, for example, send a "pick" or "touch" event to an object. Here is a small example for a light switch [9]:

```
Assembly (Name=switch) {
   Visual { Geometry { "switches/rocker" }}
   Orientation { -10, 0, 0}
   Event {
      Create { dvAssign ("%state", "Off"); }
      Touch { dvCallElse ("seq (%state,  Off) ",
                              *, On, *,  Off);
                  dvAssemblyEvent (light, toggle);
            }
      On { dvAssemblyOrientation (., 10, 0, 0);
            dvAssign ("%state", "On");
         }
      Off { dvAssemblyOrientation (., -10, 0, 0);
            dvAssign ("%state", "Off");
         }
   }
}
```

This compact piece of code defines a switch, with its geometry stored in "switches/rocker," and with an initial rotation downwards to represent the off position. Then four event-handling functions are declared. "Create" and "Touch" are standard events generated by the dVS runtime, and "On" and "Off" are custom events created by and for this object. When the switch is created, the Create event handler is called, setting a local variable called state to "Off". The Touch handler is called when the user touches the switch, and does two things. First, the light sends itself either an "On" or "Off" event (depending on the value of the variable state). Then it sends a "toggle" event to an assembly called "light" (which presumably has an event handler defined for the custom event "toggle," just like switch does for "On" and "Off"). Finally, the event handlers for "On" and "Off" change the orientation of the assembly so that the switch will appear to have been flipped, and then update the value of state.

The light switch is only a simple example, and does not demonstrate the full power of the dVISE scripting language. A few features of particular interest are:

- The ability to use any assembly definition as a template to create new, slightly different, assemblies.

- The ability to define constraints on object motion.

- The ability to do animations (via the "Tick" event), and the ability to define key frames for animations.

Strengths

- **High-level Scripting Interface:** dVISE's high-level interface can decrease development time, and also shields developers from many of the painful details of the underlying system. Because the scripts and the imported object geometry information are stored in system-independent files, no rewriting or recompiling is needed to port the application to other supported platforms. Further, distribution and networking is invisible to the developer (networking agents on each machine update the others about generated events and data updates; the developer need not be aware which machine an event was generated from).

- **Product Simulation:** Division emphasizes dVISE's use for product simulation and engineering reviews. dVISE's ability to import data from CAD programs is very powerful and flexible. Of particular interest is the ability to use networked systems to allow groups separated by distance to work together in a simulated world and perform collaborative design reviews.

- **Portability:** dVISE is available on a particularly large number of different platforms.

- **Graphics Scalability:** One unique feature of the dVS run-time is that different platforms can have very different levels of rendering power without special concern on the application's part. For example, a high-end workstation might use textured, Phong-shaded, antialiased rendering, while a PC might just display smooth-shaded polygons.

Limitations

- **Custom Language:** dVISE's scripting language, although fairly simple and well-suited to its intended purposes, is still another language with another set of commands and syntax rules for a developer to learn. However, Division has made GUI-based tools to simplify the development process.

- **Highly Specialized:** Because of dVISE's strong emphasis on product simulation, it may not be well-suited for very different tasks, such as those for which loading fixed models is not helpful.

References

Division homepage: http://www.division.com

[9] S. Ghee, "Programming Virtual Worlds." ACM SIGGRAPH '97 Course Notes, 1997.

[10] "Building and Interacting with Universal Virtual Products." White Paper, http://www.division.com/5.tec/a_papers/uvp.htm .

3.6 Lightning

Summary

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

Source

Lightning is under development at Fraunhofer Institute for Industrial Engineering. For availability and technical information, consult their web page at http://vr.iao.fhg.de/vr/projects/Lightning/OVERVIEW-en.html .

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.

In theory, 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, there is a Device Manager that controls all the sensor objects.

Output is controlled by various Render Managers – "render," in this case, used in a very general sense. For example, audio objects are rendered by the Audio Render Manager. A Visual Render Manager exists based on SGI's Performer software. The system has been 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.

- **Performer-Based:** The current SGI version of Lightning uses Iris Performer for graphics rendering. This results in a very high level of graphics performance.

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.

References

Lightning home page: http://vr.iao.fhg.de/vr/projects/Lightning/OVERVIEW-en.html .

J. Landauer, R. Blach, M. Bues, A. Rösch, and A. Simon, "Toward Next Generation Virtual Reality Systems." *Proc. IEEE International Conference on Multimedia Computing and Systems*, Ottawa, 1997.

R. Blach, J. Landauer, A. Rösch, and A. Simon, "A Highly Flexible Virtual Reality System." 1998.

3.7 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.

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. All others should contact the University for licensing information. More information, including licensing information and news, is available at the MR Toolkit home page, http://www.cs.ualberta.ca/~graphics/MRToolkit.html .

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 spaceballs 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 interprocess 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 for 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* [14] 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+ [15] 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) [16] 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.

References

MR Toolkit home page: http://www.cs.ualberta.ca/~graphics/MRToolkit.html

MRObjects home page: http://www.cs.ualberta.ca/~graphics/mrobjects/

[13] C. Shaw, M. Green, J. Liang, and Y. Sun, "Decoupled Simulation in Virtual Reality with the MR Toolkit." *ACM Transactions on Information Systems*, Volume 11, Number 3: 287-317, July 1993.

[14] C. Shaw and M. Green, "The MR Toolkit Peers Package and Experiment." *IEEE Virtual Reality Annual International Symposium (VRAIS '93)*, 463-469, September 1993.

[15] S. Halliday and M. Green, "A Geometric Modeling and Animation System for Virtual Reality." *Virtual Reality Software and Technology (VRST 94)*, 71-84, Singapore, August 1994.

[16] Q. Wang, M. Green, and C. Shaw, "EM – An Environment Manager for Building Networked Virtual Environments." *IEEE Virtual Reality Annual International Symposium*, 1995.

3.8 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.

Availability

WTK is a commercial VR development environment available from 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 huge range of devices. A full up-to-date listing is available at their web site. (http://www.sense8.com)

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 in 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.

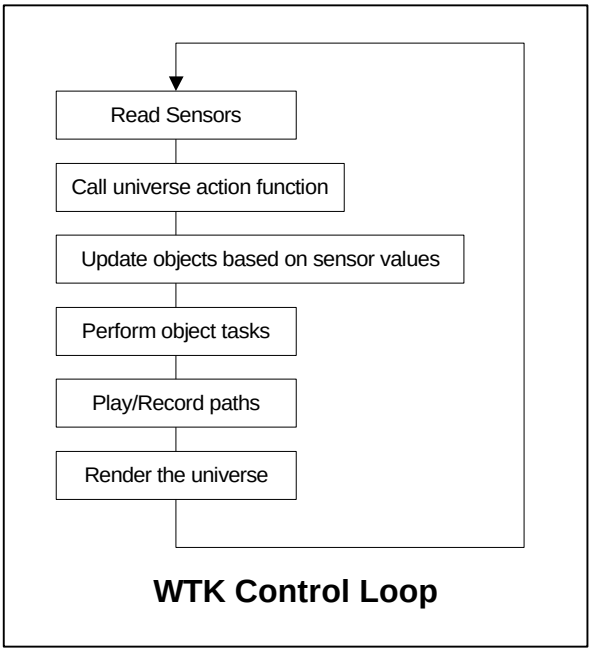
WTK provides functions that allow loading of many database formats into WTK. It includes loaders for many popular data file formats. WTK also allows the user to edit the scene graph by hand if that level of control is needed. Developers can create geometry on the vertex and polygon levels or they can 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:



WTK Control 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. But 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

- **Widely Used**: WTK is a highly used 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 if there is a hardware change.
- **Performance**: WTK does not perform as well as some other VR libraries, most notably the libraries based upon Iris Performer.

References

[17] "WorldToolKit Release 8: Technical Overview", http://www.sense8.com

4. VR Juggler

Summary

The VR Juggler is a VR library under development at the Iowa Center for Emerging Manufacturing Technology (ICEMT) at Iowa State University.

Availability

VR Juggler will be available through ICEMT.

Platform

Primary development for VR Juggler is being done on the Silicon Graphics platform. However, our goal is to provide support for multiple platforms; our current porting priorities are HP-UX and Windows NT. As it is developed, the library code is being ported to and tested on each of these three platforms.

Supported VR Hardware

One of our primary objectives is to support projection-based systems such as the C2, a CAVE-like device at ICEMT. We use general definitions of display surfaces, which allows the library to be used with almost any screen configuration. This same interface also supports workstation displays and HMDs.

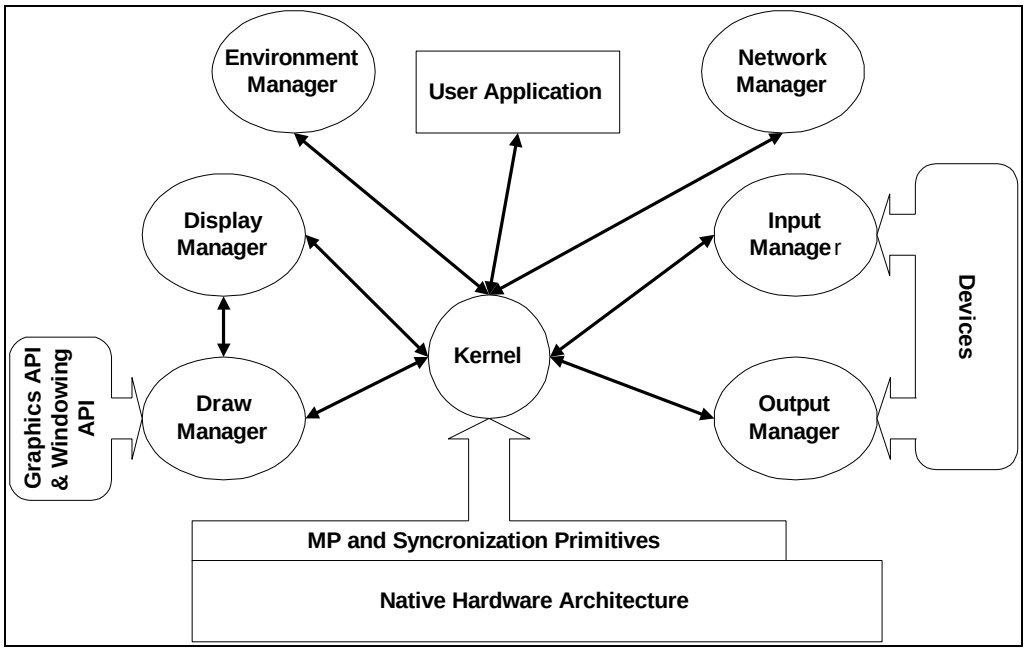
Our current set of tracking classes supports the Ascension Technologies Flock of Birds, Logitech 3D mouse, and Fakespace BOOM. We also have support for Immersion boxes, Virtual Technologies CyberGloves, and Fakespace PINCH gloves.

Objectives

The goal of this library is to address the needs outlined in the introduction. We want to produce a framework for the next generation of VR applications, a library that addresses all of the needs of VR application development in a flexible and extendable way.

Overview

VR Juggler has been designed as an object-oriented system, divided into distinct and well-defined components called *Managers*. Each manager is designed to encapsulate and hide specific system details. There are managers to deal with windowing systems, to interface with graphics libraries, to encapsulate input devices, and to configure and control the system. A kernel unites the managers and brokers all communication within the system.



Managers

Input

The *Input Manager* controls all manner of input devices - trackers, joysticks, mice, keyboards, etc. These devices are divided into different categories, including position devices (such as trackers), digital devices (such as a mouse button), and analog devices (such as a steering wheel or joystick). Applications can request access to certain devices (e.g. "Give me the position tracker for the user's head"), but a layer of abstraction is always maintained between the application and the device processes. This is what enables us to reconfigure devices during execution without disturbing the application. For example, a troublesome tracker can be restarted or replaced with a keyboard-based position simulator; no special application code is required to support this.

Output

The *Output Manager* controls all the output devices in much the same way that the Input Manager controls the input devices. When applications request access to a desired output device, an abstract handle is returned whereby the application can use the output device in whatever way it sees fit.

Draw

Draw Managers are designed to encapsulate specific graphics APIs, such as OpenGL and Iris Performer. They handle the details of setting up viewing parameters for each frame, calling the application-supplied drawing functions, and controlling stereo viewing. The Draw Manager also manages API-specific windowing and graphic context creation.

Display

The *Display Manager* encapsulates all the information about graphics windows. This includes information such as size, location, graphics pipeline, and the viewing parameters being used. The Draw Manager uses this information to configure the instantiated windows and rendering contexts. The Display Manager allows the system to add and remove displays at run-time. For example, when a display is added from the graphic interface, the Environment Manager passes a new display to the Display Manager. It is the Display Manager's responsibility to alert any other Managers in the system about the new addition.

Kernel

Finally, the *Kernel* controls the entire run-time system and brokers all inter-Manager communication. Because of this, the individual Managers are loosely coupled and very independent. This independence is an aid to development because of the flexibility it gives to the design. Changes to one part of the system, such as the addition of a Draw Manager for a new graphics API, have very little effect on the rest of VR Juggler.

Environment

The *Environment Manager* is VR Juggler's run-time control system. It presents a graphical interface written in Java, which communicates with the library's Environment Manager object through a network connection. The Environment Manager supplies data to the Graphic User Interface (GUI) and passes on instructions from the GUI to the rest of the system. From this interface, a user can view and dynamically control every aspect of a running application. To assist in troubleshooting performance issues, the Environment Manager includes a real-time performance display.

Configuration

The *Configuration Manager* is a highly-extensible database of configuration information. It provides access to all the various types of information needed to set up a VR environment - everything from tracker serial port speeds to monitor display modes. Configuration information is categorized into "chunks" for easier access - display chunks, tracker chunks, C2 chunks, HMD chunks, and so on. Applications can also use this system, adding their own chunks of data that can be loaded with the same interface and edited with the same graphical tools.

Network

The *Network Manager* is a generic object-oriented interface to the networking abstraction of the system. Because the network manager is an abstraction, it may use any networking method to distribute the data. For example, it might use a simple sockets based protocol or it may be implemented on MPI.

The network manager provides the system with all necessary networking for application and kernel use. The network manager can be used directly by an application, or an application can use other generic distribution methods provided by the library. In either case, all of the networking goes through the network manager.

The Application

The application only has access to the Kernel. The Kernel provides all information that the user application may need. Each graphics API provides a different application object template, so that the application is customized for the features of that API. For instance, an Iris Performer application may have a function that returns the base scene node for the library to render in each channel.

System Primitives

VR Juggler is implemented on top of a set of low-level objects that control process management, process synchronization, and memory management. This allows VR Juggler to be easily ported to other hardware architectures by reimplementing these classes to take into account any hardware-specific issues. Because the low-level primitives are extended for each system, VR Juggler can be adapted to many different architectures while achieving maximum performance on each one.

How VR Juggler Addresses the Needs of VR Development

The introduction to this paper included a list of fifteen things to consider when looking at VR development environments. In describing the available environments, we attempted to at least touch on all these issues. Since VR Juggler has not been published as extensively as these other systems, we would like to take this opportunity to include a little more detail, and quickly state the exact positions taken on each of these issues in our own development effort.

Cross-Platform Development

We are doing our development primarily on SGI workstations. 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 us to easily port the library to other platforms by replacing the system-specific classes derived from the abstract bases. Currently the library has support for SGI, HP, and Windows NT.

VR Hardware, Abstraction, and Run-time Flexibility

In order to support a wide range of VR devices, the display manager uses a generic surface description that can be used for any projection surface. Currently, we use VR Juggler with projection-based systems such as the C2 or CAVE. It also has support for HMDs. By using a generic display device description, we can easily configure the library to run on any VR display device.

Input devices are hidden behind generic Proxies, which give the application 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, the library can immediately begin to take advantage of it.

This Proxy system gives VR Juggler a high degree of run-time flexibility. The physical device classes can be moved around, removed, restarted, or replaced without affecting what the application sees. The proxies themselves remain the same, even when the underlying devices are changed.

The Environment Manager GUI will, when completed, offer an easy-to-use interface for reconfiguring, restarting, and replacing devices and displays at run-time. This will allow users to interactively reconfigure a VR system while an application is running. This ability also increases the robustness of applications because it allows devices to crash or otherwise die without taking down the entire application.

Locally Distributed Applications

Support for distributed applications is planned for the second major release of VR Juggler, tentatively scheduled for late 1998. We are currently evaluating various methods of application distribution that will be transparent to developers.

Distributed Environments

Support for distributed environments will come from the Network Manager. Each high-level API will need to distribute applications differently depending upon how they store the application data. For example, when using scene graph based graphic APIs it is possible to distribute an application by sharing the scene-graph data in a way similar to Avocado. If the API does not have any direct support for VR data structures, as in OpenGL, then the application is responsible for distributing all the data correctly. Because each high-level API will need to distribute an application differently, VR Juggler will just provide cross-platform support for networking.

Rapid Prototyping

In addition to supporting non-stereo workstation displays, VR Juggler will include simulators for each class of input devices. Since the devices are hidden behind Proxies, the application will be unaware of whether it is running in a full VR setup or a stripped-down environment. If the developer wants to experiment with different types of devices, then she can use the GUI interface to switch devices at run-time. This can allow rapid testing of many interaction methods.

High-level and Low-level Interfaces

VR Juggler does not include its own modeling language or have built-in support for collision detection and object behavior, though such a high-level system could be implemented using VR Juggler as its base. However, it does offer a more abstracted view of devices and displays than many of the low-level development environments that we have discussed.

Since VR Juggler does not define its own graphics API, it can not easily support a scripting language for writing applications. A tool built on top of VR Juggler could provide this level of support.

Graphics Interfaces

VR Juggler currently supports OpenGL and SGI's Performer software. It has been designed to be extended though, and allows the addition of new graphics APIs without major changes to the library. In order to add a new API, the developer only needs to create a new draw manager class and a new application framework class. Since all managers in the library interact with the abstract interface of the draw manager, the rest of the library would be unaffected.

Interaction

Support for interaction in VR Juggler is fairly low-level; the developer must write code to look at the state of the input devices and decide what affect they will have on the environment. There is no event model and no built-in way to associate behaviors with graphic elements. These types of features could be supported in a higher level API running on top of VR Juggler.

APIs and Languages

VR Juggler applications are written in C++, as is the library itself. We have attempted to take full advantage of object-oriented design principles while creating the API.

Extensibility

Adding new devices of an already supported general type (such as new position inputs, or new displays) is designed to be very easy and transparent to applications. This is because the library uses generic base class interfaces to interface with all objects in the system. This allows the instantiated objects to be implemented in whatever way is desired.

Minimal Limitations

VR Juggler gives the developer full access to the graphics API. Therefore, the developer has total freedom when defining the interactions between the user and objects in the virtual world. We try not to limit the application developer in any way.

Performance Monitoring

VR Juggler has built-in performance monitoring capabilities. It is able to record data about the time spent by various processes, and it can acquire performance data for the underlying graphics hardware (as available). It also has the ability to timestamp data to measure latency (for example, the latency between the generation of tracker data and its use to generate the display). The GUI includes the ability to display the performance data at run-time.

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.

Research versus Commercial Sources

VR Juggler is the product of a research environment. As such, there is no paid technical support staff, and documentation is not the developers' highest priority. There are some advantages, though. The developers may be interested in collaborating with outside sources, and the source code may be made available, allowing application developers to modify the Juggler architecture for their own environments.

Current State

As of this writing (April 1998), an initial version of VR Juggler is being tested and debugged. This version supports single-machine applications on SGI platforms using either OpenGL or Performer. Performance monitoring and dynamic application control through the GUI is being implemented. Support for distributed applications will be enabled in the second major release, estimated for completion later in 1998.

Conclusion

The VR Juggler architecture is a flexible and efficient API for VR application development. Our new object-oriented design is very portable, and presents a simple, easy-to-learn interface to the developer. Performance is also one of our top priorities. We believe that Juggler's design has the ability to equal the performance of the currently available VR environments. We are analyzing performance at each step of the design and implementation process to make sure we provide an efficient implementation. We hope that VR Juggler's use will lead to more portable and scalable VR applications, and allow VR developers to concentrate on the worlds they want to create, instead of the systems on which they run.

Bibliography

[1] http://www.apple.com/quicktime/qtvr/index.html

[2] http://cosmosoftware.com/developer/moving-worlds/

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

Iris Performer

[4] J. Rohlf and J. Helman, "IRIS Performer: A High-Performance Multiprocessing Toolkit for Real-Time 3D Graphics." Proc. Siggraph 94, ACM Press, New York, 1994, pp. 381-394.

Alice

[5] R. Pausch, et al., "A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality." IEEE Computer Graphics and Applications, May, 1995.

Avocado

[6] "Virtual Spaces: VR Projection System Technologies and Applications." Tutorial Notes, Eurographics '97, Budapest 1997, 75 pages. http://viswiz.gmd.de/~eckel/publications/eckel197c/eckel97c.html

CAVE Library

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

[8] C. Cruz-Neira, T. DeFanti, and D. Sandin, "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE." ACM SIGGRAPH '93 Conference Proceedings. Pp. 135-142, August 1993.

DVISE

[9] S. Ghee, "Programming Virtual Worlds." ACM SIGGRAPH '97 Course Notes, 1997.

[10] "Building and Interacting with Universal Virtual Products." White Paper. http://www.division.com/5.tec/a_papers/uvp.htm

Lightning

[11] J. Landauer, R. Blach, M. Bues, A. Rösch, and A. Simon, "Toward Next Generation Virtual Reality Systems." *Proc. IEEE International Conference on Multimedia Computing and Systems*, Ottawa, 1997.

[12] R. Blach, J. Landauer, A. Rösch, and A. Simon, "A Highly Flexible Virtual Reality System."

MR Toolkit

[13] C. Shaw, M. Green, J. Liang, and Y. Sun, "Decoupled Simulation in Virtual Reality with the MR Toolkit." *ACM Transactions on Information Systems*, Volume 11, Number 3: 287-317, July 1993.

[14] C. Shaw and M. Green, "The MR Toolkit Peers Package and Experiment." *IEEE Virtual Reality Annual International Symposium (VRAIS '93)*, 463-469, September 1993.

[15] S. Halliday and M. Green, "A Geometric Modeling and Animation System for Virtual Reality." *Virtual Reality Software and Technology (VRST 94)*, 71-84, Singapore, August 1994.

[16] Q. Wang, M. Green, and C. Shaw, "EM – An Environment Manager for Building Networked Virtual Environments." *IEEE Virtual Reality Annual International Symposium*, 1995.

WorldToolkit

[17] "WorldToolKit Release 8: Technical Overview", http://www.sense8.com