VR Juggler: A Virtual Platform for Virtual Reality Application Development

Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert
Albert Baker, Carolina Cruz-Neira
Virtual Reality Applications Center, Iowa State University
allenb, cjust, patrick, kevn, carolina, baker@vrac.iastate.edu

Abstract

Today, scientists and engineers are exploring advanced applications and uses of immersive systems that can be cost-effectively applied in their fields. However, one of the impediments of the wide-spread use of these technologies is the extensive technical expertise required of application developers. A software environment that provides abstractions from specific details of hardware configurations and low-level software tools is needed to provide a common base for the prototyping, development, testing and debugging of applications. This paper describes VR Juggler, a virtual platform for the creation and execution of immersive applications, that provides a virtual reality system-independent operating environment. We will focus on the approach taken to specify, design, and implement VR Juggler and the benefits derived from our approach.

1. Introduction

Today's virtual reality (VR) application developers are expected to have expertise in the problem domain, and they also must have expertise in the development of sophisticated software systems that utilize features such as multiple processes, message passing, shared memory, dynamic memory management, and a variety of process synchronization techniques. They may also be concerned with very low-level issues such as device drivers for particular VR I/O devices or techniques to generate multiple stereoscopic views.

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

Currently there are a variety of software toolkits for virtual reality development [2], but most of them are either focused to specific uses and requirements or are monolithic packages that offer little flexibility to developers. EAI's Wordltoolkit [4] is a good tool, particularly because it enables novice VR developers to quickly prototype an application, however, it expects the application data to come from CAD packages, providing little support for scientific data generated from direct computations. Another popular commercial tool is the CAVE Library [3]. It is a low-level library geared to multiple screens application development. Its main limitation is that it is statically tied to a set of specific hardware configurations, making it impossible to add new configurations or components without making significant changes to the library's core code. Other popular toolkits for VR application development include the MRToolkit [5], and dVise [6].

This paper presents VR Juggler, an extensible virtual platform for VR application development. It provides an object-oriented development environment to support the efficient development of time-critical, interactive, immersive applications independently of the underlying technologies. We discuss the VP concept in the context of prototyping, debugging, and running immersive applications and focus on the approach taken to specify, design, and implement VR Juggler.

2. A virtual platform for VR

A VP provides a development and execution environment that is independent from hardware architecture, operating system (OS), and available VR hardware configurations. It provides a unified operating environment in which developers can write and test applications using the available resources while guaranteeing the portability of the application to other resources.

A VP must address the following key technical challenges to allow developers to build advanced virtual environments in complex VR systems:

1. Abstract the complexities of VR systems with local and distributed components

Proceedings of the Virtual Reality 2001 Conference (VR'01)
0-7695-0948-7/01 $17.00 © 2001 IEEE

2. Provide for scalability of VR systems in terms of the number of display surfaces, computer systems, human/computer interface equipment, networks, and software tools.
3. Provide flexibility to define a variety of hardware configurations
4. Allow for run-time changes in the hardware and software configuration of the environment
5. Provide the ability of running multiple applications simultaneously
6. Provide tools for evaluating and tuning the performance impact of the different layers that form an application

The rest of this section introduces VR Juggler's approach to meet these challenges . More technical detail will be provided in subsequent sections of this paper.

2.1. Operating system independence

Independence of the underlying technologies is one of the principal requirements to support cross-platform application development. It is also critical to release application developers from the responsibility of tuning the performance of their application to the specific platforms being used.

To permit system independence, VR Juggler isolates system-specific dependencies behind a simple VP interface, implemented as a kernel. In VR Juggler, the interface of the kernel object defines the VP for VR development and encapsulates the system-specific details of VR Juggler so the application does not have to depend upon those details.

2.2. Device abstraction

A VP provides abstractions for VR devices based on their functionality. VR devices can be abstracted into several basic classes of input and output like positional, orientation, digital, analog, glove, and gesture.

VR Juggler defines base class interfaces for each device class in order to make all devices of a given class look the same to the developer. For example, in any application, a simulated positional device driver has the same interface as a magnetic tracker. Since VR Juggler only interacts with devices through this interface, application developers are not tied to specific hardware.

2.3. Operating environment

The VP provides a simple operating environment for VR applications. This operating environment allows for multiple running applications and components.

VR Juggler allows environments to have several highly specialized applications running simultaneously by considering each application an instance of the application object. This is similar to the desktop paradigm: in a typical desktop session, users have many individual programs such as web browsers, e-mail readers, program launchers, chat clients, and text editors active simultaneously, and then switch among them to complete a task. VR Juggler allows this same type of multi-program productivity with VR applications. In a typical immersive working activity, there could be one tool that allows launching new applications, another may allow the user to receive communications from people in other virtual environments, while yet another takes verbal notes about the data in running applications. There are a wide range of utility applications that can be helpful to users in a virtual environment [7].

2.4. Support for multiple graphics APIs

Another key component of any VR application is the graphics Application Programming Interface (API) used to create the visual elements. A VP must allow developers to use any graphics API they choose.

VR Juggler supports multiple graphics APIs by encapsulating all graphics API-specific behavior in draw managers. Because the kernel represents only the part of the VP that hides system details, we must add an additional interface that is specific for each supported graphics API. The draw manager's interface can be considered a smaller VP that presents the application with an API-specific abstraction.

2.5. Overview of VR Juggler's virtual platform

The application interface to the VP (Figure 1) consists of the kernel interface that provides the hardware abstraction for the VP and the draw manager that provides the abstraction for the graphics API.

Since the kernel controls all VR Juggler components except the graphics API, its interface provides the VP for the hardware-specific details of the environment. Because the kernel interface is the only way the application accesses the hardware, it is possible to extend or change the implementation details of any component of VR

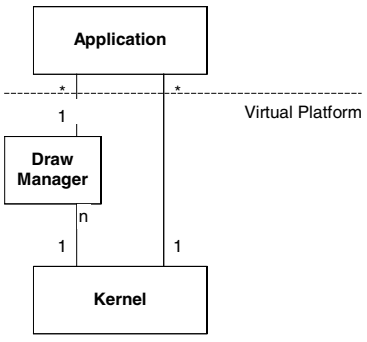

Figure 1: Application/VP interface

Juggler as long as the kernel interface remains the same.

This combination of the kernel and draw managers provides VR Juggler applications with system independence. Once a VR Juggler application is written for one system, it can run on any other system supported by VR Juggler.

3. Application development

This section describes VR Juggler from an application developer's perspective. We start by describing the rationale for making the application an object and describe how an application object is defined. We then discuss the benefits of application objects.

3.1. Application object

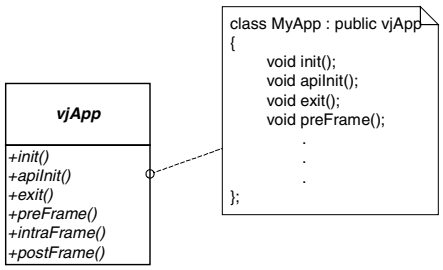

Figure 2: Application object

In VR Juggler, user applications are *objects* (see Figure 2). VR Juggler uses the application object to create the VR environment with which the user interacts. The application object implements interfaces needed by the VP to create the virtual environment. (An interface is a collection of operations used to specify a service of a class or a component [8].) The kernel maintains control over the environment and calls the methods defined in the application interface. When the kernel calls the application's methods, it temporarily gives up control to the application object so the application can execute.

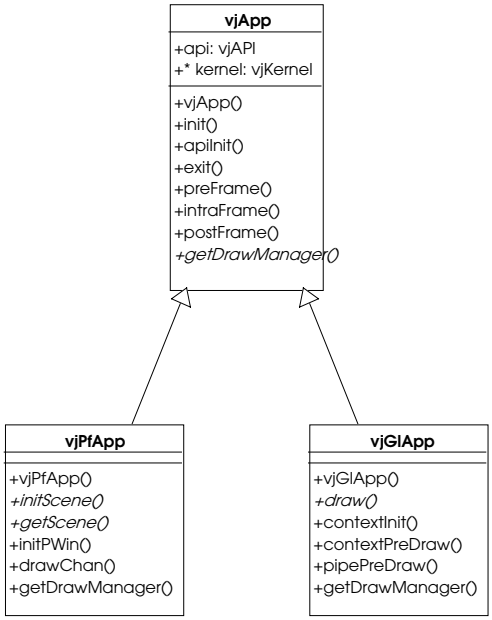

Figure 3: Application class hierarchy

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

The VR Juggler kernel and draw manager communicate with an application through pre-defined interfaces that all applications must implement (see Figure 3). When the kernel and draw manager need the application to do processing or to return information, they call the member functions of the interface. VR Juggler includes base classes for the interfaces needed for the kernel and draw manager interaction with the application. An application inherits from and extends these base classes in order to realize the required interfaces.

The kernel calls each of the member functions based on a strictly scheduled frame of execution[1]. During the frame of execution, the kernel calls the application methods and performs internal updates (see Figure 4). Because the kernel has complete control over the frame, it can make changes at predefined "safe" times when the application is not doing any processing (see *checkForReconfig()* in Figure 4). During these "safe" times, the kernel can change the VP configuration as long as the interface the application sees remains the same.

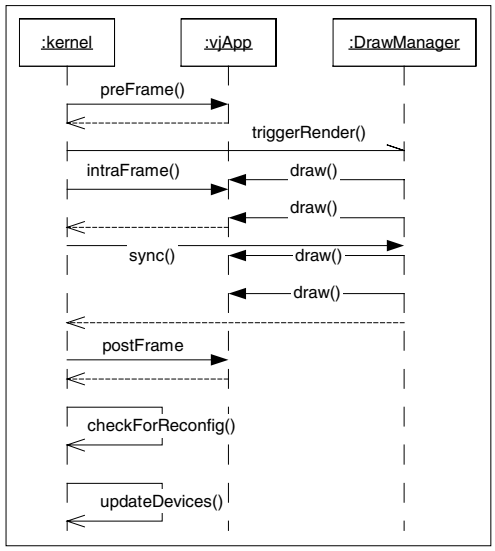

Figure 4: Kernel frame

[1] A frame of execution in VR Juggler is the time it takes the to complete one set of updates within the virtual platform. This includes the time to render the environment, to resample the devices, and to do any processing necessary for that frame.

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

3.2 Benefits of application object

The most common approach for VR application development is to have the application define the main function and have the application call library functions when needed. The library in this model only executes code when requested to do so by the application because the application is in control of the main thread of execution. This model makes the application developer responsible to coordinate the execution of the different components of the VR system and, therefore, responsible to create efficient, complex applications.

VR Juggler, being a VP, does not use this model because it needs to maintain control over the different components to provide the flexibility needed to make changes to the VP at run-time.

By controlling execution, the kernel always knows the current state of the applications, therefore it can safely manage the run-time reconfigurations of the virtual environment. Under this model, nearly every parameter to configure the execution environment can be changed at run time. It is possible to switch applications, start new devices, reconfigure devices, and send reconfiguration information to the application object.

Application objects lead to a robust architecture as a result of low coupling and well-defined inter-object dependencies. The application interface defines the only communication path between the application and the VP. By restricting interactions to the interfaces of the kernel, draw manager, and application, the system restricts object inter-dependencies to those few interfaces. This decreased coupling allows changes in the system to stay local. Changes to one object will not affect another unless the change involves the interface of one of the objects. This leads to more robust and extendable code.

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

Since applications use a distinct interface to communicate with the VP, changes to the implementation of the VP do not affect the application. This makes it simple to make significant changes to the implementation of the VP without affecting any applications that currently run on the platform. These changes could include bug fixes, performance tuning, new device support, or any number of other changes. We believe that this ability to modify the system's behavior at run-time is one of the major strengths of VR Juggler.

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

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

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

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

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

Figure 5: Sample application object

4. VR Juggler design

This section covers the design of VR Juggler in more detail. We will start by discussing the *microkernel* architecture used in VR Juggler. That is followed with a description of the internal managers that maintain the input devices, store the display information, and interact with the external GUI control system used to reconfigure the architecture. Next, we describe the external draw manager and the application class that connects to it. We will conclude with a brief discussion of two features of VR Juggler enabled by its architecture: run-time reconfiguration and multi-user applications.

4.1. Microkernel

VR Juggler's VP is based on a specialization of the microkernel architectural pattern [9]. The microkernel controls the entire run-time system and manages all communication within the system. The VR Juggler microkernel architecture (Figure 6) has a core kernel object that implements the central services needed for VR application development.

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

External managers provide an interface to the VP that is specific to the application type. Client applications communicate with the VR Juggler system through the

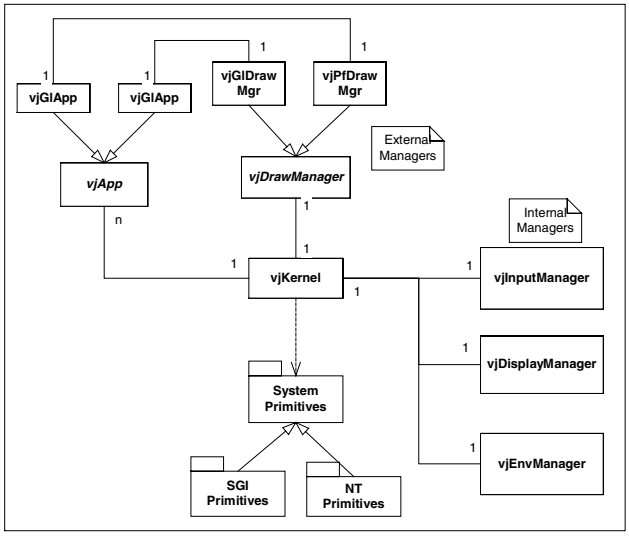

Figure 6: Microkernel architecture

interfaces of the external managers and of the kernel. Currently the only external managers are the graphics API-specific draw managers.

4.1.1. Kernel as a mediator

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

The kernel in VR Juggler acts as a mediator [10] by encapsulating how all the other managers in the system interact. There are no direct dependencies between the managers. The kernel can change the way the system frame executes without changing the way the managers behave or relate to each other.

4.1.2. Kernel portability

The VR Juggler kernel is layered on top of a set of low-level primitives that ease porting and allow for performance tuning on each hardware platform. The primitives control process management, synchronization, and other hardware-dependent issues (Figure 6). Because these primitive classes account for the majority of hardware-specific implementation differences, they ease the porting of VR Juggler to other architectures. While porting, each low-level primitive is extended and optimized in order to achieve high performance on each system.

4.2. Configuration information

All configuration information is contained within small units called *chunks*, which are divided into properties. Each property has a type and one or more values. A chunk contains all the configuration information for a particular part of the VR system or application. For example, the chunk given in Figure 7 defines configuration information for a window. It has three properties: name, size, and origin. The size property has two values of type int.

Chunk: Window		
Name	String	
Size	Int	Int
Origin	Int	Int

Figure 7: Window chunk

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

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

4.3. Internal managers

4.3.1. Input manager

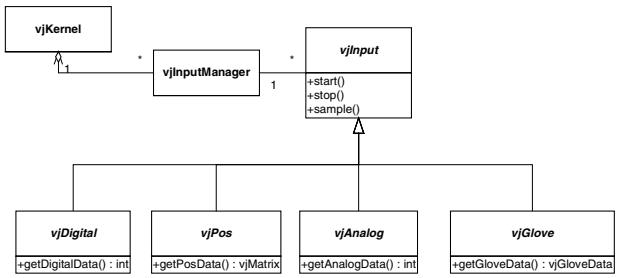

Figure 8: Input device class hierarchy

The Input Manager controls all manner of input devices for the kernel. The input devices are divided into distinct categories, including position devices (such as trackers), digital devices (such as a wand or mouse button), analog devices (such as a steering wheel or joystick), and glove devices (such as a CyberGlove™). VR Juggler defines a class hierarchy for the each of these types of input devices (see Figure 8). There is base class for all input devices that specifies the methods that must be implemented to start, stop, and update the device (see vjInput Figure 8). There is also a base class defined for each distinct category that specifies the generic interface that any device of that category must support. For example, all positional devices must support a *getPosData()* member function that returns a position matrix.

To add support for new devices, a new class is created for the specific device. The new class is derived from the base classes of the input categories that the new device can return (Figure 8). The new class has to define member functions that implement the base input device interface as well as define the methods for returning the input type of each of the parent classes.

The application uses device proxies [10] to interface with all devices. Before an application gets data from a device, it must first get a proxy to the device. The application requests the device by using the name that the device was given in the current configuration. The input manager looks up the requested device and returns a proxy to that physical device.

Device proxies allow the application to be decoupled from the devices in use. The device referred to by a proxy can change at run-time. For example, a proxy may initially be linked to a hardware tracker in the system. If the user wants to change tracking systems, then VjControl is used to point the proxy to a different tracker. The next frame, the application still uses the same proxy but the data returned is now coming from a different device. This change has no impact on the application execution since it does not know that the change has occurred. The proxy abstraction allows the configuration of the VP to change during execution without disturbing the applications.

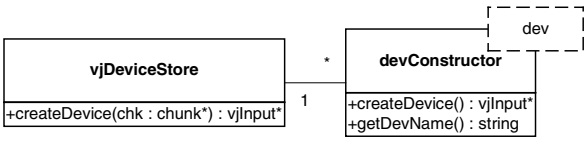

Figure 9: Device store

VR Juggler uses a device store [11] to allow the system to keep all device drivers separate from the main library and to load device drivers dynamically (Figure 9). The device store is a factory object [10] that keeps track of device constructors and the associated device drivers that have registered with the system

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

4.3.2. Environment manager

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

4.3.3. Display manager

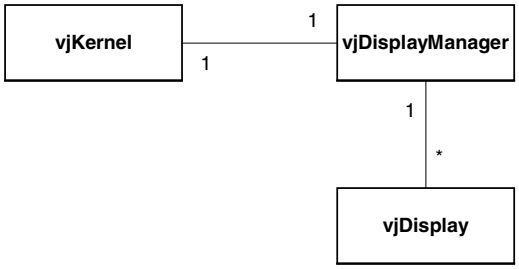

Figure 10: Display Manager

The display manager encapsulates all the information about the display windows' settings (Figure 10). This includes information such as size, location, graphics pipeline, and viewing parameters being used. The display manager is also responsible for performing all viewing calculations for its windows and is used to configure the draw manager.

4.4. External managers

4.4.1. Draw manager

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

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

4.4.2. Application

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

VR Juggler treats the application as it does any other component plugged into the kernel. The kernel keeps the application synchronized with the rest of the system by invoking application callbacks at pre-defined times during the frame of execution. Because the application plugs into the kernel, it can be removed or replaced at any time.

This allows applications to be changed while VR Juggler is active.

The base interface is extended through sub-classing to create application interfaces that are specialized for a specific graphics API. For instance, an OpenGL Performer application may have a function that returns the base scene node so the draw manager can render the graph in each channel.

4.5. Run-time reconfiguration

All VR Juggler managers support run-time reconfigurability. Run-time reconfiguration is supported using the chain of responsibility pattern [10]. Each manager supports a configuration interface that allows adding and removing of configuration chunks within the current configuration. When the kernel receives a configuration request, it forwards the request to each manager and to the current application. The receiving manager or application then processes the request or uses the same chaining method to pass the request on to its dependencies.

Input devices can be reconfigured at run-time without affecting the current application. A reconfiguration request can be sent to the input manager requesting that a specific proxy be pointed at a different device. Displays can also be added and removed at run-time. This feature is particularly useful for multi-screen VR systems because screens can be activated and deactivated as the application runs. Applications can also receive configuration information. This allows applications to respond to changes at run-time in the same way that the rest of the VP does.

4.6. Local Multi-User support

VR Juggler supports multiple users rendered from the same machine. Because the architecture of VR Juggler has no direct ties between the managers, there is no requirement that views are rendered for only one user. All that is needed to render the view of multiple users is to associate the position of the user with the window rendering their view. VR Juggler does not limit the number of trackers that can be active during an application execution, so multiple tracked views can be generated within an application.

5. Discussion

5.1 Virtual platform

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

Although there is a popular belief that object-oriented abstraction introduces severe penalties in program performance, we believe that the object-oriented design approach for VR Juggler as a VP provides the best avenue to achieve its goals. We have placed a great deal of effort on optimization of our abstraction levels to minimize the performance impact. We have conducted performance evaluations of VR Juggler applications; our results show that the overall performance is affected very little by the abstraction in the library.

5.2 Hardware abstraction

Displays are abstracted to allow support for any VR device. The display manager uses a generic surface description that allows for any number of projection surfaces. Currently, we use VR Juggler with projection-based system such as CAVEs and benches. It also has support for HMDs and desktop VR. By using a generic display description, we can configure an application to run on any VR display device.

For some types of display devices, it is necessary to change the interaction paradigm used in the application. Currently we are investigating ways to allow applications to modify their interaction methods based on device type.

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

5.3 Run-time flexibility

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

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

5.4 Performance tuning

VR Juggler includes built-in performance monitoring capabilities. These include the ability to accumulate data about time spent by various processes, performance data for the underlying graphics hardware (as available), and measure tracker latency (the time between generation of

tracker data and the display of data generated from the tracker data). VjControl can display the performance data at run-time or the performance data can be captured and analyzed later.

The environment manager allows users to 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.

5.5 Cross-platform

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

5.6 Extensible

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

6. Current state

Currently the VR Juggler VP is supported on the Irix, Linux, and Windows NT platforms. It has support for OpenGL, OpenGL Performer, and VTK. It is freely distributed under the LGPL open source license and can be downloaded at http://www.vrjuggler.org.

VR Juggler does not provide yet have a complete component based interface. This means that VR Juggler does not support binary compatibility.

7. Conclusion and future directions

The VR Juggler architecture is designed to set a new standard API for VR development. Our object-oriented design is very portable and presents a simple, easy-to-learn interface to the developer. VR Juggler provides several levels of abstraction with little or no measurable performance penalty. We hope that VR Juggler will provide a platform for a new generation of highly portable and scalable VR applications, as well as a new generation of VR developers who can concentrate on the worlds they want to create and not the systems on which they run.

We are investigating a component-based approach for the low-level components of VR Juggler. Publicly available tools such as Bamboo [12] are good candidates to provide the component-based infrastructure.

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

References:

[1] C. Cruz-Neira "A Look Behind the Scenes of Virtual Reality Applications". *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST '96. Hong-Kong, June 1996, pp. 161-162.

[2] A. Bierbaum and C. Just. "Software Tools for Application Development". *ACM SIGGRAPH 98 Course #14: Applied Virtual Reality.* ACM SIGGRAPH 98 Conference, Orlando, July 1998. pp. 3.1-3.45

[3] C. Cruz-Neira, *Virtual Reality Based on Multiple Projection Screens: The CAVE and its Applications to Computational Science and Engineering,* doctoral dissertation, University of Illinois at Chicago, Department of Electrical Engineering and Computer Science, 1995.

[4] Sense8 WorldToolkit R8 Reference Manual, 1997.

[5] C. Shawn, J. Liang, M. Green, and Y. Sun. "The Decoupled Simulation Model for Virtual Reality Systems". *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference.* May 1992, pp.321-328.

[6] S. Ghee. *Programming Virtual Worlds.* ACM SIGGRAPH 97 Conference, Los Angeles, July 1997.

[7] T. Imai, A. Johnson, J. Leigh, D. Pape and T. DeFanti "The Virtual Mail System," Proceedings of the *IEEE VR 99* , March 1999, Houston, pp. 78.

[8] J. Rumbaugh, I. Jacobson and G. Booch "The Unified Modeling Language Reference Manual," 1999

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal "Pattern-Oriented Software Architecture: A System of Patterns," 1996

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides "Design Patterns," 1995

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

[12] K. Watsen and M. Zyda "Bamboo - Supporting Dynamic Protocols For Virtual Environments," *Proceedings of the IMAGE 98 conference,* 1998