
VR Juggler 1.0 Technical Overview

\$Date: 2002/06/16 22:19:05 \$

Table of Contents

Overview	2
Fundamentals	2
What is VR Juggler?	2
Virtual Platform: A virtual platform for virtual reality application development	2
Configuration information	5
VR Juggler Microkernel	5
Modularity	6
Mediator	6
Kernel portability	6
Consequences	6
Internal Managers	7
Input manager	7
Environment manager	8
Display manager	8
External managers	8
Draw manager	8
Other external managers	8
Application objects	9
Base application interfaces	9
No main() -- Don't call me, I'll call you.	9
Benefits of application objects	10
Run-time (re)configuration	10
Config chunk handler interface	11
Pending configuration queue	12
Dependency management	12
Run-time reconfiguration discussion	14
Multi-threading	14
How does everything get started?	15
System interaction	15
How well does VR Juggler meet the design goals?	16
Hardware abstraction	16
Run-time flexibility	16
Performance tuning	16
Cross-platform	17
Extensible	17
A. Appendix: Needs of VR system	17
Primary needs	17
Performance	17
Extensibility	17
Flexibility	18
Simplicity	18
Robustness	18
Performance	19
Low latency	19
High frame rate	19
Support for hardware	20
Performance monitoring	20
Extensibility	20

Hardware abstraction	20
Simple extension	21
Do not require application changes	21
Flexibility	21
Scalability	21
Cross-platform	22
Run-time changes	22
Support use of other application toolkits	23
Do not be overly restrictive	24
Simplicity	24
Short learning curve	24
Rapid prototyping using simulation	25
Robustness	25
Failure protection	25
Maintainability and correctness	25

Overview

This document is the technical for VR Juggler (VRJ). It describes the design choices and implementation details of VR Juggler. We describe what the major components of the system are, why the components are there, and how the components are implemented and used with the rest of VR Juggler.

This guide is designed to be used by developers who want to extend VR Juggler and people who are just interested in the low-level details of the system. As such, it is expected that readers are already comfortable with the basics of VR Juggler such as writing an application and configuring the system. This guide will not describe these basic details of VR Juggler but will instead focus on the behind the scenes details of the underlying system.

Before we get start describing the system in detail, we are going to take a moment and explain what VR Juggler is and describe some of the fundamental concepts and metaphors of the system. Hopefully this will make the design more clear and provide some insight into our design philosophy.

Fundamentals

What is VR Juggler?

VR Juggler is 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.

Virtual Platform: A virtual platform for virtual reality application development

The fundamental metaphor of VR Juggler is that VR Juggler is a virtual platform for VR application development.

A virtual platform (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.

The VR Juggler virtual platform (JVP) addresses the following key technical challenges (See Appendix A, Appendix: Needs of VR system for more information about these needs)

1. Abstract the complexities of the current VR system
2. Allow the use of any graphics API
3. Provide for scalability of VR systems and resources in terms of the number of display surfaces, computer systems, human/computer interface equipment, networks, and software tools.
4. Provide flexibility to adapt and cope with a wide variety of hardware configurations
5. Allow for run-time changes in the hardware and software configuration of the environment
6. Provide the ability of running multiple applications simultaneously
7. Provide tools for evaluating and tuning the performance of the VR Juggler system and the applications that use it.

The purpose of the VR Juggler virtual platform (JVP) is to separate the hardware-dependent and hardware-independent software components of the VR software system. The virtual platform provides a simple operating environment for virtual reality application development. By using the JVP, a developer can write an application once on a local VR system and run it on any other VR system.

The JVP design has the following characteristics.

Virtual platform API

The basic JVP system () is composed of an application object, a draw manager, and the VR Juggler kernel. The interface between the application object and the JVP consists of the kernel interface that provides the hardware abstraction for the virtual platform, and the draw manager that provides the abstraction for the graphics API (Figure 1).

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

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

The virtual platform interface means that application code does not have to change when new system features are added or even when running on a different VR system. The virtual platform in VR Juggler separates the application developer from the system details that can change. Since the virtual platform consists of the kernel and the draw manager, we have the freedom to change any details of the VR Juggler system as long as the interfaces to the draw manager and the kernel remain the same. As long as the interface looks and behaves the same, the application can never see or rely upon any details hidden by the virtual platform. This provides VR Juggler applications system independence. Once a VR Juggler application is written for one system, it can run with VR Juggler on any other system.

Benefits of virtual platform

The following are the major benefits of the of the virtual platform design in VR Juggler.

- Architecture and OS independence

The JVP allows development of applications that are free from architecture and OS dependence leading to truly

cross-platform applications. This allows applications written with the virtual platform to run on architecture that the JVP has been ported to. It also allows the virtual platform to be tuned for each local platform in order to run VR Juggler applications with high-performance.

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

- Device abstraction

The JVP provides standard abstractions for many classes of VR devices such as: positional, digital, and glove. By using these common abstractions for device classes, the virtual software system hides the details of the actual devices in use. The application can make use of these device abstractions to get data from the devices.

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

- Allows for use of multiple graphics APIs

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

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

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

- Operating environment

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

Discussion

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

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

times.

Configuration information

All configuration information is contained within small units called config "chunks", and these chunks are divided into properties that hold values. Each property value has a default type associated with it. Although the properties have these default types, they are implemented as a variant data type which allows them to return data in whatever data type is needed.

A chunk contains all the configuration information for a particular aspect of the VR system or application. For example, the chunk given in Figure 2 defines configuration information for a window. It has three properties: name, size, and origin. The size property has two values of type int.

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

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

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

VR Juggler Microkernel

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

It was concluded that a specialization of the microkernel architecture [system of patterns] would be the best solution to this problem.

The microkernel controls the entire run-time system and manages all communication within the system. The VR Juggler microkernel architecture (Figure 3) has a core kernel object that implements the central services needed for VR application development: input devices, display settings, and configuration information.

- Internal Managers

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

- External Managers

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

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

Modularity

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

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

Mediator

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

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

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

Kernel portability

The VR Juggler kernel is layered on top of a set of low-level primitives that ease porting and allow for performance tuning on each hardware platform. The primitives control process management, synchronization, and other hardware-dependent issues (Figure 3). Because these primitive classes account for the majority of hardware-specific implementation differences, they can greatly ease the porting of VR Juggler to other architectures. During the porting process, each low-level primitive is extended and optimized in order to achieve high performance on each system.

Consequences

A microkernel design has several important consequences to the design as discussed in [system of patterns]

Benefits

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

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

Costs

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

Internal Managers

As stated in the section about the microkernel, internal managers are responsible for system tasks that are beyond the scope of the kernel. The internal managers capture the commonality of tasks such as managing the current input devices or tracking the current state of the configuration system. The remainder of this section describes the major internal managers.

Input manager

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 all of these types of input devices (see Figure 4). There is a base class for all input devices that specifies the methods that must be implemented to start, stop, and update the device (see `VjInput` Figure 4). 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 derive from `VjPosition` and support a `getPosData()` member function that returns a position matrix.

Adding a device

To add support for new devices, developers create a new class for the specific device. The new class must be derived from the base classes of the input types that the new device can return (Figure 4). For example, a device that can return analog and positional data is derived from `VjAnalog` and `VjPosition`. The new class has to implement the member functions of the base input device interface as well as the methods for returning the input type of each of the parent classes.

Device proxies

The application uses device proxies [Design patterns] 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 using the name given to it in the current configuration. The input manager looks up the requested device name, and returns a proxy to the corresponding physical device.

The main advantage of device proxies is that they allow the application to be decoupled from the physical device being used. The device referred to by a proxy can change at run-time. For example, a proxy may initially be linked to a hardware tracker in the system. If the user wants to change tracking systems, then the user points the proxy to a different tracker. The next frame, the application still uses the same proxy but the data returned is now coming from a different device. The proxy abstraction allows the configuration of the virtual platform to change during execution without disturbing the application.

Device store

VR Juggler uses a device store [Object store] to allow the system to keep all device drivers separate from the main library and to dynamically load device drivers at run-time (see `VjDeviceFactory` in Figure 5). The device store is a factory object [GOF] that keeps track of device drivers and associated device constructors that have registered with the system. A device constructor is a proxy that hides the exact type of a device driver and the method used to create new instances of the device. There is no coupling between the library and the device drivers, so the set of device

drivers can vary independently.

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

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. The application only has to be compiled with the associated device library or load the device library at run-time to have access to the new devices.

Environment manager

The environment manager allows vjControl to connect to a running VR Juggler application. 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 the vjControl user interface, it is possible to view and dynamically control every aspect the running virtual platform.

Display manager

The display manager (see Figure 6) encapsulates all the information about the display windows' settings. This includes information such as size, location, graphics pipeline, and the viewing parameters being used. In addition to holding the display parameters, the display manager is also responsible for performing all viewing calculations for the windows it controls. This is done as part of the main kernel control loop.

The display manager is also used for communicating configuration requests to the draw managers in the system. When a new window is created in the system, the display manager handles the configuration request by adding the window to its internal state information. Then it triggers an update notification that notifies the draw manager of the new display window.

External managers

Draw manager

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

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

Other external managers

Currently VR Juggler has no other external managers, but we expect that this will be a major area for future extension.

New external managers would function much like the draw manager by providing applications with interfaces that

are specific to the type of service offered to the application. As long as the applications make use of the encapsulated features using the interface provided, VR Juggler will take care of the details of keeping the manager synchronized with the rest of the system.

An example of an external manager that is currently under development is a sound manager. The sound manager will provide a unified interface for loading and play sound samples. The actual implementation will adapt to the local system configuration and use whatever method is available to handle the sound. Since user applications will only access the sound system through this interface, their applications will move transparently from one sound system backend to another.

Application objects

In VR Juggler, user applications are objects (see Figure 7). The VR Juggler system uses the application object to create the VR environment in which the user interacts. The application object inherits from based application objects that define an interface that must be implemented by the application object. The kernel maintains control over the environment and calls the methods defined in the application interface. When the kernel calls the application's methods, it gives up control to the application object so the application can execute the code needed to create the virtual environment.

Base application interfaces

The first step in writing an application object is to derive from the base classes that define the kernel and draw manager interfaces the application needs to implement. There is a base class for the interface that the kernel expects and a base class for interfaces needed by each of the available draw managers. For example in the `userOglApp` class in Figure 9 the interface needed for the Kernel is `vjApp` and the interface needed by the draw manager is `vjGlaApp`. Since all applications must interact with the kernel, all applications are required to implement the `vjApp` interface. Only OpenGL application objects need to implement the `vjGlaApp` interface.

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

The kernel calls each of the member functions of the interface based on a strictly scheduled frame of execution. During the frame of execution, the kernel calls the application methods and performs internal updates (see `updateDevices()` in Figure 10). 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 10). During these "safe" times, the kernel can change the virtual platform configuration as long as the interface remains the same.

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

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

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

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

In common programs, the `main` function signals the point where the thread of control enters the application. After the `main` function is called, the application starts performing any application processing necessary. When the OS starts the program, it gives the `main` function some processing time. Then after the process's quantum expires, the operating system switches to another process.

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

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

Benefits of application objects

Application objects provide many unique benefits that are not possible in traditional procedural systems. Many of these benefits can be traced back to the flexibility allowed by the microkernel architecture and application object design.

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

- Run-time changes

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

- Low coupling and increased robustness

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

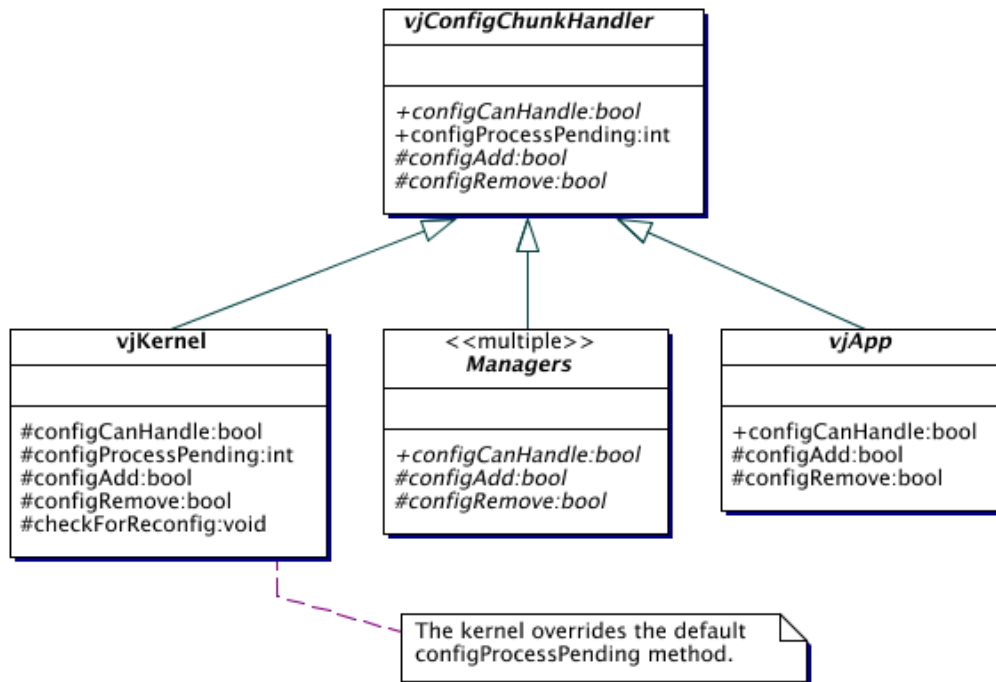
- Dynamic loading

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

- Allow implementation changes

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

Run-time (re)configuration

Figure 11. vjConfigChunkHandler interface

All VR Juggler managers support run-time reconfigurability. Input devices can be reconfigured at runtime 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.

Config chunk handler interface

VR Juggler provides a common framework for processing configuration and reconfiguration requests (see Figure 11). The framework defines an interface (`vjConfigChunkHandler`) that specifies the methods a "handler" component must implement to support run-time reconfiguration. All reconfigurable objects must conform to this interface. New objects can add support for run-time reconfiguration by simply inheriting from the `vjConfigChunkHandler` class and providing implementations for the methods defined by the interface.

The methods in the handler interface are:

```
bool configCanHandle(vjConfigChunk* chunk):
```

This function is used to determine whether the handler object knows how to handle the configuration chunk given. If the object wants to handle the chunk, then this method should return true. If not, then it returns false.

```
bool configAdd(vjConfigChunk* chunk):
```

The system is requesting that the handler reconfigures the system to add the configuration specified by the given chunk. It returns true if the change is successful, and false otherwise.

```
bool configRemove(vjConfigChunk* chunk):
```

The system is requesting that the handler reconfigures the system to remove the configuration specified by the given chunk. It returns true if the change is successful, and false if the change is unsuccessful or cannot be completed.

The class interface creates a simple interface for allowing objects to become configurable. Since all configurable components support a common interface, the system can simply keep a list of reconfigurable components. It is not necessary to know the actual type of the components that are being configured.

Pending configuration queue

Reconfiguration is implemented using a pending config queue that is contained in the config manager (see `vjConfigManager` in Figure 12). The queue contains a list of all currently pending reconfiguration requests. As with all other managers, the kernel controls the configuration manager. The kernel is in charge of synchronizing the system and sending reconfiguration requests.

The system initially starts with only the kernel and several system managers instantiated. When the kernel receives a new configuration request, it passes it to the config manager which adds the config chunk to a pending reconfiguration queue. The request remains there until the kernel attempts to reconfigure the system with the pending requests. When a request is successfully processed, it is removed from the queue. If a request is not able to be successfully satisfied, then it remains in the queue until the system is able to satisfy the request.

Each frame, the kernel checks for new configuration requests (see `checkForReconfig` in Figure 13). If there are any new entries, then the kernel attempts to process the request. The kernel steps through all known managers (that implement the `vjConfigChunkHandler` interface). The kernel asks each one in turn whether they can handle the configuration request. If they can, then the kernel passes the request on to the manager. The manager then processes the request using whatever methods it requires. For example, the input manager maintains an internal list of configurable devices. It looks for the device that corresponds to the given chunk, and configures it.

When the manager returns from processing the request, the kernel continues on to the next manager and attempts to process the same request. The kernel does not stop processing with the first manager that can handle the request, because multiple managers may need to respond to the same configuration requests.

After the kernel has iterated through all the known managers, it checks to see if any of the managers successfully processed the configuration request. If a manager successfully processed the request, then the kernel removes it from the pending queue. The configuration request is also stored in an active configuration data structure so the system may use the information again in the future. If no manager was able to complete the request, then the request remains in the queue where the kernel will try to process it again in a future frame.

This method of processing the configuration information is based on the chain of responsibility pattern [design patterns GOF]. It differs only in that the handlers in VR Juggler do not have a strict successor chain. Instead, the kernel keeps a list of several base handlers. The kernel then traverses this list to find a valid handler for a given reconfiguration config chunk. These handlers may then in turn process the request in any way they wish. VR Juggler does not impose any specific behavior on the reconfigurable components.

It is also possible for a system component to directly query the config manager if an object would rather not have the kernel manage its reconfiguration.

Dependency management

It is not difficult to imagine situations where the configuration of one component relies upon the successful loading and configuration of another component. For example, in VR Juggler users access all devices through proxies. The device proxy serves as a handle to a device that allows the system to access the device indirectly. Users configure proxies by specifying the device that the proxy should reference. If the system attempts to configure a proxy before the device being referenced has been loaded, the proxy's configuration will fail. Any system that implements configuration or reconfiguration of the system must have a system in place to handle these inter-dependency issues.

Dependency checking

To handle dependency resolution, VR Juggler performs dependency checking between system components. Dependency checking refers to the process the system goes through when it runs a check to see if all the system resources required by a new configuration request are available before it allows that request to be processed.

Dependency checking is supervised by the `vjDependencyManager`. The dependency manager allows the system to send it a query about any configuration chunk. The dependency manager will check to make sure that any dependencies that need to be satisfied for the given chunk have been satisfied, and will return approval to the caller.

The dependency manager checks dependencies by maintaining a list of dependency checker objects that it uses to check for dependency fulfillment. All dependency checker objects implement a common interface specified by the `vjDepChecker` class (see Figure 14).

This interface defines two methods that are used for dependency checking:

```
bool canHandle(vjConfigChunk* chunk)
```

This function is used to determine whether the checker object knows how to evaluate dependencies for the configuration chunk given. It returns true if the object knows how to evaluate the dependencies, and false if not.

```
bool depSatisfied(vjConfigChunk* chunk)
```

This function actually performs the dependency evaluation. If the dependencies are satisfied, then it returns true. If dependencies are not satisfied, then it returns false.

System components that require advanced dependency checking are responsible for registering their dependency checkers with the dependency manager. The VR Juggler manager that most directly deals with the components often handles this registration. For example, whenever a new device registers with the input manager, the device also registers any dependency checkers with the dependency manager.

The manager processes dependency requests by querying each dependency checker object to find out what types of configuration chunks it can evaluate.

When the manager receives a dependency query, it attempts to find a dependency checker that can handle the given configuration chunk (see Figure 13). If it finds a dependency checker, then it sends a request with the chunk to the checker. The checker then checks for whatever system state is required and returns an answer, which the manager returns to the originator of the query. If the dependency manager does not find a checker, then it uses the default dependency checker to check for many of the most common dependency issues.

Automatic unloading

One issue that we have not touched upon yet is what to do if a user removes a component from the system through reconfiguration, but another component relies upon the removed item. As an example, consider again the case of a proxy that is configured to reference a given input device. After the system is fully configured, the device is removed via reconfiguration. How does the system keep the proxy in a valid state? How does it re-connect the proxy if the original component is later re-added to the software system?

VR Juggler handles these issues through "smart unloading" of system components. When a component's dependencies are reconfigured or removed, smart unloading allows the dependent component to reconfigure itself dynamically in an attempt to satisfy its dependencies by modifying its behavior. If the dependencies cannot be satisfied by dynamic modification, then the component is unloaded and placed back into the pending queue. This effectively backs-out the original configuration requests until all the dependencies are once again satisfied.

Because this process could lead to a cascade of unsatisfied dependencies, it continues iteratively until all components of the system are once again in a stable state.

This area of the system is a current area of active research. We are investigating better ways for components to dynamically change the way that they work in an attempt to deal with unsatisfied dependencies via graceful degradation. We are also working to better techniques to deal with some of the more highly complex dependencies.

Run-time reconfiguration discussion

Results of the initial implementation have been very positive. We have been able to create a flexible architecture that allows for the majority of system components to be reconfigured at run-time. We are still refactoring the design to increase flexibility and add more reconfiguration abilities to the system.

The reconfiguration system has been used to create application "switchers" at VRAC. When we give tours of the VR systems at VRAC, we are faced with the problem of needing to rapidly switch between applications and VR system settings. Before we had run-time reconfiguration, we would have to shutdown the application and the VR system every time we wanted to switch applications. This led to unacceptable downtime and increased risk of potential problems when restarting the application.

Using run-time reconfiguration, we have been able to solve this problem by creating switcher applications. Our switcher applications provide a single unified virtual environment that contains and manages any number of stand-alone user applications. Once the switcher application has been started, users of the VR system can quickly swap between the stand-alone applications without having to leave the switcher virtual environment or change system settings. This allows people giving tours to just start a single application that allows them to run any number of other applications within the same environment.

We have also benefited immensely from the increased robustness that the reconfiguration system affords while debugging new hardware systems and drivers. Because of the reconfiguration system, the applications keep running even when the hardware fails and has to be reconfigured or entirely restarted.

Run-time reconfiguration has also allowed us to do run-time performance tuning using the abilities provided by VR Juggler. VR Juggler provides a performance monitoring system that allows users to interactively evaluate the performance of their applications [Juggler performance Just IPT 2000]. Using this tool, developers can pinpoint performance bottlenecks. Then using run-time configuration they can make changes in an attempt to increase performance.

The implementation of run-time reconfiguration has not been without its problems though. There have been several obstacles we have had overcome and work around during the implementing the system.

One area of difficulty that we have found is that some of the popular application programming interfaces (APIs) used in a VR application do not lend themselves well to reconfiguration. Some graphics APIs (such as Iris Performer) expect to be in complete control of system resources and as such do not provide a way to release the API's resources when VR Juggler does not need them any more. We are actively pursuing ways to alleviate these problems by communicating with API developers and by investigating systems that are more flexible.

An unexpected result was that average VR users did not jump into using run-time reconfigurability immediately. Dynamic systems are still a new concept and it will take some time for users to become comfortable with and begin actively taking advantage of the new abilities afforded by such a system. As we are training out users and show them the benefits of reconfigurability, they are becoming increasingly excited and have started to use reconfigurability in their applications.

Multi-threading

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

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

To avoid this problem, several current VR development environments decouple each of the tasks so that each operation can execute individually without incurring delay by waiting for other operations [Shaw decoupled 1993] [CAVE Cruz thesis]. Each task can be viewed as separate component that executes relatively independently

of the other components. The system core is responsible for making sure that the components are synchronized when needed. This allows the application to run as fast as possible because, the display may be able to run at 60hz while the tracking system runs at 50hz and a haptic controller runs at 500hz. If the devices were not in separate threads, they would have to run at least as slow as the slowest device, and would actually run slower because they would have to wait for all devices.

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

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

How does everything get started?

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

Example 1. Kernel Startup

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

An application is given to the kernel at a later time through the kernel interface. Once the kernel has an application, it begins executing application methods within the kernel execution frame. The application can be given to the kernel as a dynamic object or as an object allocate in the kernel boot loader code. In the case that object is allocate in the loader, a common main() function (see Example 1) can hold both the kernel start code and the code to give the kernel the application.

System interaction

In order to better explain interaction within the VR Juggler architecture, we will now describe how the system starts up and loads a single OpenGL application (see Figure 15). In order to simplify the diagrams we will not go into the details of configuration, we also leave out some method invocations, and we do not deal with multiple application objects.

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

The next step in startup is to create and initialize an application. First, an application object must be instantiated. Then the application object is given to the kernel to start executing. The first thing the kernel does with an applica#

tion is to create the draw manager that the application needs. Once this is done, the kernel then executes the application's initialization methods and signals the draw manager to do the same.

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

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

How well does VR Juggler meet the design goals?

Hardware abstraction

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

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

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

Run-time flexibility

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

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

Performance tuning

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

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

Cross-platform

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

Extensible

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

A. Appendix: Needs of VR system

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

Primary needs

Performance

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

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

Current VR software systems have been successful at achieving good performance. Unfortunately, many of these systems do so while neglecting several fundamental needs of a software system such as: reusability, extensibility, flexibility, portability, and robustness. In some cases, system developers sacrifice these needs in an attempt to increase performance by tying the software system as closely to the hardware as possible. Another reason that current systems may not implement these features is that it is much more difficult to design a system that supports these features. Since the primary focus of VR research to date has been hardware systems and not software systems, these types of features have not received the attention that they deserve in a VR software system. We believe that a high-performance VR software system does not need to sacrifice any of these features in order to maintain high performance. In addition, we believe that these features are vitally important for creating a long lasting standard VR software system.

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

Extensibility

Extensibility in a VR development environment allows user applications to survive technological changes of the fu#

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

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

Flexibility

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

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

In addition, the design of the system itself should not lock developers into writing only one given type of application. For example, the development environment must make it just as easy to write a passive architectural walk-through application as it is to create an interactive scientific visualization application. This requires a development environment that is not only flexible about what hardware it is running on, but is also flexible about what type of applications and toolkits are running within the software system.

Simplicity

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

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

Robustness

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

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

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

Performance

Low latency

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

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

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

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

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

High frame rate

There are many potential areas in a VR software system where latency can be introduced. In order to reduce latency, the sources of the latency must be thoroughly understood. Figure A.1 shows a diagram that outlines the sources of latency in the system. The diagram shows the path of a single set of sampled input data. The effects of the data are traced through the system.

The diagram shows the number of "frames" of several system components. The input devices are running asynchronously to the rest of the system which is coordinating through the use of synchronization points during processing. The chart highlights the tracker data that is valid at the first synchronization point (at the end of display frame 1). The system then uses this input data to start the rendering and computation of the next interval. When this interval completes, the user can see a rendered display (display frame 3) that has used the original tracker readings, but the display does not have any updates that are dependent upon computing a new application state from the input. Display updates of this type are not available until display frame 4.

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

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

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

Support for hardware

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

VR development environments differ widely in the interfaces provided for creating an application. Some provide a very high-level view, where users create applications with custom scripting languages and graphical tools, and the system itself takes on most of the responsibility of simulation, geometry, and interaction. Other interfaces float just above the hardware level, using well-known graphics APIs and programming languages to ensure the greatest performance. Often, the higher-level tools will enable faster development with a shallower learning curve. The other side of the argument is, "If you want something done right, do it yourself." The more one of these systems is willing to do by itself, the more likely it is that it will do something unwanted, or do it in a way that is not optimal for a particular application. The key to balancing this trade off is to make application development as easy as possible, but all the developer enough flexibility to use whatever optimizations they need.

Performance monitoring

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

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

The performance information can also be used to find and alleviate performance problems with a specific VR system's configuration. Often small changes in configuration options can have a dramatic impact on the performance of applications in a given VR system. For example, it may be possible to tweak the configuration of graphics windows in a way that achieves a higher visual frame rate. This type of configuration tuning can increase the performance of all applications that run in the local VR system.

Extensibility

Hardware abstraction

In order to be usable, the VR development environment must provide support for the physical hardware devices in

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

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

Simple extension

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

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

Do not require application changes

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

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

Flexibility

Scalability

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

A scalable system also eases application development when VR system resources are scarce. Most sites only have

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

Cross-platform

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

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

Run-time changes

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

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

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

Setup

Run-time reconfiguration can prove invaluable when setting up and configuring a new system. Reconfiguration allows users to change device configuration parameters while an application is running. For example, users can use this ability to interactively tweak tracker settings at run-time. A person setting up a system can run a calibration program that draws a coordinate axes at the position of a tracker. By using this visual feedback, a user can interactively test whether the offset and rotation parameters for the current tracking system have been configured correctly. This type of interactive testing is helpful in determining correct system settings.

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

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

Reconfigurable VR software systems also enable the use of advanced projection systems where the physical settings of the projection surface itself are changing at run-time. Examples of this type of system include desk-based systems

with a movable projection surface and large scale CAVE-like devices that allow users to move the walls. Run-time reconfiguration allows users to reconfigure the desk projection surface while an application is running; or if a driver is available, the running system can actively monitor the desk's current settings and automatically update the projection parameters to reflect any changes.

Software and hardware testing

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

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

Developers can make use of this flexibility to test an application using many different VR system configurations. For example, it may be helpful to start the application first in a simulator, reconfigure it so that it is running with an HMD, and then reconfigure it again so that it is using a CAVE or some other large-scale projection environment. Applications can be tested in all of these environments without ever halting execution.

Performance tuning

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

Application adaptations

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

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

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

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

Support use of other application toolkits

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

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

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

Do not be overly restrictive

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

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

Simplicity

Short learning curve

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

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

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

Rapid prototyping using simulation

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

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

Robustness

Failure protection

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

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

Maintainability and correctness

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

Maintenance presents a problem in a large system because of inter dependencies within the software system. In a complex system, it is very difficult to completely understand every part of the system. If system components are highly intertwined, changes in one component could affect the correctness of another component. These dependencies are often very difficult to predict and take into account.

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

Testing correctness in a large system is important because you need some level of confidence that the system will work. A common way to manage this is to separate the major system components into separate entities that can be

individually tested and verified. The idea is that if a system is made up of many components and each of these components individual works correctly then, the combined system will work correctly (assuming the linking code works correctly). To test such a system, the components should be tested at many levels of granularity. For example in an object-oriented system you may want to test at object, module, and sub-system levels.