Published in final edited form as:
Presence (Camb). 2010 April 1; 19(2): 162–178.

Lessons about Virtual-Environment Software Systems from 20 years of VE building

Russell M. Taylor II,
University of North Carolina at Chapel Hill

Jason Jerald,
University of North Carolina at Chapel Hill

Chris VanderKnyff,
University of North Carolina at Chapel Hill

Jeremy Wendt,
University of North Carolina at Chapel Hill

David Borland,
Renaissance Computing Institute

David Marshburn,
University of North Carolina at Chapel Hill

William R. Sherman, and
Desert Research Institute

Mary C. Whitton
University of North Carolina at Chapel Hill

Abstract

What are desirable and undesirable features of virtual-environment (VE) software architectures? What should be present (and absent) from such systems if they are to be optimally useful? How should they be structured? To help answer these questions we present experience from application designers, toolkit designers, and VE system architects along with examples of useful features from existing systems. Topics are organized under the major headings of: 3D space management, supporting display hardware, interaction, event management, time management, computation, portability, and the observation that less can be better. Lessons learned are presented as *discussion* of the issues, *field experiences*, *nuggets* of knowledge, and *case studies*.

1 Introduction

Over the past 20 years, the authors and our colleagues have built and evaluated dozens of virtual environment (VE) systems. The purposes of these systems have been varied: developing VE hardware (Eyles, Molnar, Poulton, Greer, & Lastra, 1997; Fuchs et al., 1989; G. Welch et al., 1999; R. B. Welch, Blackmon, Liu, Mellers, & Stark, 1996), VE toolkits (Robinett & Holloway, 1992; Russell M. Taylor II et al., 2001), interaction techniques (Burns et al., 2006; Kohli & Whitton, 2005; Lok, Naik, Whitton, & Brooks, 2003; Mine, 1997a, 1997b; Mine, Jr., & Sequin, 1997), building walk-through applications (Aliaga et al., 1999; Mine & Weber, 1995),

Corresponding author: Russell M. Taylor II, CB #3175, Sitterson Hall, University of North Carolina, Chapel Hill, NC 27599-3175, (919) 962-1701, taylorr@cs.unc.edu.

scientific visualization applications (Jen et al., 2004; Leech & II, 1992; Marshburn et al., 2005; Ouh-young, Beard, & Brooks, 1989; Qi, Taylor II, Healey, & Martens, 2008; Quammen et al., 2008; Russell M. Taylor II et al., 1993), and locomotion interfaces (Feasel, Whitton, & Wendt, 2008; Peck, Fuchs, & Whitton, 2009; Razzaque, 2005; Usoh et al., 1999). Each of these systems has had real-world evaluation through either controlled user studies or daily use by external clients. Several have been re-implemented using different programming languages (Burette et al., 2007; Jen et al., 2004), rendering libraries (Ouh-young et al., 1989; Robinett & Holloway, 1992; Russell M. Taylor II et al., 1993) or VE toolkits (Whitton, 2004).

1.1 Organization of the paper

This paper collects and organizes experiences gained from doing this work, also drawing on discussions with colleagues (Russell M. Taylor II, 1995), positive examples of work done elsewhere, and negative design choices that prevented adoption of our and others' VE toolkits.

These experiences have been classified into four types:

- ***Discussions*** lay out our thoughts on issues, including both desirable features and dangerous features of VE toolkits. These are based on our overall experience, but also include extrapolations beyond our direct experience. This is the default classification for the text and is often unlabeled at the start of each section.
- ***Field Experiences*** describe lessons learned based on experience with systems deployed in active use;
- ***Nuggets*** are brief insights from current and past designers, describing features they found useful or describing hazards to be avoided; and
- ***Case Studies*** provide examples of systems we have encountered that provide solutions to problems encountered in the design of VE systems (some of which are non-VE systems).

Although the lessons learned are sometimes application-specific, taken together they guide our current decisions about VE toolkits and systems. Our intent in this paper is not to cover all topics important to VEs (notably absent are audio, collaboration, and persistent state), but to make our experiences available for those who design VE toolkits in the future. The paper includes our successes, our failures, what we loved, what we hated, what we still yearn for, and what we could not tolerate.

We begin with a discussion of 3D space management; proceed to the topics of supporting display hardware[*]; interaction[*]; event management (including input device abstraction); time management[*]; separating rendering from computation; portability[*]; and conclude with a counter-intuitive truth about VE toolkit design that a VE library that includes fewer features can be better than one that includes more. We also include case studies of porting applications from one toolkit to another[*]. Within each section, the experience is labeled by type (field experience, case study, nugget, or discussion).

2 Managing Space

At its heart, each VE toolkit manages a hierarchy of spaces; it enables the placement of viewpoints, screens, and rendered objects based on reports from tracking devices and user interaction. The most basic interactions involve self-motion within the world (head tracking, hand tracking, and navigation through flying, scaling, or grabbing the whole world). Systems

[*]Partially or wholly located in the supplemental material accompanying the online version of this article.

often include the ability to render both 3D objects representing the user (hand, vehicle) and the world, as well as 2D user-interfaces or information displays.

We have found the basic VE transformation hierarchy and methods of interaction described in (Robinett & Holloway, 1992) to adequately support our various virtual-environment applications over the years. A version of the left-hand side of Figure 1 from that paper is shown here (modified from the original based on as-built experience):

One way the diagram differs from the original is the addition of optional screens in Room space for fixed-screen projection systems such as our *GROPE*, (Brooks, Ouh-Young, Batter, & Kilpatrick, 1990), *PIT* (Arthur et al., 1998), *nanoWorkbench* (Grant, Helser, & II, 1998), and other displays such as *CAVE*s (Cruz-Neira, Sandin, & DeFanti, 1993). A second modification is the addition of Sensor spaces, required because a tracker sensor's pose is often not co-located with the user's head or hand.

Nugget: You should be able to track body parts in addition to the head and hand.

2.1 Transformation Hierarchy vs. Scene Graph

The transformation tree in Figure 1 is all that is required to enable viewpoint determination and navigation within a static scene. It requires no knowledge of or interoperability with whatever transformation hierarchy or scene graph the application is using to describe objects within the world. The UNC *Vlib* toolkit's implementation of the model presented in Figure 1 included the ability to grab and manipulate application objects, based on the *PPHIGS* scene graph used on the *Pixel-Planes 5* (Fuchs et al., 1989) graphics supercomputer. This scene graph included the ability for *Vlib* to traverse the entire tree structure (including user-specified objects), enabling it to automatically support grabbing and scaling of user objects. This capability disappeared when *Vlib* was ported to the *OpenGL* rendering framework, although the library and world-space-root manipulation continued to work without rewriting user applications.

Extending the transformation hierarchy to include the application's scene graph requires traversing a difficult and narrow road. ***Ditch on one side of the road:*** The selection of a specific scene graph for a VE toolkit will exclude the toolkit's use by the majority of applications (all those using other scene graphs) and will limit its useful lifetime. ***Ditch on the other side:*** If a VE toolkit is to work with multiple scene graphs and rendering libraries, then there is no completely automatic way to enable the toolkit to manipulate application-space objects.

Discussion—We believe that the least-bad resolution to the scene graph issue will be to provide a transformation hierarchy for the VE toolkit (which is required for the determination of viewpoint and navigation) and then provide hooks for application developers to use, but not to use any scene graph natively within the toolkit. As the most basic display recipe, the toolkit should provide the application with a viewpoint, screen, and modeling transformation suitable for rendering the objects from each point of view (hand space, room space, world space) for each eye. Helper methods can provide the ability to import and export the transformations to a variety of formats (*Vlib* supported *PPHIGS*, *OpenGL*, row-major and column-major 4×4 transforms).

The most basic hook between the application scene graph and the VE hierarchy is the construction of a hierarchy that shadows that of the application's scene graph, with the application copying transformations back and forth as they change. A better model could be to provide integration of the toolkit with the most common scene graphs, using their existing callback or update mechanisms to hide the data transformation from the application developer. In this solution, the application developer owns the right-hand side of the graph (shown as a cloud in the figure) and can decide how much interactivity to provide via the toolkit, and how much to implement on their own.

We recommend that a new toolkit be initially implemented with direct hooks to support rendering using *OpenGL*, *Direct3D*, and *VTK* as the rendering libraries to ensure that it can be generalized, and to include rendering to a 3D audio-only environment to ensure that it does not make assumptions about visual-only rendering.

Field experience—An example of problematic tying to a specific rendering library came when one of the force-feedback vendors changed from a toolkit that used its own internal representations of the objects in the world to one that was built on top of *OpenGL*. Application code now had to link with *OpenGL*. *DirectX* applications and those that ran on server machines with no display were in a bind; even applications that used *OpenGL* for rendering had to draw the world twice: once to an off-screen buffer for the haptic point of view and again for the visual points of view.

Nugget: In general, doing things in different spaces (user hand vs. object, for example) is very difficult. Any support for making this easier will be a big help. **Nugget:** The use of a self-aware transformation hierarchy has been shown to make it much easier to design interactions and animations: *Alice/Diver* has a generalized transformation tree that enables the specification of any object's motion or behavior in any other object's coordinate system ("keep the bunny head pointed towards the carrot") (Gossweiler, Long, Koga, & Pausch, 1993). **Nugget:** "Smart models" (like the Division *MAZ* files or *Inventor* files) that contain more information than just geometry may be desirable. Animation and dynamic behavior (specifying that an object should follow gravity or can be grabbed) could be supported. (More modern systems like VRML and X3D do enable such specifications.)

2.2 Calibration

Proper calibration of VE systems is more important than that of other graphics systems because a consistent image update with proper motion is essential for the experience to match that of moving around in the real world. This requires proper matching of many system parameters, including: field of view, eye separation, head-sensor location (with respect to the eyes), and screen placement (with respect to the eyes or tracker).

There are two stages of calibration: coarse and fine. The coarse stage determines the basic poses of tracker sources and sensors with respect to the room and makes the world move in approximately the correct direction as the user moves and rotates head and hand. The other is a fine calibration, some of which depends only on the devices and some of which (eye separation, eye location with respect to the head tracker) depends on the individual. The methods described in (R. L. Holloway, 1995) and (Azuma & Bishop, 1994) can form a starting point for the development of such fine-tuning procedures. It turns out that accurate calibration is a much more difficult to do than it seems it should be.

Field Experience—Rich Holloway described the sources and relative magnitudes of these errors for the case of a see-through HMD in (R. L. Holloway, 1995) and (R. Holloway, 1997). He discussed the required tolerances for each measurement in the system (including optical distortion and other parameters not listed above), along with a method of calibration of a complete system using a highly-accurate mechanical tracker. Two surprising results were that eye tracking may not be necessary if the center of the eye is used as the center of projection and that the addition of a World space to the calibration system can add errors.

The major result of his experience was that delay (latency) swamps all other error sources. The sources of delay include tracker delay, host-computer delay (buffering, O/S), image-generation delay, video-synchronization delay, frame delay, and delay inside the image device itself. The unfortunate reality is that, for the motions he observed, the worst case was 1mm off-set error for every 1ms of total system delay – which turns out to be 1.6cm just during the scan-out time of a 60Hz display!

To help combat this error, Ron Azuma looked at how far ahead we can predict the motion of a user's head (Azuma & Bishop, 1994). He found that prediction could reduce registration errors by 2.5-3 times over no prediction and that by mounting inertial sensors on the head to track acceleration and rotation, prediction errors could be further reduced by a factor of 2-3. These improvements depend on how far into the future the prediction is made, but remain better than no prediction out to at least 200ms. He also describes a technique for determining the static registration of his system that could be incorporated into the calibration portion of other VE libraries.

Nugget: The toolkit must provide the capability to both calibrate the transformations associated with the trackers and screens and to test the calibration of the system. This could be done using a test application, but it needs to have some way to bypass/control the calibration parameters and save the new setup. For example, it should be possible to adjust the transformation that describes how the tracker sensor is mounted on a hand-held device (Russell M. Taylor II, 1995).

Some modern toolkits such as *FreeVR*, *Virtools*, and *VRUI* include routines to help with calibration. Others only provide calibration tables whose values must be filled in by measurement or using custom application code.

3 Supporting Display Hardware

A discussion of some of the issues related to supporting various display hardware configurations can be found in the supplemental material accompanying the online version of this article.

4 Object Interaction

A description of some of the issues surrounding a VE toolkit's support of user interaction with objects in the scene can be found in the supplemental material accompanying the online version of this article.

5 Event Management

Communication between components in a VE toolkit and the application code is a key feature to enabling extensibility: when a toolkit-provided button is pressed, application code needs to be notified. Between the different components of a VE toolkit itself, collision events (such as hand-object collisions), system control elements (such as tracker information and button presses), new simulation data, and other events drive interaction and system response. VE libraries should carefully consider how this event handling is implemented: it is highly visible within application code and directly affects programmer's experiences.

Field Experience

Event communication (a specific case of the Observer pattern) has been implemented many ways – to varying effect. We describe three that we have had experience with (callbacks, inheritance, and slots-and-signals) and the implications they have for application code.

Callbacks—Many callback systems are implemented in a way that requires the object handling the callback to continue to exist. In C/C++, this means they are either C-style global methods, or static class method. Either way, the method does not belong to an instance of an object, requiring either a global pointer to an object instance, or a void*-style input parameter (which points to the "owning" instance). Furthermore, when implemented in such a way that a non-static class method may be passed as an input to a callback, the callback reference can remain even after that instance is out of scope – potentially leading to program failure at strange, hard-to-debug times after the object has been deleted.

Inheritance is a second common implementation of inter-object communication (for example implementations, see *Java*'s java.util.EventListener interface, *FLTK*'s handle method within all Fl_Widget subclasses, or *VR Juggler* (Bierbaum et al., 2001)). In the inheritance implementation, the class interested in events from some toolkit object inherits from that object type and implements specific methods. In the two examples above, the process is implemented as a single method with an input type which specifies the type of action performed. The user code must check for one (or many) "interesting" event types and act in response to only that type – leading to long case-selection blocks in the event handler. To obtain cleaner code, these events can be separated into different event-specific methods (for instance, separate methods for buttonPressed and for buttonReleased). Even so, if a single class wishes to listen to events from two different sources – and both dictate a method with the same signature (common if the two sources are the same type) – this can lead to confusing code.

Slots and Signals—The cleanest approach to communication between objects we've seen is known under various names: Trolltech's *Qt* calls them "slots" and "signals"; *C#* calls them "events" and "delegates"; herein we will use *Qt*'s names. When an event occurs within some part of the system, an object emits a signal. Interested parties connect their slots (user-specified methods which accept the output from signals) to signals they are interested in and receive updates when signals are sent. The slots and signals maintain information about their connections so that when one goes out of scope, it unregisters itself from all connections – avoiding the crash failure mode of standard callbacks. Furthermore, because the user specifies the name of each slot – independent of the signal name – and the connection between slots and signals, a single user instance can receive inputs from toolkit-owned, same-named signals at different slots.

While *C#*'s events and delegates are natively supported within the language, most other languages do not have native support: *Qt* preprocesses custom language features to produce C ++ code for a class that implements slots or signals. However, this slots and signals communication pattern does not *require* language support.

Finally, while this discussion began by specifying single-process, inter-object communication, between-process, inter-object communication could follow the same pattern if inter-process messages were received at a single endpoint within a process and then emitted as a signal from that single endpoint.

Nugget: It would be good to map user events to application events through well–defined hooks. This would allow an application to run with different menu systems or interaction paradigms without much change.

5.1 Input Device Abstraction Case Study: VRPN

VE systems make heavy use of devices such as trackers, wands, game pads and/or joysticks and trackballs. In many instances, there are multiple products from multiple vendors that can serve the same function in the VE system.

Field Experience—It can be desirable to provide low-latency, robust, and network-transparent access to VE input devices because some devices only work on particular architectures or can attached to only particular brands of computers. We have encountered the following issues:

- VE toolkits should provide a uniform interface to input devices providing identical or highly similar functionality. An application that uses different devices for the same function should not require custom code for each device. Good software development practice suggests that unless there is a compelling reason, applications should not be written for specific devices, but for more generic abstractions.
- Some devices require specialized connections (PC joysticks) or have drivers only for certain operating systems yet are useful to drive applications on other platforms.
- Some VE devices perform more reliably when left on continuously, and require lengthy reset procedures when closed and re-opened.
- VE applications require minimum latency, and need to know at what time events occur in the system.

Design and Implementation

The above criteria, along with others, led us to develop the *Virtual-Reality Peripheral Network* (*VRPN*) (Russell M. Taylor II et al., 2001), an architecture where input/output devices at each display station are connected to one or more local device servers. The design decisions resulted in a package that is compatible with many different software systems: members of the VE community have used *VRPN* with *CAVELib, VR Juggler, FreeVR, VRUI, Panda3D, VMD*, the Virtools VR Pack, Syzygy, AVANGO, OpenSceneGraph, WorldViz, OpenTracker, DART, DIVERSE, VTK, and other packages. The design supports many types of devices: it has drivers for nineteen trackers and twenty-four other device types. In several cases the drivers have been implemented by the vendor of the device. We summarize here some of the design decisions that we feel led to this wide applicability. These decisions and others are described in detail in (Russell M. Taylor II et al., 2001).

Device Factoring—It has been fruitful to think of *VRPN* as providing interfaces to a set of functions, not as providing drivers for a set of devices. Particular devices can be factored into one or more *canonical device types*. Each type specifies a consistent interface and semantics across devices implementing that function. The most-commonly-used types are *Tracker* (location plus orientation), *Button* (press and release events), *Analog* (one or more values), *Dial* (incremental rotation), and *ForceDevice* (an output device used to specify forces and force-fields).

Factoring a set of devices to canonical types requires mapping the different capabilities of each device onto a common interface. There is a tension between providing a very simple interface (which does not enable access to particular advanced features) and providing a feature-rich interface (where many devices do not implement many of the features, forcing application code to deal with special cases). *VRPN* dealt with these issues by:

- factoring devices based on their functions,
- enabling devices to export multiple interfaces,
- silently ignoring unsupported message types (for downward compatibility), and
- providing application-level access to all messages (for extensibility).

A *VRPN* driver for a multi-function physical device exports interfaces for multiple *VRPN* device types. For example, the driver for SensAble Technologies' *Phantom* haptic display exports Tracker, Button, and ForceDevice interfaces under the same device name. The application deals with a *Phantom* as if it were three separate devices, one for each of its functions. Because all devices mapped to the same server automatically share a communication interface, this is done without increasing bandwidth or latency.

A special case of multiple interfaces is the *layered device*. In this case, higher-level behavior is built on top of an existing device. An instance is the *AnalogFly* server, designed to enable flying using joysticks: the joystick driver reports analog values for each of its axes and the AnalogFly integrates these values into Tracker messages. Clients can connect both to the low-level device (to read the buttons on a joystick, for example) and to the higher-level tracker device.

Do one thing and do it well: VRPN provides *only* a device-layer interface, not a scene graph or a set of interaction techniques or graphics output. This was not a design choice as much as a fortunate characteristic, but it was clearly a feature that enabled wide adoption of the toolkit (discussions on the mailing list have made it clear that if VRPN had included a specific scene graph or graphics library or directory services choice, then it would have become unattractive to users of other scene graphs or directory services).

6 Representing and Managing Time

A discussion of the representation and management of time (including motion paths) in a VE toolkit can be found in the supplemental material accompanying the online version of this article.

7 Computation: Separating Simulation and Rendering

Latency, the delay between when a user makes an action and when that action is reflected in the state of the virtual environment and on the display, is the primary enemy of effective VE systems (R. Holloway, 1997) and computation time can be a major component of end-to-end latency. Simulation and rendering, two of the major computational processes in VE systems, almost always operate at different rates. To minimize the impact of one computation on another, it is therefore desirable to separate them into different threads or processes.

In our experience, the aggregate processing requirements of virtual environment software usually exceed the capability of one thread, one process, or even one computer. Furthermore, the benefits of being able to run different parts of a VE application in different processes have been widely recognized in the literature (Adachi, Kumano, & Ogino, 1995; Bryson & Johan, 1996; Mark, Randolph, Finch, Verth, & II, 1996; Shaw, Liang, Green, Sun, & 1992; Sokolewicz, Wirth, Böhm, & John, 1993; Ståhl & Andersson, 1994). There are two essential needs to be satisfied when considering multithreading: multiprocessing and distributed processing. The first is to provide enough raw compute capability. There must be enough CPU and GPU cycles, enough memory and memory bandwidth, enough network bandwidth, enough main processor cycles, and so forth to meet the throughput needs of all application components. Second, the computational resources must be deployed so that each component of VE application is able to achieve its required update rate.

For VE applications that include haptics, a separate haptics thread running faster than the graphics rendering thread is required (Mark et al., 1996). Similarly, applications with simulation or device-control threads that operate more slowly than rendering or user interaction often require their own thread (Mark et al., 1996). Several other pieces of VE functionality may warrant separate threads, as well: a thread for 2D user interface elements; simulation and animation threads; a thread for device communication, or even one thread per device. In addition, particular VE application may include unforeseen pieces of functionality requiring a separate thread; a good VE system will enable this gracefully.

Examples of starkly different update rates include running a haptic server (1kHz update rate) with a rendering application (30 Hz update rate) (Mark et al., 1996), or an interactive graphics application with a simulation (which may run at several seconds per frame) (Bryson & Johan, 1996).

Alice/Diver included a rendering process separate from the application loop to enable real-time rendering even when the application loop slows down (Gossweiler et al., 1993). Many VE libraries, e.g., *CAVElib*, *VR Juggler*, *FreeVR*, and *VRUI*, do this by default; any application written with those libraries automatically split rendering into a separate thread.

Field experience

Our *nanoManipulator* system (Russell M. Taylor II et al., 1993) is an example of a multi-threaded, multi-processing application. A haptic thread is started by the SensAble Technologies device driver to update a *Phantom* haptic device at the desired 1000 Hz. A device-control process runs on a computer controlling a scanned-probe microscope to receive commands and send data. Rendering and 2D GUI interaction occur in yet another process (often on a third computer). This configuration was effective in most cases, but the 2D GUI and 3D rendering were handled in one thread that also communicated with the other two external threads through reliably-buffered channels. This caused system lock-ups when a user navigated through a series of 2D menus, causing the 3D rendering to hang and the network buffers between the microscope and haptic device to go un-serviced – resulting in system lock-up until the menu choice was completed. In this particular case, the result was barely acceptable because the user's attention was focused on the 2D interface widgets and not the 3D rendering.

Case Study: VRPN Error Reporting

When an application uses servers on remote machines and there is an error condition, there is the question of how to indicate this to the user (who may not even be logged into the machine). Early versions of *VRPN* did not provide support for error reporting back to the application. *VRPN* now handles this by enabling each object to send text messages. These messages have associated severity (normal, warning, or error), and they propagate across network connections. *VRPN* includes a static text-printing object that prints these messages on the application end. By default, it prints warnings and errors to standard output. This mechanism enables device drivers and servers anywhere in the system to send human-readable warning and error reports. This has proven to be very useful when debugging system behavior.

Another example of distributed error notification is the *VRUI* (Kreylos, Bethel, Ligocki, & Hamann, 2000) toolkit which allows warning signals—including audio and flashing objects—to be delivered to the VE user while immersed. For example, *VRUI* warns users when one of the 6-DOF trackers gets within one foot of a display screen. The application flashes a green grid so that the user can see where the screen is located and thus avoid damaging it.

Discussion

Distributing the work of a VE across threads, processes and machines, while fixing or ameliorating a number of problems, can introduce a number of other problems, for instance deadlock, improper synchronization, and locking issues. These issues are all documented in parallel computing, distributed computing, and concurrent computing texts and literature. VE systems, in providing for parallelism, must provide ways to control that parallelism effectively.

To a large extent, VE systems just use and export whatever parallel facilities are provided by the programming language (e.g. *Java*) or operating system. Beyond that, though, VE systems should provide parallelization facilities that are specifically adapted to the tasks of VEs. The client-server model of *VRPN* is particularly appropriate for distributed devices, for instance.

Nugget: For the case of VE devices, client and server should be run as separate processes when they have very different update rates, when server initialization takes a long time, when message timing is critical, or when the server requires frequent access to a device.

Nugget: A good VE system will include the capability to render in a separate thread, i.e. separate rendering from all other application computation. Rendering is the most pervasive VE feature and is also the feature for which degradation in performance is most noticeable. For immersive VEs in particular, rendering freezes or stalls induce *breaks in presence*, the term coined by Slater and Steed to mean attending more to the real world than the virtual (M Slater & Steed, 2000).

Nugget: A good VE system should also provide a way to identify which pieces of VE functionality run in which threads; should provide a way to name, start, and control the update rate of those threads; should ensure the VE code is thread-safe; and should provide to the those who extend the VE system mechanisms to ensure that their own code is thread-safe.

8 Portability

The VE applications developed within our groups have used a variety of operating systems, rendering libraries, and languages. Long-lived applications such as the *nanoManipulator* (Russell M. Taylor II et al., 1993), have been implemented on as many as six combinations as platforms come and go. The libraries on which the applications are built need to be ported to each new combination as well. The issues for each are different, and we describe them in order.

8.1 Operating System/CPU

Case Study: *VRPN*—Our *VRPN* (Russell M. Taylor II et al., 2001) was initially developed for four different architectures (*MIP, Sparc, Intel*, and *PA-RISC*), including a mixture of big- and little-endian machines. It was also compiled for a variety of operating systems (*Irix, HPUX*, and *Windows*). We found that enforcing wide applicability from the beginning produced a code base that was readily ported to new architectures (*ARM, PowerPC*) and operating systems (*Linux, MacOSX, WinCE*). It also turned up a variety of bugs (such as non-initialization of variables) that show up on some architectures but not on others.

When the initial implementation targets two or more alternatives, it is more portable.

8.2 Languages

Many VE systems involve the use of two languages: the language that the basic toolkit is written in (requiring low latency and low-level connections to device drivers) and scripting language (s) that support animation and perhaps interaction development within the environment (supporting rapid prototyping and flexibility). This raises the question of what the base language for a VE toolkit should be, and also how to best support one or more scripting languages. The variety of favorite languages for different groups indicates that a VE toolkit must work with a number of (existing and future) languages if it is to be widely adopted.

Field Experience—Regarding the language used to implement the toolkit, we have seen a transition in languages from C and Fortran to C++, Tcl/Tk, Java, C#, and Python. Several languages have come into existence since the early 1990s and new ones are still arriving, Ruby being a recent example. The changes in languages suggest that long-term viability of a VE toolkit will require that it be able to grow to support languages that did not exist at the time it was developed. Such adaptability will require the capability for automatic translation found in tools such as *SWIG* (SWIG), which automatically converts annotated C++ code to a variety of other languages. This implies that today (mid-2009) the base language should probably be C ++, unless new translators arrive that accept other languages for conversion. C++ is also presently the language that is most compatible with low-level device drivers that must at some level be part of the system.

Case Study: VTK—The *Visualization ToolKit* (*VTK*) takes the approach of using C++ as a base language with automatic translation to other languages (Law, Martin, & Schroeder, 1999). This has enabled our NIH National Research Resource team at UNC to implement visualization and data-manipulation applications in C++, Tcl/Tk, Python, and Java. In one case an application was prototyped in Python and then rewritten in Java (and is now being considered for implementation in *Qt*/C++) without rewriting the basic C++ and *VTK*-pipeline-based application core. Such portability enables us to develop local expertise on *VTK* basics, while providing the freedom to implement projects in the language that best suits the problem, data types, and programmer skills.

Case Study: *Alice*—The *Alice* toolkit provides an excellent example of providing both a low-level language and an interpreted language to control interaction and animation (Pausch et al., 1994). This layered approach enabled teams including non-programmers to develop interactions based on abstract interfaces to the lower-level toolkit. In the course of its development, *Alice* has supported two different scripting interfaces: the original Python interface was recently changed to Java (using *Jython* to maintain support for Python scripts).

Nugget: The ability of a system to be converted to multiple languages can enable flexibility in choice of a scripting language. The fact that the basic graphics and device functions are written in C++ enables the rapid processing needed for real-time, low-latency operation while an interpreted language is fast enough for the animation/interaction functions.

8.3 Licensing

Case Study: *VRPN*—To encourage wide adoption among both open-source toolkits and commercial vendors, and to enable use by companies that sell virtual-environment systems, we chose to release *VRPN* into the public domain. The *GNU General Public License* (at that time the *GNU Public License*) was considered and not selected because it was too restrictive for comfortable use by commercial entities. Other licenses, such as the *BSD* license model, would also allow this flexibility.

9 Extensibility

Nugget: The application programmer should have to learn as little as possible to attain a certain goal (rather than having to understand the whole API to make what seems to be a minor change to one part of it).

9.1 Case Study: VTK

The *Visualization ToolKit* (*VTK*) is a good example of appropriate code factorization that affords extensibility to provide the types of features needed by VE applications (Kitware, 2003). A discussion of the techniques that deserve to be emulated can be found in the supplemental material accompanying the online version of this article.

10 Case Study: Selecting a Toolkit for *EVEIL* and *EVEIL2*

One of the most popular virtual environments at UNC is the “Pit,” a variation of a similar environment first developed at University College London (Mel Slater, Usoh, & Steed, 1995). UNC’s year 2000 version of the Pit was used to validate the use of physiological stress measures as surrogates for questionnaire based measures of presence (Meehan, 2002). In the environment, participants were asked to drop objects onto targets located on the virtual floor, located twenty feet below a narrow ledge. The natural-walking interface coupled with the simulation's visual fidelity proved highly immersive (as evidenced by the fear response it induced). Four years later, we redesigned the application to use modern libraries. We selected *Wild Magic 3.0* (Eberly, 2008) as our rendering toolkit for three reasons: heritage, minimalism, and documentation.

Heritage

The UNC 2000 *Pit* demo had used the original *Wild Magic*, and our implementation team had some experience already with the second version of that engine. While the internals of the renderer had changed dramatically between the three versions, the API design remained relatively static. We would not have to radically redesign our application loop.

Minimalism

Game engines are often designed around showcase games (*e.g.*, *Unreal Engine 3* and the game *Gears of War*), and integrate the games' requirements deeply in their own structure. We rejected one notable first-person shooter game engine for precisely this reason: despite extensive toolset support, it made such deep assumptions about avatar motion that integrating it with *VRPN* would require significant effort. We rejected another open-source simulation engine for a similar reason: it integrated several well-designed and modular libraries into a cohesive whole, but the resulting package was wholly monolithic; even stereo rendering support required bypassing most of the integration code.

Documentation

We evaluated one high-publicity game engine aimed at hobbyists and independent developers. Although the engine shipped with documentation, its coverage was uneven. The developers had concentrated their reference and tutorial documentation on the art content pipeline and in-game scripting system. Details on the engine's entity replication system and rendering internals were scant at best and these systems were almost treated as inviolate black boxes. As these were the two areas we most needed to modify, we decided to pass on the engine rather than puzzle out the architecture.

Previous iterations of our code base had made significant changes to the underlying *Wild Magic* code to support such features as stereo rendering. Combined with minimal use of revision control systems, it quickly became impractical to upgrade the *Wild Magic* components to newer releases. This had the expected adverse effect on code maintainability; not only were engine fixes rarely integrated into our custom branch, but there was a great tendency to add side effects and subtle dependency requirements into the modified code. When we designed our new Effective Virtual Environment Intermediate Layer (EVEIL) toolkit, we made an explicit design decision to bar such practices in the new code. *Wild Magic*, *VRPN*, and other libraries were to remain unmodified, even if this led to a larger number of mostly-trivial derived classes in *EVEIL*. This approach paid off immediately, as we were able to upgrade between minor releases of the renderer and other libraries with minimal difficulty. Debugging also became easier, as we were able to treat these packages as black boxes and focus mainly on their interaction with our code.

Although we had succeeded with *EVEIL* in creating a reusable software framework for virtual environments, the underlying renderer was again beginning to show its age by late 2007. Shader, shadow, and multi-texturing support were minimal, and we were now contemplating new studies that required a higher degree of visual fidelity in these areas than our renderer could provide. *Wild Magic 3* had been superseded by *Wild Magic 4*, a breaking change that would require significant work to integrate with *EVEIL*. Because an upgrade would essentially involve porting *EVEIL* to a new renderer, we once again considered the other game engines available rather than unhesitatingly switching to *Wild Magic 4*.

Emergent Game Technologies, makers of the *Gamebryo* game engine, had recently unveiled an academic license program: qualified educational institutions could receive a binary-only license for *Gamebryo* free of charge (Emergent Game Technologies, 2007). We requested an evaluation copy, and eventually decided to switch to *Gamebryo* and create *EVEIL2*. This switch had several advantages:

As a leading commercial game engine, *Gamebryo* actively pursued the state of the art in gaming graphics. The latest shader programming models, including those in *DirectX 10*, were fully supported. We now had a shadowing system, per-pixel lighting support, and other graphical capabilities previously deemed too expensive or time-consuming to implement with *Wild Magic*.

Given the massive art requirements for top-of-the-line games, Emergent had invested heavily in its engine's content pipeline. *Gamebryo*'s exporter supported most *3D Studio Max* and *Maya* functionality without crashing.

Finally, we also had active support resources. While as non-paying customers we were ineligible for direct technical support, we had access to online forums where our questions would be answered by Emergent employees and other *Gamebryo* customers. Additionally, two of our group members had previously interned at Emergent, thus pre-seeding our group with *Gamebryo* proficiency. (Disclaimer: One of the authors is now himself an intern at Emergent.)

10.1 Results of EVEIL and EVEIL2

Despite these advantages, we did run into some difficulties when adapting *Gamebryo* for *EVEIL2*. The license we signed was one source of stress; we were one of the first academic licensees and thus ran into potential conflicts with the wording of earlier versions of the license. The license also includes NDA provisions, making it somewhat more difficult to set *EVEIL2* up at other sites. While binaries are freely distributable under the terms of the academic license, the protection system for *Gamebryo 2.3* made it difficult to distribute compiled code to non-licensees. (This defect was later corrected in the next version of *Gamebryo*.)

Gamebryo does not support *OpenGL* as a rendering interface, implementing only a *Direct3D* backend on PCs. Quad-buffering for hardware stereo support is therefore unavailable and multi-monitor 3D acceleration is impaired under *Windows XP* regardless of the specific engine used. We eventually resorted to nonstandard features of the video driver to achieve a reasonable frame rate, and are investigating newer versions of *Windows* and the video driver to see if the flaw is still present. Also, without source code some bugs are naturally harder to find. While lack of source removes the temptation to modify the underlying engine altogether, we were occasionally reliant on the forums to answer simple questions about *Gamebryo* we would have been able to answer ourselves with brief code inspection.

The primary design goals of *EVEIL* remained constant for the new version. We still wanted a lightweight wrapper around *VRPN* and wanted our renderer to speed the creation of virtual environments. In this we succeeded, but scheduling constraints resulted in a more lightweight application layer than we had originally intended. Many of the advanced rendering options are implemented in the *Gamebryo* sample application layer, which we were forced to re-implement as part of adding stereo support. Although *Gamebryo* provides a level-editing tool and many post-processing effects, they are not currently available to *EVEIL2* apps. Mindful of our commitment to modularity, we have left these options open for future development.

10.2 Further Observations on *EVEIL* and *EVEIL2*

Further observations on *EVEIL* and *EVEIL2* are available in the supplemental material accompanying the online version of this article.

11 Less can be better

Nugget: It is at least as bad to put too much into a VE toolkit as to put too little.

This nugget manifests itself in two ways: excluding audiences by selecting an implementation they don't like, and complexity of configuration and building.

Case Study: *VRPN*

Two examples of complexity reduction turned up during our development of the *VRPN* library (Russell M. Taylor II et al., 2001). The first was our decision to produce a heavyweight server-side version of the library (which had to link with all of the vendor-specific device libraries) and a more lightweight client-side version (which only had to link with the *VRPN*-defined interfaces). This reduced the size of client executables, but more importantly it removed the build and link complexity from the application programmer. This separation was not carried over to the *Windows* build environment, with the result that developers complain about the difficulty of building applications (it requires re-building the library with different features disabled).

A particularly frustrating example of build complexity was the set of constraints that kept us from using the *Standard Template Library* in *VRPN*. Initially, this was because of the two competing versions of the *STL* (one included vector.h, one vector); continually it has been because different vendors implement *STL* in incompatible ways. Other libraries used incompatible versions of the *STL*, effectively making it impossible to use both *VRPN* and the other library. We solved this within *VRPN* by keeping all traces of the *STL* from the header files (abstracting objects in the header files as needed) so that application code never tried to include them. We also avoided all use of *STL* within *VRPN* itself, which had the unfortunate consequence of forcing reimplementation of what is essentially the vector class.

Replacing STL by a more modern portable library, such as *Boost*, would avoid these inconsistencies but would introduce more of the first type of complexity, where a user of *VRPN* also has to download and install another external package that does not come built in to their compiler. It is surprising how even one or a few such dependencies can form practical barriers that reduce the number of users of a library. Addressing this complexity by using an auto-fetch and auto-build system such as *CMake* requires yet another tool to be downloaded, installed, and understood and often increases the perceived difficulty of using the library.

Field Experience

At the Desert Research Institute, we attempted to build and run one of the available VE integration libraries but in the end were unable to get it running and configured in our *CAVE*™. The first hurdle, which we did overcome, was the sheer number of dependencies that had to be set up correctly to get the toolkit to build. We never did get it configured to work properly in our *CAVE*™, so we are unable to support it as a toolkit choice for our users.

Field Experience

An example of excluding audiences is listed above in the case study on *EVEIL*; one toolkit in particular was rejected because it included hard-coded entity behaviors. Entity architectures simplify the main application loop by separating the components of the environment (items, avatars, and non-player characters) into modular, loosely coupled pieces. While not strictly required for multi-user simulations, entity systems are usually an integral component of game engine networking libraries. Because commercial games must run smoothly over the Internet, a network notoriously prone to jitter and latency, these networking libraries typically include prediction algorithms for minimizing bandwidth use and movement discontinuity. Unfortunately, they also tend to interfere with the *VRPN* tracker update model. We rejected one first-person shooter game engine for precisely this reason; we would have had to make significant changes to the player movement system to add head tracking to avatars.

We considered another open-source gaming and simulation engine on the advice of a collaborator. At first glance, it seemed ideal for our purposes: a thin wrapper layer integrating a renderer, physics engine, animation system, and more. The project leads had chosen a different approach from our eventual path, though. They encapsulated these systems within their own class hierarchy, discouraging use of the underlying engine components. Very few of these classes were modifiable through inheritance and sub-classing, meaning that extensive alterations would be required to implement systems as simple as stereo rendering. When the full extent of these problems became clear, we decided to abandon our prototype and look elsewhere for an engine.

Nugget: A VE toolkit should make difficult things easier without making easy things more difficult, and it must not make desirable things impossible.

12 Other sources of nuggets and experience

Gary Bishop and others' report on research directions for virtual environments has comments from 18 researchers in the field (Bishop et al., 1992). Mark Mine's report describes ten years of work at Disney creating location-based and ride-based virtual environments (Mine, 2003). Jim Chung and others' HMD virtual worlds paper describes the issues being dealt with even before the work described here, including latency, resolution, and field-of-view challenges that are still relevant today (Chung et al., 1989). Of course, books on VE systems, such as (Sherman & Craig, 2002), provide in-depth coverage of all aspects of development.

Supplementary Material

Refer to Web version on PubMed Central for supplementary material.

Acknowledgments

The requirements sections include input from conversations with people at EVL (U. Illinois) and the ARPA Enterprise (D.C.). They included direct conversations with Gary Bishop, Erik Erikson, Mark Finch, Stefan Gottschalk, Trey Greer, Richard Holloway, Kimberly Jones, Anselmo Lastra, Bill Mark, Leonard McMillan, Mark Mine, Marc Olano, Eddie Saxe, Andrei State, Greg Turk, Brian Upton and Hans Weber.

The "time in VR" group included Gary Bishop, Amy Henderson, Jon Leech, Vern Chi, Ron Azuma, Kevin Jeffay, Richard Holloway, Stefan Gottschalk, Marc Olano, Mark Mine, and Ellen Scher, and Hans Weber.

The *EVEIL* development group included Christopher Oates and Luv Kohli.

The work described here was supported by grants from the following agencies: DARPA, NIH, NSF, ONR, DOD, and ARPA.

The UNC work took place in the context of ongoing research projects being supervised by Frederic P. Brooks, Jr. and Henry Fuchs. Their vision and constant push for excellence drove the progress reported here and the collegial, collaborative atmosphere they fostered promoted a free sharing of ideas and techniques that enabled us to learn these lessons.

Bibliography

Adachi, Y.; Kumano, T.; Ogino, K. Intermediate Representation for Stiff Virtual Objects; Paper presented at the Proc. IEEE Virtual Reality Annual International Symposium (VRAIS'95); Research Triangle Park, NC. 1995, March 11-15;

Aliaga, D.; Cohen, J.; Wilson, A.; Baker, E.; Zhang, H.; Erikson, C., et al. MMR: An interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration; Proceedings of the 1999 ACM Symposium on Interactive 3D Graphics; 1999; p. 199-206.p. 237

Arthur, K.; Preston, T.; T, RM., II; Brooks, Frederick P., J.; Whitton, Mary C.; Wright, WV. Designing and Building the PIT: a Head-Tracked Stereo Workspace for Two Users; Paper presented at the 2nd International Immersive Projection Technology Workshop; Ames, Iowa. May 11-12, 1998; 1998.

Azuma, R.; Bishop, G. Improving Static and Dynamic Registration in an Optical See-Through HMD; Paper presented at the Proceedings of SIGGRAPH '94; Orlando, Florida. 1994, July 24--29;

Bierbaum, A.; Just, C.; Hartling, P.; Meinert, K.; Baker, A.; Cruiz-Neira, C. VR Juggler: a virtual platform for virtual reality application development; Paper presented at the IEEE Virtual Reality 2001; 2001;

Bishop G, Bricken W, Frederic P, Brooks J, Brown M, Burbeck C, Durlach N, et al. Research Directions in Virtal Environements: Report of an NSF Invitational Workshop held March 23-24, 1992 at UNC Chapel Hill. Computer Graphics 1992;26(3):153–177.

Brooks, FP., Jr.; Ouh-Young, M.; Batter, JJ.; Kilpatrick, PJ. Project GROPE - Haptic displays for scientific visualization; Paper presented at the Computer Graphics: Proceedings of SIGGRAPH '90; Dallas, Texas. August 1990; 1990.

Bryson, ST.; Johan, S. Time Management, Simultaneity and Time-Critical Computation in Interactive Unsteady Visualization Environments; Paper presented at the IEEE Visualization '96; October 1996; 1996.

Burette A, Feng D, Marshburn D, Jen D, Weinberg R, T RM II. ToolBox: Stepping into the third dimension. Journal of Neuroscience 2007;27:12757–12760. [PubMed: 18032646]

Burns E, Razzaque S, Panter AT, Whitton MC, McCallus MR, Brooks FP. The hand is Slower than the Eye: A quantitative exploration of visual dominance over proprioception. Journal on Presence: Teleoperators and Virtual Environments 2006;15:1–15.

Chung, JC.; Harris, MR.; Brooks, FP.; Fuchs, H.; Kelly, MT.; Hughes, J., et al. Exploring Virtual Worlds with Head-Mounted Displays; Paper presented at the Proceedings of SPIE: Non-Holographic True Three-Dimensional Displays; 1989;

Cruz-Neira, C.; Sandin, DJ.; DeFanti, TA. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE; Paper presented at the SIGGRAPH '93; Anaheim California. 1993, August 1-6;

Eberly, DH. Wild Magic real-time 3D graphics engine. 2008. Retrieved January 20, 2009, from http://wwww.geometrictools.com

Emergent Game Technologies. Emergent puts award-winning development tools in students' hands with innovative academic program. 2007. Retrieved January 25, 2009, from http://www.emergent.net/en/News/Press-Release-Archives/Emergent-Puts-Award-Winning-Development-Tools-in-Students-Hands-with-Innovative-Academic-Program/

Eyles, J.; Molnar, S.; Poulton, J.; Greer, T.; Lastra, A. PixelFlow: The Realization; Paper presented at the 1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware; August 1997; 1997.

Feasel, J.; Whitton, MC.; Wendt, JD. LLCM-WIP: Low-latency, continuous-motion walking-in-place; Paper presented at the Proceedings of IEEE Symposium on 3D User Interfaces; Reno, NV. March 2008; 2008.

Fuchs, H.; Poulton, J.; Eyles, J.; Greer, T.; Goldfeather, J.; Ellsworth, D., et al. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories; Paper presented at the SIGGRAPH '89; 1989;

Gossweiler, R.; Long, C.; Koga, S.; Pausch, R. DIVER: a DIstributed Virtual Environment Research Platform; Paper presented at the IEEE 1993 Symposium on Research Frontiers in Virtual Reality; 1993;

Grant, B.; Helser, A.; T, RM, II. Adding Force Display to a Stereoscopic Head-Tracked Projection Display; Paper presented at the proceedings of VRAIS '98; Atlanta. 1998;

Holloway R. Registration error analysis for augmented reality. Presence 1997;6(4):413–432.

Holloway, RL. Registration Errors in Augmented Reality Systems. The University of North Carolina; Chapel Hill: 1995.

Jen, D.; Parente, P.; Robbins, J.; Weigle, C.; Burette, A.; Weinberg, R., et al. ImageSurfer: A Tool for Visualizing Correlations between Two Volume Scalar Fields; Paper presented at the IEEE Visualization 2004; Austin, Texas. 2004, October 10-15;

Kitware, I. The Visualization Toolkit User's Guide. Kitware, Inc.; 2003.

Kohli, L.; Whitton, MC. The Haptic Hand: Providing User Interface Feedback with the Non-Dominant Hand in Virtual Environments; Paper presented at the Proceedings of Graphics Interface 2005; 2005;

Kreylos, O.; Bethel, EW.; Ligocki, TJ.; Hamann, B. Virtual-reality based interactive exploration of multiresolution data. In: Farin, G.; Hamann, B.; Hagen, H., editors. Hierarchical and Geometrical Methods in Scientific Visualization. Springer-Verlag; Heidelberg, Germany: 2000. p. 205-224.

Law, C.; Martin, KM.; Schroeder, WJ. Multithreaded Streaming Pipeline Architecture; Paper presented at the IEEE Visualization 99; San Fransisco, CA. 1999, 115-122;

Leech, J.; T, RM, II. Interactive Modeling Using Particle Systems; Paper presented at the Proceedings of the 2nd Discrete Element Methods conference at MIT; Boston. 1992, November;

Lok B, Naik S, Whitton M, Brooks F. Effects of Interaction Modality and Avatar Fidelity on Task Performance and Sense of Presence in Virtual Environments. Journal on Presence: Teleoperators and Virtual Environments 2003;12(6):615–628.

Mark, W.; Randolph, S.; Finch, M.; Verth, JV.; T, RM, II. Adding Force Feedback to Graphics Systems: Issues and Solutions; Paper presented at the Computer Graphics: Proceedings of SIGGRAPH '96; 1996, August;

Marshburn D, Weigle C, Wilde BG, Desai K, Fisher JK, Cribb J, et al. The Software Interface to the 3D-Force Microscope. Proceedings of IEEE Visualization 2005:455–462.

Meehan M, Insko B, Whitton MC, Brooks FP. Physiological Measures of Presence in Stressful Virtual Environments. ACM Transactions on Graphics 2002;21(3):645–652.

Mine, M. Unpublished Ph.D. University of North Carolina; Chapel Hill: 1997a. Exploiting Proprioception in Virtual-Environment Interaction.

Mine, M. Computer-Aided Design 1997. 1997b. ISAAC: A Meta-CAD System for Virtual Environments.

Mine, M. Towards Virtual Reality for the Masses: 10 Years of Research at Disney's VR Studio; Paper presented at the Proceedings of the workshop on Virtual Environments 2003; Zurich, Switzerland. 2003;

Mine, M.; B, FP., Jr.; Sequin, CH. Moving Cows in Space: Exploiting Proprioception as a Framework for Virtual Environment Interaction; Paper presented at the Proceedings of SIGGRAPH 97; Los Angeles, CA. 1997;

Mine M, Weber H. Large Models for Virtual Environments: A Review of Work by the Architectural Walkthrough Project at UNC. Presence 1995;5:136–145.

Ouh-young, M.; Beard, DV.; Brooks, FP. Force Display Performs Better Than Visual Display in a Simple 3-D Docking Task; Paper presented at the Proceeding of IEEE Robotics and Automation Conference; Scottsdale, AZ. May 14-18, 1989; 1989.

Pausch, R.; Conway, M.; DeLine, R.; Gossweiler, R.; Maile, S.; Ashton, J., et al. Alice & DIVER: A Software Architecture for the Rapid Prototyping of Virtual Environments; Paper presented at the Course notes for SIGGRAPH '94 course "Programming Virtual Worlds"; 1994;

Peck TM, Fuchs H, Whitton MC. Evaluation of Reorientation Techniques for Walking in Large Virtual Environments. Transactions on Visualization and Computer Graphics 2009;15(3):383–394. [PubMed: 19282546]

Qi, W.; Taylor, RM., II; Healey, C.; Martens, J-B. User Centered Design for Medical Visualization. Idea Group Publishing; 2008. 3D Interaction with Scientific Data through Virtual Reality and Tangible Interfacing.

Quammen, CW.; Richardson, A.; Haase, J.; Harrison, B.; Taylor, RM., II; Bloom, KS. FluoroSim: A Visual Problem Solving Environment for Fluorescence Microscopy; Proceedings of Visual Computing for Biomedicine; Delft, Netherlands. 2008; p. 151-158.

Razzaque, S. Redirected Walking. The University of North Carolina at Chapel Hill; Chapel Hill: 2005.

Robinett, W.; Holloway, R. Implementation of Flying, Scaling, and Grabbing in Virtual Worlds; Paper presented at the Proceedings of the ACM Symposium on Interactive 3D Graphics; Cambridge, MA. 1992;

Shaw, C.; Liang, J.; Green, M.; Sun, Y. The decoupled simulation model for VR systems; Paper presented at the Proceedings of CHI ‘92; 1992; p. 321-328.P. o. C.

Sherman, WR.; Craig, AB. Understanding Virtual Reality: Interface, Application, and Design. Morgan Kaufmann; 2002.

Slater M, Steed A. A Virtual Presence Counter. Presence: Teleoperators & Virtual Environments 2000;9 (5):413–434.

Slater M, Usoh M, Steed A. Taking steps: The influence of a walking technique on presence in virtual reality. ACM Transactions on Computer-Human Interaction 1995;2(3):201–219.

Sokolewicz, M.; Wirth, H.; Böhm, K.; John, W. Using the GIVEN++ Toolkit for System Development in MuSE; Paper presented at the Proceedings of First Eurographics Workshop on Virtual Reality; Polytechnical University of Catalonia. September 1993; 1993.

Ståhl, O.; Andersson, M. DIVE - a Toolkit for Distributed VR Applications; Paper presented at the Proceedings of the 6th ERCIM workshop; Stockholm. 1994, June 1-3;

SWIG. from http://www.swig.org/

Taylor, RM, II. Requirements and Availability of Application Programmer’s Interfaces for Virtual-Reality Systems. Department of Computer Science, University of North Carolina; Chapel Hill: 1995. Technical Report No. TR95-009

Taylor, RM., II; Hudson, TC.; Seeger, A.; Weber, H.; Juliano, J.; Helser, AT. VRPN: A Device-Independent, Network-Transparent VR Peripheral System; Paper presented at the Proceedings of the ACM Symposium on Virtual Reality Software & Technology; VRST, Banff Centre, Canada. 2001, November 15-17;

Taylor, RM., II; Robinett, W.; Chi, VL.; Frederic, P.; Brooks, J.; Wright, WV.; Williams, RS., et al. The Nanomanipulator: A Virtual-Reality Interface for a Scanning Tunneling Microscope; Paper presented at the SIGGRAPH 93; Anaheim, California. August 1-6, 1993; 1993.

Usoh, M.; Arthur, K.; Whitton, MC.; Bastos, R.; Steed, A.; Slater, M., et al. Walking > Walking-in-Place > Flying, in Virtual Environments; Paper presented at the Computer Graphics Series Proceedings; 1999;

Welch, G.; Bishop, G.; Vicci, L.; Brumback, S.; Keller, K.; Colucci, D. n. The HiBall Tracker: High-Performance Wide-Area Tracking for Virtual and Augmented Environments; Paper presented at the Symposium on Virtual Reality Software and Technology 1999 (VRST 99); University College London. 1999, December 20-22;

Welch RB, Blackmon TT, Liu A, Mellers BA, Stark LW. The effects of pictorial realism, delay of visual feedback, and observer interactivity on the subjective sense of presence. Presence: Teleoperators and Virtual Environments 1996;5(3):263–273.

Whitton M. Making Virtual Environments Compelling. Communications of the ACM 2004;46(7):40–47.

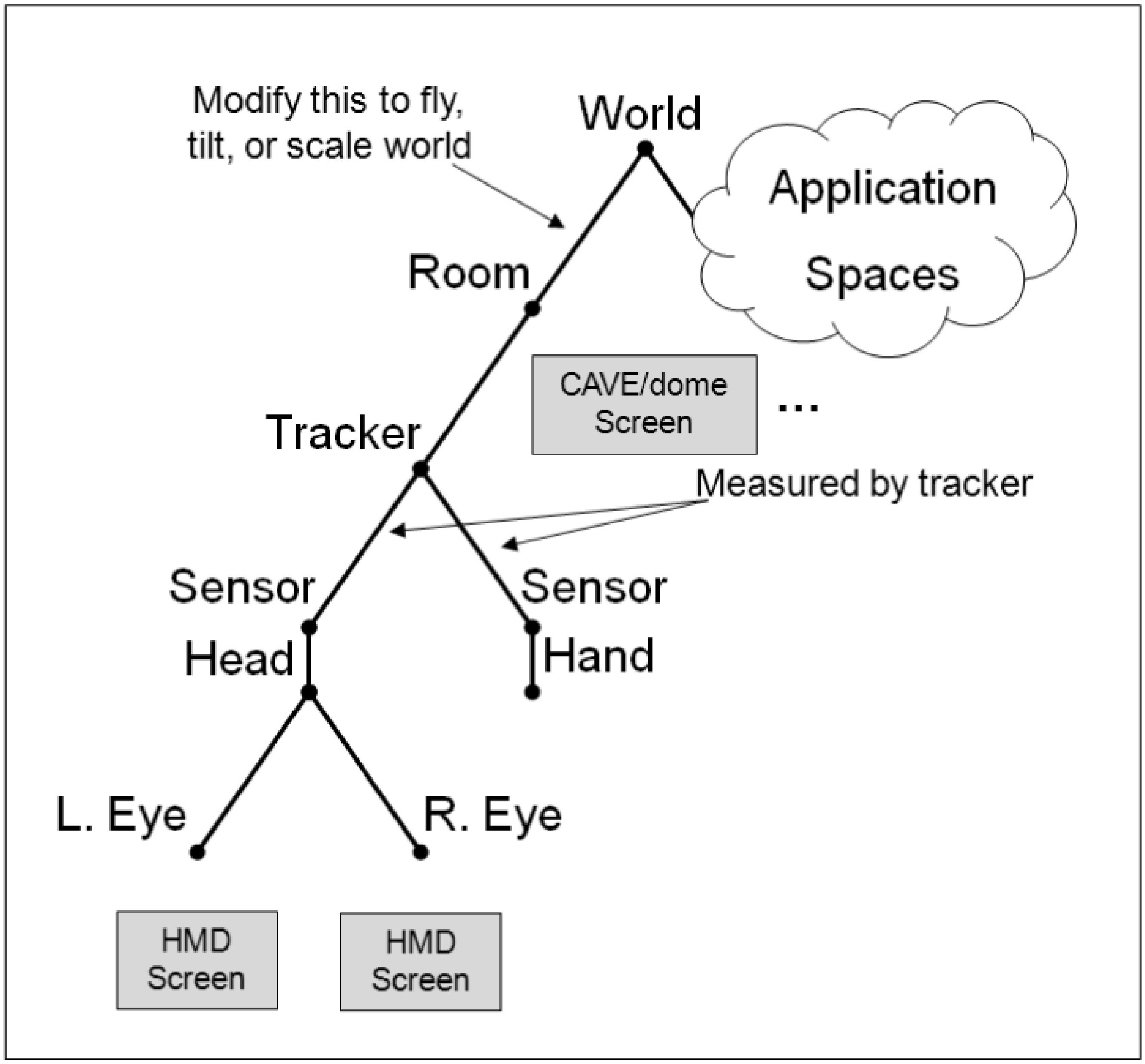

Figure 1.
Modified VE-space tree from (Robinett & Holloway, 1992). Dark tree shows spaces: World space is where the VE-toolkit-specific spaces meet the application spaces; Room space is based on the vehicle the user flies around in; there is one Tracker space for each tracker in the system, as many Head and Hand spaces as are tracked, and two eyes for stereo displays. Display screens for HMD systems are located in Eye space for the appropriate eye; displays for *CAVE*s, domes, *ImmersaDesk*s, and other projection systems live in the Room; together with the eye positions, they determine the viewing transformations.