G. Drew Kessler
dkessler@eecs.lehigh.edu
Department of Electrical Engineering
and Computer Science
Lehigh University

Doug A. Bowman
Dept. of Computer Science
Virginia Polytechnic Institute
and State University

Larry F. Hodges
College of Computing
Georgia Institute of Technology

The Simple Virtual Environment Library:

An Extensible Framework for Building VE Applications

Abstract

As virtual environment (VE) technology becomes accessible to (and affordable for) an ever-widening audience of users, the demand for VE applications will increase. Tools that assist and facilitate the development of these applications, therefore, will also be in demand. To support our efforts in quickly designing and implementing VE applications, we have developed the Simple Virtual Environment (SVE) library. In this article, we describe the characteristics of the library that support the development of both simple and complex VE applications. Simple applications are created by novice programmers or for rapid prototyping. More-complex applications incorporate new user input and output devices, as well as new techniques for user interaction, rendering, or animation. The SVE library provides more-comprehensive support for developing new VE applications and better supports the various device configurations of VE applications than current systems for 3-D graphical applications. The development of simple VE applications is supported through provided default interaction, rendering, and user input and output device handling. The library's framework includes an execution framework that provides structure for incrementally adding complexity to selected tasks of an application, and an environment model that provides a layer of abstraction between the application and the device configuration actually used at runtime. This design supports rapid development of VE applications through incremental development, code reuse, and independence from hardware resources during the development.

1 Introduction

The development of VE applications is an area inviting experimentation. In particular, the field is open to new applications, different device configurations, new techniques for rendering, interaction, and model maintenance. Such experimentation is most productive when new ideas and designs can be implemented quickly and then compared to previous implementations. However, developing even simple VE applications with limited or no software support requires a considerable amount of expertise and development time. In addition, it is quite difficult to reuse modules of VE applications in projects that use different configurations.

To support the rapid development of novel VE applications that may introduce new environments, behaviors, or software techniques, we have developed

Presence, *Vol. 9, No. 2, April 2000,* 187–208

the Simple Virtual Environment (SVE) library and runtime system. We designed SVE to meet the following goals.

1. Provide software support for the creation of simple VE applications by novice developers.
2. Provide a framework for extending simple VE applications to include additional technology or more-complex behavior.
3. Allow an application to be developed independently from the system configuration used at runtime.

Supporting the rapid development of simple VE applications is critical for quickly bringing people up to speed with the technology and interaction and rendering techniques common to VE applications. A framework for extensions to the default behavior allows for rapid, focused experimentation with new techniques in different aspects of the application; further, it allows these new techniques to be easily incorporated into VE applications that can benefit from them. One area of experimentation is determining, for a particular VE application, the best configuration of input devices (such as tracking, glove, and button devices) and output devices (such as head-mounted displays (HMDs), stereoscopic or single-view projection screens, monitors, handheld displays, and audio speakers). Our design allows a variety of device configurations to be specified at runtime by providing a separation between the devices used and the environment model, and by defining how the device input affects the model.

Like many 3-D scene-rendering systems, the SVE library provides a runtime system that generates a first-person point of view of a 3-D environment, described by a scene graph of geometric objects, and the library allows the user to fly through the environment (in the user's gaze direction) as a response to user input. The library, therefore, supports the rapid development of architectural walkthrough applications, as described by Brooks (1986). The main effort of developers for these types of VE applications is spent generating the geometric model of the environment using an independent CAD software package. To optimize the rendering process so that a high display rate can be obtained, the SVE run-time system utilizes many common algorithms and model representations, such as pregenerating display lists and geometric transformations for objects, and quickly culling objects not in view using simple bounding-sphere or -box representations of objects.

What sets SVE apart from other 3-D rendering systems is that it provides comprehensive support for extensions that can be developed independently and easily integrated, and that it allows for a variety of input and output device configurations to be selected for a particular application with little or no additional programming. Device independence is achieved by using the SVE environment model as the interface between input and output devices and the application. The environment model describes the dynamic state of the application's 3-D environment and includes a representation of the user in the environment.

As Figure 1 shows, the SVE framework provides the environment model and an execution cycle that ensures default routines are invoked at the right time to handle input, output, and other actions that examine and / or modify the environment model. The SVE system also provides distributable components to interface with devices connected to remote computers. In addition, the figure demonstrates that the SVE framework allows an application to provide its own routines, invoked by the execution-cycle module, to handle devices, interactions, animations, and rendering by interfacing with the environment model.

After discussing related work in the next section, we present the design of the environment model and the user representation in that model. We demonstrate how our design allows for a wide variety of display and user input devices and configurations, which can be changed with little or no additional application programming. In Section 4, we describe how the framework of the library supports extensions in all aspects of VE applications, how the framework supports new complex mechanisms such as interaction techniques and polling device handling, and how the environment model provides a common database that allows extensions to interface with each other as well as the system. We conclude by summarizing the aspects of the SVE system that make it an effective development tool.

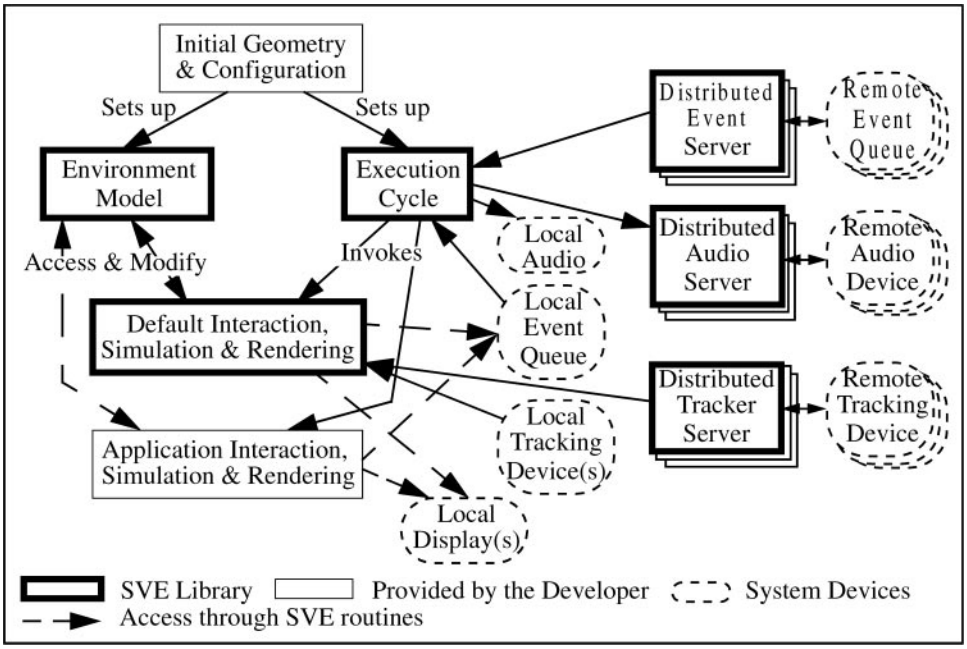

Figure 1. *The SVE framework.*

2 Related Work

As the technology for VE applications became available and was integrated, software systems and tools were created to support the development of applications using the technology. The first of such systems was the RB2 (Reality Built for 2) system created at VPL (Blanchard, 1990), which provided for the development of simple VE applications by novices to VE technology. Application developers, though, were constrained by a limited set of data-transformation nodes provided by the visual programming interface, Body Electric, which allowed for modifications to the scene and viewpoint as reactions to input data. The MR toolkit (Shaw et al., 1993), a widely used VE application development library developed at the University of Alberta, allowed for complex VE applications by providing the components necessary to transform existing 3-D visualization applications into immersive VE applications. The toolkit provided a low-level interface to distributed components of the VE application. Later additions provided support for geometric objects with behaviors, programmed using OML (Object Modeling Language, which was also developed at the University of Alberta), and provided support for a high-level description of the OML objects and their interaction in the environment using the Environment Manager (EM) tool (Wang, Green, & Shaw, 1995). Developing applications using the MR toolkit can be done at a low level, which does not provide geometry and rendering support, or at a high level using EM and OML. Using EM and OML, however, requires extra effort in learning a new language for describing organization and behaviors. Low-level application code can instance high-level OML objects, but it is not clear that a high-level application programmed and configured using EM is extensible to include low-level code, such as application-specific rendering or interface techniques.

Many VE application-support software systems decompose the VE application into components that can be executed concurrently, perhaps as part of a distributed system. One such system, VEOS (Bricken, 1994), partitions the VE application into self-contained entities that have an internal ‘‘persist’’ behavior and a ‘‘react’’ behavior that takes action on events identified by its ‘‘interact’’ behavior. The self-contained and hierarchical nature of the entities (entitities can be contained in other entities), and the use of the LISP language to define be-

haviors, provides a modular design that allows for an efficient distributed application. However, performance shortfalls of the system have led to its discontinued use. Another such system is VR-DECK (Codella et al., 1993), which provides for events to be passed between software components that may be distributed about a network. A VE application is defined by a set of components and their event connections. However, the components do not share a common view of the environment and each must rely on other components to produce the information about the environment required for the component's task.

The DIVE (Carlsson & Hagsand, 1993; Hagsand, 1996), AVIARY (Snowdon, 1994), and dVS (Ghee, 1995) systems provide a task-level decomposition of a VE application. Each system provides components common to VE applications, such as tracking device interfacing, geometric object maintenance, and visualizer components that render the user's view. An application is created by programming application components that introduce and manipulate geometric objects in the environment shared by the other components by using input from input-collection components. These systems provide for the development of complex, distributed VE applications but require considerable work to extend (as new components need to be written). Like the MR toolkit, the dVS system has a higher level add-on, called dVISE, which allows for the specification of interactive, animated environments in ASCII-text world definition files. Therefore, application development is done at a low level using dVS routines, or at a high level, after learning the dVISE environment model. Because dVS is not designed with ease of use in mind, extending past the capabilities provided by dVISE will require a large step in understanding the support system.

The Alice system (Pausch, 1995) provides an environment to rapidly prototype VE applications. An application developed using Alice consists of a number of scripts written in Python, an object-oriented, interpreted language, and possibly subroutines written in C or C++ which are referred to by the Python script. The Alice system provides Python classes that can be instantiated to introduce geometric objects into the environment and to define behaviors for geometric objects. Due to its interpretive nature, VE applications written using Python and the Alice system can be quickly altered to introduce new interactions and behaviors, but they will provide slower simulation speeds than compiled code. The rendering and view-determination tasks are decoupled from the simulation to achieve high frame rates independent of the simulation computation, but this decoupling prohibits extensions by the application developer to the rendering techniques used for the geometric objects.

The Avocado framework (Tramberend, 1999) also provides a script interface, using Scheme, which can access subroutines or objects written in C++. The framework is primarily designed to share scene graphics and application data in a distributed VE system, but it does not describe a framework for user representation in the environment model or for independence from input devices.

A few 3-D graphical application development toolkits exist that allow the designer to describe a 3-D model at a higher level than 3-D graphics libraries that simply draw primitives such as lines and polygons. These toolkits include Inventor (Strauss, 1993), Performer (Rohlf & Helman, 1994), Mirage (Tarlton & Tarlton, 1992), and Java3D (Sowizral et al., 1998). They provide a collection of object types that represent graphical shapes, lights, and groupings, which the application can create instances of and include in a scene graph. The Inventor and Mirage toolkits also include camera object types that determine the eye point and gaze direction in the scene for display rendering. The Java3D API improves on the camera model by including a ''view platform'' object in the scene graph, which is associated with a view object that defines how the user's view is generated from the point of view of the view platform. The view object can be configured for head-mounted display or head-tracked, fixed display configurations. On the other hand, Performer treats the viewing parameters separately from the scene graph.

Although these toolkits can be used to develop VE applications with the addition of software to interface with the tracking, input, and output devices used, they are primarily designed for scene animation and visualization on a flat screen. On the other hand, the WorldToolKit (WTK) library developed by Sense8 (1998) provides a 3-D scene renderer that does provide for incorporating

tracking device input and other input to manipulate the viewpoint and geometric objects in the environment, as well as allowing for different display outputs. However, developers using the WTK library must maintain in their program the association of the user's viewpoint, body-monitoring devices, and the objects in the environment—associations that the SVE environment model maintains automatically. In addition, the SVE library provides a more complete framework for extensions and supporting application behavior.

The Bamboo system (Watsen & Zyda, 1998) supports the development of networked VE applications through a mechanism to combine required modules designed for specific subtasks of the application, and through providing a structure to the execution of those modules. Although it provides an extensible execution framework, it does not provide a structured model of the environment that, in the SVE library, serves as a common interface between the modules. Consequently, the Bamboo system depends on modules that provide the rendering of the environment and the support of simple interactions to be designed to interact with each other. In the SVE library, these basic functions of a VE application are provided by default routines (which can be replaced or augmented) that use the common environment model.

3 The SVE Environment Model

The SVE environment model was designed to provide a separation between an SVE application and the interface configuration of the application. An SVE application can be easily configured to use any number of positional tracking devices, including six-DOF trackers, gloves that report finger movements, and conventional mouse and keyboard control. In addition, an SVE application can be configured to generate a display for an HMD, a stationary display (desktop or wall-projected), or a handheld display. For example, the SVE library has been used to develop an immersive, airplane environment displayed on an HMD to a person being treated for a fear of flying (Hodges et al., 1996). Another application built using SVE—this one providing scientific visualization of 3-D information—used a desktop, stereo display (Obeysekare, 1996). The configuration can include selection from a range of rendering features, such as lighting, Gouraud shading, and texture mapping, to match the capabilities of the hardware running the application. When appropriate, an application can have a different behavior for different configurations, but for most common situations, the design of the SVE environment model makes special handling of different configurations generally unnecessary.

The separation of an application from its configuration increases the speed with which developers can create new applications by allowing development when the destination hardware configuration is not completely available, and by allowing experimentation with different configurations. For example, most of the development of a simulated animal's behavior in reaction to its environment and the user's distance and gaze direction can be accomplished without testing the application with the developer in a tracked HMD. Instead, the user's position and orientation can be controlled by mouse input. In addition, allowing an application to easily switch between configurations would make it easy to compare a user's response to an environment presented in an HMD with the environment projected in stereo onto a wall.

The SVE environment model contains a default set of elements that represent the user's presence in the environment. This user model serves as the interface between an application and the system configuration of a display and a set of input devices. Using this model, an application can easily determine the user's head, hand, or eye position and gaze direction as determined by the devices used in the configuration. We describe the design of the environment model and the user model in the next section. The use of particular displays and positional tracking devices is handled by automatically introducing a small group of elements into the environment and user model. In Section 3.2, we describe the elements introduced to represent display configurations and how they are used by the application. In Section 3.3, we describe how the system calculates the viewing parameters for rendering the user's view(s) for a particular display configuration. Finally, in Section 3.4, we describe the elements introduced to handle tracking, device input.

3.1 Modeling the Environment and the User

By their very nature, VE applications define an environment within which a user can explore and interact. This environment generally contains geometric objects, possibly with additional characteristics such as behavior or state, ambient and localized audio, and a representation of the user. Like many 3-D rendering systems, the SVE environment is modeled as a coordinate-system graph, or scene graph, of geometric objects. Each node of the graph has one parent node and any number of child nodes. The edge between a node, A, and its parent, B, is associated with a coordinate system transformation, $T_{B \leftarrow A}$, that transforms geometric points described in A's coordinate system into B's coordinate system. The transformations are stored as 4×4 matrices that generally perform affine transformations (translation, rotation, scale, and so on) on 3-D points that make up the object's geometry. Details on the use of transformation matrices in coordinate system graphs are given in the Appendix. In short, one can think of a node in the graph as representing a rigid geometry that is attached with a position and orientation offset (given by the transformation) to its parent geometry. An object's position (or, more precisely, the position of its origin) can be computed by composing the transformations between the object and the root of the coordinate system graph. In fact, an object in the SVE object tree may simply be a placeholder for a special position or coordinate system in the environment, having no visible geometry to be rendered. For example, the user representation in the SVE environment model is a collection of empty geometric objects that define the position of key components of the user. We will call this type of geometric object a *placeholder object* (PHO), although its special designation does not connote any special handling by the SVE system.

Virtual environment displays can be described as 3-D graphical displays that immerse a user within a 3-D environment. The presence of the user may define simply a viewpoint and gaze direction, or may include representations of the user's body, such as a hand, which may interact with the environment. The SVE environment model, therefore, automatically includes a subtree of PHOs representing relevant parts of the user's body, including the head, eyes, hands, and even fingers, if needed by the application. (The SVE user representation is similar to the coordinate system graph described by Robinett & Holloway, 1995.) This subtree is shown in Figure 2, attached as a child of a Workspace object that defines the location of the user's space in the encompassing environment. The purpose of each object in the tree is described in Table 1. Note that the graph shown in the figure is the initial state of the user representation. An application is free to change the graph, as appropriate. For example, if the user ''enters'' an airplane object, the Workspace object could be attached to the airplane object (by making the airplane object the parent of the Workspace object) so that the user will move with the airplane.

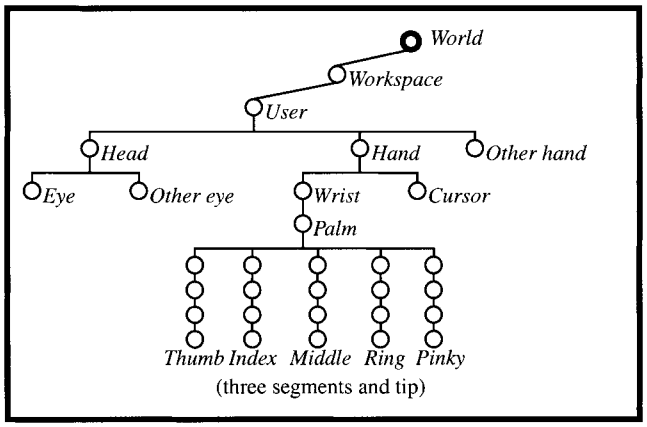

Figure 2. *Default object tree in the SVE environment model.*

The user model serves as the interface between the application and the devices used in the configuration for positional user input and display output. An application can discover the position and orientation of the user's head by asking for the position and orientation of the origin of the Head PHO in the world environment, its ''world position.'' This information can be easily obtained by calling an SVE routine that calculates the transformation matrix, $T_{W \leftarrow HMD}$, from the Head coordinate system to the World coordinate system. If a tracking device is attached to the user's head, the Head object will move appropriately. If, instead, the user moves the entire user representation using the mouse, the Head

Table 1. *Objects in the Default SVE User Representation (Only the Index Finger Shown)*

Object	Represents (SVE name)	Parent
World	The base coordinate system of the entire environment. (SVE WORLD)	none
Workspace	The space that the user moves in. The origin of this object's coordinate system generally represents a point on the floor in the workspace of the user. That point corresponds with a point in the virtual environment, defined by the coordinate-system transformation of this object to the World object. (ORIGIN)	World
User	The user in the environment. The origin of this object generally represents the position of the eyepoint in the most basic configuration (no tracking input devices, display on a desktop monitor), as the origins of all objects in the user tree generally are at the same point. (USER)	Workspace
Head	The head of the user, perhaps wearing an HMD. (SVE HMD)	User
Hand	The primary hand of the user. (SVE hand)	User
Other Hand	The secondary hand of the user. (SVE other hand)	User
Eye	The eyepoint of the user. The position of this object may represent the left or right eye, or the single eyepoint for monoscopic viewing. (SVE eye)	Head
Other Eye	A second eyepoint for display configurations that render both the left and right eyes of a stereo view. (SVE other eye)	Head
Cursor	A position that follows the hand, perhaps at a certain offset distance. Used for interactions with the environment. (SVE cursor)	Hand
Wrist	The wrist (and perhaps lower arm). Used for configurations that have wrist-bend sensors and that place a tracking device on the lower arm. (SVE wrist)	Hand
Palm	The palm of the hand, origin at the wrist attachment. (SVE palm)	Wrist
Index	The first segment of the index finger, origin at the attachment with the palm (the metacorpophalangeal joint). (SVE index)	Palm
Index pip	The second segment of the index finger, origin at the proximal interphalangeal joint. (SVE index pip)	Index
Index dip	The last segment of the index finger, origin at the distal interphalangeal joint. (SVE index dip)	Index pip
Index tip	The tip of the index finger. (SVE index tip)	Index dip

object will simply follow the User object, which will rotate with the movement of the mouse. In both configurations, the application uses the same method to determine the position and orientation of the user's head. Similarly, the application can implement interactions between the tip of the user's index finger and other geometric objects regardless of how the finger is controlled (by a tracked glove device, by a hand tracker and assuming a rigid hand, or by assuming a fixed distance from head to finger tip).

3.2 Handling Different Display Configurations

The SVE system can render one or more views, each of which are defined by an eyepoint (or eyepoints for field-sequential stereo images), a viewplane, and the root of an object tree to display. By default, the SVE system creates one view whose eyepoint corresponds to the Eye (and, if necessary, the Other Eye) PHO(s) in the user model, whose object tree is rooted at the World

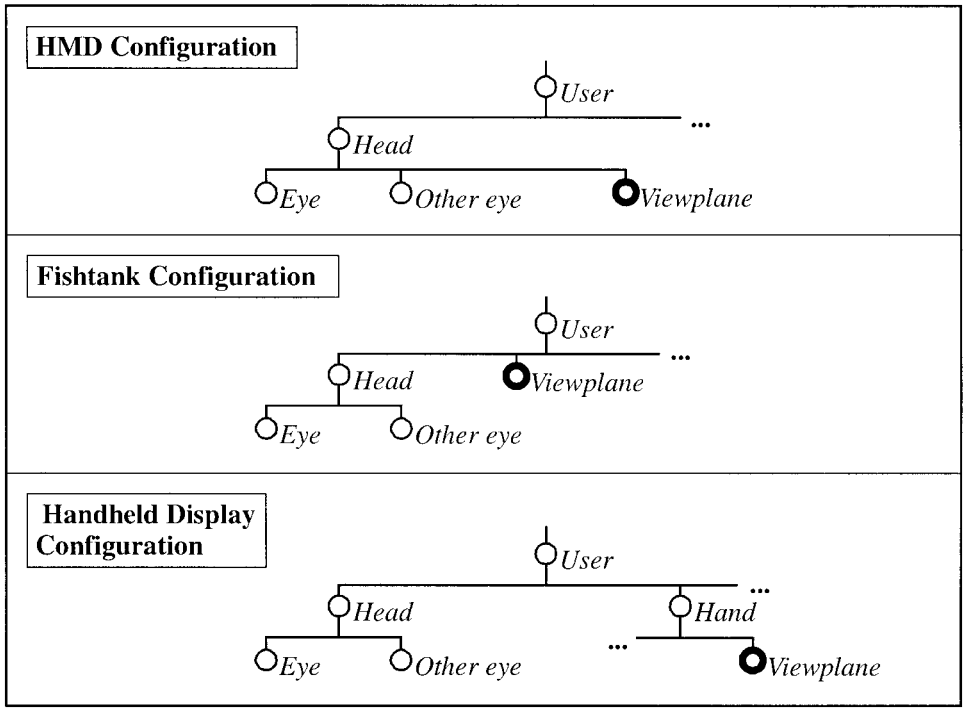

Figure 3. *Placement of the viewplane object in the SVE environment model for various display configurations. (Some default objects are not shown.)*

object. The viewplane of the new view is represented by the View Plane PHO, which is placed according to the display configuration. Our approach is similar but more general than the method presented by Southard (1995), as it does not enforce a particular tracking-device configuration and it ensures that the near and far clipping-plane distances are scaled with the eye PHO coordinate system. In addition to the standard view(s) from the user's eyepoint(s), an application may create new views that correspond to different perspectives, or views of entirely different environments.

For each view, the system calculates a viewing transformation that is used to transform geometric objects from their local coordinate system to the eye's coordinate system so that the primitives of the geometry are rendered from the user's point of view. The view's perspective is produced using a viewing transformation that is defined by the position and orientation of the object(s) representing the eyepoint(s), the object representing the viewplane, and the dimensions of the window, which are defined in the viewplane's coordinate system.

Our technique for defining the viewing configuration is different from the camera model used by Robinett & Holloway (1995), as well as many 3-D rendering systems such as Inventor (Strauss, 1993) and Alice (Pausch, 1995). In the camera model, the window through which the user sees the environment is placed at a set distance from the user's eye position. The camera model is appropriate for VE applications that use an HMD—where the window to the virtual world can be statically located in relation to the user's eye as the location and orientation of the HMD device is known, the window is positioned according to the specification of the HMD, and the locations of each of the user's eyes in relation to the HMD device can be approximated. However, this model does not support the ''fishtank'' display configuration (Deering, 1992)—where the window to the virtual world remains stationary within the user's workspace as it represents a computer monitor, projection screen, or desktop display—or handheld displays (Rekimoto, 1997), where the window to the virtual world is rendered to a handheld display that moves with the user's hand motions.

Our method also provides more flexibility than the Java3D API viewing configuration (Sowizral et al., 1998), which supports HMD and fishtank display configurations, in that the viewplane and eye, as well as trackers, are treated as first-class objects rather than attributes of a ''View object,'' that is an attribute of a ''View Platform'' object in the scene. Therefore, configurations such as multiple wall-projections using a single eyepoint or handheld displays are better supported by the SVE model.

The introduction of the viewplane object fits well in our design of the user representation in the environment model. Our design is intended to provide a model that could be more easily related to an application designer's understanding of the user's presence in the environment, rather than a model based on a particular tracking device set-up or display type. As is shown in Figure 3, the viewplane object is placed differently in the user model depending on the type of display being used. However, the process through which the viewing transformation is generated is the same: The window extents are transformed into eye coordinates and are used, along with near and far clipping-plane distances, to define a viewing volume. (See Figure 4.)

In addition to the positions of the View Plane object, other display configuration parameters give the extents of the window on the plane. For HMD configurations, the viewplane object is positioned to correspond to the optical projection plane of the HMD (with window extent values that provide the correct vertical and horizontal field-of-view angles for the HMD) and is ''rigidly attached'' to the Head object. For fishtank displays, the viewplane is the representation in the virtual world of the monitor screen in the real world. The viewplane does not move with the user's head, but stays stationary in the user's reference frame. Therefore, the viewplane object is attached to the Workspace object (or the User object, if the display and the user are attached to a platform that moves about the workspace). For handheld displays, the viewplane object is attached to the Hand or Other Hand object, and so on. The accuracy of the stereo rendering for the fishtank configuration or the registration of the handheld display to the physical world is, of course, dependent on the accuracy of position values given for the viewplane and tracking-reference frame.

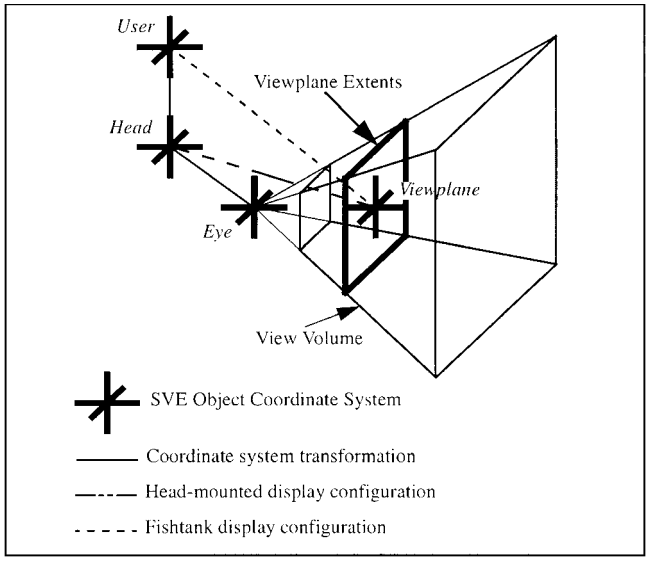

Figure 4. *Object specification of view volume.*

3.3 Calculating Model and Viewing Transformations

Given a view that includes components representing the eyepoint and viewplane, the SVE system generates a model transformation that generates 3-D points in eye coordinates from points in world coordinates and a viewing transformation that generates 3-D points by the graphics pipeline to clip against a $2 \times 2 \times 2$ viewing volume. The model transformation is constructed differently from systems using a camera model, while the viewing transformation that is constructed is a standard transformation for off-axis window viewing. The three coordinate systems (world, eye, and uniform) are depicted in Figure 5.

The model transformation is determined by constructing a transformation from eye coordinates to world coordinates, and then inverting it. The eye-coordinate system is described in the world-coordinate system by an eyepoint (the world position of the eye object) and an orientation coincident with the orientation of the viewplane (given by the world orientation of the viewplane object). Since the eyepoint is assumed to be at the origin

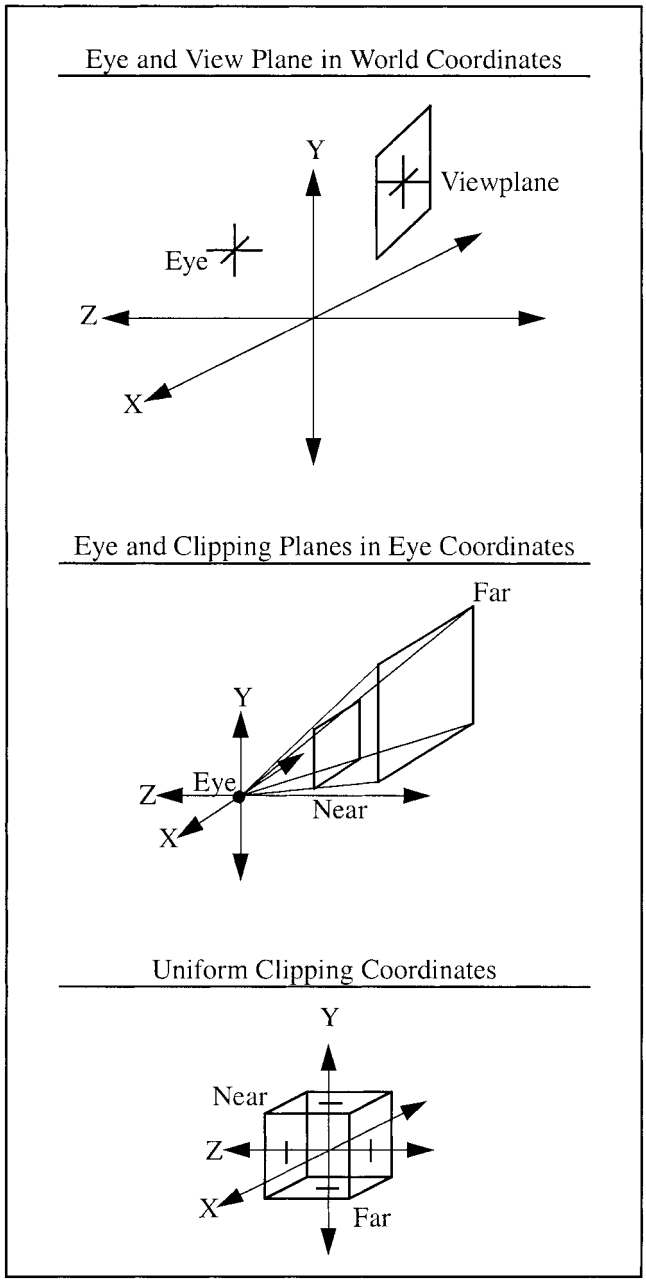

Figure 5. *World, eye, and clipping coordinate systems.*

of the eye-coordinate system, and the direction of the eye gaze is determined by the viewplane (the eye looks in the direction of the viewplane), the orientation of the eye-coordinate system is not meaningful for generating the model transformation. Any orientation difference between the eye- and world-coordinate systems will, however, affect the position of geometry associated with the eye object and its children. If the viewplane object is in a subtree of the eye object, as might be the case for an eye-tracked display system, the calculated world position of the viewplane and its window corners will automatically take into account the orientation of the eye-coordinate system. The off-axis projection transformation used for the viewing transformation is based on coordinates for the corners of the display that are on a plane perpendicular to the *z* axis. Because the window corners are given in the *x*-*y* plane of the viewplane-coordinate system, the model transformation includes the viewplane-to-world orientation transformation.

The model transformation must be a rigid-body transformation, which preserves distances and angles. This is because the underlying graphics system (Irix GL or OpenGL) performs lighting calculations before applying the viewing transformation (called the ''Projection Matrix''), which are based on distance and angle relationships to geometry. Light positions, like vertex positions, are given in world coordinates. The world-to-eye transformation, therefore, should not change angles or distances, if the effect of lights on the geometry is to be correct. For this reason, only the position of the eyepoint and the orientation of the viewplane are used in the model transformation.

The model transformation can be calculated by constructing a rotation matrix from the normalized column vectors of the viewplane object-to-world matrix transformation and premultiplying that with a matrix that translates by values for the location of the eye-coordinate system origin in world coordinates. The resulting transformation matrix, which transforms points in eye coordinates into corresponding points in world coordinates, is inverted. Equation (1) shows the column vector of the viewplane object-to-world matrix, $T_{W \leftarrow VP}$, for column i, which is normalized in Equation (2). The eyepoint, EP, is obtained from the last column of the eye object-to-world matrix, $T_{W \leftarrow E}$ (Equation (3)). The world-to-eye transformation, $T_{\text{Eye} \leftarrow W}$, is the inverse of the eyepoint translation and the viewplane rotation, which is the transpose of the rotation multiplied by the translation in the negative direction of the eyepoint. The SVE system actually generates the model transformation from the result of these operations, as shown in Equation (5).

$$\vec{\mathrm{V}}_i = [V_{1,i},\, V_{2,i},\, V_{3,i}] \tag{1}$$

$$\vec{\mathrm{V}}'_i = \vec{\mathrm{V}}_i / \|\vec{\mathrm{V}}_i\| \tag{2}$$

$$EP = (E_{1,4},\, E_{2,4},\, E_{3,4}) \tag{3}$$

$$T_{Eye\leftarrow W} = \left(\begin{bmatrix} 1 & 0 & 0 & EP_1 \\ 0 & 1 & 0 & EP_2 \\ 0 & 0 & 1 & EP_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{\mathrm{V}}'_{1\,1} & \vec{\mathrm{V}}'_{2\,1} & \vec{\mathrm{V}}'_{3\,1} & 0 \\ \vec{\mathrm{V}}'_{1\,2} & \vec{\mathrm{V}}'_{2\,2} & \vec{\mathrm{V}}'_{3\,2} & 0 \\ \vec{\mathrm{V}}'_{1\,3} & \vec{\mathrm{V}}'_{2\,3} & \vec{\mathrm{V}}'_{3\,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^{-1} \tag{4}$$

$$\begin{aligned} T_{Eye\leftarrow W} &= \begin{bmatrix} \vec{\mathrm{V}}'_{1\,1} & \vec{\mathrm{V}}'_{1\,2} & \vec{\mathrm{V}}'_{1\,3} & 0 \\ \vec{\mathrm{V}}'_{2\,1} & \vec{\mathrm{V}}'_{2\,2} & \vec{\mathrm{V}}'_{2\,3} & 0 \\ \vec{\mathrm{V}}'_{3\,1} & \vec{\mathrm{V}}'_{3\,2} & \vec{\mathrm{V}}'_{3\,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -EP_1 \\ 0 & 1 & 0 & -EP_2 \\ 0 & 0 & 1 & -EP_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \vec{\mathrm{V}}'_{1\,1} & \vec{\mathrm{V}}'_{1\,2} & \vec{\mathrm{V}}'_{1\,3} & -(\vec{\mathrm{V}}'_1 \cdot EP) \\ \vec{\mathrm{V}}'_{2\,1} & \vec{\mathrm{V}}'_{2\,2} & \vec{\mathrm{V}}'_{2\,3} & -(\vec{\mathrm{V}}'_2 \cdot EP) \\ \vec{\mathrm{V}}'_{3\,1} & \vec{\mathrm{V}}'_{3\,2} & \vec{\mathrm{V}}'_{3\,3} & -(\vec{\mathrm{V}}'_3 \cdot EP) \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{5}$$

The viewing transformation is a standard transformation that maps points in the viewing volume in eye coordinates to a uniform volume, $-1 < x < 1$, $-1 < y < 1$, and $-1 < z < 1$. This mapping allows points outside of the viewing volume to be easily clipped from the rendering process, and for easy depth determination (in which the distance from $z = -1$ is the depth). The viewing transformation is constructed using the near and far clipping-plane distances, and the points through which the corners of the viewing-volume pyramid pass through the near clipping plane. These corner points, given by the diagonally opposite points (L (left), B (bottom), $-N$ (near)) and (R (right), T (top), $-N$ (far)), are determined from the intersections at $z = -N$ of lines from the origin (the eye) and the world positions of the corners of the viewplane window, transformed through the model transformation.

Although the scale difference between the eye- and world-coordinate systems does not affect the model transformation (unless the viewplane object is in a subtree of the eye object), the near and far clipping-plane distances are affected by an eye scale difference. The scale at which the world is viewed roughly describes the amount of detail the viewer is interested in. At a small scale, the viewer is interested in high detail close by. At a large scale, the viewer is interested in low detail over large distances. The near clipping plane defines what can be seen close up, and the distance between the near and far clipping planes defines the amount of detail that can be seen (by defining the number of discrete depth levels). The scale of the eye object describes the scale at which the world is being viewed. For example, if the eye is attached to a model of an airplane cockpit, the world is being viewed from the current scale of the airplane. Changes in the airplane's scale should not affect whether the cockpit controls are clipped out or rendered at the correct depth. For this to be the case at any scale, the near and far clipping planes must be given in the eye-coordinate system as distances towards the viewplane in a direction normal to the view plane.

The near and far clipping-plane distances are calculated by determining the unit vector normal to the viewplane in world coordinates, multiplying that by the world-to-eye coordinate transform, $T_{E\leftarrow W}$, and dividing the near and far clipping distances by the length of the resulting vector. As shown in Equation (6), the first two steps reduce to multiplying the normalized third column of $T_{W\leftarrow VP}$ by the inverse of the eye-to-world transformation (only the top left 3×3 submatrix needs to be inverted). The near and far distances, N and F, are then calculated as shown in equation (7).

$$\begin{aligned} \overrightarrow{Z_{eye}} &= M_{E\leftarrow W} \cdot \begin{bmatrix} \vec{\mathrm{V}}'_{1\,1} & \vec{\mathrm{V}}'_{2\,1} & \vec{\mathrm{V}}'_{3\,1} & 0 \\ \vec{\mathrm{V}}'_{1\,2} & \vec{\mathrm{V}}'_{2\,2} & \vec{\mathrm{V}}'_{3\,2} & 0 \\ \vec{\mathrm{V}}'_{1\,3} & \vec{\mathrm{V}}'_{2\,3} & \vec{\mathrm{V}}'_{3\,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \\ &= (M_{W\leftarrow E})^{-1} \cdot \begin{bmatrix} \vec{\mathrm{V}}'_{3\,1} \\ \vec{\mathrm{V}}'_{3\,2} \\ \vec{\mathrm{V}}'_{3\,3} \\ 0 \end{bmatrix} \end{aligned} \tag{6}$$

$$N = (\text{near}) / \|\overrightarrow{Z_{eye}}\|, \quad F = (\text{far}) / \|\overrightarrow{Z_{eye}}\| \tag{7}$$

The viewing matrix transformation is constructed by premultiplying a shear matrix, which moves the center of the viewplane window to the $-z$ axis, by a scale matrix, which moves the sides of the view-volume pyramid to coincide with the $x = z$, $x = -z$, $y = z$, and $y = -z$ planes, and finally premultiplying the result by a projection matrix. These matrices are given in Equation (8).

$$T_{Clip \leftarrow Eye} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{F+N}{F-N} & -\frac{2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix}}_{\text{Project}} \cdot \underbrace{\begin{bmatrix} \frac{2N}{R-L} & 0 & 0 & 0 \\ 0 & \frac{2N}{T-B} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Scale}} \underbrace{\begin{bmatrix} 1 & 0 & \frac{R+L}{2N} & 0 \\ 0 & 1 & \frac{T+B}{2N} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Shear}} \quad (8)$$

$$= \begin{bmatrix} \frac{2N}{R-L} & 0 & \frac{R+L}{R-L} & 0 \\ 0 & \frac{2N}{T-B} & \frac{T+B}{T-B} & 0 \\ 0 & 0 & -\frac{F+N}{F-N} & -\frac{2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

This is the transformation generated by the ''window'' command in SGI's Graphics Library (transposed, as matrices are postmultiplied in GL) (McLendon, 1991). This transformation is different from the one presented by Deering (1992) only in that, here, the eyepoint is at the origin rather than the viewplane being on the *x*-*y* plane, as it is in Deering's work. A 3-D point in eye coordinates is transformed by this viewing matrix transformation, and each component of the result is divided by the fourth, *w*, component to obtain the corresponding location in clipping coordinates:

$$P_{Clip} = T_{Clip \leftarrow Eye} \cdot T_{Eye \leftarrow W} \cdot T_{W \leftarrow O} \cdot P_O \,. \quad (9)$$

The model and viewing transformations are used in conjunction with the object's world matrix transformation, as shown in Equation (9), to obtain a 3-D point in clipping coordinates given a point in the object's local coordinates. Note that the model and viewing transformations are generated each frame, while the object-to-world transformation is generated, cached, and regenerated only when one of the transformations between the object and the root of the object tree changes.

3.4 Incorporating Tracker Information

The SVE environment model provides a framework for defining an environment that includes geometric objects and a user representation independent of the hardware configuration. However, one defining characteristic of VE applications is the ability to associate tracking devices or other positioning interaction techniques, to the user's head or hand or to a particular geometric object in the environment. A common method to introducing tracking information into an environment model is to ''attach'' the information to a geometric object. With this method, the position and orientation of the ''attached'' geometric object is set to the tracker's position and orientation. As a result, the coordinate system of the object's parent corresponds to the tracking device's reference coordinate system. For example, the default user object tree would incorporate head-tracking information by positioning the User object at the location of the tracking device reference frame (the transmitter, in the case of electromagnetic trackers), and ''attach'' the Head object to the tracker. Unfortunately, this method has a few potential problems:

- The Head and Cursor objects, as children of the User object, must share the same reference frame. This will not be possible if they are controlled by different tracking devices that have separate reference frames.
- The Head object's position would be overwritten every frame when using head tracking. This operation prevents orientation corrections for the tracker mounting on the physical HMD through setting the Head object's position to make that correction.
- If the application wishes to have an object that is not

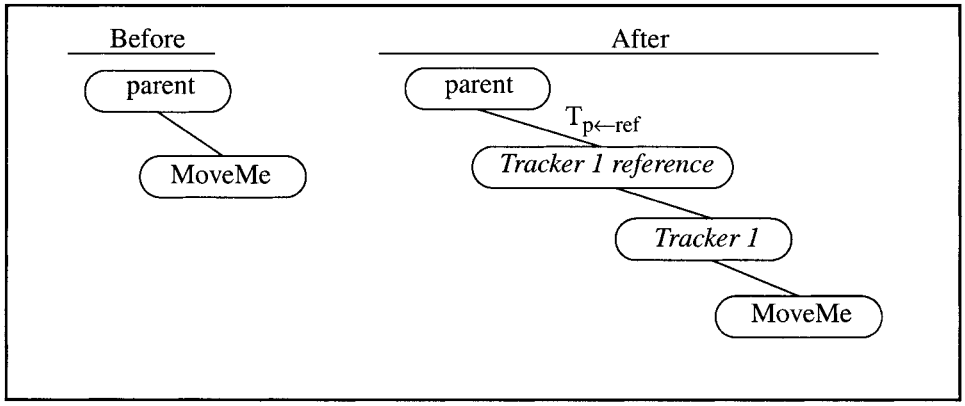

Figure 6. *Inserting tracker 1 to control object "MoveMe."*

a child of the User object follow a tracking device, the application would need to include a reference frame object as that object's parent and position it to the appropriate location in the world.

Our solution to these problems is to incorporate tracking information to control an object in the model by inserting two new PHOs into the object tree at run-time. (See Figure 6.) One object, called Tracker X Reference (where *X* is a unique ID), is created as a child of the controlled object's parent. This object represents the tracking device reference frame. The other new object, called Tracker X, is created as a child of Tracker X Reference, and represents the tracking information from the tracking device. The object to be controlled is then linked to the Tracker X object as a child. Given this method, tracking devices can be introduced into the application at run-time to cause *any* object in the model to be tracked.

The locations of the reference frame of the device can be initially given in relation to a coordinate system that represents the user's workspace (generally the Workspace object). This method allows tracking devices of different reference frames to be correctly used in a common workspace reference frame. As shown in Equation (10), the transformation from the tracker reference object to its parent, $T_{p \leftarrow ref}$, can be calculated from the world position of the parent object, $T_{W \leftarrow p}$, the world position of a workspace object (which could be the parent object), $T_{W \leftarrow ws}$, and the given initial transformation from the reference coordinate system to the workspace-coordinate system, $T_{ws \leftarrow ref}$:

$$T_{p \leftarrow \text{ref}} = (T_{W \leftarrow p})^{-1} \cdot T_{W \leftarrow ws} \cdot T_{ws \leftarrow \text{ref}} \qquad (10)$$

4 Framework for Extension

Much of the success of the SVE library as a development tool for VE applications can be attributed to the library's framework for providing basic services and for extending those services in a modular and independent manner. The mechanisms for extending the library include allowing for additional behaviors at appropriate phases of the execution, allowing additional named data values to be associated with components of the environment model, and providing a structured framework for modules that provide interaction with input devices and that provide for particular user-interaction techniques. The SVE extension framework is an open architecture that gives structure to extensions to make them modular, easy to develop, and reusable. The set of extension opportunities provided by the framework is more complete than that of other VE development systems. The extensibility of the SVE library allows it to support a wide variety of VE applications in the face of new environment behaviors, hardware devices, and rendering and interface techniques, which is critical in a field that is still in the experimentation stage.

The mechanism used to allow for extensions is

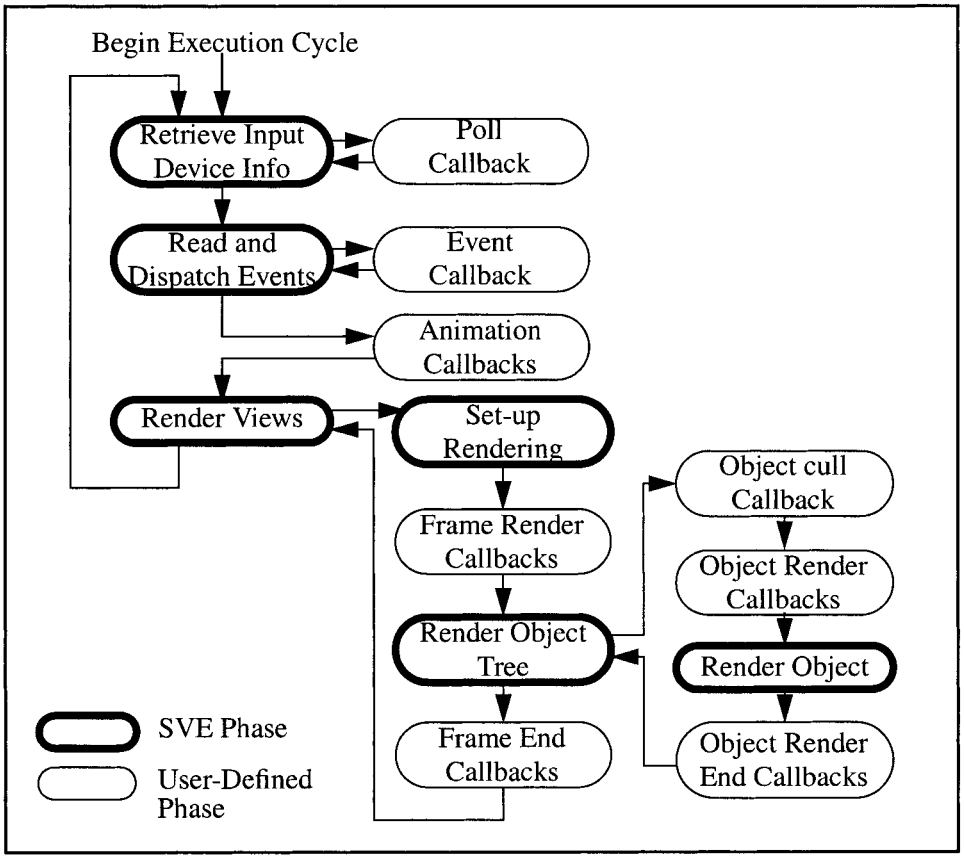

Figure 7. *The SVE execution cycle.*

through application-defined callback routines that are provided to the SVE runtime system to be invoked at certain phases of execution. Callback routines are commonly used for application-specific extensions to a system. For example, event-driven 2-D interface managers allow applications to respond to user interactions with the interface (Myers, 1989) by defining callback routines. In the next section, we describe how callback routines are used by VE applications using the SVE library. The library also allows the data associated with the environment model to be extended through named properties assigned to geometric objects. This mechanism is described in Section 4.2. Finally, the SVE library provides further structure in its framework to support more-complex extensions (new polling input devices and interaction techniques). This part of the framework is described in Section 4.3.

4.1 Callback Routines

The philosophy behind the SVE library design is to provide reasonable default behavior in all aspects of the VE application and to provide the capability of overriding or augmenting the defaults. For example, the default reaction to pressing the middle mouse button, ''flying'' in the facing direction, can be changed to ''walking'' along a groundplane along the facing direction. The default mechanism used to cull from the rendering pipeline geometric objects that are entirely out of the viewing volume can be augmented to also cull objects that are too small to be seen.

4.1.1 The Execution Cycle. The phases of the SVE execution cycle are given in Figure 7. The application overrides or augments the default behavior for a

particular phase by defining and registering a callback routine. When the appropriate phase in the execution cycle is reached, the application-defined callback will be summoned, and parameters will be given to provide the necessary context for the routine's task. As the figure shows, the application can define routines that respond to particular events (such as mouse movement, button and key presses, and geometric object selection), routines that provide animated behaviors, that perform a task just before (or just after) a frame or a particular geometric object is rendered to the display, or that determine if a geometric object should be culled from the rendering pipeline altogether (and if its children in the scene graph should be culled, as well). In addition, the application can provide a callback routine that will be called when a geometry file given to describe a geometric object is not understood by the SVE library, allowing the application an opportunity to translate it into the appropriate geometric primitives.

An application can register any number of callback routines for a particular execution phase or event. When the phase is reached, the routines are called in the reverse order in which they were given (LIFO, or stack, order). During the ''read and dispatch events'' phase, a limited number of events are removed from the event queue, and each is dispatched by calling the routines associated with the particular event type, in LIFO order, until one indicates that it has ''consumed'' the event. For example, a routine may respond to a button press only when the user's hand is inside a particular object, consuming that event. Otherwise, the button-press event is passed on to other routines that may respond to it.

This method of ordering callbacks in LIFO order enables incremental and modular development. New behaviors can be added without removing other, working behaviors. If routines are designed to be mostly independent (which is facilitated by limiting them to observing and modifying the environment model), then they can be developed independently and brought together after they are fully developed and tested. In addition, a set of routines that cooperate to perform a particular task can be reused in different applications.

4.1.2 Examples. For example, different interaction methods for navigating through an environment were implemented using SVE callback routines and compared to each other (Bowman, Koller, & Hodges, 1997). Similarly, new object-manipulation techniques have been implemented and evaluated using the SVE library, including a novel reaching technique (Bowman & Hodges, 1997a) and an object-centric viewing technique (Koller, Mine, & Hudson, 1996).

Through the use of a set of SVE callback routines, the VAnno toolset (Harmon et al., 1996) allows a user to place several different types of audio annotations within a 3-D environment. The toolset also offers a user-interface component with which the user of the VE application can record and place annotations, play or edit existing annotations, and specify system properties. This toolset can be useful for applications in which several users are collaborating on a design or analysis, and has been used in a system called the Virtual Data Visualizer (van Teylingen, Ribarsky & van der Mast, 1995) and in an architectural-walkthrough application.

A stylus-interaction API developed for SVE applications (Bowman & Hodges, 1997b) introduces a set of callback routines to handle the interaction of a tracked stylus held by the user with certain geometric objects in the environment. To use the stylus library, the programmer simply associates callbacks with these environment objects using calls to the stylus API. The application then receives events when the stylus enters or exits the objects, or when the stylus button goes up or down within the object.

4.1.3 Using Callbacks. The stylus-interaction technique provided by the API can be used to implement simple buttons, menus, or image maps with draggable icons. It is powerful enough to allow the replication of any 2-D, mouse-based interface in a virtual environment, as well as many novel 3-D interfaces. It has been used in a Virtual Venue application (Bolter et al., 1995; Bowman, Hodges, & Bolter, 1998), which provides an ''information-rich'' virtual environment (the Aquatics Center at Georgia Tech) to allow users to explore and learn more about the environment through selecting objects of interest, floating icons, or spatial hy-

Figure 8. *The virtual airplane application.*

perlinks on a handheld tablet. It has also been used in an architectural design environment (Bowman & Hodges, 1997b). The usefulness of encapsulating callback routines that implement particular interaction techniques-(like the stylus interaction) for inclusion in many applications prompted us to design a framework specifically for interaction components and techniques. This framework is described in Section 4.3.

Another common use of the callback framework is to represent different phases in an application's execution with different sets of callback routines, and switching between sets when changing phases. For example, an airplane simulation developed to treat people who were afraid of flying (through graded exposure to flying experiences) was designed as a finite-state machine (FSM), in which the nodes of the FSM represented states such as taxiing, taking off, cruising, circling the airport, and landing. (Figure 8 shows the patient's view of the cabin during taxiing.) The behavior of the airplane in the environment (and associated sound effects) was implemented in a set of animation callbacks, one for each phase. When the application transitioned from one phase to another, the current animation callback was removed, and the appropriate animation callback enacting the behavior of the new phase was registered. This design and implementation strategy allowed independent development of each phase of the application and some flexibility to change the FSM by adding or removing phases of the application.

Apart from simply invoking an application-defined routine at the right point in the execution of the simulation-rendering cycle, the value of defining callback routines is that they are provided the context needed for their tasks. All callbacks are given a reference to a global state structure which includes the scene graph of the different views that the callback will likely alter or examine. The state structure also includes the time that the current cycle and previous cycle began, which can be used by the animation callbacks to synchronize their behaviors to a common clock. The state structure given to an event callback includes the event that occurred and its associated data. The rendering callbacks are called after the rendering context has been initialized with the viewing parameters of the view being rendered. This allows applications to call SVE routines to render particular geometric objects or to call routines of the underlying graphics library (GL or OpenGL), where graphics primitives will be drawn to appear in the coordinate system of the environment, or of the particular geometric object being rendered.

Providing a rendering context to rendering callbacks allows VE applications using the SVE library to easily extend beyond the library's rendering capability. This feature was used in the VGIS (Virtual Geographic Information System) application, which provides a 3-D immersive or workstation-window view of terrain with embedded information that can be queried and displayed (Koller et al., 1995). The system was developed for battlefield management, and therefore includes symbolic representations of units and unit groups (Figure 9). Although the SVE library was used to provide for user modeling and interaction as well as rendering of simple geometries, the terrain geometry had to be handled separately to be rendered in real time. A custom algorithm was developed (Lindstrom et al., 1996) to render the terrain. The rendering operations and the rendering used for an overlaid interface component (in the center and corners of Figure 9) were implemented as rendering callbacks. Another application used the SVE library to render data-flow geometry generated by an AVS module as a geometric object in the environment which could be

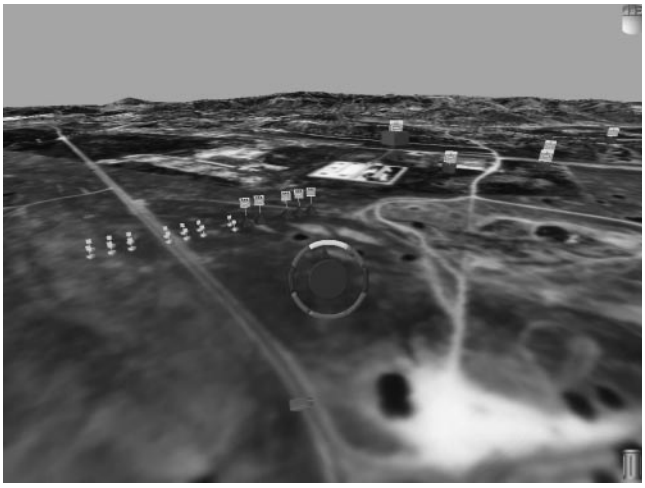

Figure 9. *The virtual geographic information system application.*

viewed and manipulated like other objects (Williams et al., 1995). This application demonstrates the value of having rendering callback routines for individual objects.

4.1.4 Incorporating Remote Processes. The SVE execution cycle is performed by a single process model (although distributable components to perform input gathering and audio output tasks are provided and can be run in parallel). It has been argued (Shaw, 1993; Pausch, 1995) that separate processes should be used for the rendering process and the simulation process that continually updates the environment model, so that the frame rate of the display rendering is decoupled from the simulation update rate. However, such a distributed design introduces subtle complexities. For example, if a rendering process receives head-tracking information directly from the tracking process (as in the DIVER system (Gossweiler et al., 1993)), then a separate simulation process will not be able to prevent the user's head from going through walls, even if it detects a collision between the head position and the wall. Rather than impose a solution to these issues, the SVE library utilizes one process for simulation and rendering. As a result, an application routine must be designed to complete quickly for each execution, as it will not be preempted. The application's simulation process, however, could simply exchange messages with a remote simulation process that performs the actual computation. The SVE library has been used in conjunction with RAVEL (Kessler, Hodges, & Ahamad, 1998), a system that supports distributed, networked components of a VE application. The SVE library was used in the development of components that render the user's view and components that examine and manipulate the environment's geometric model.

4.2 Properties

Different VE applications will often want to associate a different set of attributes to the geometric objects that populate the environment. For example, a CAD application may wish to designate certain objects as ''parts'' that have properties such as ''material,'' ''manufacturer,'' and ''cost.'' An application could maintain a list of parts and their properties itself, but many VE systems provide a ''user pointer'' associated with every object in the environment model, which can be used to refer to the application's object-specific information. (For example, Java3D (Sowizral et al., 1998) provides this feature.)

Using a user pointer, however, is not an option for a module that is designed to extend a set of applications, because some of those applications may depend on using the user pointer themselves. For example, a module may be developed to enforce the laws of physics for a set of objects in the environment of any application it is included with. That module may wish to associate with objects properties such as ''mass,'' ''velocity,'' and ''acceleration.'' In addition, some of these properties may be defined by the application (such as ''mass''), and others may be generated by the module (such as ''velocity'').

The SVE library provides a mechanism to associate a named data item (or array of items) with any object in the environment. Items can be characters, integers, floating-point numbers, geometric object references, or generic pointers. The item type of a property can be queried. Array properties can store such information as character strings and position matrices. An interest callback routine can be associated with the property and will be called whenever the property changes. This mechanism is similar to the object / property / event architec-

ture of the WorldToolKit, release 8 (Sense8, 1998), where property changes are identified as events. The SVE library provides an additional capability which allows for generate and update callbacks to be associated with properties. A generate callback is called whenever the value of the property is requested, and an update callback is called whenever the property is set to a new value (essentially an interest callback that is guaranteed to be called first). These callbacks allow for external storage of information and supports lazy updates, where information is not recalculated as a result of a change until it is requested. This capability provides for more-efficient data storage and access, while providing a uniform method to access object information.

The property mechanism is implemented to allow efficient access and update to properties, while also utilizing memory resources efficiently. The property values are stored in memory locations referred to by addresses in a dynamic array that is associated with each geometric object. The index of a particular property name is fixed when the first value is stored using that property name. (A property name's index value can be used instead of the property name in all future accesses to object properties of that name.) The dynamic array begins at the largest multiple of ten that is smaller than the minimum index of an object's property list, and ends at the smallest multiple of ten that is larger than the maximum index of the object's properties. Therefore, the property value array stored with each object will grow as properties are associated with it. When the object does not have a value for a property index that has been assigned, this bounded array may contain elements that are not used. In practice, however, it is expected that most objects will contain almost the same set of properties in any one application, and the number of array elements that are empty will be small.

4.3 Modular Frameworks

As we have shown, applications developed with the SVE framework can provide one or more callbacks and a set of properties to create the behavior necessary for the application's appearance and tasks. In addition, the framework supports the development of a module containing callbacks and defining properties to perform a particular task that can be reused in many applications. In our experience developing VE applications, we have identified two types of modules that occur frequently: interaction techniques and polling device interfaces. Our framework provides extra support for defining and using these types of modules.

4.3.1 Interactors. A graphical 2-D application generally contains many components for user interaction, such as buttons, menus, and scroll bars. Development tools for graphical interfaces define a set of interactor classes from which actual components can be easily instantiated to be used in the application. The SVE library supports the development of 3-D interactors through an extension called SVIFT, the Simple Virtual Interactor Framework and Toolkit (Kessler, 1999). This framework allows for interactor types, or classes, to be defined as a set of routines that handle instance creation, event handling, resizing, extent queries, property changes, and deletion. The SVE library supports the instantiation of interactors through application code or through file descriptions. Through this mechanism, a set of routines that define the behavior of a user interaction can be defined, grouped, and then reused in multiple applications.

Although the SVIFT extension to SVE allows for graphical interactors like floating buttons, labels, menus, and containers to be defined, the framework supports more-general interaction techniques. The framework defines interactors as responding to events, such as button presses or geometric object selection, and environment model changes, such as the movement of an object of the user model, by producing additional events or making a change to the environment model. The framework, therefore, supports interactors that act like tools, such as a ''ray gun'' that can select objects, and interactors that have no appearance, such as a position-constraint maintainer that does not allow a particular object to move outside a plane, line segment, or volume.

Even though they may not have a geometry, all interactors defined with this framework are associated with at least one geometric object. Therefore, they can be included in the environment model, have a position in the

environment (perhaps as a PHO), and store properties. In addition, an interactor can be designed to work with a set of any geometric objects (which can include other interactors). For example, this feature allows the menu to contain buttons, submenu buttons, and noninteractive separators with no extra work. Interactors provide additional, responsive behavior to the environment, but still include their state as part of the environment model, which may be used by other interactors or application routines.

4.3.2 Polling Devices. The SVE system, by default, handles the task of periodically polling devices, such as trackers and mice, for their current state and incorporating that information in the environment model or with associated events. If a new type of device is used, however, new routines will need to be written to open the device, poll for the current state, and close the device. The SVE library provides a framework for defining this set of routines for a particular device and allowing particular devices of that type to be instantiated. The SVE runtime system will automatically open, poll, and close instantiated devices through the provided routines at the appropriate points of the execution cycle. Once the routines for a particular device have been defined, they can be easily reused by any other application wishing to use the device.

5 Conclusion

The SVE library was designed to support the experimental development of VE applications, display mechanisms, and interaction techniques. Many of the characteristics of the library that make it a good tool for developing experimental VE applications also make it a good tool for general VE application development and can be incorporated in other VE development tools. In particular, using an environment model as a source of input and a place for output provide a common interface layer for VE tasks, and also provides the freedom to mix and match different input and output devices to define an application's configuration. One key to this mechanism is including a model of the user in the environment which can be manipulated by user input devices and used to drive output devices. Another key is allowing for arbitrary, nongraphical properties to be stored with the model.

The other aspect of the SVE library that makes it an effective tool is its extensibility from a functional default behavior. The library provides a runtime system that allows, with almost no programming, for a simple, fly-through VE application. The library also provides a framework that allows for the independent development of different mechanisms to render parts of the scene, to respond to events and changes in the environment model, to provide dynamic behaviors for parts of the environment model, and to interface with new I/O devices. Through this framework, new mechanisms can be designed to be reusable in many applications, even if they involve a group of routines implementing complex interaction techniques.

The SVE library has been used to produce numerous successful VE applications and VE techniques. As a research project itself, it has changed from its original form to meet the needs of the applications and application developers. The library has undergone seven major revisions, and is currently on version 2.1. New developments, such as the interactor framework (Kessler, 1999), are being developed through the extension mechanism of the library, rather than adding to the SVE library itself.

Appendix A

The SVE library maintains a model of the environment as a collection of individual geometric objects and placeholder objects (PHOs), which have no geometry, arranged in a rooted tree structure. The relationship between object nodes in the tree is one of attachment, in which a child object has a location, orientation, and scale in relationship to its parent object. Therefore, when a parent moves, rotates, or changes scale, the position, orientation, and scale of its children are affected accordingly. This structure is known as a *coordinate-system graph*, which is a certain type of scene graph. The object nodes of the tree define a local coordinate system for its

geometry through a spatial transformation from its parent coordinate system (stored as a 4×4 matrix transformation for homogenous coordinates, which also allows for other relationships such as shears along axes). The SVE geometric object tree is not a general scene graph because changes to the graphics state by an object's description (such as a coordinate-system transformation or color change) do not propagate to parent or sibling objects in the tree.

The transformations given in the coordinate-system graph allow points of the geometry of all objects to be transformed to the associated point in a single coordinate system, such as the ''world'' coordinate system (generally the root of the tree), or the coordinate system of the user's eye (represented as a PHO in the tree). Therefore, the appropriate relationships between objects can be seen by the user and detected by the program. Using the notation in Foley et al. (1996), the matrix transformation $T_{B \leftarrow A}$ transforms a point in the child's (A) coordinate system to the corresponding point in the parent's (B) coordinate system. Thus, if points are represented as column vectors, we can write Equation (11):

$$P_B = T_{B \leftarrow A} \cdot P_A. \tag{11}$$

Two transformations can be composed into a single transformation, as shown in Equation (12):

$$T_{C \leftarrow A} = T_{C \leftarrow B} \cdot T_{B \leftarrow A}. \tag{12}$$

The combined transformations from the coordinate system of object O to the world-coordinate system of the tree root, therefore, is represented as $T_{W \leftarrow O}$. Transformations ''down'' the object tree, where points given in a parent's coordinate system are transformed to a child's coordinate system, are represented as the inverse of the child-to-parent transformation, or by reversing the subscripts, as shown in Equation (13):

$$(T_{B \leftarrow A})^{-1} = T_{A \leftarrow B}. \tag{13}$$

The relationship between the coordinate systems of any two nodes can be obtained by combining the coordinate-system relationships between nodes along a path up from one node to the root, and down along the path to the other node, as shown in Equation (14):

$$\begin{aligned} T_{D \leftarrow C} &= (T_{W \leftarrow D})^{-1} \cdot T_{W \leftarrow C} \\ T_{D \leftarrow C} &= T_{D \leftarrow W} \cdot T_{W \leftarrow C}. \end{aligned} \tag{14}$$

(See Robinett and Halloway (1992) for further discussion of using coordinate-system graphs for the geometric object representation in VE applications.)

The tree of geometric objects as a whole describes the ''world'' and can be defined using a world-description file that the application loads after initialization, or by adding geometric objects to the world tree one at a time in the application, or by a combination of these methods. Geometric objects have many properties, including a unique name, a geometric description, parent and child links in the tree, a local coordinate system, boundaries that surround the object's geometry, boundaries that surround the geometries of the object and its children's geometries, and flags that indicate if an object is currently visible, highlighted, or selectable. A PHO can serve as a coordinate system for its child objects. Objects with geometries may be defined by an object file, or may be created piece by piece in the application. The object file can use the Wavefront format (OBJ) or an SVE object format, which allows for objects with special properties such as text, texture faces with changing images, visibility limits, or boundaries that do not conform to the object's geometry.

Acknowledgments

The SVE library owes its existence to Don Allison, Doug Bowman, Elizabeth Bright, Eric Brittain, Jim Durbin, Kevin Hamilton, Drew Kessler, David Koller, Rob Kooper, E. J. Lee, Peter Lindstrom, Tom Meyer, Greg Newton, Jouke Verlinden, Zach Wartell, and Ben Watson.

We would also like to thank the reviewers for their helpful suggestions and comments.

References

Blanchard, C., Burgess, S., Harvill, Y., Lanier, J., Lasko, A., Obermann, M., & Teitel, M. (1990). Reality built for two:

A virtual reality tool. *ACM SIGGRAPH Special Issue on the 1990 Symposium on Interactive 3D Graphics,* 35–36.

Bolter, J., Hodges, L. F., Meyer, T. C., & Nichols, A. (1995). Integrating perceptual and symbolic information in VR. *IEEE Computer Graphics and Applications, 15* (4), 8–11.

Bowman, D., Hodges, L. F., & Bolter, J. (1998). The virtual venue: User-computer interaction in information-rich virtual environments. *Presence: Teleoperators and Virtual Environments, 7* (5), 478–493.

Bowman, D., & Hodges, L. F. (1997a). An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. *Proceedings of the ACM Symposium on Interactive 3D Graphics,* 35–38.

——— (1997b). Toolsets for the development of highly interactive and information-rich virtual environments. *International Journal of Virtual Reality, 3*(2), 12–20.

Bowman, D., Koller, D., & Hodges, L. F. (1997). Travel in immersive virtual environments: An evaluation of viewpoint motion control techniques. *Proceedings of the IEEE Virtual Reality Annual International Symposium* (*VRAIS*) '97, 45–52.

Bricken, W., & Coco, G. (1994). The VEOS project. *Presence: Teleoperators and Virtual Environments, 1*(2), 111–129.

Brooks, F. P., Jr. (1986). Walkthrough—A dynamic graphics system for simulating virtual buildings. *Proceedings of the 1986 Workshop on Interactive 3D Graphics,* 9–21.

Carlsson, C., & Hagsand, O. (1993). DIVE—A platform for multi-user virtual environments. *Computers & Graphics, 17*(6), 663–669.

Codella, C. F., Jalili, R., Koved, L., & Lewis, J. B. (1993). A toolkit for developing multi-user, distributed virtual environments. *Proceedings of the IEEE Virtual Reality Annual International Symposium* (*VRAIS*), '93, 401–407.

Deering, M. (1992). High resolution virtual reality. *Proceedings of ACM SIGGRAPH 92,* 195–202.

Foley, J., van Dam, A., Feiner, S., & Hughes, J. (1996). *Computer Graphics: Principles and Practice* (2nd ed. in C) (pp. 222–226). Reading, MA: Addison-Wesley.

Ghee, S. (1995). dVS: A distributed VR systems infrastructure. *ACM SIGGRAPH 95 Course Notes.*

Gossweiler, R., Long, C., Koga, S., & Pausch, R. (1993). DIVER: A distributed virtual environment research platform. *IEEE Symposium on Research Frontiers in Virtual Reality,* 10–15.

Hagsand, O. (1996). Interactive multiuser VEs in the DIVE system. *IEEE MultiMedia,* 30–39.

Harmon, R., Patterson, W., Ribarsky, B., & Bolter, J. (1996). The virtual annotation system. *Proceedings of the IEEE Virtual Reality Annual International Symposium* (*VRAIS*), *96,* 239–245.

Hodges, L. F., Rothbaum, B. O., Watson, B. A., Kessler, G. A., & Opdyke, D. (1996). A virtual airplane for fear of flying therapy. *Proceedings of the IEEE Virtual Reality Annual International Symposium* (*VRAIS*) *96,* 86–93.

Kessler, G. D., Hodges, L. F., & Ahamad, M. (1998). RAVEL, a support system for the development of distributed, multiuser VE applications. *Proceedings of the IEEE Virtual Reality Annual International Symposium* (*VRAIS*) 98, 260–267.

Kessler, G. D. (1999). A framework for interactors in immersive virtual environments. *Proceedings of IEEE Virtual Reality '99,* 190–197.

Koller, D., Lindstrom, P., Ribarsky, W., Hodges, L. F., Faust, N., & Turner, G. A. (1995). Virtual GIS: A real-time 3D geographic information system. *Proceedings of Visualization '95,* 94–100.

Koller, D., Mine, M., & Hudson, S. (1996). Head-tracked orbital viewing: An interaction technique for immersive virtual environments. *Proceeding of the ACM Symposium on User Interface Software and Technology* (*UIST*) *'96,* 81–82.

Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., & Turner, G. A. (1996). Real-time, continuous level of detail rendering of height fields. *Proceedings of ACM SIGGRAPH 96,* 109–118.

McLendon, P. (1991). *Graphics Library Programming Guide* (p. c-4). Mountain View, CA: Silicon Graphics, Inc.

Myers, B. A. (1989, January). User-interface tools: Introduction and survey. *IEEE Software,* pp. 15–23.

Obeysekare, U., Williams, C., Durbin, J., Rosenblum, L., Rosenberg, R., Grinstein, F., Ramamurthi, R., Landsberg, A., & Sandberg, W. (1996). Virtual workbench—A nonimmersive virtual environment for visualizing and interacting with 3D objects for scientific visualization. *IEEE Visualization '96 Annual Conference Proceedings,* 345–349.

Pausch, R., Burnette, T., Capehart, A. C., Conway, M., Cosgrove, D., DeLine, R., Durbin, J., Gossweiler, R., Koga, S., & White, J. (1995, May). Alice: Rapid prototyping for virtual reality. *IEEE Computer Graphics & Applications,* pp. 8–11.

Rekimoto, J. (1997). NaviCam: A magnifying glass approach to augmented reality. *Presence: Teleoperators and Virtual Environments, 6*(4), 399–412.

Robinett, W., & Holloway, R. (1992). Implementation of flying, scaling, and grabbing in virtual worlds. *Proceedings of the 1992 Symposium on Interactive 3D Graphics,* 189–192.

——— (1995). The visual display transformation for virtual reality. *Presence: Teleoperators and Virtual Environments, 4*(1), 1–23.

Rohlf, J., & Helman, J. (1994). IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Proceedings of SIGGRAPH 94,* Orlando, FL, 381–394.

Sense8 Corporation. (1998). *WorldToolKit™ Release 8 Technical Overview.* Mill Valley, CA.

Shaw, C., Green, M., Liang, J., & Sun, Y. (1993). Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems, 11* (3), 287–317.

Snowdon, D. N., & West, A. J. (1994). AVIARY: Design issues for future large-scale virtual environments. *Presence: Teleoperators and Virtual Environments, 3*(4), 288–308.

Strauss, P. (1993). IRIS inventor, a 3D graphics toolkit. *ACM SIGPLAN Notices* (*OOPSLA '93 Conference Proceedings*), *28* (10), 192–200.

Southard, D. A. (1995). Viewing model for virtual environment displays. *Journal of Electronic Imaging, 4*(4), 413–420.

Sowizral, H. A., Nadeau, D. R., Bailey, M. J., & Deering, M. F. (1998). Introduction to programming with Java3D. *ACM SIGGRAPH 98 Course Notes.*

Tarlton, M. A., & Tarlton, P. N. (1992). A framework for dynamic visual applications. *ACM SIGGRAPH Special Issue on the 1992 Symposium on Interactive 3D Graphics,* 161–164.

Tramberend, H. (1999). Avocado: A distributed virtual reality framework. *Proceedings of IEEE VR'99,* 14–21.

van Teylingen, R., Ribarsky, W., & van der Mast, C. (1995). *The virtual data visualizer* (Technical Report GIT-GVU-95-16). Graphics, Visualization, and Usability Center.

Wang, Q., Green, M., & Shaw, C. (1995). EM—An environment manager for building networked virtual environments. *Proceedings of the IEEE Virtual Reality Annual International Symposium* (*VRAIS*) *'95,* 11–18.

Watsen, K., & Zyda, M. (1998). Bamboo—A portable system for dynamically extensible, real-time, networked, virtual environments. *Proceedings of the IEEE Virtual Reality Annual International Symposium* (*VRAIS*) *'98,* 252–259.

Williams, C., Obeysekare, U., Kessler, D., Rosenblum, L., & Hodges, L. F. (1995). *Incorporating virtual environments into scientific visualization using AVS and georgia tech's simple virtual environment library.* Panel presentation at the *AVS '95* conference.