Interactive Exploration in Virtual Environments

Belleman, R.G.

Publication date
2003

Chapter 4

Enabling technology for interaction in virtual environments*

"People shouldn't have to read a manual to open a door, even if it is only one word long (push / pull)."

Donald A. Norman.

4.1 Introduction

Virtual Environments (VEs) are used in areas where the immersion of a person in a synthetically generated environment provides more insight in a specific problem over classic methods using a desktop [247]. Often, these type of problems entail the analysis of three or more dimensional structures that are difficult to comprehend using projections on two dimensional screens. The primary objective of an immersive VE is to engulf a user in a computer generated synthetic environment. A successful immersive experience is achieved when the user experiences the sensation of "presence", of being part of the artificial environment. Interaction is a key requirement in achieving this goal.

In this chapter we begin by identifying the interaction methods that are important to do useful work in a VEs. In the sections that follow, we describe various interaction techniques that can be used to create efficient, flexible and information rich immersive exploration environments.

4.1.1 Increasing awareness through interaction

Different sensory modalities can be used to present data to the human senses. In Virtual Reality (VR) research the objective is to subject the human sensory system with impulses that leads the user to believe he is present in the synthetic world. As

*Parts of this chapter have been published in *R.G. Belleman, J.A. Kaandorp, D. Dijkman and P.M.A. Sloot.* "GEOPROVE: Geometric Probes for Virtual Environments", number 1593 in *Lecture Notes in Computer Science*, pages 817–827, 1999.

in the real world, the experience of presence is greatly enhanced when the environment responds to our interaction. Interaction in an immersive VE should *enhance* the immersive experience and most definitely not break it through interaction methods that force the user to step out of the VE and into the real world, not even for a brief moment. Unfortunately, very few readily available toolkits exist for VE application developers that facilitate the construction of interactive VEs. The ones that *do* exist use specialized hardware or physical devices that the user wears whilst in the VE, offer limited flexibility, are too specific to the application area for which they were developed or they require a substantial software engineering effort from the developer [35, 43, 127, 149, 161, 201].

Input devices, gestures, intention and mapping

Input devices are the primary interfaces between the real and virtual world that enable users to interact with a VE. Input devices offer a degree of expressiveness that is proportional to the "degrees of freedom" (DOF) they express. A simple button represents either a pressed or unpressed state and therefore does not provide sufficient expressiveness to denote, say, a two dimensional vector like a two dimensional analog joystick does. Higher degrees of expressiveness can be achieved by combining input devices.

A frequently used input device used for interaction in a VE are 6 degrees of freedom (3 translation and 3 rotation) tracking sensors. These devices create a relation of the position and orientation of physical objects (such as the user's hand) with objects in the virtual world. Common ways in which these devices are used to interact with VEs are through proximity tests that allow virtual objects to be "grabbed" or intersection tests that detect that the device is pointing at a virtual object. Input devices used in VR systems are often a combination of a tracking sensor with "manipulators" like buttons, small joysticks or flex sensors (as in gloves) that can be manipulated by the user.

VR input devices are used to monitor the user's "gestures" which in turn convey "intention" to the environment. The user's gestures are analysed by the environment to ascertain the user's intentions which are then "mapped" to interaction methods. An easy to use VE accurately maps the user's gestures to the associated interaction method so that the environment fulfills the user's intention [162]. Moreover, the interaction methods provided to the user should provide clear and intuitive clues to their "affordance"; how they should be used and what they can be used for [171]. For example, clinching the fingers of a glove in the proximity of a virtual object or clicking a button while pointing at an object could result in the object being selected.

4.1.2 Interaction methods

Some interaction methods in a VE will be predetermined by the task-specific actions that have to be performed. However, most environments have a need for interaction methods that are generic for most applications. Unfortunately, there is no unifying

framework for interaction in VEs as there are now for desktop PCs [162, 201]. Due to its immersive nature, there are, however, a number of interaction methods that we feel are mandatory to do useful work in a VE.

Motion parallax

An important interaction method allowing a user to comprehend multi dimensional structures is by allowing the user to look around the visual representation of these structures. Just as in the real world, the difference in perspective that results from the changing position of the user's eyes in relation to the object provides information on the size and relative location of substructures, thereby providing useful depth-cues. This method of interaction is often referred to as "motion parallax". In VEs, motion parallax is achieved by changing the rendered images based on the location and orientation of the head. To that end, a tracking sensor is mounted on the head, allowing the VE to obtain this information.

Object manipulation

The content of virtual environments consists of objects that are representations of the structures of interest to the user. The manipulation of these objects can form the basis of many interaction methods, much in the way we manipulate objects in the world around us. An obvious (and intuitive) way to implement interaction in immersive VEs would, therefore, be through the manipulation of virtual objects in the environment [22,24,135,163,164,167,188]. In general, the following object manipulation tasks can be identified:

- create, delete - The instantiation or removal of an object from the environment changes the context of the environment, which in turn can be a reason to trigger an action. For example, the deletion of an object could result in the deletion of an associated menu (and vice versa). We will come back to what we mean by "context" in section 4.4 (page 81).

- select - Selection is equivalent to identifying an object as being in the focus of the user's attention. If the object is a visual representation of an action, selecting an object may result in the execution of an action (much like pressing a button).

- move, rotate, scale - Objects in a VE have a position, orientation or scale which may be changed. This change itself may represent a change in context (like moving a chess piece on a board) or represent a quantitative value (like the displacement of a button over a certain distance in a slider).

- parameterization - Besides position, orientation and scale, objects often have other, more object specific attributes associated with them like shape, or color. Interaction methods to change the value of an attribute is a useful type of interaction to signify parameter changes. The type, range and accuracy of the

attribute puts constraints on the interaction methods that are most suitable to set the value of an attribute.

In section 4.2 we describe an architecture that extends an existing scientific visualization environment for use in a VE. This architecture allows for the manipulation of scientific visualization results to facilitate the construction of information rich and flexible exploration environments for scientific research.

Navigation and wayfinding

Users are constrained by the physical confinements of the VR devices they are using. For example, in the case of HMDs the user will not be able to move beyond the length of the connected cables, in projection based systems, the projection screens limit the user's movements. The virtual environment that they explore is often larger than these confinements so that interaction methods are required that allow the user to navigate through the virtual world and reach objects outside physical reach or to constrain the movements of the user in relation to objects [132, 187, 198]. While the user is navigating through a VE, it may be hard to find one's way around and to locate "what is where" in the environment, especially in large VEs [55, 73, 74, 179].

Quantitative and text input

Text and number entry are important in situations where precision is critical. For some applications an approximate value may be sufficient so that some form of user-interface widget (such as the slider or dial we know so well from desktop user interfaces) could be used. Unfortunately, none of the existing user-interface libraries that exist today can be readily incorporated into a VE. Also, in other situations where exact values are critical, interaction methods are required that allow the user to enter individual digits or characters. Some approaches to achieve exact value input involve physical input devices that are taken into the VR system, such as keyboards, tablets and even small computers (like PDAs) [252, 259]. For projection based environments this is fine, but for head-mounted-displays this solution fails since the physical devices cannot be seen by the user. In section 4.3 we address these problems through an architecture that allows existing desktop applications and 2D user-interface libraries to be used in a VE.
Voice input is an alternative method that is suitable for exact value input in some cases [42]. Number entry is viable using most voice recognition systems through a well-defined vocabulary [107]. Text input however forms a far more difficult problem since a predefined vocabulary can, in most cases, not be defined. Although most automatic speech recognition (ASR) systems provide a dictation mode that allows free text to be entered, the accuracy of these systems in this mode leaves a lot to be desired [208, 264]. In section 4.4.2 we describe a speech recognition technique for quantitative and text input that exploits environment context to increase recognition reliability and allows multimodal interaction in a VE.

Quantification

Although stereopsis allows humans to perceive depth, it is know that distances and sizes in virtual environments are frequently underestimated due to a lack of adequate visual stimuli for scale [200]. An instrument for obtaining quantitative information from a visual representation is therefore a valuable asset. Section 4.5 describes GEOPROVE, a geometric probing software architecture for interactive data exploration environments, virtual environments in particular. This architecture allows researchers to probe visual presentations in order to obtain quantitative information. The properties that we want to measure could be obtained automatically using data analysis techniques, but this often requires designing and implementing specialized algorithms that are dedicated to the specific task. Quite often these techniques rely on heuristic algorithms that are difficult to design, implement and control.

4.2 Interactive scientific visualization in VEs

The primary method used to create virtual environments is through the rendering of visual constructs. Programming environments exist that are ideally targeted towards the construction of "content" for a VE, but in general these environments do not provide methods to incorporate interactive scientific visualization into the VE [212,218,219,236]. Likewise, many desktop environments exist that provide flexible methods to do scientific visualization, but few of these provide support for direct interaction and manipulation of graphical objects in VEs [103,172,209,242].
A combination of scientific visualization methods and interaction methods that allow a user to manipulate these visualizations from within a VE would allow for the construction of interactive exploration environments for use in scientific research. Using such a combination, environments could be constructed where scientific data is represented through well established visualization methods and the interaction capabilities support exploration.

4.2.1 The Visualization Toolkit

For our research we have chosen the Visualization Toolkit (Vtk) for our scientific visualization purposes [106,209]. Vtk is an open source object-oriented software system for 3D computer graphics, image processing and scientific visualization. The availability of the source code to Vtk allows us to extend its functionality so that it can be used for interactive scientific visualization in VEs. In addition, Vtk is supported on most major operating systems which greatly facilitates application development and portability. The combination of Vtk and the CAVE library produces an extremely versatile and powerful software environment for scientific visualization in VEs.
The Vtk pipeline is an extension to the pipeline described in section 1.3.2 (page 7) and is illustrated in Figure 4.1. The third column of this Figure also illustrates how consecutive stages in the pipelines are created, parameterized and connected in a program, in this case in the Tcl language [175,258]. Vtk pipelines start with source

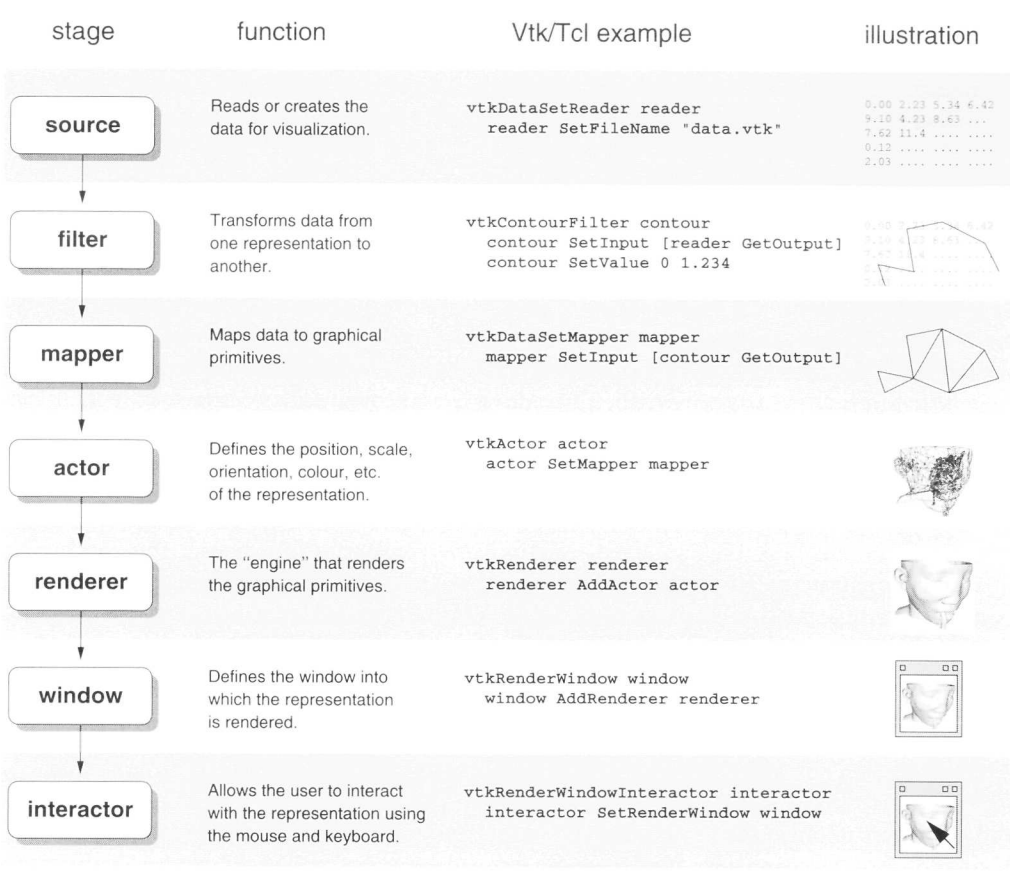

Figure 4.1: The Visualization Toolkit (Vtk) pipeline.

objects that either produce source data themselves or that read input data from external sources such as simulations or CAD programs. The output of a source object goes through several filters that alter the geometry. In most cases it is these filters that are the embodiment of a visualization algorithm. A mapper is used to map the output data from the filters to geometric constructs that can be rendered by a graphics rendering pipeline. The actor object is used to represent the resulting visualization and is used to define the type and properties of the representation. The renderer forms the interface between the visualization and the graphics rendering pipeline. On the desktop, the renderer displays the resulting images in a window while interaction with the visualization using the mouse and keyboard is implemented through an interactor. Other objects types in Vtk define light sources, cameras and mathematical functions.

4.2.2 SCAVI: Speech, CAVE and Vtk Interaction

Previous work by others resulted in a library to render Vtk objects in a CAVELib application [94]. This library copies all Vtk actors into a shared memory area to make them accessible to all display processes for rendering (please refer to Appendix A for details on CAVELib). Dynamically changing data is handled automatically. Not all of Vtk's functionality is supported (most notably cameras and lights are unsupported); we have made several extensions to the original version to support actor opacity, actor removal, visibility and texture mapping of actors. Although this work allows us to render Vtk visualizations in CAVELib applications, it did not provide methods to interact with them. We have developed a framework that allows rendering and interaction with scientific visualizations produced by Vtk. This framework, called "Speech, CAVE and Vtk Interaction" (SCAVI), provides the VE application developer with functionality to interact directly with the visualized objects using the CAVE wand and buttons and speech commands [95].

Direct object manipulation

Direct object interaction in SCAVI is implemented using a hierarchical intersection scheme: For each actor defined in the renderer, SCAVI first checks whether the line that starts at the location of the front of the wand and points in the direction defined by the wand's orientation intersects the bounding-box of the actor. If one or more intersections are found, a second line intersection test is performed on these objects and the distance from the wand to the intersection points (if any) are calculated. The object with the closest distance to the wand is considered to be "in focus". Only actors that are defined as "visible" and "pickable" can become focused (triggering a focus event) whenever there is a direct line-of-sight from the front of the wand to that object. Pressing and holding the first wand button allows a focused object to be grabbed and dragged to another location. Pressing the second wand button on a focused object selects that object, such that more elaborate manipulation can be performed. Only one actor can be selected at a time. Manipulations on actors that are in focus (by pointing at them) take precedence over actors that are selected. The application developer can request a reference to a selected actor or the actor that is in focus for subsequent action. Figure 4.2 illustrates a simple application that allows primitive objects (spheres, cubes, cones, etc.) to be manipulated interactively.
Every object in SCAVI is a Vtk actor with several enhancements defined in a C++ class called `vtkCAVEActor`. All SCAVI objects are kept in a dynamically linked list (DLL), which is not kept in shared memory making it only available to the main CAVE process and none of the display processes. The main extra features of a SCAVI object over a Vtk actor is the ability to add several event handlers to an object and the ability of the interaction component to identify and perform interaction on an object. Each SCAVI object can be given a name so that voice identification using speech technology or enumeration in a virtual menu becomes possible. Besides regular Vtk functionality, such as changing the color of the object or the visibility state, several functions have been rewritten to accommodate for the extended functionality, such as

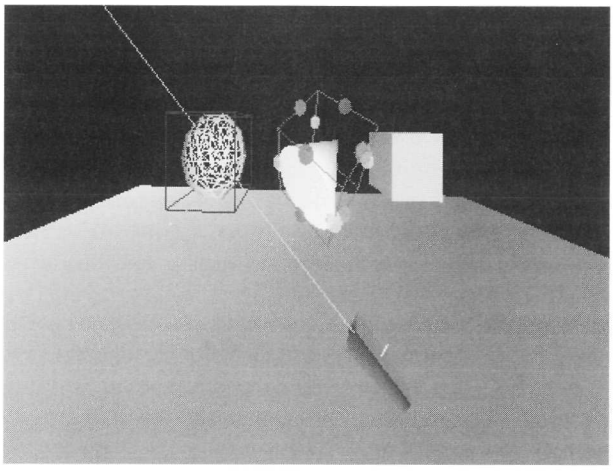

Figure 4.2: SCAVI *in use to manipulate Vtk actors.*

delete and copy. The rotating, scaling and translation functions are also rewritten, which makes it possible to transparently use either the transformation matrix at the end of the Vtk pipeline (i.e. from the actor) or after the source input. The benefit of using a transformation matrix directly after the source input is that use of subsequent Vtk filters will be aware of changes in the object's position, scale and orientation.

Although the concept of a "scene graph" is not explicitly implemented in Vtk (and most other scientific visualization environments), this functionality can be simulated through the use of a coordinate transformation filter (such as the Vtk class `vtkTransformPolyDataFilter`). This filter accepts vertex coordinates and a transformation matrix as input and produces transformed coordinates as output.

The transformation of coordinates requires a matrix-vector product for each coordinate and so the time required to transform larger objects gets increasingly large. This poses a problem in particular when a user manipulates the position, rotation or scale of an object. In a "real" scene graph system, this displacement is easily performed by inserting a transformation between the object and its parent that reflects the displacement of the hand from the moment the object was grabbed. With the use of a transformation filter, the new position of all polygons in the object must be recalculated at each frame by a transformation filter. This method may yield acceptable results for small objects, but for large objects the time spent by the transformation filter will make fluent interaction impossible.

In SCAVI this problem is circumvented by the following trick; as the user manipulates an object, a transformation matrix is associated with the actor representing the object so that the visual appearance of the object mimics the displacement intended by the user. The transformation matrix associated with the actor is used by the graphics rendering library (OpenGL) which performs the transformation in hardware, on hardware that supports it, and is therefore much quicker compared to the software

solution. As soon as the user releases the object, the actor's transformation matrix is copied and given to a transformation filter which then applies the transformation to all vertices in the object. During the interaction, the visual appearance of the object thus transforms, following the user's gestures, while the actual transformation of coordinates takes place just once when the user releases the object. As a result, there will be a slight delay from the moment the user releases the object and the time the object is actually transformed. Within this time, no interaction with the object is possible.

4.3 Bridging the gap: 2D applications in VEs

Many applications in everyday use are designed for desktop systems with 2D graphical user interfaces (GUIs). While these applications may not always be of particular use in VEs, some provide capabilities that can be very useful to the VE user or application developer. Web browsers, document readers and movie players, for example, are often used by desktop applications as "standard" utilities to provide on-line documentation or tutorials to the user. Desktop users also benefit greatly from "productivity applications" like calculators, file browsers or conferencing tools, to name but a few, that provide a rich set of accessories that enable the user to perform diverse tasks that are not offered within a single application. The same is true for VE applications; much of the functionality offered by the desktop utilities just mentioned would be quite useful to users immersed in a VE. Unfortunately, few of these utilities currently have 3D counterparts for use in immersive VEs.

A solution to this is to bring a desktop PC system or a handheld computer like a Personal Digital Assistant (PDA) into the VR installation to gain access to these utilities [252, 259]. However, this solution is impractical in combination with head-mounted displays (HMDs) since the display obscures the physical devices from view. But also in projection based VR systems, the user needs to "escape" from the VE to access the system in the real world. Others have looked at solutions in which the application is rewritten, or more specifically; the part that builds the GUI, so that the GUI is presented in the VE [6, 7]. This either requires that the utility is available in source form, and the VE application developer is up to this task, or that 3D equivalents of 2D GUI toolkits are available [43]. Few GUI toolkits for use in VEs exist, however, and the ones that do, often do not offer the same richness that is needed to effectively express the same functionality. Moreover, applications that are distributed in binary form only can not be adapted at all.

Although one can argue whether 2D interaction metaphors are efficient for use in immersive environments, the advantages are multitude. Users are accustomed to applications with 2D GUIs, so presenting them with existing utilities that can be used in a VE provides the user with familiar applications that he knows to use well. The immersive experience of the VE is maintained as the user is no longer forced to step into the real world to interact with a GUI on a desktop. VE application developers would be able to exploit existing applications, as do desktop applications developers,

and focus on the task for which the VE is intended. Application developers would be able to quickly construct GUIs using the flexibility of existing 2D GUI toolkits instead of implementing 3D equivalents.

4.3.1 XiVE: X in Virtual Environments

X in Virtual Environments (XiVE, pronounced "zive") provides access to existing 2D desktop applications by "swallowing" their GUI into the VE. The ideas behind XiVE have earlier been described by others.
Dykstra in [70] describes a modified X server that renders images into memory instead of to a graphics device and accepts device input events through a FIFO queue. The VE application then maps the images rendered by the X server onto textures in 3D space while interaction with the applications is made possibly using the FIFO queue. Developed at Chalmers Medialab, 3Dwm uses the Virtual Network Computing (VNC, [38]) remote display system protocol to distribute a bitmapped desktop across a network [71, 159]. XiVE uses a more generic and flexible approach by exploiting the client/server protocol that is used in the X Window System. The X Window System (from hereon referred to simply as "X") was designed with network transparency in mind. This means that an application (the "client") can run on one host and open windows on a "server" running on any accessible host (meaning: reachable across a network and having permission to access the server's resources).

4.3.2 Design of XiVE

XiVE maintains a list of windows that are to be presented in the virtual environment (the "monitored list"). Each window in this list contains a display and identifier that together uniquely identify a window on an X server. For each window, XiVE's update mechanism grabs the contents of the window from the X server and stores its pixel representation in memory. This pixel representation is converted into an OpenGL texture and rendered onto a rectangle in virtual space (see Figure 4.3) [261]. Each window has a transformation matrix associated with it so that a window can be placed at any location, orientation and scale in virtual space, integrated with the other content generated by the VE. Once the windows to monitor are identified, XiVE repeatedly grabs their contents and renders them as textures in the VE. At the same time, XiVE monitors the VE's interaction devices for intersection with the displayed textures and manipulator events (such as button presses). When this happens, XiVE generates synthetic motion, button and keyboard events (using the XTEST extension in the X server) to mimic the behavior of a conventional mouse or keyboard input device.

Identifying windows and displays

Windows and displays are identified by the hostname of the system running the X server and a numeral to identify the display, screen, window or a shared memory

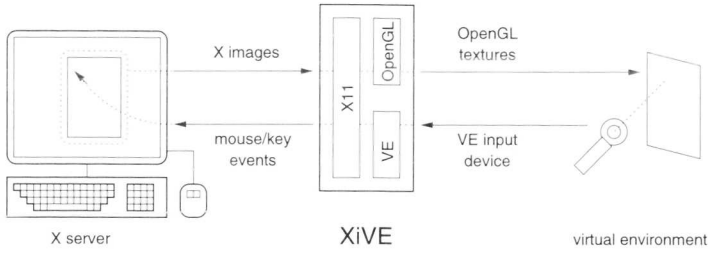

Figure 4.3: XiVE *grabs window images from X servers and renders them in the virtual environment as OpenGL textures. Interaction with the applications is made possible through synthetic keyboard and mouse events using the* XTEST *extension of the X server.*

identifier, using the following notation:

```
[hostname]:display[.screen][#window|%memid].
```

The members between square brackets are optional; if `hostname` is empty, the local host is assumed; if `screen` is empty, the default screen of the display is used; if `window` is empty, all windows on the display are used; if `memid` is given, a shared memory rendering X server is used. Using this identifier, windows and displays on an X server can be rendered in a VE using any combination of the following configurations (see also Figure 4.4):

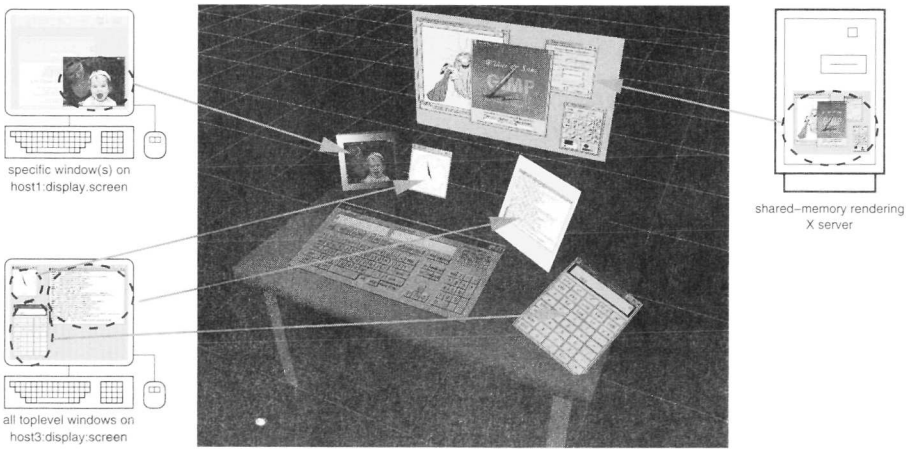

Figure 4.4: XiVE *can monitor windows from specific windows on an X server (top left), all top-level windows on an X server (bottom left), or shared-memory rendering X servers (top right). See also colour reproduction on the back cover.*

1. **Specific windows** - For example: `dexter:2#0x23` which identifies window `0x23` on the default screen of display `:2` on host `dexter`. The window is represented by a textured rectangle in the VE. Note that the "root window" on an X server is a special window, represented by a unique window identifier, that contains all other windows on the screen.

2. **Top-level windows** - For example: `dexter:0.2` which identifies all top-level windows on screen `2` of display `:0` on host `dexter`. Each window is represented by a textured rectangle in the VE. The initial placement of each new window mimicks that on the X server so that overlapping windows on the X server overlap in the same way in the VE. This ensures that pull-down menus and pop-up windows appear at the same relative location. This initial placement can be overridden using the transformation matrix so that, for example, the window can be moved elsewhere through user interaction.

3. **Shared memory rendering X server** - For example: `:9%12345` which identifies a shared memory rendering X server running on the local host with display number `:9` and shared memory id `12345`. The root window of a shared memory rendering X server is represented as a single textured rectangle. The window image can directly be accessed by a XiVE application, since it is located on the same machine, so that it does not have to be copied using a network transfer which improves update rate. The display identifier is required in this case only to enable interaction with the client applications rendering on the X server. An example of an X server that renders into memory instead of on a graphics display is `Xvfb`, which comes with most X distributions [140].

4.3.3 Performance issues

Two factors characterize the performance of a XiVE application. The first is *update rate*; the frequency at which new content of an X window is represented in the VE. Low update rates may result in important information being missed while at the same time it may make interaction with the application difficult, especially when visual feedback is of importance during the interaction with the X application. The second is *frame rate*; the frequency at which a representation of an X window can be rendered in the VE. Since most immersive environments render the environment from a user-centered point-of-view (using the position and orientation of the user), the frame rate should be at least 10 frames per second for a responsive environment. Low frame rate results in decreased response from the environment which in fully immersive systems can cause motion sickness or at the very least negatively influences interaction with the VE.

The main factors that limit performance are (1) the limited performance of the network connection between the host running the XiVE application and the host running the X server, (2) the delay caused by the conversion of a grabbed image into a representation that can be used as input format for OpenGL textures, (3) the delay caused by the definition of the texture, and (4) the time that is required to render a textured

rectangle in the VE. XiVE uses a number of techniques to reduce these overheads during the four steps that are required to update a representation of a window in the VE:

1. **Obtaining a window image from the X server** - In a situation where the X server is not running locally, the image must be obtained using a network transfer. High network latency reduces response time, low bandwidth reduces the frequency of updates that can be rendered by XiVE. These network limitations can be reduced by executing an X server on the same machine as where XiVE is used so that communication takes place internally, or by using an X server that renders images into shared memory which can be accessed without any network communication.

2. **Conversion of an image to a format suitable for texture rendering** - Images retrieved from an X server are stored in color indexed format so that every pixel is represented by a value that represents a color through a lookup table. OpenGL version 1.1 and later supports this kind of image format in different flavors and XiVE attempts to detect which are supported by the implementation of OpenGL. If no support for color indexed textures is detected, XiVE falls down to an algorithm that converts the color indexed images to RGB format which is supported by all OpenGL implementations. The latter format takes time to generate and requires considerable more memory to store and so XiVE benefits from OpenGL libraries that have support for color indexed textures.

3. **Texture definition** - Defining textures in OpenGL often involves downloading the texture into dedicated texture memory on a graphics interface. To avoid unnecessary overhead, XiVE detects whether a window's contents has changed before redefining its texture. Although this requires that a new image is obtained from the X server, this update scheme executes in a separate thread of execution in XiVE. This thread detaches from the rendering processes and constantly monitors the windows maintained by XiVE and redefines a texture only when the contents of a window has changed.

4. **Texture rendering** - Texture rendering is one of the most expensive operations in computer graphics. XiVE benefits greatly from graphics hardware that is capable of performing texture mapping in hardware.

Performance results

Tables 4.1, 4.2 and 4.3 illustrate the overhead in these four steps for a profiled XiVE application running on a Sun UltraSPARC-IIi at 300 MHz, running Solaris 8 and equipped with an OpenGL 1.2 accelerated Elite3D graphics interface. The XiVE application grabs the root window from an X server running an X server at 800 by 600 with 16 bit pixels (937.5 kB total image size) and renders this window using either RGB or color indexed (CI) textures.

	RGB (time)	(%)	CI (time)	(%)
image transfer	1040 ms	49.7%	1040 ms	58.7%
conversion	750 ms	35.9%	430 ms	24.3%
texture definition	300 ms	14.3%	300 ms	16.9%
rendering	1 ms	0.0%	1 ms	0.0%
total	2091 ms		1771 ms	

Table 4.1: Overhead of the four steps to update a window representation using a 10 Mbit/s ethernet connection between XiVE *and the X server.*

In Table 4.1 the two systems are connected via a 10 Mbit/s ethernet. The table shows that most overhead is caused by the transfer of the window image from the X server. Since the update rate is directly related to how fast images can be obtained from the X server, this configuration is useful only in cases where update rate is not of high importance. Note that the time to render the window is only 1 ms which leaves ample time to render other geometry in the VE and still maintain a sufficiently high refresh rate. The table also shows that the time required to convert the transfered image into a suitable format for texture rendering is significantly lower in the case of color indexed textures.

	RGB (time)	(%)	CI (time)	(%)
image transfer	36 ms	3.4%	36 ms	4.7%
conversion	730 ms	68.4%	430 ms	56.1%
texture definition	300 ms	28.1%	300 ms	39.1%
rendering	1 ms	0.0%	1 ms	0.0%
total	1067 ms		767 ms	

Table 4.2: Overhead to update a window representation using internal network communication.

In situations where high update rates are critical, the transfer overhead can be significantly reduced by running the X server on the local host, as shown in table 4.2. As the X server in this situation is running on the same host as the XiVE application, all image transfers occur using internal communication. The table clearly shows that the transfer overhead is significantly reduced. In fact, this overhead can be completely canceled through an X server that renders into shared memory; as XiVE can access this memory directly, no images need to be transferred at all, as can be seen from Table 4.3.

4.3.4 Conclusions

We have described an architecture that allows existing applications with 2D graphical user interfaces to be used in 3D immersive virtual environments. This architecture bridges a gap between the 2D user interface libraries that exist for desktop application development environments and VE development environments that in general

	RGB (time)	(%)	CI (time)	(%)
image transfer	0 ms	0.0%	0 ms	0.0%
conversion	730 ms	70.8%	430 ms	58.8%
texture definition	300 ms	29.1%	300 ms	41.0%
rendering	1 ms	0.0%	1 ms	0.0%
total	1031 ms		731 ms	

Table 4.3: Overhead to update a window representation using a shared-memory rendering X server.

lack these facilities. Through XiVE, VE application users are presented with user interfaces and applications they know well from conventional desktop applications which greatly facilitates interaction in the VE and may result in increased productivity. The design of XiVE allows existing VE applications to be easily extended with this feature at little loss of performance. For example, Figure 4.5 shows XiVE used in the VRE application described earlier in section 2.3 (page 29) to set a threshold for the isosurface modeling visualization pipeline via an interface designed in Tk [175].

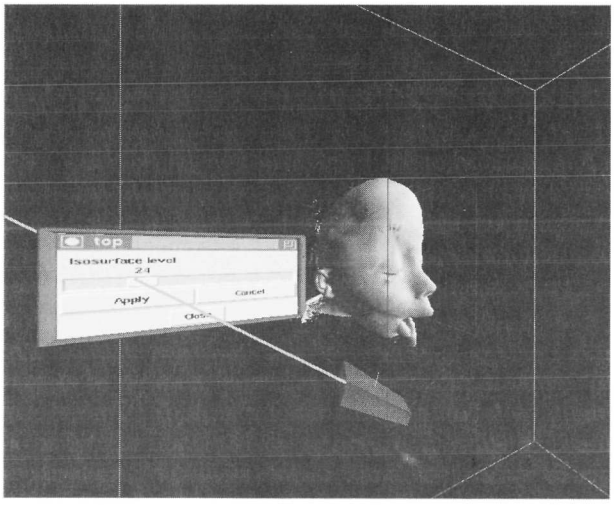

Figure 4.5: XiVE used in the VRE environment (see section 2.3) to set an isosurface threshold via a user interface defined in Tk [175].

4.4 Context sensitive interaction

The applicability of an interaction method is largely determined by its degrees of freedom, in providing sufficient expressiveness so that multiple functions can be triggered by similar but different gestures, and by its robustness to discern one gesture from

another. With the increase in functionality that can be triggered in an application using an interaction methodology, the chances increase that a user triggers a function that was not intended. A relatively unexplored paradigm for interaction in a VE that can be used to reduce this, is to restrict whether gestures are recognized based on the context of the application. Although it may seem contradictory to constrain a user in a restricted set of gestures, the aim of the technique described here is to provide the user with more efficient, goal-directed interaction methods by eliminating those that are out of context [23]. In the following we will describe the concept of "application context" in a VE and how context can be used to improve the efficiency of an interaction method.

4.4.1 Determining application context

During the use of a VE, the environment takes on a new state as the user interacts with the environment. Each change as a result of user interaction or autonomously operating entities in the environment, alters the state of the environment, potentially bringing the user into a different context. Accurate determination of context is not always a trivial task. In general, properties that have the most significant impact on context and that can be easily obtained from the VE are:

- The state of entities in the environment - The existence or properties of entities in the environment imply that there is a chance that the user may want to interact with the entities or its properties. Therefore, the interaction methods supporting the interaction on this entity should be enabled. Likewise, if entities or properties are removed, its associated interaction methods may be disabled.
- The user's focus of attention - When a user is located in the vicinity of, or has his focus on a certain area in the environment (either by looking at, being in the vicinity of, or pointing at entities in the VE), there is a chance that this user will want to perform interaction on entities in this area, and not on those which are, for example, out of view.

Determining these properties is possible if the VE provides access to the data structures that determine them.
Enabling every possible interaction method in the environment may result in incorrect interpretations of the user's intentions and may therefore result in an unintended response from the environment. By disabling interaction methods that are not applicable to the context of the environment at some particular moment, the chances that gestures are mistakingly recognized for something which was not intended can be decreased.

The context engine

We have built a "context engine" that monitors the state of the environment asynchronously [267]. This context engine provides information to "agents" that dynamically adapt the interaction methods enabled by the environment at any time. Based

on that, interaction methods are enabled or disabled automatically as the context of the VE changes. Still subject to ongoing research at the University of Amsterdam, the agents have the ability to perceive state changes through monitors, take actions that affect conditions in the environment and perform reasoning to interpret perceptions, solve problems and determine actions. As a test case, we have applied the context engine to improve the reliability of interaction via speech recognition.

4.4.2 Context sensitive speech recognition

Speech recognition involves the transformation of an acoustic speech signal into written text. Development of speech recognition has been ongoing for over 25 years and only recently speech applications are coming into widespread use. Speech recognition can be very useful in applications where hands and eyes are constantly busy, which is often the case in immersive environments, or for people with disabilities that inhibit their motor skills. Psychological studies have shown that users of virtual environments prefer to use a combination of speech and hand gestures so that they can concentrate on the objects in the virtual environments and the task at hand [96, 97]. Also, speech input does not require special training of the user or unfamiliar input devices. Moreover, since speech recognition systems operate on a different input modality (namely sound), they are not hampered by limitations of graphics resources, thereby making it a suitable alternative input modality for situations where visual fidelity (such as frame rate) is critical.

Common problems with speech recognition

Acceptance of speech technology is growing although there are still some key issues holding it back, i.e. recognition accuracy and ambiguity [42, 177, 208, 228]. Common situations where accuracy is important is where the speech recognition system must be able to discern words that have approximately the same pronunciation (like "dye" and "die"). Ambiguity occurs when the same verbal command is used to trigger different functionalities. For example, consider an application that supports a "volume" command. In one particular context this command could refer to the output gain of an audio device while in another it could refer to a quantitative property of a three-dimensional object. To resolve this ambiguity, the user would have to add extra information (e.g. "audio volume" or "object volume") in order to unambiguously identify the target for this command. Although the recognition will perform better with this extra information added, the user is expected to have a thorough understanding of the speech recognition system's grammar. This will not make the system easier to use. These problems can be reduced by incorporating application context into the speech recognition systems.

Related work

In 1980, Bolt has shown the added benefit of interaction through a combination of speech and context information derived from hand gestures in his "Put That There"

demonstration system [21]. The application and benefits of speech input in visualization and virtual environments is described in [215, 239]. Oviatt et al. integrate complementary modalities in a manner that supports mutual disambiguation of errors to improve performance [177, 263]. Suhm et al. describe a multimodal error correction that uses context provided by additional modalities (like key, mouse and pen input gestures) to correct errors [228].

Applying context to speech recognition

Context sensitive speech recognition (CSSR) systems can have large vocabularies (i.e. thousands of words), but only a subset of that vocabulary is activated at a particular time. The complete vocabulary is subdivided into subsets that contain commands that are applicable to the context for which they are intended. While the application is running, the context engine determines the context of the application and enables only those subsets of the vocabulary that are applicable to the application context. As only a part of the complete vocabulary is enabled, the chance that a verbal command is incorrectly recognized is reduced, thereby increasing the overall accuracy of the speech recognition system. Note the resemblance with conventional desktop applications where the user is also frequently inhibited from invoking some functionality that are "inappropriate" by disabling the options.

Performance results

A speech recognition library capable of context sensitivity (CAVETalk) has been developed. The core of CAVETalk uses IBM's ViaVoice as third-party recognition software [47, 104, 105]. ViaVoice is a speaker independent speech recognition system that can handle isolated or continuous recognition. User specific training is not necessary, but can be done to improve recognition accuracy. Vocabularies in ViaVoice are defined in Backus-Naur Form (BNF). Using this form, it is possible to define extremely large and flexible vocabularies in a very concise notation, as shown in an example in Figure 4.6.

An experiment was conducted in which the user was presented with a verbal command set that consisted of 57 words in total, divided into five subsets. The user was asked to issue the commands in random order, as provided by an external randomization program. During a single experiment, the user was asked to issue each command twice; once when all words in the vocabulary would be enabled, the second time only the subset of the vocabulary that contained the command would be enabled. The user did not know whether the whole vocabulary or only a subset would be enabled.

In total, 13 experiments were performed. In the non-context situation, the mean number of correctly recognized commands was 45.2 (79.4%) with a standard deviation of 4.4. In the context situation, the mean number of correctly recognized commands was 49.5 (86.8%) with a standard deviation of 4.1, an average improvement of 7.4%. Although this improvement may seem marginal, the vocabularies in this experiment were intentionally chosen to contain words that had little similarity in pronunciation. For vocabularies that contain words with a similar pronunciation the results

```
<<start>>
  = "create" "a" ? <target> <destination>?
  | "delete" <denom>? <target> <source>?
  | "move" <denom>? <target> <source>? <destination>
  | <quit> .

<target> = "sphere" | "cube" .

<denom> = "the" | "that" | "this" .

<source> = "at the" <location> "of" <denom>? <target> .

<destination>
  = "here"
  | "to the"? <direction>
  | "to the" <location> "of" <denom>? <target> .

<direction> = "up" | "down" | "left" | "right" | "front" | "back" .

<location> = "top" | "bottom" | "left" | "right" | "front" | "back" .

<quit> = "quit" <program>? | "exit" <program>? .

<program> = "program" | "application" .
```

"create cube", "create a sphere", "create a cube to the left of the sphere", "create a sphere here", "move cube right", "move that cube to the back of this sphere", "move the cube at the left of that sphere to the right of this cube", "move the cube up", "move that sphere here", "delete the cube", "delete the sphere at the left of that cube", "exit program", "quit".

Figure 4.6: Example of a vocabulary in BNF notation used by the ViaVoice speech recognition system. With this vocabulary, 24558 different sentences can be recognized, some of which are shown at the bottom.

are much higher, provided these words are in different subsets and are not activated at the same time.

User reactions

In the environments we developed, users are able to navigate through the VE with great ease by pointing at objects in the environment and saying "come here" or "go there". The use of context also dismissed the need for "push-to-talk" solutions where the voice recognition system's attention is explicitly told when to listen to verbal commands [142], as the system only listens when there is clear defined context. This allows users to freely discuss the things that are going on in the environment with

their peers, without having to worry that the environment responds unintentionally.

4.5 GEOPROVE: Geometric Probes for Virtual Environments

In many scientific computing problems, the level of complexity in the generated data is too vast to analyse numerically. For these situations, interactive scientific visualization is an essential method to present and explore the data in a way that allows a researcher to comprehend the information it contains. Immersive virtual environments such as the CAVE [52] further enhance a researcher's perception and are therefore often used to obtain better insight in multi-dimensional datasets for which desktop visualization environments are too restrictive.
In most scientific visualization environments, a visualization pipeline transforms numerical data into geometric constructs that are rendered into visual presentations. These presentations allow researchers to qualitatively analyse their data. Many visualization environments stop at this point and provide little means to obtain quantitative information on what is being presented. Through the use of stereoscopic images it is possible to estimate quantitative properties, such as the (relative) size and distance of virtual objects [200]. This may be acceptable to some applications, for others however, an instrument for obtaining quantitative information from the visualization is a valuable asset. Examples of this are applications where simulations are verified to the real-life phenomena that are being modeled, applications for diagnostic purposes based on medical data obtained from medical scanners (i.e. CT, MRI, etc.), or computer aided design tasks.
In this section we describe GEOPROVE, a geometric probing software architecture for interactive data exploration environments, virtual environments in particular. This architecture allows researchers to probe visual presentations in order to obtain quantitative information. We will show the application of this probing system with the test-case described earlier in section 2.4 (page 42).

4.5.1 Related work

While most scientific visualization environments provide some probing functionality, most of these act as subset selectors that extract selected regions from larger data sets for localized visualization, complementing global visualization methods [172, 242]. The Visualization Toolkit (Vtk) for example provides probe filters for the computation of point attributes in local areas. Point attributes are computed at input points specified by a probe consisting of a geometric structure by interpolating into the source data. Vtk also contains methods that calculate properties such as the volume, surface area and normalized shape index of closed triangle surfaces [209].
The work by van Leeuw et al., takes this method one step further by using visual probes that consist of a set of geometric primitives. Multiple characteristics of a small

area in a flow field transform the geometric primitives in the probe to visualize velocity and local change of velocity [56]. Although these visual probes provide excellent means for exploring local properties of a dataset, they are often used for localized visualization only, and not for obtaining quantitative measurements.
The work by Brady et al., shows a CAVE application for the visualization of biomedical images obtained from medical scanners that allows features to be manually traced and labeled [25]. This software has been used for obtaining the lengths of biological structures and for segmenting medical images.
Germans et al. describe an approach to obtain measurements from the visual domain which is similar to what we describe here [194].

4.5.2 Geometric probing

In our system, geometric probes consist of one or more markers. These markers are used to sample properties of the presentations in the virtual environment. A property can either be the coordinates of a position in the environment, or a value obtained from data at this position. The property obtained from the markers are used in an evaluation function producing the result of a measurement. The evaluation function defines a relation between markers in a probe, which we illustrate here by connections between markers.
Determined by the spatial configuration of the markers, probes have dimensions and a certain degree of freedom by which they can be positioned over an area of interest. For example, a probe consisting of exactly one marker has 3 degrees of freedom in a 3D environment (translation in 3 directions) and can thus either be used to obtain the position (x,y,z) of a feature in 3D space or a mapped quantity $f(x,y,z)$ at this position, where f provides a mapping of a position to a quantity (i.e. a scalar, vector, tensor). An evaluation function takes this sample and produces the result of the measurement.

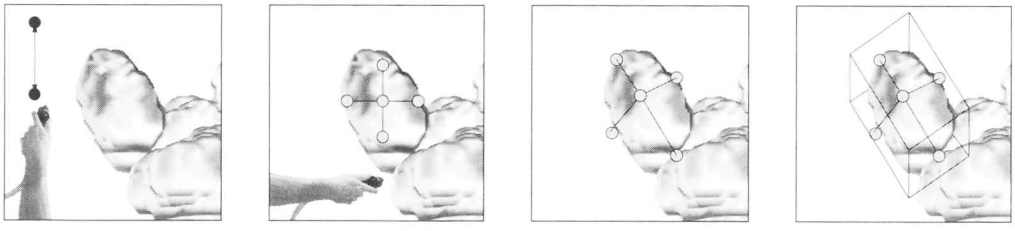

Figure 4.7: Probing procedure, as an example applied here to determine the bounding volume of part of a structure. From left to right: calibration, interactive placement of the probe, registration of the probe to the data, calculation of bounding box.

A measurement procedure with probes in a virtual environment requires the following course of actions (see also Figure 4.7):

1. Calibration - As in any measurement, the properties that are calculated based on sampled quantities need to be referenced to a well defined unit. This unit

of reference is especially required when the measurements from two different objects are to be compared. For the same type of measurements, calibration will only have to be performed once.

2. Placement of the probe - Interactive placement of a probe in a virtual environment makes use of devices whose position and orientation are tracked in three-dimensional space. Through these devices, a user is able to place a probe roughly over the region of interest.

3. Registration of the probe - The interactive placement of a probe is not accurate in most cases because of inaccuracies in the tracking hardware or inexperience of the user. Registration of the probe involves refining the position, scale and rotation of the probe, either interactively or aided through registration functions.

4. Calculation of the result - Once the probe is in place, the result can be calculated. Depending on the type of probe, the calculation is either performed purely based on the position and orientation of the probe, or the positions of each marker are first mapped to a quantity.

5. Presentation of the result - When the calculation is finished, the results need to be presented to the user in some meaningful way. In addition, the user should be able to log the measurement on file for later analysis and some method of annotation is required so that the user can relate back to the measurement once they are analysed elsewhere.

# markers	1	2	3	4	5
probe					
positional property	position	length, distance	angle, curvature	bounding rectangle	saddle point
mapped property	value	derivative	extrema	surface	Gaussian distribution

Table 4.4: Examples of probes with different number of markers and examples of positional and mapped properties that can be measured with these probes.

Table 4.4 shows some examples of probes consisting of a number of markers and examples of properties that can be obtained. Most of the properties in this table can be relatively easily obtained. For the first implementation of this architecture, with the coral growth data exploration system as a test-case, we limit ourselves to measurements that require probes consisting of 2 markers (length, distance) and 3 markers (angles). However, in the design of GEOPROVE we have attempted to keep the architecture generic so that the addition of probes for the acquisition of other properties can be achieved with little effort.

Software architecture

Interactive visualization applications benefit from a design in which the computation and visualization processes are implemented by separate communicating threads of control [31]. The library used in CAVE environments supports primitives for this design [249]. If implemented carefully, this configuration allows interactive virtual environments to be built that have a high frame rate and minimal interaction delay. However, it does have implications for the design of the GEOPROVE architecture. The software architecture that we have developed is shown schematically in Figure 4.8.

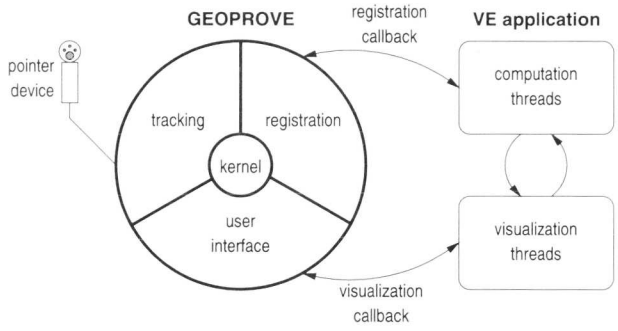

Figure 4.8: The software architecture for GEOPROVE *and its interface to a virtual environment application.*

The heart of GEOPROVE consists of the *kernel* that logs the positions and annotations of the probes, performs calculations and maintains a record of the measurements for storage and retrieval. The *tracking* part provides the kernel with position and orientation information of pointer devices with which probes can be positioned in the virtual world. The *registration* part manages the positioning of probes based on information provided by the application. This allows probes to be "snapped" in real time to computed data so that accurate measurements can be obtained. The *user interface* part provides access to GEOPROVE, including the rendering of probes, the presentation of results and annotations, and methods to drive GEOPROVE.
Measurements can be obtained on two levels; on the first, most basic level, measurements are based purely on the position and orientation of pointer devices that are tracked in three-dimensional space. On the second level, measurements take place based on structures defined by the application. This two-level scheme is described in more detail in the following two sections.

Tracker based probing

On a most basic level, probes are positioned based solely on the location of pointer devices that are being tracked in the visualization environment domain. Note that in this situation measurements are taken without a direct relation to the application other than the user's perception of the environment. This allows simple mea-

surements to be made such as distances and angles without any feedback from the application.
As this method of probing is independent of what is being presented, probes can not be registered on data from the application. Therefore, its accuracy and usefulness is inherently dependent on the quality of the tracking systems, the experience of the user and the application for which it is used. Most tracking systems are sensitive to noise, therefore GEOPROVE supports scaling the coordinates obtained from the tracker system such that a more accurate positioning of a probe can be performed. In this research, the physical devices that are tracked consist of a head tracker and a "wand", a device that is similar to the desktop mouse but which is tracked using a six degrees-of-freedom sensor [205].
The main benefit for this kind of probing lies in its simplicity and its allowance for the fast acquisition of positional measurements such as lengths, angles, spatial derivative approximations and special geometries like the fractal box dimension [78]. Furthermore, as the implementation of this kind of probing can be isolated into GEOPROVE itself, it minimally interferes with existing VE software.

Mapping markers to quantities

Quantities that relate to properties that are defined by the application can only be obtained by interrogating the visualization or computation thread. Since GEOPROVE does not have direct access to the data maintained in the application, the quantification of a marker from its position has to be handled by the application via a callback function.
Some quantities can be best obtained from the abstractions that are made from application data when these are visualized. An example of this is the calculation of a surface area which can be approximated using isosurfaces that are extracted from grid based data for visualization through e.g. a surface extraction algorithm. For other measurements, the data contained in the computation may be of higher quality.

Registration

For some types of measurements it may be necessary to perform calculations on specific features of the underlying data. In these cases the probe needs to be aligned to these features before calculations can be performed. Depending on the probe's shape and its degrees of freedom, the registration of a probe to the underlying data sets takes the same form as the techniques that are used in *geometric hashing* [260]. In short, this method first takes two markers of the probe as a handle to match "points of interest" in the geometric dataset. Using geometric transformations (the most common being translation, rotation and scaling) a basis is constructed which determines the exact position of all markers in the probe. Through a voting mechanism a histogram is then constructed of candidate bases from which the best candidate for registration is chosen. Using this method, positions acquired from tracker sensors can be registered to data structures that have been used for visualization or computation.

Although this technique is in general computationally expensive, we may exploit "focus locality" by disregarding the geometry which is far from the user's focus.

Presentation, logging and annotation

GEOPROVE does not render its own user-interface. Instead, it relies on the VE application to do this. This allows GEOPROVE to be seamlessly integrated into existing applications. Most of our applications contain standard user interface components that can be used by multiple processes at the same time (see also Figure 4.10). Existing software can be instrumented with probing facilities with minimal effort. In the current version, only four function calls need to be added to the source code of an existing application to obtain the most rudimentary features. These functions consist of: (1) the initialization of GEOPROVE, (2) a display handler that renders feedback to the user for both interaction and presentation of results, (3) a user interaction handler, and (4) a registration function that allows markers to be snapped to visualized geometry.
Probe locations and measurement results are stored in a log file. During runtime, the user can browse through this log via the user interface and view entries or delete entries. The user also has the option to add annotations to a measurement via a "snapshot"; a virtual photograph made from the perspective of the user. Both low resolution (for runtime inspection) and high resolution (for off-line inspection) pictures are supported. We are currently adding the option to record a speech-sample to annotate measurements with extra information.
When the application is terminated, the log is written to a file, containing the measurements and references to possible snapshots and speech annotations. The log can then be analysed on a workstation, saving valuable CAVE time.

4.5.3 Position accuracy of the SARA CAVE tracking sensors

We have performed experiments to measure the accuracy of the tracker installation in the SARA CAVE. We are not so much concerned with the difference between the reported tracker position and the physical position of the tracker, since this can (at least partially) be corrected with visual feedback (e.g. by drawing a cursor at the reported tracker position). What is important for a probing system is the ability for users to indicate a certain position in 3D space. The accuracy with which this can be done depends on several factors: the resolution of the tracker system, the quality of the visualization system, and the skill of the user.
Our measurements are based on a task where a sphere was drawn at one of 196 target positions, and another sphere (drawn at the reported position of the wand) had to be 'superimposed'. Figure 4.9 shows a visual representation of our results. These measurements were obtained by sampling the 196 target positions over 6 independent experiments by two experienced CAVE users. Each target position was sampled between 10 to 15 times. These measurements clearly show that large deviations from target positions take place most often in the corners and at the entrance of the CAVE,

making it almost impossible to accurately position a marker at these locations. We have observed "jumping" behaviour in the reported position of the wand where slight changes in hand position result in very different reported positions from the tracker. The worst deviation we have measured was as high as 47 cm. This error can be attributed to a large concrete pillar at the entrance of the CAVE which is far away from the magnetic source and very likely contains metallic reinforcements that distort the magnetic field.

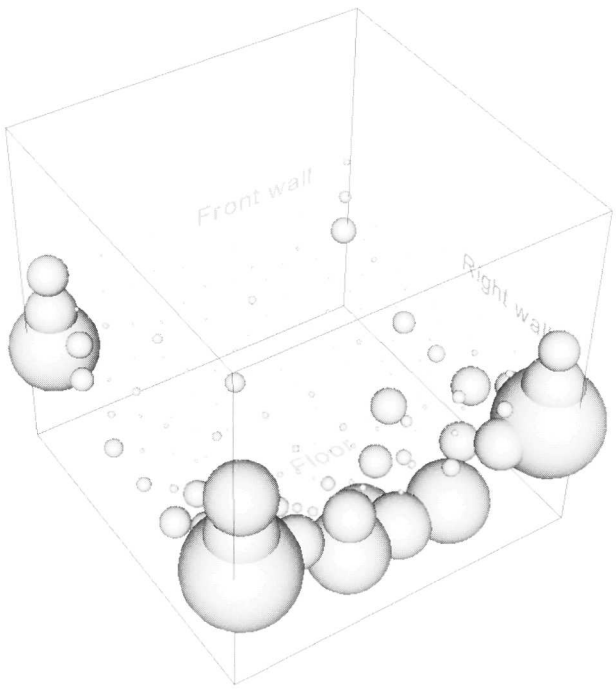

Figure 4.9: Mean deviation of reported tracker positions against target positions in the SARA CAVE.

To compensate for these large errors, we have opted for a refinement system in which the user is able to refine initial probe positions through down-scaling the tracker output. This way, large displacements of the hand result in smaller displacements of the probe, allowing probe positions to be refined irrespective of tracker hardware restrictions.

The best results are obtained in the center of the CAVE. Here target positions can be indicated with very small deviations (< 1 cm). Ways to increase the accuracy of the tracker systems used in projection-based VR systems such as the CAVE are reported in [54].

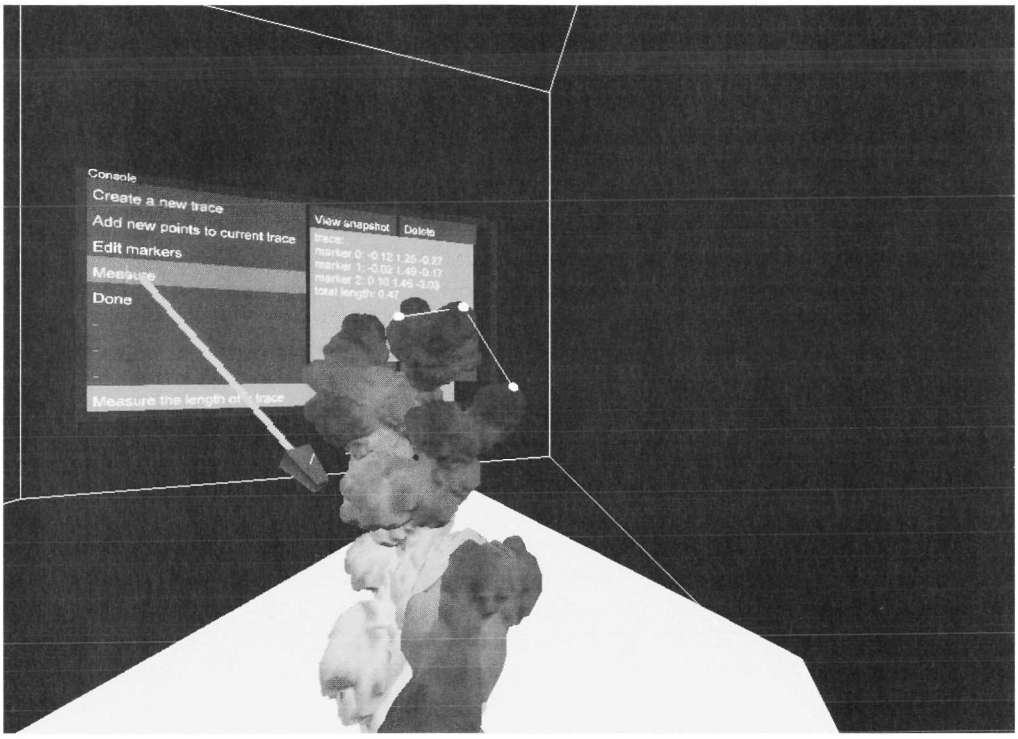

Figure 4.10: A CAVE simulator snapshot of an application instrumented with GEOPROVE. *Three markers (shown in white) of a trace are registered (or "snapped") to the visual geometry. The window in the back shows the user interface to* GEOPROVE *and presents the length of the trace.*

4.5.4 Results

We instrumented the interactive coral growth exploration environment described in section 2.4 (page 42) with GEOPROVE to obtain measurements of the shortest distances between neighbouring branch ends ("branch spacing") (see also Figure 4.10) [13]. The importance of this spatial observable is described in [211].
Figure 4.11 shows histograms of 106 measurements obtained from a CT scan of a real coral structure (*Pocillopora damicornis* in a sheltered environment) and 155 measurements obtained from 8 simulated coral structures under similar conditions.
It can be observed from the measurements shown in Figure 4.11 that there is a remarkable difference between the measurements of the branch spacing done in the CT scan of *Pocillopora damicornis* and the simulated growth forms. Three observations can be made from these measurements:
(1) The measurements from the simulated structures seem to be bimodally distributed while the CT scan measurements are unimodal. This may be the result of mea-

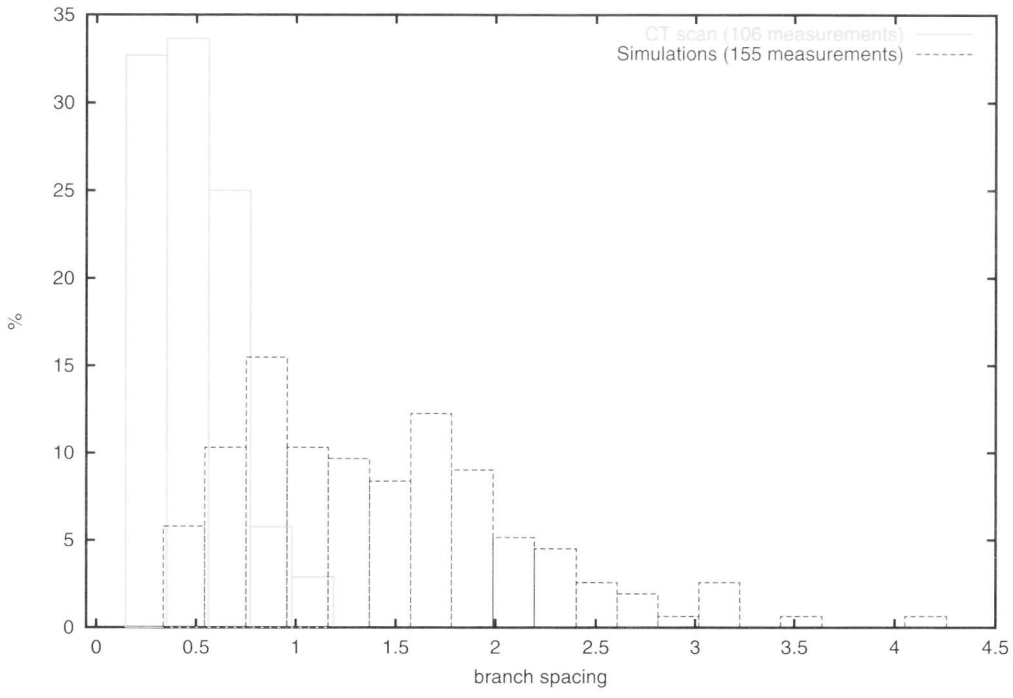

Figure 4.11: Comparison of branch spacing in a CT scan of Pocillopora damicornis *in a sheltered environment with 8 simulated structures under similar conditions.*

surement artifacts or it may be caused by the simulation mechanics. To fully understand the reasons for this we would need to obtain a larger number of more detailed measurements, which we have not yet been able to acquire.
(2) The mean branch spacing of the two distributions differs significantly (≈ 0.5 versus ≈ 1.5). This may be due to the scaling of the measurements: in order to obtain a scale that makes the simulated structures comparable to the real objects, the measurements from the simulated structures have been scaled using a factor obtained from the ratio in dimensions of the simulated structures and the real coral.
(3) In real *Pocillopora damicornis* the variability is relatively low compared to the simulated forms. These measurements seem to indicate that there is a mechanism which regulates growth of branches in the immediate vicinity of other branches, this mechanism is not present in the current simulation models. In the study by Rinkevich and Loya [196] it is proposed for the branching stony coral *Stylopora pistillata* that there is a chemical signal mechanism which regulates growth of branches, resulting in a relatively low variability and even a remarkable uniformity of the branch spacing. Their experiments indicate the possible appearance of a chemical signal which is being secreted into the water column and works as repellent, growth suppress-

ing, agent. A similar mechanism might be present in *Pocillopora damicornis*. The morphological measurements indicate that in future versions of simulation models of stony corals a regulation mechanism is required in order to obtain a better approximation of the actual growth process.

4.5.5 Conclusions

We have presented a software architecture that allows us to instrument interactive virtual exploration systems with probes to obtain quantitative information based on visual presentations. We have used this system to instrument an existing application with little effort. We have limited our system to simple measurements with probes that consist of one, two or three markers. It is relatively easy to extend this system with probes that consist of more complex markers, enabling more advanced measurements to be performed.

4.6 Summary and conclusions

This chapter presented several interaction techniques to enrich the immersive experience of a virtual environment. As in the real world, interaction with a virtual environment greatly increases a user's awareness. Our primary goal was to develop techniques for the purpose of scientific exploration, but most can be applied to other types of virtual environments as well.

By adopting the Visualization Toolkit (Vtk) as our basis for the construction of virtual environments using SCAVI, an application developer has immediate access to a wide range of computer graphics, image processing and scientific visualization functions. Our direct object manipulation extensions to Vtk allow flexible scientific exploration environments to be built within a short time span. The resulting interaction capabilities come close to those offered by commercially available VE toolkits such as the WorldToolkit and Performer, but it also lacks important features, such as a scene graph system. The basic interaction methods are relatively simple and are all based on the wand's buttons and joystick, but have proven to be effective for the implementation of expressive interactive environments.

XiVE provides a unique capability for VE application designers by allowing existing 2D GUI toolkits and applications to be integrated into virtual environments. Its general design allows a developer to design graphical user interfaces using the same programming techniques as are used for desktop applications and include them in virtual environments with great ease. XiVE has recently been released to a selected number of VE application developers for evaluation purposes. Based on their experiences and comments, improvements will be incorporated after which the code will be released to the public.

The concept of context in a virtual environment has, as of yet, only been applied to improve the accuracy of a speech recognition system. Results have shown that the performance of speech recognition does improve, though not by much. The added value

of speech recognition in virtual environments, however, is unquestionable. Through the use of a different modality than graphical representations (i.e. sound), interaction with a virtual environment remains possible even when low frame rates make interaction through graphical constructs difficult. However, experiences with the current version of our speech recognition system show that variations in pronounciation between different speakers remain a problem.

GEOPROVE provides interaction methods for obtaining quantitative measurements from visual representations in a virtual environment, which is essential for scientific exploration. Its design allows VE applications to be extended with this functionality with little effort. The current implementation of GEOPROVE provides its own graphical user interface to facilitate integration in existing VE applications, but it has been designed in such as way that this can be replaced by other interfaces.