VR-Techniques for Industrial Applications

Gabriel Zachmann
Fraunhofer Institute for Computer Graphics,
Visualization and Virtual Reality Dept.,
64283 Darmstadt, Germany

Virtual Prototyping (VP) is, so far, the most challenging class of applications for virtual reality (VR). A VR system suitable for VP must be able to handle very large geometric complexities which, in general, cannot be reduced significantly by common rendering techniques such as texturing or level-of-detail. Furthermore, VP is the most "interactive" type of applications for VR compared to other areas such as architecture, medicine, or entertainment. This is due to the simulation of rather complex tasks involving many objects whose behavior and properties should be imitated as closely as possible. Finally, in order to sustain continuously a feeling of presence and immersion, the VP system must be able to achieve a frame rate of no less than 15 frames per second at any time.

This chapter will deal with the issue of interaction in complex virtual environments (VEs) for VP. First, we will review briefly the novel I/O devices currently being used in VR, and classify VEs with respect to the devices being used and with respect to the real environment being simulated. Then, several interaction techniques will be discussed, both for VEs in general and for VP in particular. The last two sections give a more technical view on the issue of how to describe VEs and on the overall architecture of VR systems.

1 Characterization of VEs

Virtual environments can be distinguished by their relation to a real environment.

- The VE is actually a *projection* of some real environment. That real environment might be of very different scale [28] or at some distance from the user [5]. The latter is usually described by the term *tele-presence*.
- The VE does *not exist* but is otherwise (fairly) realistic. It might have existed in the past, or it might exist in the future (which is actually the purpose of VP).
- The VE is quite unreal. This is commonly the case in entertainment which strives to provide the participants with an exciting and exotic world.

There are, of course, intermediate or mixed forms. An example of this is "augmented reality"[8]: the user sees his real environment through special glasses which allow the superposition of computer generated images. Thus, the image of the real world can be augmented by pictures, signs, pictograms, hints, instructions, etc. However, interaction is usually restricted to the real environment.

Different VEs (as distinguished above) require different kinds of interaction:

- the user manipulates *existing*, real objects. Examples are: repair of a satellite by means of a vehicle armed with tools; steering of a vehicle where humans cannot enter (e.g., the moon, radioactively polluted terrain, etc.); exploration and modification of the structure of materials at atomic scale [28].
- the user manipulates *non-existent* objects. Examples might be: examination and modification of the interior design of a building which is still under planning; visualization of fabrication processes, such that deficiencies or dangers for human operators can be detected at an early stage; simulation of surgical operations.

Another classification scheme can be based on the amount of *distribution*. The simplest VEs are local, single-user. Distributed VEs might still be single-user, but the application is distributed among several machines or processors. In *multi-user* VEs, several users share the same experience while being at (possibly) remote places.

2 Techniques to achieve immersion

Virtual environments can be classified according to several orthogonal criteria: the real environment being simulated, the amount of *immersion* and *presence* they offer (see Figure 1), and the degree of distribution.

2.1 Immersion

A key feature of VR technology is *immersion*. This term defines the feeling of a VR user, that his virtual environment is real. [1]

A high degree of immersion is equivalent to a realistic or "believable" virtual environment. In the following we will briefly describe several effects which detract from the experience of immersion (ordered by significance), and how they can be avoided:

1. Psychological experiments indiciate that the most important effect is *feedback lag*. There are mainly three factors which contribute to this lag:
    - Rendering time.
    - Tracking systems usually produce delayed data. The more sensors they have to track, the longer the delay. In addition, filtering introduces lag. Attempts have been made to overcome this problem by trying to predict the position/orientation [17].
    - Other computations like collision detection or simulation of physics.

[1] More precisely, we define complete *immersion* analogously to Turing's definition of (artificial) intelligence: if the user cannot tell which reality is "real", and which one is "virtual", then the computer generated one is totally *immersive*.

2. *Narrow field-of-views* is a rather severe shortcoming which makes the user feel as if he looked through a tunnel into the world, especially, if he uses a head mounted display.

   To overcome this, HMDs usually have got some kind of wide-angle optics. Large projection screens can alleviate the "tunnel" effect of monitors; a very large field-of-view is achieved by a surround-screen projection (cave) as developed by [6].

3. A *monoscopic view* deprives the user of ability to estimate distances in the depth range from 20cm to about 5m. A stereoscopic view can be obtained quite easily by simply rendering the same scene twice with slightly shifted viewpoints (stereoscopic rendering). One should use the parallel-lens algorithm to achieve a good stereoscopic effect [18, 1, 27].

   Unfortunately, the conflict between the computer generated disparity in the user's eyes and the eyes' accommodation to the screen surface (or the HMD's virtual display distance) will still remain. From our experience, this is not a problem, although the discrepancy is not realistic.

4. *Low display resolution*, from our experience, is the least significant factor concerning immersion. First generation HMDs based on LCDs have very low resolution which is made worse because of the wide-angle optics. CRT based HMDs offer full resolution, but are heavier.

Since our visual perception is our primary sense, research focuses very much on fast, realistic, high-resolution rendering. However, other senses must also be stimulated to achieve full immersion; among them are audible feedback, tactile, and force feedback.

2.2 Presence

While immersion is an objective measure, presence is the subjective sense of *being in* the VE [26]. Presence requires a self-representation in the VE – a virtual body (often, only the hand is represented in the VE, since the whole body is not tracked). It requires also that the participant identify with the virtual body, that its movements are his/her movements.

3 Devices

Virtual Reality offers a novel human-computer interface by utilizing novel I/O devices. The output devices and their determining characteristics commonly used are:

Display	**Characteristics**
HMD, Boom Cave, Stereo-Projection, Workbench Monitor	Resolution Color / Monochrome Field-of-View Contrast and Brightness Distortion

Very important to most VR applications is the tracking device which measures the head's, hand's, or the complete body's position and orientation. Several technologies and characteristics are:

Tracking technology	Characteristics
magnetic optical mechanical other (ultra-sonic, inertial, gyroscopic)	volume noise and distortion latency accuracy occlusion

Other input devices are:

Device	Characteristics
glove desktop positioning device tetherless positioning device other (buttons, microphone, ...)	dimension accuracy practicality

The amount of *immersion* and *presence* is determined, among others, by the I/O devices being used [11]: the ultimate immersion can be achieved by head-mounted displays, followed by arm-mounted (Boom), several cave variants through screen-based (Workbench or just stereo-projection) systems. Unfortunately, the degree of presence is (almost) reciprocal to the degree of immersion: with a workbench, we do have full presence, but almost no immersion. In a cave, we have optimal presence because the real body is actually *inside* the VE, but we cannot achieve full immersion. With an HMD we have full immersion but almost no presence.

Using output devices for other human senses such as hearing or feeling greatly increases the effect of immersion.

The situation is quite similar on the input side: all input devices so far offer different advantages and disadvantages. While one application might demand a fully immersive VE, another one might be best implemented by stereo-projection and spacemouse.

The result of this observation is that there cannot be a perfect, general input or output device. Instead, the appropriate devices are determined by the application. For assembly simulations it is important to achieve a very high level of immersion in yet to be built "environments". Style and desgin evaluation doesn't necessarily need perfect immersion, but it does need high quality images and a good sense of presence. Entertainment needs a high degree of immersion in unreal environments which will never exist. Augmented reality usually provides a very high level of presence in the existing environment, while there is not much of a virtual environment at all. Figure 1 gives a graphical depiction of our classification of some typical areas of VR applications.

As can be seen from the figure, VP is an area which has a very broad range of requirements: on one hand, there are applications more biased towards functional simulation, and thus requiring a high degree of immersion; on the other hand, there are applications such as ergonomy studies or interior design which require a high degree of presence.

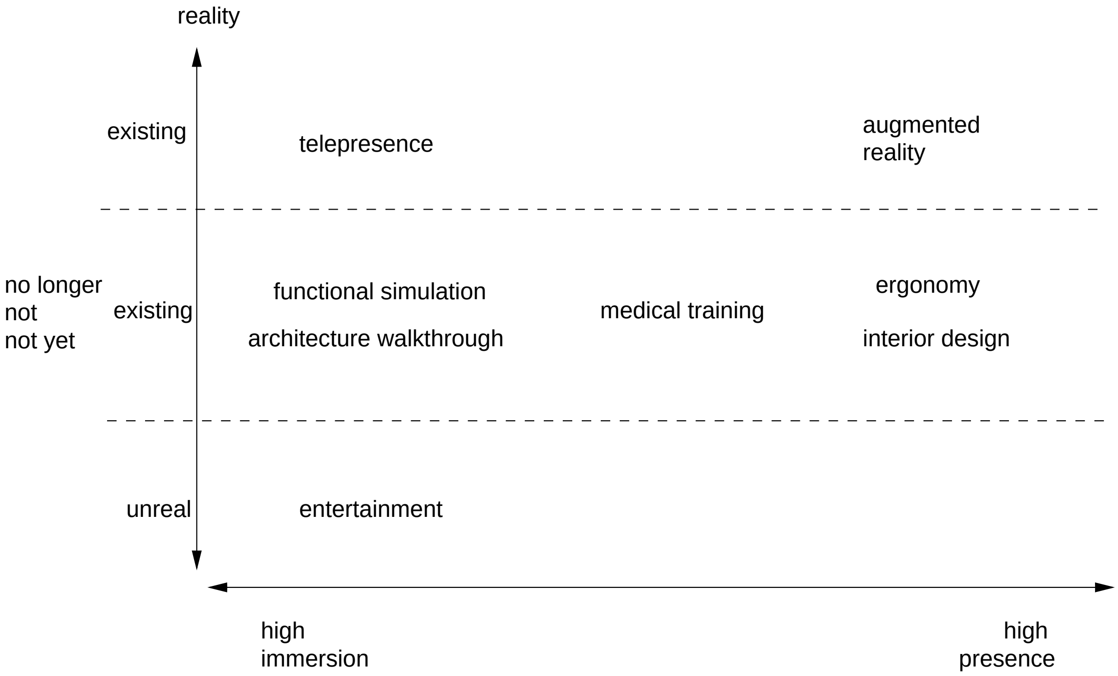

Figure 1: This graph depicts the classification of some typical applications of VR with respect to the "reality" of the environment and with respect to the immersion/presence required.

3.1 Distortion Correction of magnetic fields

Electro-magnetic trackers have become the most wide-spread devices used in today's virtual reality systems. They are used to track the position and orientation of a user's hands and head, or to track instruments such as an endoscope or scissors. They're also being deployed in real-time motion capture systems to track a set of key points and joints of the human body. Commercial optical tracking systems are becoming more mature; however, they're still much more expensive than electro-magnetic systems, but are not yet quite as robust in terms of drop-outs.

Unfortunately, there is one big disadvantage of electro-magnetic trackers: the electro-magnetic field itself, which gets distorted by many kinds of metal. Usually, it is impossible to banish all metal from the sphere of influence of the transmitter emitting the electro-magnetic field, especially when using a long-range transmitter: monitors contain coils, walls, ceiling, and floors of the building contain metal trellises and struts, chairs and tables have metal frames, etc. While tracking systems using direct current seem to be somewhat less susceptible to distortion by metal than alternating current systems, all ferro-magnetic metal will still influence the field generated by the transmitter to some degree.

A distortion of the magnetic field directly results in mismatches between the tracking sensor's true position (and orientation) and the position (orientation) as reported by the tracking system. Depending on the application and the set-up, mismatches between the user's eye position (the *real viewpoint*) and the virtual camera's position (the *virtual viewpoint*) impair more or less the usability of VR. For example, in assembly tasks or serviceability investigations, fine, precise, and

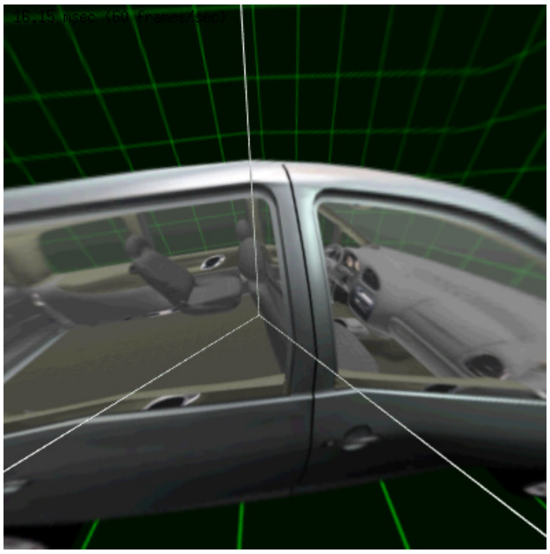

Figure 2: Without correction of the magnetic field tracking, an off-center viewer in a cave will see a distorted image. The effect is even worse when the viewer moves, because objects seem to move, too.

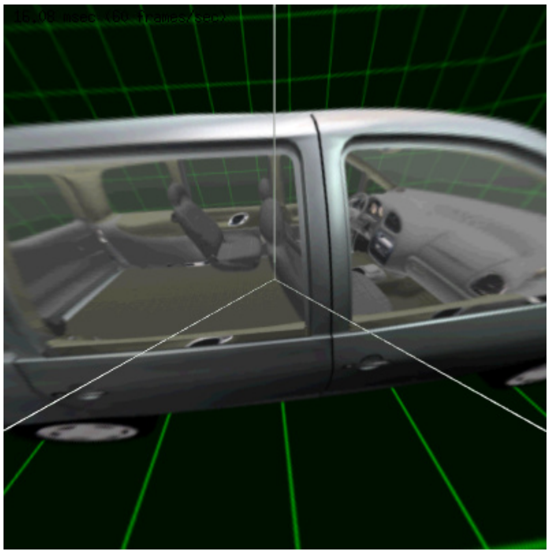

Figure 3: With our method, the perspective is always correct, even when the viewer moves. (See also the color plates.) Data courtesy of Volkswagen AG.

true positioning is very important [7]. In a 2.4m cave or at a workbench, a discrepancy of 7 cm (3 in) between the real viewpoint and the virtual viewpoint leads to noticeable distortions of the image[2], which is, of course, not acceptable to stylists and designers. For instance, straight edges of objects spanning 2 walls appear to have an angle (see Figure 2), and when the viewer goes closer to a wall, objects "behind" the wall seem to recede or approach (see [14] for a description of some other effects). Mismatches are most fatal for Augmented Reality with head-tracking in which virtual objects need to be positioned exactly relative to real objects [22].

We have developed an algorithm which overcomes these adverse effects [34]. It is very fast (1-2 msec), so no additional latency is introduced into the VR system. With our algorithm, the error of the corrected points from their true positions does not depend on the distance to the transmitter, nor does it depend on the amount of local "warp" of the magnetic field.

4 Interaction Techniques

By interaction we mean any actions of the user aiming at modification or probing the virtual environment. In order to achieve a good degree of immersion, it is highly desirable to develop interaction techniques which are as intuitive as possible. Conventional interaction devices (key-

[2]This is just a rule of thumb, of course. The threshold at which a discrepancy between the real and the virtual viewpoint is noticeable depends on many variables: expertise, distance from the cave wall or projection screen, size of the cave, etc.

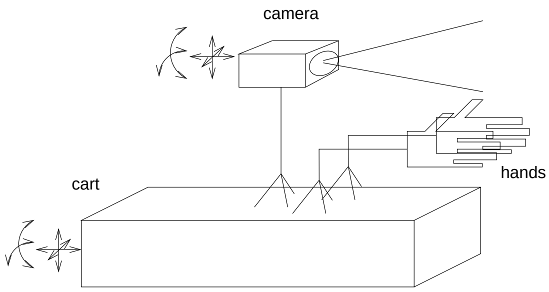

Figure 4: All navigation modes can be deduced from the *flying carpet* model. Not all modes utilize all of the "devices" shown.

Figure 5: With a Boom for interior design, the eyeball-in-hand navigation technique is used.

board, mouse, tablet, etc.) are not fit for natural interaction with most VR applications. One of the more intuitive ways is the "virtual trackball" or the "rolling trackball", which both utilize the mouse[16, p. 51 ff.].

The shortcoming of all of the above mentioned devices is their low number of input dimensions (at most 2). However, new devices like SpaceMouse, DataGlove, tracking systems, Boom, Cricket, etc., provide 6 and more dimensions. These allow highly efficient, natural interaction techniques. Some of them will be described in the following.

4.1 Navigation

By navigation we mean all forms of controlling the viewpoint in a virtual environment, or steering of a real exploration device (e.g., the repair robot or the microscope needle). Navigation is probably the simplest form of interaction, which can be found in all VR applications.

Virtually all navigation techniques can be deduced from a single model, which assumes the virtual camera mounted on a virtual cart, also sometimes referred to as *flying carpet* model (see Fig. 4). See also [32, 25, 9].

- In *point-and-fly* the user moves the cart by pointing in the desired direction with the navigation device (e.g., glove or cricket) and making a certain gesture or pressing a certain button. If a glove is being used, the speed of the motion can be controlled by the flexion value. If head tracking is enabled, the camera will be controlled by the head tracker.

  This navigation technique is the one most widely used.

  [19] have suggested a more sophisticated point-and-fly mode: the user points at the desired object and the VR system computes a "swerved" path which will place the user eventually in front of the object. Again, the speed can be controlled.

Sometimes it is desirable to constrain the cart at a certain height, for example at eye level above the virtual ground, while the user can still move around.

- *eyeball-in-hand*: this paradigm is implemented by feeding the tracking system's output (e.g., the position of an electro-magnetic sensor or a Boom), directly to the viewpoint and the viewing direction, while the cart remains fixed. This technique is most appropriate for close examination of single objects from different viewpoints, e.g., interior design (see Fig. 5).

- *scene-in-hand*: this is the complementary technique to eyeball-in-hand. Sometimes this can be quite useful for orientation or coarse object placement [23].

- Sometimes it is desirable to be able to control the viewpoint "without hands". In that case, *speech recognition* can be used in order to move the cart by uttering simple commands such as "turn left", "stop", etc. This has become feasible with today's user-independent speech recognition systems and fast processors.

In order to attain maximum flexibility, it is highly desirable that all of the above mentioned navigation modes can be mapped to all possible configurations of input devices. While there are certain combinations of navigation mode and input device configuration which will be utilized much more often than others, a general mapping scheme comes in handy at times.

Of course, there are many parameters which affect user representation and navigation: navigation speed, size and offset of the hand, scaling of head motion, eye separation, etc. These have to be adjusted for every VE.

4.2 Gesture recognition

Usually, static gestures like "fist" or "hitch-hike" are used to trigger actions. Gestures can be augmented by taking also the orientation of the glove's tracker into account, i.e., a hitch-hike gesture pointing up is different from a hitch-hike pointing down (for example, to make an elevator go up or down). There is also research going on to recognize dynamic gestures which consist of a continuous sequence of static gestures and tracker positions.

There is a little bit of confusion about terminology here. Sometimes, static gestures are called *postures* and dynamic gestures are just called *gestures*. We will call gestures plus orientation *postures*.

Gestures can be defined as ellipsoids in $\mathbb{R}^d$ (where $d = 10$ or $d = 20$, typically); then, they can be recognized by a simple point-in-ellipsoid test. Of course, other norms can be utilized as well, for example the l^∞-norm is computationally much more efficient.

Another, more robust approach exploits the fact that almost all gestures are located near the "border" of $[0, 256]^{20}$, i.e., their flex values are (almost) maximal or minimal. So, $\mathbb{R}^d$ can be subdivided into certain half-spaces, quarter-spaces, etc.

Other approaches are back-propagation networks (perceptrons) [30] and hidden Markov models [21]. Still, with all algorithms, glove calibration for each user is necessary.

Gestures are very well suited to trigger simple actions, like navigation or display of a menu. However, experience has shown that VR systems should not be over-loaded with gesture driven

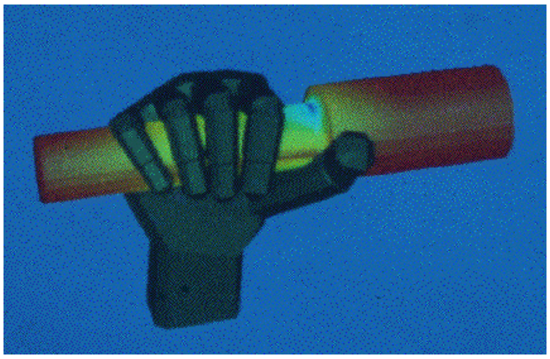

Figure 6: When objects should be grabbed, interaction with virtual environments can actually be more efficient than in the real world.

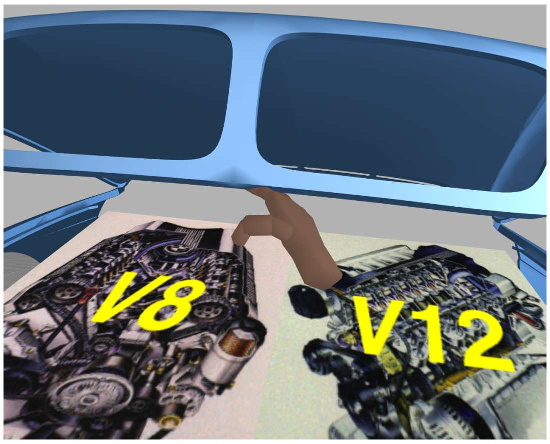

Figure 7: Objects should behave naturally, so interaction with them can be natural. Here, the hood of the car can be opened by just lifting it.

interaction [24]. A casual user will confuse gestures, and the every-day user will find them rather unnatural. In fact, a set of gestures which trigger specific actions can be considered another type of (invisible) menu, which is no more realistic or natural than well-known 2D desktop menus. Other techniques such as speech recognition or natural object behavior should be utilized whenever there is no special reason to use gestures.

4.3 3D menus

Menus provide a 1-of-n choice. 3D menus are a straight-forward extension of the well-known 2D menus [15]. Usually their appearance is triggered by a gesture or a button. Several possibilities exist in order to select a menu item with the pointing device (glove, flying joystick, etc.): shooting a ray through the index finger (if a glove is being used), shooting a ray from the eye through the index finger, or actually touching the 3D button with the finger.

However, we feel that 3D menus should be avoided. They are a relic of 2D desktop interaction and there are usually more efficient ways to interact in 3D. Furthermore, almost all 1-of-n choices can be done much more efficiently by speech recognition.

4.4 Natural object behavior

The goal of this technique is to avoid burdening the user with acquiring any skills other than the ones used in every-day life, i.e., objects in the virtual world should behave just like they do in the real world. However, to achieve such realism, great computational resources are needed, both in terms of CPU power and in terms of clever algorithms.

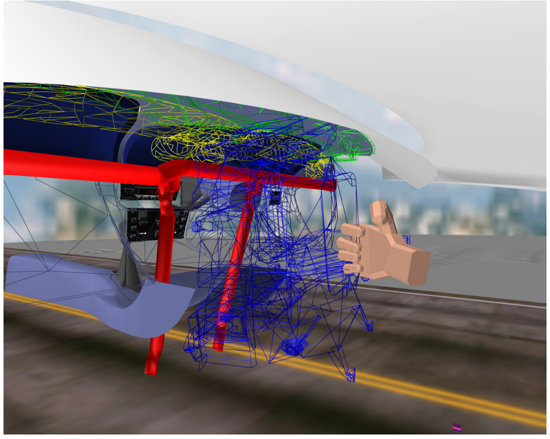

Figure 8: Collision response while moving an object must not interfere with smooth interaction. Here, collision response was chosen to be highlighting of colliding objects by wireframe and via sound feedback.

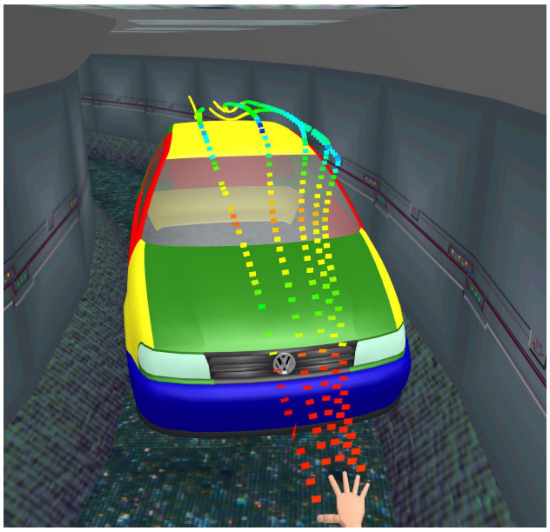

Figure 9: Particle sources can be placed in a virtual environment interactively, while the particle tracing module computes particle paths on-line. So, the simulation of a flow field can be visualized within the environment itself, which is the inside of a car in this case.

Grabbing, pushing, or pulling an object are every-day tasks. As in the real world, a user should be able to do this in the virtual environment by just pushing or pulling it with a virtual hand (see Figures 6, 7).

Although it is highly desirable that objects behave naturally, we feel that the designer of a virtual environment has to decide on a case-by-case basis how close to reality certain interactions should be. For example, for many tasks, it is perfectly sufficient to let the user grab an object by just touching it while making a fist gesture. However, for other scenarios, e.g. design evaluation, the grabbing should be modeled as close to reality as possible.

Another property of realistic object behavior is that they do not penetrate each other. The basis of almost all natural object behavior is real-time *collision detection* for complex objects [12]. However, if force feedback is not available, *collision response* must be modeled carefully to facilitate efficient interaction. For example, it might be very hard to place an object within a dense environment, if colliding objects stick to each other [29]. Instead, a more suitable collision response might be to highlight colliding objects and accompany the collision by some sound feedback (see Fig. 8).

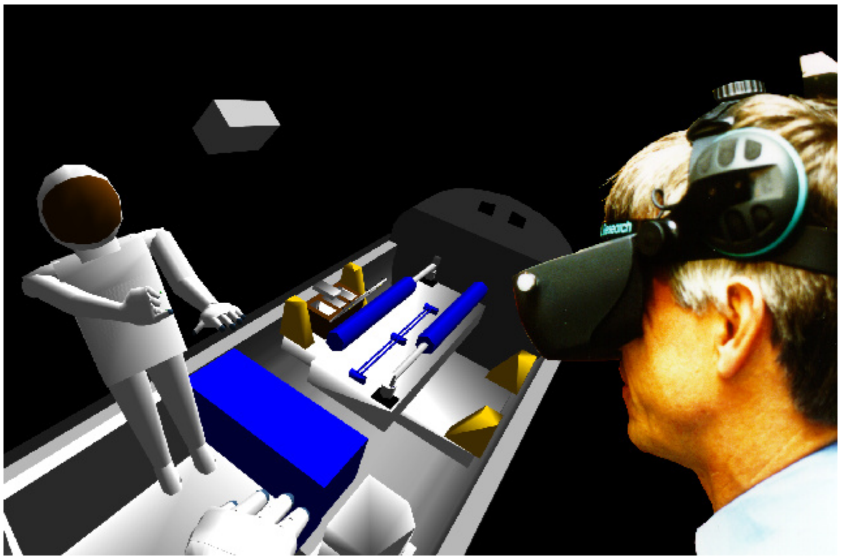

Figure 10: In a shared virtual environment, two astronauts perform a repair task on the Hubble telescope while being at different locations (Germany and Texas). Scenarios like this can be used for training or simultaneous engineering.

4.5 Interaction for scientific visualization

This is a rather new field in the realm of virtual environments. By using 6D (or more) devices, new and potentially much more efficient techniques for interaction with simulation results can be devised.

We have ported several visualization techniques for flow fields to virtual environments. Common techniques are particle tracing, stream lines, and streak lines. For each of them, sources within the field have to be placed. Using interaction techniques for virtual environments, it is very easy to place these: the user just grabs the object which represents the source and places it somewhere else in space. Or, even more efficient, the fingers of the virtual hand itself become sources (see Fig. 9).

4.6 Cooperation in VEs

So far, computer-supported cooperative work (CSCW) does not play a major role in today's product design. As companies operate more and more globally, however, design teams will get more and more distributed. Eventually, there will be a need for CSCW techniques in VP systems, so that several designers or stylists can discuss a new digital prototype while being at remote sites, possibly located on different continents. Therefore, a VR system should be able to allow multiple users to interact with each other and with the same VE (see Figure 10 for an example).

From a classification point of view, it does not make a big difference if there are multiple participants or just one. However, from a technical point of view, it *does* make quite a difference.

When dealing with multiple participants, many new technical issues arise: for instance, the time lag introduced by networks should be neutralized; appropriate representations of the other users should be created.

A great challenge is *floor control*, i.e., mutually exclusive access to objects, in real-time across networks. Intertwined with that is the problem of when access should be mutual exclusive: sometimes participants *do* want to access an object simultaneously (e.g. for repair), sometimes it is not necessary (because the attributes being altered are "orthogonal"). Several methods have been devised to insure consistency:

- *Ownership token passing* [31]. Smooth dead-reckoning is done for local ghosts. The owner of the token of an objects broadcasts updates for that object. When and how the token for an object will be passed is up to the implementation and might be determined by the application.
- *Read/write* permissions à la Unix. Objects can be marked writable for a specific group of users. Every process which changes an attribute of an object and has write permission, broadcasts updates.
- *Distributed locks.* When a process wants to change an object, it will first lock it's local database, then request a distributed lock on that object.

  With this approach it might be difficult to assert responsiveness at all times. In order to avoid delays in the simulation loop, an application could try to look ahead and request a *lock in advance* if it can be foreseen that a user will probably try to execute an action which requires a particular lock.

The methods above can be combined – however, whatever measures we take, we must make sure that the overall system is *responsive* at all times.

5 Description of VEs

Creating virtual worlds is still a cumbersome and tedious process. Below, we describe a framework which facilitates creating virtual environments. VE "authors" should be allowed to experiment and play interactively with their "worlds". Since this requires very low turn-around times, any compilation or re-linking steps should be avoided. Also, authors should not need to learn a full-powered programming language. A very simple, yet powerful script language will be proposed, which meets almost all needs of VE creators [33] (we will not, however, discuss any syntactical issues, since they can be found in the reference).

In order to achieve these goals, we identify a set of basic and generic user-object and object-object interactions which, experience has taught us, are needed in most applications.

For specification of a virtual world, there are, at least, two contrary approaches:

- The *event based* approach is to write a *story-board*, i.e., the creator specifies which action/interaction happens with a certain event.

  A story-driven world usually has several "phases", so we want a certain interaction option to be available only at that stage of the application, and others at another stage.

- The *behavior based* approach is to specify a set of *autonomous objects* or *agents*, which are equipped with receptors and react to certain inputs to those receptors (see, for example, [4]).

  So, overstating a little, we create a bunch of "creatures", throw them into our world, and see what happens.

In the long term, you probably want to be able to use both methods to create a virtual world.

Here, we will focus on the *event based* approach. The language for specifying those worlds will be very simple for several reasons: VE authors "just want to make this and that happen", they don't want to learn Python or C++. Moreover, it is much easier to write a true graphical user interface for a simple language than for a full-powered programming language.

All concepts being developed here have been inspired and driven by concrete demands during recent projects. Most of them have been implemented in an *interaction-module*, which is part of our whole VR system.

5.1 The action-event paradigm

A virtual world is specified by a set of *static* configurations (geometry, module parameters, navigation modes, etc.) and a set of *dynamic* configurations. Dynamic configurations are object properties, user-object interaction, action dependencies, or autonomous behavior.

The basic idea of dynamic configurations is that certain *events* trigger certain *actions*, *properties*, or *behavior*; e.g., when the user touches a virtual button, a light will be switched on, or, when a certain time is reached an object will start to move. Consequently, the basic building blocks of our virtual worlds are *actions*, *events*, and *graphical objects* – the *AEO triad*[3] (see Figure 11).

Unlike other systems [13, 2, 10], our actions are *not* part of an object's attributes (in fact, one action can operate on many objects at the same time).

In order to be most flexible, the action-event paradigm must satisfy the following requirements:

1. Any action can be triggered by any event.

2. Several events can trigger the same action. An event can trigger several actions simultaneously (many-to-many mapping).

3. Events can be combined by boolean expressions.

4. Events can be configured such that they start or stop an action when a certain condition holds for its input (positive/negative edge, etc.)

5. The status of an action can be the input of another event.

We do not need any special constructs (as in [20]) in order to realize *temporal operators*. Parallel execution of several actions can be achieved trivially, since one event can trigger many

[3]In object-oriented programming parlance, actions, events, as well as graphical objects are *objects*. However, in the following we will use the term *object* only for graphical objects.

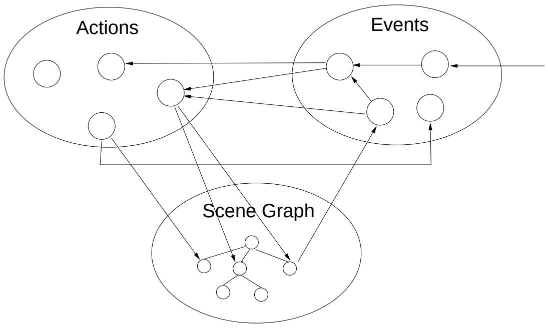

Figure 11: The *AEO triad*. Anything that can "happen" in a virtual environment is represented by an action. Any action can be triggered by one or more events, which will get input from physical devices, the scene graph, or other actions. Note that actions are not "tied-in" with graphical objects, and that events are objects in their own (object-oriented) right.

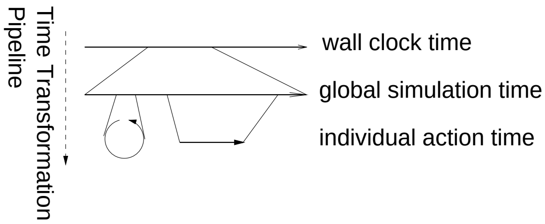

Figure 12: A simulation of virtual environments must maintain several time "variables". Any action can have its own *action time* variable, which is derived from a global *simulation time*, which in turn is derived from wall clock time. There is a small set of actions which allow the simulation to set/change each time transformation individually.

actions. Should those actions be triggered by different events, we can couple them via another event. Sequential execution can be achieved by connecting the two actions by an event which starts the second action when the first one finishes. Similarly, actions can be coupled (start-to-start or start-to-stop) with a delay.

5.2 Time

Many actions (besides navigation, simulation, and visualization) depend on time in some way. For example, an animation or sound sample is to be played back from simulation time t_1 through t_2, no matter how much computation has to be done or how fast rendering is.

We maintain a global *simulation time*, which is derived from wall-clock time. The transformation from wall-clock time to simulation time can be modified via actions (to go to slow-motion, for example, or to do a time "jump").

Furthermore, we keep an unlimited number of time variables. The value of each time variable is derived from the global simulation time by an individual transformation which can be modified by actions as well (see Figure 12).

Those times can be used as inputs to events, or to drive simulations or animations. Thus, time can even be used to create completely "time-coded" parts of a virtual reality show.

5.3 Events

Events are probably the most important part for our world description – they can be considered the "sensory equipment" of the actions and objects. They have the form

event-name: *trigger-behavior input parameters* where *event-name* is for further reference in the script. When an event "triggers" it sends a certain message to the associated action(s), usually "switch on" or "off".

It is important to serve a broad variety of inputs (see below), but also to provide all possible trigger behaviors. A trigger behavior specifies when and how a change on the "input" side actually causes an action to be executed. Let us consider first the simple example of an animation and a keyboard button:

> *animation on as long as button is down,*
> *animation switch on whenever button is pressed down,*
> *animation switch on whenever button is released,*
> *animation change status whenever button is pressed down,*

These are just a few possibilities of input→action trigger-behavior. The complete syntax of trigger behaviors can be found in [33].

It would be possible to have the world builder "program" the trigger-behavior by using a (quite simple) finite state machine (as in dVS, for instance [10]). However, we feel that this would be too cumbersome, since those trigger behaviors are needed very frequently.

In addition to the basic events, events can be combined by logical expressions. This yields a directed "event graph". This graph is not necessarily acyclic.

Our experience shows that it is necessary to be able to activate and deactivate actions. This is needed, for example, to enable unmounting of a part only after some screws have been removed. This is done via a certain action, which (de-)activates other actions.

A Collection of Event Inputs

Physical input includes all kinds of buttons (keyboard, mouse, spacemouse, boom), flex and tracker values, gestures, postures (gesture plus orientation of the hand), voice input (keyword spotting, enhanced by a simple regular grammar, which can tolerate a certain (user-specified) amount of "noise" words).

Geometric events are triggered by some geometric condition. Among them are virtual buttons, virtual menus, portals, and collisions.

Any action's status (*on* or *off*) can trigger an event. Some actions have an action-specific status, which can be used also.

All time variables (see above) can be the input of an event. This allows for one-shot or cyclical triggering of actions.

Sometimes we need to "monitor" certain object attributes and issue an action when they change, while we don't care which action (or even other module) changed them. Attributes are not only graphical attributes (transformation, material, wireframe, etc.), but also "interaction" attributes, such as "grabbed", "falling", "constrained", etc. Object attributes might be set by our interaction module itself (by possibly many different actions), or by other modules.

5.4 Actions

Actions are the building blocks for "life" in a virtual environment. Anything that happens, as well as any object properties, are specified through actions. The set of actions should be very generic, but not too low level. If they are too low level, then building VE with them is probably no more efficient that using a programming language. If they are too high level, then they are probably too specialized and cannot be used in a broad variety of applications.

Actions are usually of the form

action-name **:** *function objects parameters options*

All actions should be made as general as possible, so it should always be possible to specify a *list of objects* (instead of only one). Also, objects can have any type, whenever sensible (e.g., assembly, geometry, light, or viewpoint node). The *action-name* is for later reference in the script.

There are several cases where inconsistency has to be dealt with in a VR system. One such case arises when several actions transform the same object. For example, we can grab an object with our left hand while we stretch and shrink it with the other hand. The problem also arises, when an action takes over (e.g., we scale an object after we have grabbed and moved it). However, by implementing a standardized way of object transformation, this problem can be solved.

Another inconsistency arises when we use levels-of-detail (LODs) in the scene graph. Since any object can be a LOD node or a level of a LOD, any action should transparently apply changes to all levels, so that the author of the virtual world doesn't have to bother or know whether or not an object name denotes an LOD node (or one of its children).

A Collection of Actions.

During our past projects, the set of actions listed below briefly has proven to be quite generic.

The scene graph can be changed by the actions load, save, delete, copy, create (box, ellipsoid, etc.), and attach (changes scene hierarchy by rearranging subtrees).

Some actions to change object attributes are switch, wireframe, rotate, translate, scale (set a transformation or add/multiply to it). Others change material attributes, such as color, transparency, or texture.

The "grab" action first makes an object "grabbable". Then, as soon as the hand touches it, it will be attached to the hand. Of course, this action allows grabbing a list of sub-trees of the scene graph (e.g., move a table when you grab its leg).

With the "stretch" action we can scale an object (or sub-tree).

A great deal of "life" in a virtual world can be created by all kinds of animations of attributes. Our animation actions include playback of transformations, visibility, transparency (for fading), and color from a file. The file format is flexible so that several objects and/or several attributes can be animated simultaneously. Animations can be time-tagged, or just be played back with a certain, possibly non-integer, speed. Animations can be *absolute* or *relative* which just *adds* to the current attribute(s). This allows, for example, simple autonomous object locomotion which is independent of the current position.

As described above, there are actions to set or change the time transformation for the time variables.

Occasionally we want to constrain the movement of an object. It is important to be able to switch constraints on and off at any time, which can be done by a class of constraint actions. Several constraints on transformations of objects, including the viewpoint, have proven useful:

1. Constrain the translation in several ways:
   (a) fix one or more coordinates to a pre-defined or the current value,
   (b) keep the distance to other objects (e.g., ground) to a pre-defined or the current value. The distance is evaluated based on a direction which can be specified.

   This can be used to fix the user to eye level, for terrain following, or to make the user ride another object (an elevator, for example).

2. Constrain the orientation to a certain axis and possibly the rotation angle to a certain range. This can be used to create doors and car hoods.

All constraints can be expressed either in world or in local coordinates. Also, all constraints can be imposed as an *interval* (a door can rotate about its hinge only within a certain interval). Interaction with those objects can be made more convenient if the deltas of the constrained variable(s) are restricted to only increasing or decreasing values (e.g., the car hood can only be opened but not closed).

Another constraint is the notion of *walls*, which is a list of objects that cannot be penetrated by certain other objects. This is very useful to constrain the viewpoint or to make some objects rigid and solid.

One of the most basic physical concepts is gravity, which increases "believability" of our worlds tremendously. It has been implemented in an object property "fall", which makes objects fall in a certain direction and bounce off "floor objects", which can be specified separately for each falling object.

Object selection. There must be two possibilities for specifying lists of objects: *hard-wired* and *user-selected*.

In entertainment applications, you probably want to specify by name the objects on which an action operates. The advantage here is that the process of interacting with the world is "single-pass". The downside is inflexibility, and the writing of the interaction script might be more cumbersome.

Alternatively, we can specify that an action should operate on the currently selected list of objects. This is more flexible, but the actual interaction with the world consists of two passes: first the user has to select some objects, then specify the operation.

User modules. From our experience, most applications will need some specialized features which will be unnecessary in other applications. In order to integrate these smoothly, our VR system offers "callback" actions. They can called right after the system is initialized, or once per frame (the "loop" function), or triggered by an event. The return code of these callbacks can be fed into other events, so user-provided actions can trigger other actions.

These user-provided modules are linked dynamically at run-time, which significantly reduces turn-around time.

It is understood that all functionality of the script as well as all data structures must also be accessible to such a module via a simple, yet complete API.

5.5 Examples

The following example shows how the point-and-fly navigation mode can be specified.

```
cart pointfly dir fastrak 1 \
    speed joint thumbouter \
    trigger gesture pointfly
cartrev gesture pointflyback
cart speed range 0 0.8
glove fastrak 1
```

The hood of a car can be modeled by the following lines. This hood can be opened by just pushing it with the index finger.

```
constraint rot Hood neg \
   track IndexFinger3 \
   axis a b to c d \
   low -45 high 0 \
   on when active collision Finger13 Hood
```

The following is an example of a library "function" to make clocks. (This assumes that the hands of the clock turn in the local xz-plane.)

```
define CLOCK( LHAND, BHAND )
timer LHAND cycle 60
timer speed LHAND 1
/* rotate little hand every minute by 6 degrees in local space
*/
objattr LHAND rot add local 6 (0 1 0) time LHAND 60
/* rotate big hand every minute by 0.5 degrees in local space
*/
objattr BHAND rot add local 0.5 (0 1 0) time LHAND 60 /* define
start/stop actions */
Stop##LHAND : timer speed LHAND 0
Start##LHAND : timer speed LHAND 1
```

The `##` is a concatenation feature of `acpp`. By applying the definition `CLOCK` to a suitable object, we make it behave as a clock. Also, we can start or stop that clock by the actions

```
CLOCK( LittleHand, BigHand )
action "StartLittleHand" when activated speech "clock on"
action "StopLittleHand" when activated speech "clock off"
```

6 Architecture

So far, we have considered only a few of many modules a complete VR system comprises. For sake of completeness, we will briefly give an overview of the architecture of a VR system [3].

Figure 13 shows a hierarchy of modules: at the bottom there are service modules which are primarily concerned with I/O, such as rendering, polling of input devices, and audio feedback. In the middle level there are service modules implementing functionality which is vital

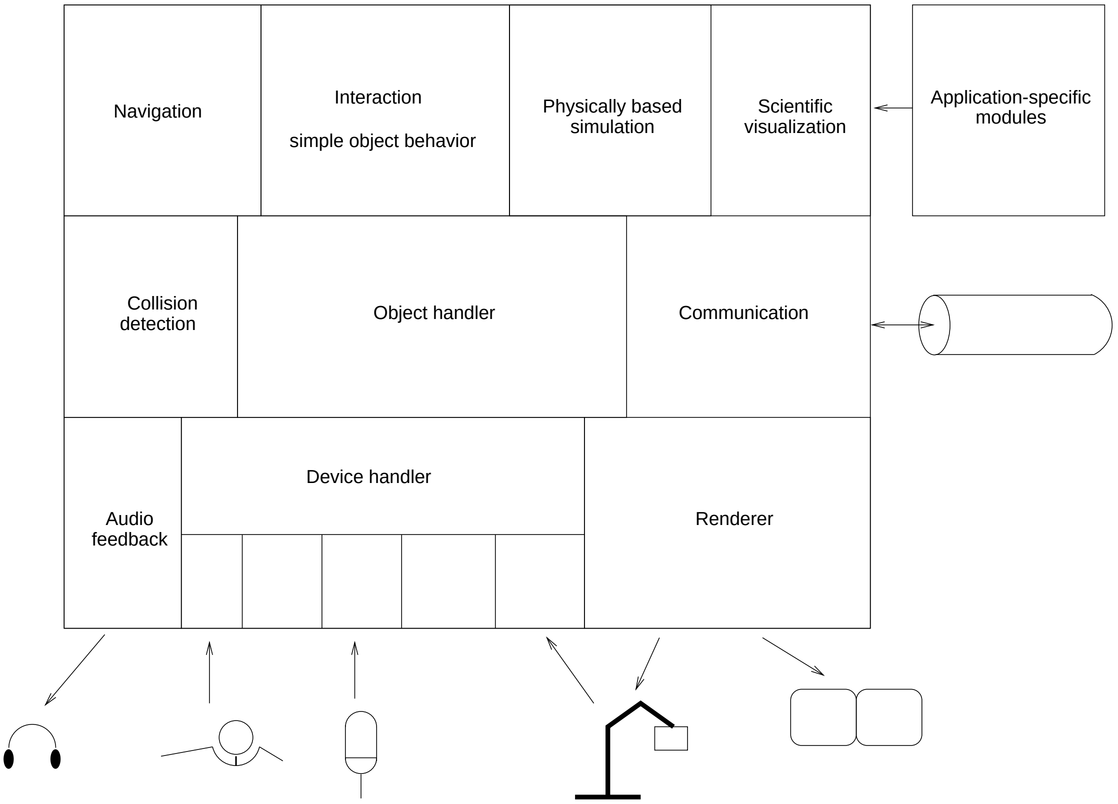

Figure 13: Architecture of a VR system. At the center is the object handler which maintains the scene graph. All modules must be able to run concurrently and asynchronously to each other, in order to achieve a constantly high frame rate.

to many higher-level modules, such as collision detection and communication with other VR systems. The object handler is a supremely important module, because it maintains the scene graph. While most high-level modules keep their own module-specific, logical representations of objects, the scene graph is still *the* common basis to all of them. At the top level reside modules implementing functionality for interaction and simulation, such as navigation, object manipulation, physical behavior, and others.

The device handler implements a device abstraction by the notion of logical devices [9], a concept well known from GKS or PHIGS. It has proven quite practical to implement each device server so that it can be run on a remote machine and communicate with the device handler via a socket or other inter-process mechanism.

As stated above, virtually any non-trivial application needs some specialized functionality which will probably not be needed by any other application. This is depicted in the figure by the "application specific module", which is not a statically linked part of the VR system, but instead loaded by the VR system at run-time on demand. These modules still have full access to all data structures of the VR system.

A VR system must be able to sustain constantly a high frame rate (20 frames/sec or higher) under all circumstances, in order to achieve immersion and efficiency. Even more important is low *lag*, i.e., the latency between a change of the input data and a change in the output images must be minimal at all times. Therefore, modules must not block or delay the flow of data from the input devices to the rendered images. In order to achieve that, most modules depicted in Figure 13 run concurrently and asynchronously.

References

[1] R. Akka. Automatic software control of display parameters for stereoscopic graphics images. Unpublished results, 1992.

[2] M. Andersson, C. Carlsson, O. Hagsand, and O. Ståhl. *DIVE — The Distributed Interactive Virtual Environment*. Swedish Institute of Computer Science, 164 28 Kista, Sweden, 1994.

[3] P. Astheimer, F. Dai, W. Felger, M. Göbel, H. Haase, S. Müller, and R. Ziegler. Virtual Design II – an advanced VR system for industrial applications. In *Proc. Virtual Reality World '95*, pages 337–363, Feb. 1995.

[4] B. M. Blumberg and T. A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In R. Cook, editor, *Siggraph 1995 Conference Proc.*, pages 47–54, Aug. 1995.

[5] G. M. H. Clifford A. Shaffer. A real-time robot arm collision avoidance system. *IEEE Transactions on Robotics and Automation*, 8(2), April 1992.

[6] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the CAVE. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 135–142, Aug. 1993.

[7] F. Dai, W. Felger, T. Frühauf, M. Göbel, D. Reiners, and G. Zachmann. Virtual prototyping examples for automotive industries. In *Proc. Virtual Reality World*, Stuttgart, Feb. 1996.

[8] S. Feiner, B. MacIntyre, and D. Seligmann. Knowledge-based augmented reality. *Communications of the ACM*, 36(7):53–62, July 1993.

[9] W. Felger, T. Fröhlich, and D. M. Göbel. Techniken zur Navigation durch Virtuelle Welten. In *Virtual Reality '93, Anwendungen und Trends*, pages 209–222. Fraunhofer-IPA, -IAO, Springer, Feb. 1993.

[10] S. Ghee. dVS – a distributed VR systems infrastructure. In A. Lastra and H. Fuchs, editors, *Course Notes: Programming Virtual Worlds, SIGGRAPH '95*, pages 6–1 – 6–30, 1995.

[11] S. Ghee, M. Mine, R. Pausch, and K. Pimentel. *Course Notes: Programming Virtual Worlds, SIGGRAPH '95*. 1995.

[12] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In H. Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 171–180. ACM SIGGRAPH, Addison Wesley, Aug. 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[13] S. Halliday and M. Green. A geometric modeling and animation system for virtual reality. In G. Singh, S. Feiner, and D. Thalmann, editors, *Virtual Reality Software and Technology (VRST 94)*, pages 71–84, Aug. 1994.

[14] L. F. Hodges and E. T. Davis. Geometric considerations for stereoscopic virtual environments. *Presence*, 2(1):34–43, winter 1993.

[15] R. H. Jacoby and S. R. Ellis. Using virtual menus in a virtual environment. *ACM Computer Graphics*, Siggraph '92, Course Notes(9), 1992.

[16] D. Kirk, editor. *Graphics Gems III*. Academic Press, Inc., San Diego, CA, 1992.

[17] J. Liang. Smoothing and prediction of isotrak data for virtual reality systems. In *Proceedings of the 1991 Western Computer Graphics Symposium*, pages 58–60, Apr. 1991.

[18] L. Lipton. *The CrystalEyes Handbook*. StereoGraphics Corp.

[19] J. D. Mackinlay, S. K. Card, and G. G. Robertson. Rapid controlled movement through a virtual 3D workspace. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 171–176, Aug. 1990.

[20] R. Maiocchi and B. Pernici. Directing an animated scene with autonomous actors. *The Visual Computer*, 6(6):359–371, Dec. 1990.

[21] Y. Nam and K. Y. Wohn. Recognition of space-time hand-gestures using hidden Markov models. In *Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST '94)*, Hong Kong, July1-4 1996.

[22] T. Oishi and S. Yachi. Methods to calibrate projection transformation parameters for see-through head-mounted displays. *Presence*, pages 122–135, winter 1996.

[23] R. Pausch, T. Burnette, D. Brockway, and M. E. Weiblen. Navigation and locomotion in virtual worlds via flight into Hand-Held miniatures. In R. Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 399–400. ACM SIGGRAPH, Addison Wesley, Aug. 1995. held in Los Angeles, California, 06-11 August 1995.

[24] O. Riedel and G. Herrmann. Virusi: Virtual user interface – iconorientierte benutzerschnittstelle für vr-applikationen. In *Virtual Reality '93, Anwendungen und Trends*, pages 227–24. Fraunhofer-IPA, -IAO, Springer, Feb. 1993.

[25] W. Robinett and R. Holloway. Implementation of flying, scaling, and grabbing in virtual worlds. In D. Zeltzer, editor, *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, volume 25, pages 189–192, Mar. 1992.

[26] M. Slatre, M. Usoh, and Y. Chrysanthou. *Virtual Environments '95. Selected papers of the Eurographics Workshop in Barcelona, Spain 1993, and Monte Carlo, Monaco 1995.*, pages 8–21. Springer, Wien, New York.

[27] D. A. Southard. Transformations for stereoscopic visual simulation. *Computers & Graphics*, 16(4):401–410, 1992.

[28] R. M. Taylor, II, W. Robinett, V. L. Chi, F. P. Brooks, Jr., W. V. Wright, R. S. Williams, and E. J. Snyder. The Nanomanipulator: A virtual reality interface for a scanning tunnelling microscope. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 127–134, Aug. 1993.

[29] G. Turk. Interactive collision detection for molecular graphics. Master's thesis, University of North Carolina at Chapel Hill, 1989.

[30] K. Väänänen and K. Böhm. Gesture driven interaction as a human factor in virtual environments. In *Proc. Virtual Reality Systems*. University of London, May 1992.

[31] Q. Wang, M. Green, and C. Shaw. EM – an environment manager for building networked virtual environments. In *Proc. IEEE Virtual Reality Annual International Symposium*, 1995.

[32] C. Ware and S. Osborne. Exploration and virtual camera control in virtual three dimensional environments. In R. Riesenfeld and C. Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 175–183, Mar. 1990.

[33] G. Zachmann. A language for describing behavior of and interaction with virtual worlds. In *Proc. ACM Conf. VRST '96*, July 1996.

[34] G. Zachmann. Distortion correction of magnetic fields for position tracking. In *Proc. Computer Graphics International (CGI '97)*, Hasselt/Diepenbeek, Belgium, June 1997. IEEE Computer Society Press.