Constructing a Highly Immersive Virtual Environment: A Case Study

Chris Faisstnauer, Dieter Schmalstieg, Tomasz Mazuryk

Institute of Computer Graphics, Vienna University of Technology, Austria

email: faisst | schmalstieg | mazuryk @cg.tuwien.ac.at

Abstract. Virtual reality (VR) applications raise enormous interest inside and outside computer science. Unfortunately, VR systems are rather complex, involving many software and hardware modules being integrated. Theoretical papers are not always of much aid in the actual implementation of VR applications. We try to fill this gap with a case study on a simple example VR system, that is used to demonstrate the most important aspects of a VR implementation, including application design, implementation strategy, selection of hardware and software, rendering, tracking and display technology. Special attention is paid to practical issues that are usually only learned by experience, and on the discussion of devices and methods that are inexpensive and readily available.

Keywords: virtual reality, virtual environment, immersion, case study

1. Introduction

The purpose of this paper is to explain the steps necessary to build a virtual environment (VE) with moderate hard- and software costs and restricted complexity. Our intention is to provide insights into practical issues of virtual environments that can help research groups, students or software companies involved in the implementation of virtual reality (VR) software projects. We concentrate on the translation of theoretical foundations into working solutions, and we report concrete experiences we had when doing an experimental implementation. As a study object, we developed a simple game placing the user in a maze, with the task of fighting computer controlled drones.

A virtual environment is a software system that creates the illusion of a world that does not exist in reality. This is done by presenting a three-dimensional representation of the environment to the user, who can navigate through the virtual world and interact with its inhabiting objects. Such an application requires a combination of user interaction (input), simulation of processes in the environment (computation), and multi-sensory stimulus (output). The main advantage of virtual reality over conventional computer graphics is the feeling of immersion created by convincing real-time stimulus presented to the user. Immersion, however, is also most difficult to achieve and sustain: It is a very demanding task to configure a system so that the diverse needs of the application (management of input, simulation, display) can satisfied with adequate performance to keep the illusion from breaking.

Covering every aspect of virtual environments is beyond the scope of this paper. We are limiting ourselves to a set of topics that as a whole allow interesting applications to be constructed. Our focus is on providing high-fidelity 3-D graphical output and interaction for a single user in a moderately complex surrounding. Related work includes an overview of software needs [1], several case studies (e.g., [2, 3]), and a tutorial for implementing multi-user simulations [4].

2. What you have to do

Getting started is always the hardest task. This section tries to give a very rough "cooking recipe" on how to make the important decisions when undertaking such a complex venture. Unfortunately, there is no single method for software engineering, and there certainly is none for virtual environment engineering. However, one should at least try to avoid some common misconceptions.

2.1 Characterize the problem and the application you want to construct

Creating working solutions always requires a solid grasp on the problem. When using novel technology, especially if it allows a lot of freedom in design (what peripherals to use etc.), it may be tempting to pick up one's favorite toys and then retrofit the application to be compatible with the available environment. Instead, a clear description of the objective is necessary, so that the design of the system can be derived from it.

2.2 Decide on the primary metaphor

How is the user represented in the VE? What must he achieve, and how can he achieve that? Once this decision is made, it should be easy to determine the required level of immersion. There is a long-lasting and ongoing battle whether total immersion (complete coverage of all or most of the user's senses) is necessary at all cost. A practical point of view may be to see it as a cost-benefit equation. If the wins in overall quality are not huge, it may not be worth the extra effort. Sometimes it may turn out that a solution using conventional desktop graphics is superior. Not every problem is suited to be tackled in 3-D and real-time.

2.3 Select the appropriate hardware

The decision should be based on the decision made in the previous step. See if you can afford it. If not, try to analyze your decision: Can you come up with an alternate, still working design within the available budget (maybe at a tolerable decrease in expected quality), or is your solution so dependent on the chosen hardware/software that the project cannot be realized at all with the budget you have at disposal? While such a consideration is true for any technical project, it is particularly crucial for virtual reality where cost literally explodes when high end components are desired.

Alternatively, if the budget is a priory fixed, or if the hardware is fixed (for example, when developing for a particular platform), try to come up with the best application design. It is not necessary to use all or even any of the gadgets that are available to you: Your application will not necessarily be better of you are trying to force-fit your problem with a design that makes use of

VR technology but is not intuitive.

2.4 Decide on the implementation strategy

Once the major design issues have been resolved, it is necessary to find an implementation strategy. Oftentimes it becomes necessary to decide on the software support that is needed. Virtual environments can be very complex software systems: How deep can you afford to dig into the computer? On the one hand, it is necessary to fine-tune almost any part of the VR application, so that the high performance demands of an immersive application are met. On the other hand, the complex structure of a VE does not allow to be concerned with too many details. The dilemma can partly be resolved by the use of toolkits that support a well-defined aspect of the VE, and are highly optimized for the task. You should always try to find a toolkit for creating the 3-D images (rendering toolkit) and for supporting you devices (device drivers in the broadest sense). Unfortunately, you may need slightly different implementations in some cases, and it is not always possible to extend, modify or patch the commercial toolkit. The highest level of support comes from integrated software solutions (e.g., WorldBuilder from Autodesk). If you are able to use such a product, you can save a lot of effort. However, the set of features supported by such a closed solution is fixed, and it may often fail to support your most innovative design ideas.

2.5 Get the hardware working

It may sound trivial in the age of "plug-and-play", but devices such as HMDs and trackers are still complicated to handle. This is partly due to the limited distribution of VR devices and a lack of standards, and will certainly improve over time, but for now a tedious process of trial-and error can hardly be avoided. It may be necessary to create custom device drivers that have specific properties, or run under specific software configurations or flavors of operating systems. Even if you can use the drivers supplied by the vendor, it is also generally necessary to fine-tune the parameters that can be set for the device (sampling frequency, sensibility etc.).

2.6 Do the implementation

Once you have a demonstration running that proves that your choice of hardware runs as a more or less harmonic ensemble, you can start the actual implementation. During the process you will probably learn that virtual environment system design is less well-understood than for example database design. Usually a lot of iterations with real user testing is necessary to get things right. It may be wise if you find a way to throw together prototypes or even barely working mock-ups of your application just to get enough response [5].

2.7 Iterate your design until satisfying

Probably the most important difference between "normal" software engineering and the construction of immersive virtual environments is the lack of a general theory, many factors have to be determined by experiment.

3. Some background on virtual environments

In this section we will discuss issues that are usually raised when designing an virtual environment or planning its implementation. We have selected several issues that seem interesting enough to justify a closer look.

3.1 Tracking technology

Immersive VR applications often require the tracking of the user's head movements to determine his current direction of gaze. Additionally other parts of the body (e.g., hands) may be tracked to allow interaction. The most important properties of trackers [6, 7] are **update rate** (defining how many measurements per second are made), **latency** (amount of time between the user's real action and the beginning of sending of the report representing this action), **accuracy (**measure of error in the reported position and orientation), **resolution** (smallest measurable change in position and orientation) and **range** (working volume of the tracker). Beside these properties, some other aspects cannot be forgotten, like the ease of use, size and weight etc. of the device.

Magnetic trackers are the most often used in VR immersive applications. They typically consist of a stationary part (emitting the electromagnetic signal) and a number of movable parts (called sensors or receivers) mounted on the tracking points. Both emitter and receiver consist of three mutually perpendicular coils. The magnetic fields generated by the emitter are picked up by the receiver and transformed into magnetic current by induction. Both AC and DC current is in use, where DC-based trackers are less sensitive to interference by metal and magnetic fields in the tracking area.

Advantages of magnetic trackers include small and light sensors, no open-line-of-sight constraint, no sensitivity to acoustic and lighting conditions, relatively high update rates and low latency. The most severe disadvantages are vulnerability to metallic objects and ferromagnetic materials and rapid deterioration of accuracy with increasing distance between sensor and emitter caused by electromagnetic distortion.

Acoustic trackers use ultrasonic waves (above 20kHz) for determining the position and orientation of objects in space. As the use of sound allows the determination of relative distance between two points only, multiple emitters (typically 3) and multiple receivers (typically 3) with known geometry are used to acquire a set of distances to calculate position and orientation from the time the sound takes to travel. Advantages of acoustic trackers are small size and weight, inexpensiveness, and independence from magnetic interference. However, they suffer from an open-line-of-sight restriction, vulnerability to acoustic interference (echoes), and rather low update rates.

Other tracking technologies include several flavors of optical tracking and mechanical tracking, but magnetic and acoustic tracking have a major non-technical advantage over these methods: They are commercially available off-the-shelf components with some maturity. Although this is a rather trivial issue, it currently limits your choices to magnetic vs. acoustic: If you can manage to avoid electromagnetic disturbance in your area of work (get a wooden desk!), you will probably opt for a magnetic tracker such as the Polhemus Fastrak or Insidetrak, or the Ascension Flock of Birds. If you can live with the line-of-sight restriction, an acoustic tracker may be appropriate, e.g., the Logitech tracker. Pay attention on how to connect these peripherals to your system: usually a serial or parallel connection or PC plug-in board are used.

3.2 Coordinate systems and their relations

The rendering transformation for computer graphics usually involves object coordinates, world coordinates, and camera coordinates. We need **object-to-world** and **world-to-eye** matrices. However, most rendering systems (e.g., Iris Performer) require the specification of the object-to-world and eye-to-world matrices. For a head-tracked HMD, the latter can be decomposed as follows [8]:

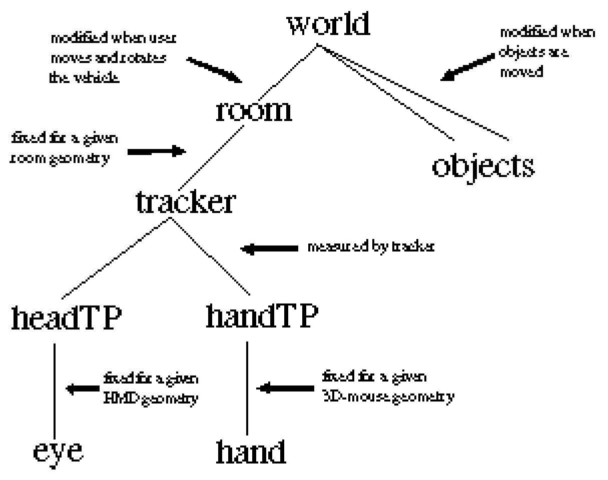
* **Eye-to-head**: defines the position and orientation of the eye in the coordinates of the tracker reference point (headTP in fig. 1) on the head (typically the sensor). The two eye-to-sensor transformations (one for each eye) are **fixed** for a given HMD geometry.

* **Head-to-tracker**: defines the position and orientation of the head in the coordinates of the tracker's stationary part (typically the emitter). This transformation changes **dynamically** as the user walks or rotates his head, and is measured by the tracking device.

* **Tracker-to-room**: defines position and orientation of the tracking system's emitter module in the physical room. In that way independence of the position of the tracker in the physical room is achieved. This transformation is **fixed** for a given physical tracking system configuration.

* **Room-to-world**: defines position and orientation of the (physical) room in (virtual) world coordinates. This is necessary because the simulation places the user in a simulated vehicle that moves in the virtual world. This transformation changes **dynamically** according to the users' actions like flying, tilting or world scaling.

If we use other trackers than for the head (e.g., hand tracking), a corresponding matrix hierarchy is used. The only difference is that a hand-to-world matrix is used like an object-to-world matrix (e.g., for display of the user's hand). Fig. 1 shows the hierarchy of transformations.


world
modified when user moves and rotates the vehicle
modified when objects are moved
room
fixed for a given room geometry
objects
tracker
measured by tracker
headTP
handTP
fixed for a given HMD geometry
fixed for a given 3D-mouse geometry
eye
hand

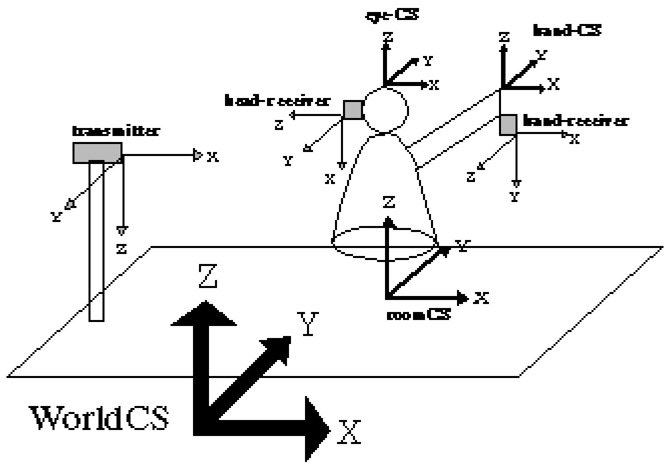


Fig. 1: Abstract and schematic view of the display transformation for VR

3.3 Display technology

The chosen display technology [9] is of crucial importance when trying to design an immersive experience. At a budget, the choice is rather limited; however, careful considerations must be paid to the human factors, because the quality of the visual presentation largely determines how successful your implementation is going to be. In the following we outline what we think are the options one generally has at a reasonable cost (i.e. the peripherals may not exceed the price of the core system).

Fish-tank VR. A large (19" and more) color monitor with high-quality can make partly immersive experiences possible without any extra costs for peripherals. However, you must be aware of the limitations: At no time is the user totally immersed in the scene, for he may always be distracted by what is going on in his immediate surrounding. Interactions requiring even a limited form of body movement are not possible. Head tracking has only limited effect. Despite the obvious limitations, the feeling of immersion can still be surprisingly high.

The monitor-only solution may be greatly enhanced by the use of liquid crystal shutter glasses allowing sequential stereoscopic view - corresponding images for the left and right eye are presented in sequential frames on the monitor in synchronization with the glasses, that use liquid crystal technology to darken the other eye in turn. The action radius is still limited by the user having to look at the monitor, but the monitor provides very good image quality with high resolution and brightness. The glasses are not more uncomfortable than normal sunglasses. Therefore they may be worn for an extensive period of time, which is beneficial if the application is to be integrated into a normal desktop workplace environment (e.g., a CAD seat). Shutter glasses are also cheaper than HMDs.

Head-mounted display. Wearing the display on one’s head brings total immersion: The user’s visual perception is bound to the images presented by the computer. The intention is to focus the user on the VE, so that he can interact with the VE just like he would with reality. However, the quality of today’s commercial products is still relatively poor. Furthermore total immersion requires protection of the user’s action volume, because he cannot see real obstacles anymore and may hurt himself. The options for HMDs include both LCD and CRT based devices. Most devices accept separate video input for each eye, usually they accept NTSC or PAL signals. Conversion from RGB signals as provided by the graphics boards is usually done by an external conversion box. Stereoscopy can either be neglected (in that case the same input is presented to both eyes), or two input signals are necessary. These can be provided by two separate image generators (either two graphics boards in one computer, or two computers that are synchronized over a network). Two signals can also be generated by special graphics boards (like the SGI multi-channel option), but this solution is often prohibitively expensive. HMDs normally include a head-mounted tracker, so that the user is always presented with an image according to his current viewing frustum.

3.4 Efficient modeling and rendering of virtual environments

Modeling toolkits. A complex system such as a VE will also require ways of integrating the geometric description of objects with their programmed behavior in a structured, standardized way: a 3-D modeling toolkit is required. Such toolkits are best developed using an object-oriented or pseudo object-oriented approach, where each kind of object (polygon, light source etc.) is modeled as a class. Instances of these classes are arranged in a hierarchical scene graph as needed (primitives as leaves, transformation nodes etc. as intermediates). Instancing of subgraphs allows the model to exploit similarities in the scene (e.g., the geometry for four identical wheels of a car need only be specified once). Such a scene graph is processed by traversal, and for each visited node the appropriate method of the corresponding class (e.g., draw) is called. Flexibility is achieved by allowing user defined traversal strategies, callback upon traversal of specific nodes, and subclassing of node types. Hierarchical bounding volumes allow efficient culling and intersection testing.

While such a modeling toolkit allows the construction of VE scenes at a sufficiently high level, they are no substitute for a good interactive modeling or CAD package. Therefore you need to make sure that the rendering system of your choice supports import for the geometry file format of your favorite modeling system. Do not forget that you need to check whether much-needed high-level information (such as which primitives belong to one object) must survive the data exchange!

Creating a toolkit for modeling and rendering is tedious and labor-intensive. It is better to use an existing solution, most of which are more than powerful enough to support your needs, and also are optimized for specific hardware configuration, which is extremely hard to do yourself. Popular choices include OpenInventor [10], Iris Performer [11], DVS [12], or Sense8 WorldToolKit.

Rendering acceleration. When designing a virtual environment experience, one may never forget that the hardware alone cannot bring high quality immersion, unless the image generation is programmed efficiently. Vendors of graphics boards constantly try to outperform each other with impressive polygon-per-second figures indicating peak performance of their products. However, professional graphics programmers know that a tenth of the figure stated by the PR information is the typical real-application performance. When dividing this number by the 20 or so frames that are necessary to maintain smooth animation for a convincing and pleasant VR experience, the polygon budget for a single frame gets really tight even for moderately complex scenes. Textures make up for the lack of detail, but low cost image generators usually do not support them (however, last-generation video games such as the Sony Saturn already support textures).

Nevertheless it is absolutely mandatory that the graphics capability at hand is exploited as much as possible. Consequently a simplistic approach along the line "throw all polygons in the rendering pipeline and forget about it" is not sufficient. Instead, it is necessary to carefully consider the trade-off between cost for preprocessing and actual rendering (which is more or less done by hardware). The most prominent methods for rendering acceleration are:

* **Visibility preprocessing**: In a large virtual environment, most of the geometry is invisible, partly because many objects lie outside the viewing frustum (typical HMDs have only a 40deg. field of view!), partly because of occluders (e.g., walls in a building). The (in)visibility of these parts of the scene can be determined by the hardware (typically Z-buffer), but the overload created from processing (scan converting etc.) the massive amount of geometry that is not visible anyhow is exactly what makes the rendering inefficient. It is therefore better to use some CPU capacity to pre-determine what is really visible or a superset thereof, and only pass this pruned dataset to the rendering hardware. This can be done by standard viewing frustum culling (e.g., using auxiliary data structures like BSP trees [13]) or by specialized data structures that exploit occlusion [14].

* **Level-of-detail rendering**: Realistic models for virtual environments can become very complex, consisting of thousands of geometric primitives [15]. If objects are far away or very small, the details cannot be seen and the effort spent on rendering them is wasted. To compensate, a model may be represented in multiple levels of detail (LOD), e.g., using an increasing number of polygons for each successive LOD. A good selection of the LOD for each object at runtime requires sophisticated heuristics [16].

4. Case study

4.1 Motivation

A game is a good example to demonstrate the important aspects we are interested in: It will only work if it is capable of providing a high level of immersion. If it fails in this respect, the user will not be captured. Furthermore, it involves all the aspects of a virtual reality application, without being too complex.

From the type of application, the requirements for software and hardware can be derived. First, one has too choose the metaphors that will be involved in the interaction of the user with the environment, in particular navigation. In our game, the user navigates in a maze and gets drawn into shooting fights with enemies. To support this situation, we need 3-D display of the maze with rapid viewpoint control, so that the user can quickly spot the enemies. We also need a simple and direct method for navigation and aiming at enemies, because these actions will be time-critical. We chose to address these requirements with a head-mounted display (HMD) with head-tracking, and a hand-held 3-D mouse. The HMD is driven by a 3-D accelerated SGI Indy workstation, which places the system cost somewhere in mid-range. Low-cost (PC-based) hardware would not have provided the necessary, prices for 3-D acceleration are dropping rapidly, and we can soon expect the hardware to be available at commodity prices.

4.2 System Overview

In this section, we discuss what hardware and software components are necessary for the construction of our example VE (fig. 2). We are decomposing the task into the "classical" domains of information processing (input, computation, output). However, note that a VE is a closely-coupled human-in-the-loop system that must run at interactive speeds.

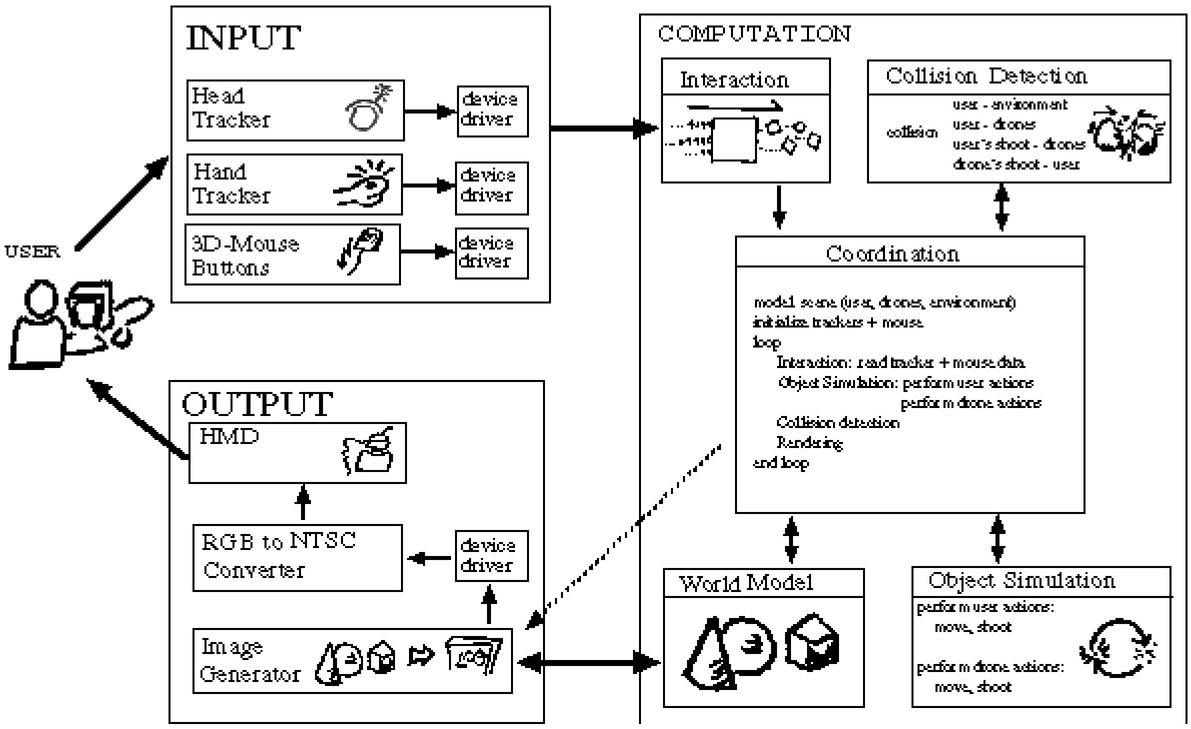


Fig. 2: System overview diagram

Input. System input comes from the interaction devices that are connected as peripherals to the computer, in our case the trackers for head and hand. Transforming the data collected by the devices into useful information (e.g., filtering out noise) can be a difficult task.

Computation. Once input data is obtained, it has to be processed so that the application can use it. An interaction module transforms the raw data (e.g., state of button changes) into semantic events ("user shoots missile"). To do this, the interaction module may require information about the current state of the simulation (e.g., the projectile fired by hitting the button may depend on the selected weapon, and whether there is ammunition available). The data is then passed on to the central coordination component, which is responsible for invoking the other modules to keep the system running. In particular, it has to look after the simulation of the objects that are present in the VE, in particular the autonomous agents (in our case, the robot drones that attack the user) and the representation of the user (e.g., simulating movement of the users vehicle based on velocity, acceleration etc.). A special task of the simulation that is both important and complex (because of the high computational effort) is collision detection and response. All these modules operate on the world model, a special database describing the details of the VE.

Output. Part of the world model is occupied by the geometric representation, which is then used for creating the images in real-time on the HMD. Needless to say, real-time rendering in convincing quality is a difficult problem into which a lot of effort is being invested.

4.3 World model and interaction metaphor

While it is often argued that VEs will revolutionize human-computer interaction because of the potential to interact with the computer in the same way we interact with the real world, the immature quality often encountered in today’s VR technology forbids complex types of interaction. It is therefore of utmost importance to carefully choose and fine-tune the metaphors used for the design of the interaction with the VE, so that it does not place too high demands on the user.

Navigation of the user in the virtual world is especially important. Without doubt the most natural navigation would be to let the user walk physically. Unfortunately, this is rarely possible because of difficulties in tracking larger areas, safety considerations when wearing HMDs, screen-bound applications when not using an HMD, length of cables etc. Therefore a navigation metaphor has to be chosen that allows the user to remain relatively stationary (standing or sitting) while navigating. A suitable solution is the simulation of a (simplified) vehicle that is operated in a similar way as the real-world counterpart. Fly-through allows movements in 3-D more or less without any restriction. Drive-throughs and walk-throughs are used if the user is meant to remain "on the ground" of the virtual environment. The usage of the vehicle is generally greatly simplified (no physics etc.) to make usage as easy as possible, unless accurate simulation of the vehicle operation is the focus of the application (e.g., flight training).

While it is crucial that the navigation problem be solved, one may not neglect the other aspects of the interaction component. Depending on what the goal of the VE is, a variety of functions may need support. In general, using interaction elements that we know from the 2-D desktop area does not work very well in an immersive 3-D environment. For example, it turns out that displaying menus in 3-D is often not perceived very well by the user. Insufficient quality (e.g., tracking inaccuracy) may make some otherwise good ideas unworkable. Instead, new methods are suddenly possible (e.g., gesture recognition). If the system has a real-time aspect (e.g., when the user is attacked in a game), the interface must be extremely simple and direct so that the user is able to respond fast. The same holds for conventional user interfaces, but in VEs the problem is more common.

For our simulation, we chose to simulate a simple "glider" vehicle that transports the user in the maze. The vehicle always moves in the direction of its nose. The vehicle is controlled by buttons of the 3-D mouse (left, right, forward, back). No physics are simulated for the vehicle. Parts of the hull of the vehicle are represented in the scene (see fig. 3), so that the user can determine the vehicle’s orientation (and hence the direction of movement). The user’s hand (with the 3-D mouse represented by a gun) is also displayed so that the user can visually coordinate his manual actions, in particular shooting (one 3-D mouse button is reserved to trigger the gun). The head-tracking allows to move the direction of sight independently of the direction of vehicle movement (e.g., one can look out of the "side" window), which is important to check for enemies behind oneself. After a short period of adjustment, the users feel quite comfortable with this simple metaphor.

4.4 Hardware and Software

For tracking, we use a Polhemus Fastrak with two sensors, one for the head and one for the hand. The tracker unit is connected to the serial port of a 486-based PC running Linux that acts as a tracker server. Using standard TCP/IP, the PC in turn is connected to the Indy workstation that runs the virtual environment and generates the images. UDP sockets are used for efficient communication of the tracker data to the workstation. We use the PC for tracking, because it provides a cheap way of distributed computation. The prediction filter applied to the tracker data consumes a substantial amount of CPU power. It is much cheaper to dedicate an inexpensive Intel CPU to the task than buy a more powerful workstation. Additionally, most of the cheaper VR peripherals are developed for the PC market, and it is therefore straightforward to connect them to PCs, whereas connecting to workstations can be annoyingly complicated.

Our system is able to compensate for the delay introduced by head tracker measurement, network transmission, and rendering using a Kalman filter based predictor as described in [17]. This method greatly reduces the effect of "swimming" of images that are not consistent with the head movement, and is also capable of reducing the noise in the measurement.

Presentation of the VE to the user is done with an EyeGen3 HMD from Virtual Research. This HMD accepts two NTSC composite signals and displays them at TV resolutions. Conversion of the monitor signal is done with an external RGB-to-NTSC converter. We use the same input for both eyes and neglect stereoscopy, predominantly for cost reasons. The HMD provides immersion even though images are not stereoscopic. Future plans involve using two Indys synchronized over a network to generate separate images for both eyes. For rendering, we chose to use the Iris Performer toolkit, because it provides excellent performance on SGI workstations.

4.5 Object simulation

The simulation of objects determines the kind of experience the user has in the virtual environment. Without simulation, the VE would only consist of "dead" geometry without interesting situations. In our test implementation, the simulation is only concerned with two types of objects: The user himself and the enemy drones.

User. Simulating the user more or less consists of appropriate responses to the user’s input. The glider vehicle that is occupied by the user is rotated and moved back and forth according to the commands given with the 3-D mouse’s buttons. No physical properties are computed for the vehicle; it starts, moves and stops completely determined by the user’s commands. This does not only simplify the simulation, it also is convenient for the user who is kept busy fighting the drones. The application is responsible for displaying the part of the user’s geometry that is visible (the shoulder, arm and hand, and the trunk of the vehicle) to aid the user’s orientation.

Drones. The drones’ simulation focuses on moving them through the maze and attacking the user on sight. This involves a strategy for direction selection: Drones always move along corridors, until they reach a crossing. In "easy mode", drones wander aimlessly, whereas in "hard mode", they move in the user’s general direction. Upon sight they directly approach the user and fire in the user’s current direction.

4.6 Collision detection

Because humans are used to the fact that solid objects cannot intersect, collision detection (and response) is very important for realism in VEs, especially because force feedback is generally unavailable. However, physical realism is not mandatory for convincing collision detection.

Our environment consists of moving entities (user, drones, projectiles), and static decoration (walls of the maze). We do not intersect the actual geometry of the moving objects, but rather approximate them with simple geometric bounding volumes. We use a sphere for the user, cones for the drones, and small spheres for the projectiles. Using simple geometric shapes, performing the intersection computations is much easier than with a potentially complex polygonal datastructure (of course at the cost of exactness!). For our application this simplification is sufficient. We can further exploit the fact that most of the potential collisions (involving user, drones, walls) can be determined from the 2-D projection of the shapes onto the ground floor. Only the projectile-object collisions need to be determined in 3-D since a shot may miss the target because it is aimed to high or to low.

4.7 Flow of action

The following fragment of pseudo code briefly outlines the steps that are executed in the main loop of the program. The most important steps are tracker reading, user and drone simulation, shot simulation, collision detection, and rendering:

```
while(user_stamina >= 0)
read_tracker_data
case user input of
left,right,forward,back: move_vehicle
trigger: fire_shot
for every drone
move_drone
if user_in_sight then fire_shot
compute_shots
for every pair of objects:
case collision_detection
vehicle_to_wall: stop_vehicle
vehicle_to_drone: lower_user_stamina
usershot_to_drone: destroy_drone
droneshot_to_user: lower_user_stamina
set camera according to head tracker
for each object: draw_object
```

Fig. 3 gives an impression of the implementation: On the right hand side the user’s arm can be seen, just before and immediately after he fires a projectile at an enemy drone.

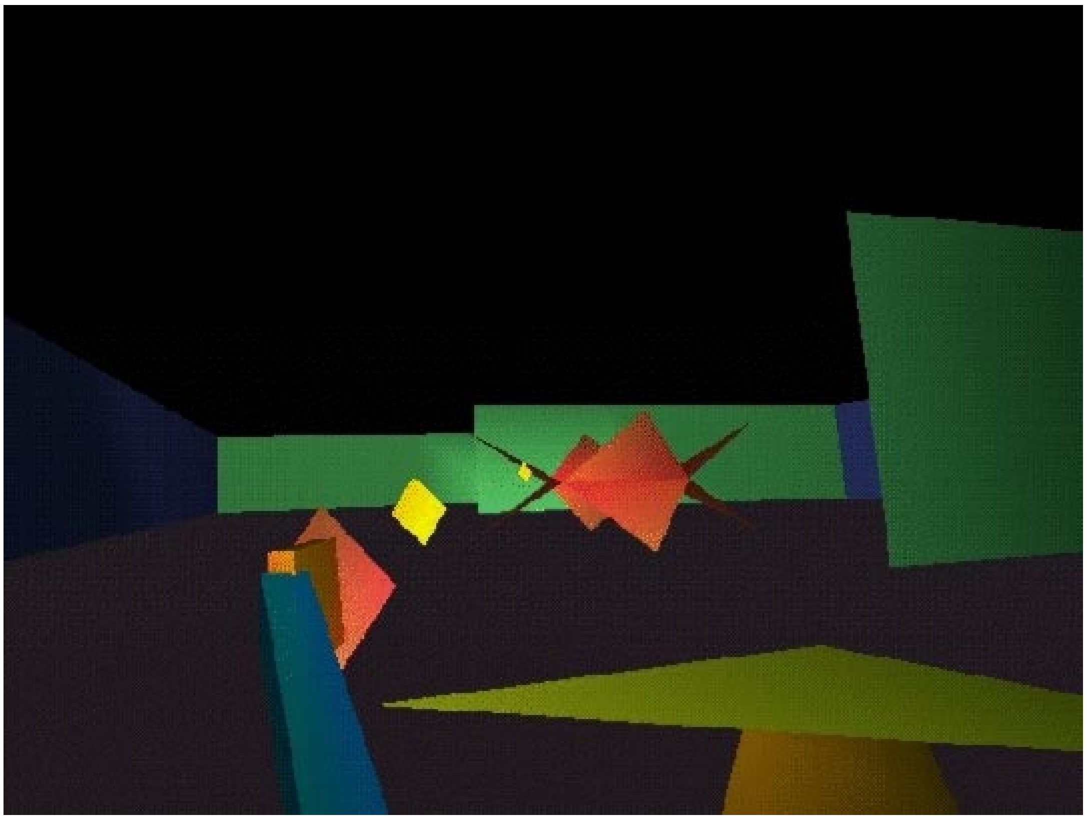


Fig. 3: Screen shots from the example implementation

5. Conclusion

We have presented a case study for the construction of a virtual environment to aid those planning to implement a similar system. A simple checklist outlines fundamental steps to take in the realization process: A characterization of problem domain and application leads to a design metaphor; when a decision is made on hardware and implementation strategy, and the hardware is proven to work, the actual implementation can be done, and the design evaluated. This "cooking recipe" is supplemented by a treatment of necessary background knowledge on virtual environments, including tracking technology, coordinate transformations for VEs, display options, modeling and rendering. A simple demo implementation

of an immersive 3-D maze game using an HMD with head and hand tracking is used to serve as an example to illustrate the concepts that have been discussed.

References.

[1] M. Zyda, D. Pratt, J. Falby, C. Lombardo, K. Kelleher: The Software Required for the Computer Generation of Virtual Requirements. Presence, Vol. 2, No. 2, pp. 131-140 (1993)

[2] C. Codella et al.: Interactive Simulation in a Multi-Person Virtual World. Proceedings of SIGCHI, pp. 329-334 (1992)

[3] C. Blanchard, S. Burgess, Y. Harvill, J. Lanier, A. Lasko, M. Oberman, M. Teitel: Reality

Built for Two: A Virtual Reality Tool. SIGGRAPH Symposium on Interactive 3D Graphics, pp. 35-38 (1990)

[4] R. Gossweiler, R. J. Laferriere, M. L. Keller, R. Pausch: An Introductory Tutorial for Developing Multiuser Virtual Environments. Presence, Vol. 3, No. 4, pp. 255-264 (1994)

[5] R. Pausch, T. Burnette, M. Conway, R. DeLine, R. Gossweiler: Alice: A Rapid Prototyping System For Virtual Reality. SIGGRAPH'94 Course, No. 2 (1994)

[6] K. Meyer, H. Applewhite, F. Biocca: A Survey of Position Trackers. Presence, Vol. 1, No. 2, pp. 173-200 (1992)

[7] R. Holloway, A. Lastra: Virtual Environments: A Survey of the Technology. SIGGRAPH'95 Course, No. 8, pp. A.1-a.40 (1995)

[8] W. Robbinet, R. Holloway: The Visual Display Transformation for Virtual Reality. Presence, Vol. 4, No. 1, pp. 1-23 (1995)

[10] P. Strauss, R. Carey: An Object Oriented 3D Graphics Toolkit. Proceedings of SIGGRAPH'92, No. 2, pp. 341 (1992)

[11] J. Rohlf, J. Helman: IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. Proceedings of SIGGRAPH'94, pp. 381 (1994)

[12] S. Ghee, J. Naughton-Green: Programming Virtual Worlds. SIGGRAPH'94 Course, No. 17 (1994)

[13] B. Naylor: Interactive playing with large synthetic environments. SIGGRAPH Symposium on Interactive 3D Graphics (1995)

[14] S. Teller, C.H.Séquin: Visibility Preprocessing For Interactive Walktroughs. Proceedings of SIGGRAPH'91, Vol. 25, No. 4, pp. 61-69 (1991)

[15] M. Deering: Data Complexity for Virtual Reality: Where do all the Triangles Go?. Proceedings of VRAIS'93, pp. 357-363 (1993)

[16] T. A. Funkhouser, C. H. Sequin: Adaptive Display Algorithm for Interactive Frame Rates During Visualisation of Complex Virtual Environments. Proceedings of SIGGRAPH'93, pp. 247-254 (1993)

[17] T. Mazuryk, M. Gervautz: Two-Step Prediction and Image Deflection for Exact Head-Tracking in Virtual Environments. Proceedings of EUROGRAPHICS'95, pp. 29-41 (1995)