A Language for Describing Behavior of and Interaction with Virtual Worlds

Gabriel Zachmann
Fraunhofer Institute for Computer Graphics
Wilhelminenstrasse 7
64283 Darmstadt, Germany
email: zach@igd.fhg.de

Abstract

Virtual environments are created by specifying their content, which comprises geometry, interaction, properties, and behavior of the objects. Interaction and behavior can be cumbersome to specify and create, if they have to be implemented through an API.

In this paper, we take the *script* based approach to describing virtual environments. We try to identify a generic and complete, yet simple set of functionality, so that non-programmers can readily build their own virtual worlds.

We extend the common object behavior paradigm by the notion of an *Action-Event-Object triad*.

Keywords: behavior, interaction, script languages, virtual reality.

INTRODUCTION

Today, virtual reality is on the verge of leaving the pure "research domain". Among others, the automotive industry is evaluating its potential in the design, development, and manufacturing processes [6, 5]. Still, there is a great lack of tools to make VR an enabling technology which would be easy to use in industry and entertainment. Creating virtual worlds is still a cumbersome and tedious process. Furthermore, any VR system meant to be used within an industrial process must face the fact that it is just one link in a long chain of software packages (CAD, CAE, FEM, etc.), which might impose a lot of constraints and requirements.

This paper tries to propose a framework which increases productivity when creating virtual environments (VEs). VE "authors" should be allowed to experiment and play interactively with their "worlds". Since this requires very low turnaround 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. As a matter of course, our virtual worlds should be input-device independent.

In order to achieve these goals, we try to 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:

- *Event based*.
  One approach is to write a *story-board*, i.e., the creator specifies which action/interaction happens at a certain time, because of user input, or any other event.

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

- *Behavior based*.
  Another 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 take 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 your virtual world.

In this paper, 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.

Overview. In the next section, we briefly discuss the approach other systems have taken. Then, we develop our general paradigm for virtual world descriptions. This will be further detailed by the next two sections. Finally, we present some real-world applications, give an outlook on our future work, and give some examples in the Appendix.

RELATED WORK

There are quite a few existing VR systems, some commercial some academic. Some of them we will consider briefly in the following.

Sense8's WorldToolkit follows the toolbox approach. Basically, it provides a library with a high-level API to handle input devices, rendering, simple object locomotion, portals, etc.

DIVE is a multi-user, distributed VR platform. The system can be distributed on a heterogeneous network (making use of the Isis library[3]). New participants of a virtual world can join at any time. They will receive a copy of the current database. All behavior is specified as a (usually very simple) finite state machine (FSM). Any FSM is part of some object's attributes. Database consistency is achieved by using distributed locks.

Division's dVS features a 2D and 3D graphical user interface to build and edit virtual worlds at run-time. Attributes of objects are geometry, light source, sound samples, collision detection parameters, etc. Objects can be instanced from classes within the description file of a virtual world. Inheritance (and polymorphism?) are supported. Several actions can be bundled (like a function in C) and invoked by user-defined events. However, the syntax seems to be rather complicated and not really apt for non-programmers.

The Minimal Reality toolkit (MR) [15, 11] is a networked system, which uses a script file to describe behavior and sharing of objects. Scripted object behavior is compiled into so-called OML code which is interpreted at run-time. For each OML instance there must be an associated C++ class.

Unlike MR, we won't develop objects ("classes") with rather high-level built-in behaviors, such as Tanks, Bombs, or Hills. Instead, we will identify actions on objects on a much lower, and therefore more generic, level.

We believe, that keeping *all* information about the virtual world (i.e., geometry, behavior, constraints) in *one* file can be a tedious and very inflexible. We strictly separate geometry, behavior, physical properties, acoustic properties, etc., in separate files, unlike [11, 1, 10]. We feel that this will save a lot of time when developing virtual environments because very often the geometry will be created by third parties or sophisticated animation software. During several development iterations, we usually get several versions of the geometry.

GENERAL CONCEPTS

A complete VR system is a huge piece of software, consisting of an object manager, renderer, device drivers, communication module, navigation and interaction module, and, probably, physically-based simulation, sound rendering, visualization, application-specific modules, etc. (see Figure 1).

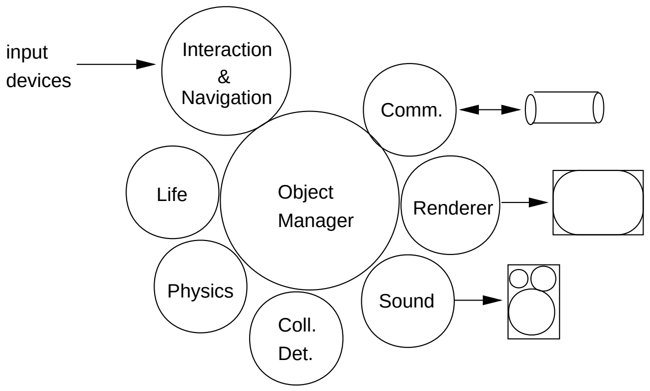

Figure 1: The object manager is the central module of probably any VR system. All other modules which build on it "simulate" or render a certain aspect of the virtual environment. Some of those are controlled by the interaction module (e.g., sound renderer and device drivers); others are "peer" (e.g., physically-based simulation).

In this paper, we will focus on the interaction module, which deals with navigation, basic interaction, and basic "life" in our virtual worlds. We will not deal with the overall system architecture.

Furthermore, we will restrict ourselves to the identification of generic functionality. Most of the time, we will *not* deal with syntax here.

The visual part of a virtual world is represented by a hierarchical scene graph. Everything is a node in this graph: polyhedra, assemblies of polyhedra, LODs, light sources, viewpoint(s), the user, etc. Most functionality and interaction presented below will operate on the scene, i.e., it will, eventually, change some attribute(s) of some object(s).

The action-event paradigm A virtual world is specified by a set of *static* configurations (geometry, module parameters, 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*[1] (see Figure 2).

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

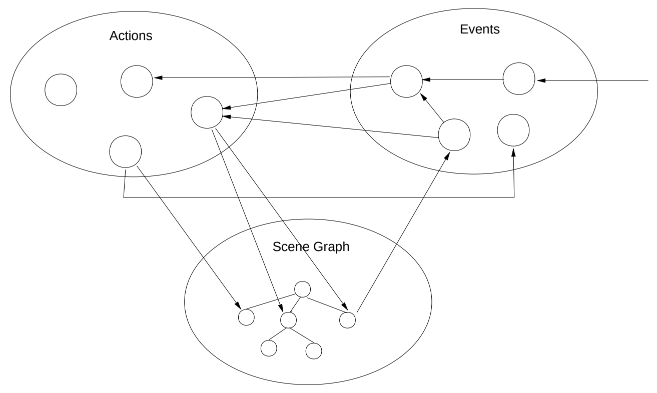

Figure 2: The *AEO triad.* Anything that can "happen" in a virtual environment is represented by an action. Any action can be triggered by one or more events, which will get input from physical devices, the scene graph, or other actions. Note that actions are not "tied-in" with graphical objects, and that events are objects in their own (object-oriented) right.

Note that 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 [12]) in order to realize *temporal operators*. Parallel execution of several actions can be achieved trivially, since one event can trigger many 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.

Because of our considerations above, we need to be able to refer to actions and events. Therefore they can be given a name. Basically, there are two ways to declare an action-event pair in the script:

action-name`:` *action ...*
event-name`:` *event ...*
action-name event-name

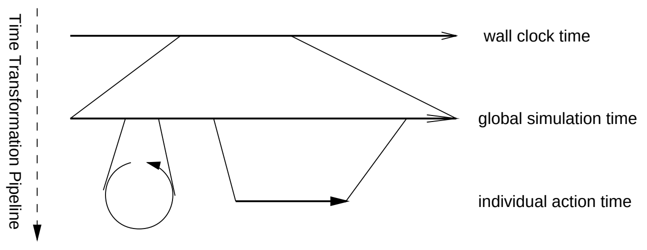

Figure 3: A simulation of virtual environments must maintain several time "variables". Any action can have its own *action time* variable, which is derived from a global *simulation time*, which in turn is derived from wall clock time. There is a small set of actions, which allow the simulation to set/change each time transformation individually.

or

action ... event ...

where *action* and *event* in the latter form cannot be referenced elsewhere in the script.

Most actions operate on objects, and many events have one or two objects as parameters. In order to achieve an *orthogonal language*, those objects can have any type (geometry, assembly, light source, etc.) whenever sensible.

Time. Many actions (besides navigation, simulation, and visualizations) 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 (almost) 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 3).

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.

Grammar. The grammar in our system is fault-tolerant and robust against variations and abbreviations, such as `playback`, `play-back`, `anim`, etc. Ordering of "commands" should (almost) never matter! (We achieve this by using a multi-pass parser.)

For easy creation and maintenance of almost identical scripts, full C-like preprocessing. The preprocessor's macro feature provides an easy way to build libraries with higher-level behavior.

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, and all but *input* are optional. 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 at 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 is

```
action
on|off|switch_on|switch_off|toggle
while_active|while_inactive|
     when_activated|when_deactivated
input
```

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). However, we feel that this would be too cumbersome, since those trigger behaviors are needed very frequently.

In order to be able to trigger any action at start-up time, there's a special "input" named `initialize`.

All actions must be able to understand the messages `on`, `off`, and `toggle`. All events must store their current state, which will be changed by input transitions.

In addition to the basic events, events can be combined by logical expressions, which 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 done so a user can be prevented from doing one thing before he does something else first. An action is deactivated when it doesn't respond to messages being sent by events. This is done via a certain action, which (de-)activates other actions. Alternatively, it is quite convenient to be able to bracket the actions you want to (de-)activate:

```
active event
actions ...
endactive
```

A Collection of Event Inputs

Physical input includes all kinds of buttons (keyboard, mouse, spacemouse, boom), flex and tracker values (are *active* when above/below threshold), 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").

Virtual buttons are just like 2D buttons of a GUI. Actually, they just check the collision between the button object and some pointing "device", usually the graphical representation for the finger tip. Any object of the scene graph can be a virtual button. Likewise, we have implemented virtual menus. Each menu item can be the input to one or more events.

Geometric events are triggered by some geometric condition. Among them are *portal*s and *collision*s.

A portal is an arbitrary object of the scene graph. The event is triggered by the inside/outside-status of an object with respect to that portal. By default, we check the center of the object's bounding box (or any other point in local space). The object can be the viewpoint. This event is very useful for switching on/off parts of the scene when the user enters/leaves "rooms" (multiple virtual worlds), or for (daisy-)chaining actions and animations (for instance, playing a sound when some object passes by).

A collision event is triggered by the exact collision of two objects (see [19, 18]).

Any action's status (*on* or *off*) can trigger an event. Some actions (like `callback`, `counter`, etc.) have an action-specific status, which can be used also. For example, counters have a current value which can be compared to another counter or a constant. The result of the comparison is the input to the event.

All time variables (see above) can be compared to a certain *time interval* (which could cross the wrap-around border of cyclic time variables!), or by one of the usual comparisons. The result (0/1) is the input to an event.

Sometimes, we want to "monitor" certain object attributes (integer, float, vector, or string valued) and issue an action when they change, while we don't care which action (or other module!) set them. The general form of an object attribute event input is

`attribute` *attribute-name* `object` *object-name comparison*

Attributes are not only graphical attributes (transformation, material, wireframe, etc.), but also "interaction" attributes, such as "grabbed", "falling", etc. This implies that the interaction module maintains an additional, augmented representation of all objects of the scene graph.

Object attributes might be set by our interaction module itself (by possibly many different actions), or by *other* modules, so object attribute events can provide a kind of simple access control mechanism in some cases.

Any action's state can be fed back into an event in order to trigger other actions. Certain actions (like `counter`) provide specific states, which we can interrogate, also.

ACTIONS

Actions are the building blocks for the "life" in a virtual environment. Anything that happens, as well as any object properties are specified through actions.

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.

Consistency. This is certainly an issue in any VR system being used for real-world applications. Here, we will not discuss the problems arising in multi-user VEs, or in systems with concurrent modules. The problem of consistency exists even within our interaction module. Some of the actions described below set transformations of objects. Obviously, those will interfere when they act at the same time on the same object. For example, in applications for automotive industries, we can grab an object with our left hand while we stretch and shrink it with the other hand. The problem arises also, when an action takes over (e.g., we scale an object after we have grabbed and moved it).

Flushing transformations or squeezing them into one matrix is not a very good idea, since we loose the original object and we have no control over the order of transformations. We do not want to loose the transformations of earlier actions, because this is valuable information we might need in a further step "outside" the VR system.

One way to deal with that is to impose a strict sequence of transformations to be used per object, at least for objects under the control of this module. We have chosen the sequence: *scaling*$\times$*rotation*$\times$*translation*.

Another issue arises when we use levels-of-detail (LODs) in the scene graph. Since any object can be a LOD node or a level of an 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).

Navigation.

The most basic "interaction" with a virtual world is navigating through it. We won't go into a thorough discussion of different techniques here. That can be found in [2, 8, 16, 14]. Instead, we will just state that a VR system must be easily configurable, and able to switch to a broad variety of devices such as mouse, spacemouse, boom, glove and HMD, microphone, etc.

For these different devices, the VR system must provide different navigation paradigms, such as eyeball-in-hand, relative mode, point-and-fly, or voice navigation (see Figure 4).

None of navigation modes must rely on the assumption that the cart is an immediate child of the root. We might

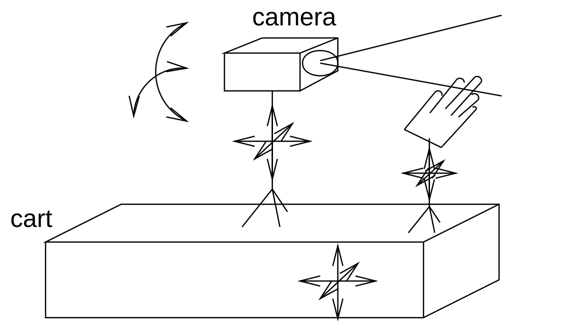

Figure 4: All navigation modes can be mapped on the *flying carpet* paradigm.

wish to make the cart a child of another object, which could be animated, for example.

There are many parameters which affect user representation: navigation speed, size and offset of the hand, scaling of head motion, eye separation, etc.

Navigation can be switched on/off by any event. With the point-and-fly paradigm, this is usually a gesture or a spoken command.

By using the abstraction of *logical input devices*, all navigation modes are completely device-independent [8, 9].

A Collection of Actions.

During our past projects, the set of actions listed briefly below 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, transparent, 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 subtrees of the scene graph (e.g., move a table when you grab its leg).

By adding just a few more properties and functionality, we can further increase believability of our worlds. Those additional properties are *friction*, which makes objects "ride on top" of others, and *pushing*, which prevents interpenetration (see [13]).

When the "stretch" action is invoked on an object (or subtree), handles will be displayed at the corners and faces of its bounding box. These can then be grabbed and will scale the object(s) as they are moved. This action is quite useful to select a certain space in the world or to create place holders from generic objects like spheres, cylinders, etc.

The "sweep" action traces out the volume when the user moves an object by replicating (simplified) copies of it. This was used to check serviceability of, for example, car engines.

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.

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

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

Constrained movement. 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 to their current value,

   (b) keep the distance to other objects (e.g., ground) to a pre-defined or to the current value. The distance is evaluated in 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 (all first-time users of VEs wonder why they can fly through walls). It can be used for any other moving object as well.

Of course, the constraints listed above are just very simple ones; for more complicated "mechanisms", a general approach will be needed, like [17] or [20, 7].

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 operates 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.

Finite state machines. The system can maintain an arbitrary number of *counters*. Those counters can be set, incremented, or decremented via certain actions. They can be used as input to events (which in turn trigger other actions).

Counters are very useful to switch from one "stage" of a "story-based" VE to the next one by the same gesture or voice command, or they can be used to build more complicated automata (a traffic light, for example).

User modules. From our experience, most applications will need some special features which will be unnecessary in other applications. In order to incorporate 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.

APPLICATIONS

An early application of our integrated VR system was shown at the "Industry Exhibition Hannover" in Spring 1995. By then, most key concepts had been designed. At the show, we tried to point out some possible applications of VR in the automotive industry.

A demonstrator for *virtual prototyping* was built using our VR system in May 1995 [5]. One of the key features was real-time collision detection [19] for clash and clearance checks (see Color Plate I). Others are volume tracing and constraints (to model the hood of the car). Flexible objects (like a hose) were implemented by an application-specific module.

For the IAA auto show in Frankfurt, Germany, in August 1995, we built the inside of a diesel engine as a virtual environment. A user could fly inside the combustion chamber and watch several combustion cycles while interacting with the air flow field. This was also shown at the Detroit Auto Show in 1996. One of the key features there was the visualization of time-variant flow-fields in the swirl port and in the combustion chamber (see Color Plate IV). All data, geometry, animations, and flow fields have been imported from simulation packages. This is where the necessity of a concept of time becomes evident: all animations (like piston head, valves, and temperature color) must coincide with the visualization, even though most of them are not specified on the complete cycle. Furthermore, global simulation time must be set sometimes to a certain value, or the "speed" of simulation time must be slowed down or stopped altogether.

PRESENT AND FUTURE

We believe that the action-event-object paradigm is very powerful, yet at the same time remaining simple and making it easy to learn to create non-trivial content in virtual environments. The set of actions and events proposed in this paper should be capable of handling most basic behavior and interaction. The concept of events gives greater flexibility and relieves the VE author from the burden of maintaining state variables. A variety of navigation modes, as well as object constraints have been described. Actions have been identified to select and manipulate objects, and to animate object attributes. Application-specific modules can be integrated smoothly. In the context of VR, concepts for handling time and consistency have been presented.

In the future, we will implement an easy-to-use graphical user interface, so virtual environments can be built on-line. It should be possible to add, delete, and change actions and events at *run-time*. A VE creator might even want to "re-wire" actions and events at run-time. To this end, the system must keep a complete history of the state of all actions, events, *and* the scene graph!

So far, we can "can" sets of actions/events by defining preprocessor macros, which are kept in libraries and can be included. With a graphical world editor, it would be nice to have access to those, too.

EXAMPLES

In this section, we will give a few excerpts from the scripts which were written to implement the virtual environments described above.

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 example will make *object* fall down and bounce off the *floor-objects* when released by the hand. Additionally, a sound will be played each time the *object* bounces.

```
grab object toggle when activated
gesture fist
gravity 0.2
fall object floors floor-objects parameters \
    switch on when deactivated
grabbed object
sound sound-file switch on when
activated collision object floor-object
```

Menu setup consists of two parts: the interaction with the menu itself,

```
menu popup MyMenu options speech "menu"
menu acknowledge MyMenu joint
thumbouter
```

and the specification of actions triggered by menu selections

```
action menu button MyMenu_1
```

In order to provide acoustic feedback when action `A` is switched on, we can write

```
sound sound-file switch on when
activated action A
```

Finally, we will present 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"
```

REFERENCES

1 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.

2 P. Astheimer, F. Dai, M. Göbel, R. Kruse, S. Müller, and G. Zachmann. *Realism in Virtual Reality*, pages 189–210. Wiley & Sons, 1994.

3 K. Birman. Replication and fault-tolerance in the isis system. *Proc. of the 12th ACM Symposium on Operating Systems*, pages 79–86, 1985.

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

6 F. Dai and M. Göbel. Virtual prototyping - an approach using vr-techniques. In *Proceedings of the 14th ASME Int. Computers in Engineering Conference*, Minneapolis, Minnesota, Sept. 1994.

7 M. Eppinger and E. Kreuzer. Systematischer Vergleich von Verfahren zur Rückwärtstransformation bei Industrierobotern. *Robotersysteme*, 5:219–228, 1989.

8 W. Felger. *Innovative Interaktionstechniken in der Visualisierung.* Springer, 1995. Reprint of the dissertation.

9 W. Felger, R. Schäfer, and G. Zachmann. Interaktions-toolkit. Technical Report FIGD-94i002, Fraunhofer Institute for Computer Graphics, Darmstadt, Jan. 1994.

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. 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.

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

13 M. J. Papper and M. A. Gigante. *Virtual Reality Systems*, chapter Using Physical Constraints in a Virtual Environment, pages 107–118. Academic Press, Inc., 1993.

14 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.

15 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.

16 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.

17 A. Witkin, K. Fleischer, and A. Barr. *Topics in Physically-Based Modelling*, chapter Energy Constraints On Parameterized Models. ACM SIGGRAPH, 1987.

18 G. Zachmann. Precise and high-speed collision detection in interactive real-time visualization systems. Master's thesis, Technische Hochschule Darmstadt, Germany, Fachbereich Informatik, 1994. ftp://ftp.igd.fhg.de/outgoing/zach/collision.ps.Z.

19 G. Zachmann. The boxtree: Enabling real-time and exact collision detection of arbitrary polyhedra. In *Informal Proc. First Workshop on Simulation and Interaction in Virtual Environments, SIVE 95*, University of Iowa, Iowa City, July 1995. The OX Association for Computing Machinery.

20 J. Zhao and N. I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics*, 13(4):315–336, 1994.