Extensible Approach to the Virtual Worlds Editing

Vít Kovalčík, Jan Flasar and Jiří Sochor*
Faculty of Informatics, Masaryk University
Brno, Czech Republic

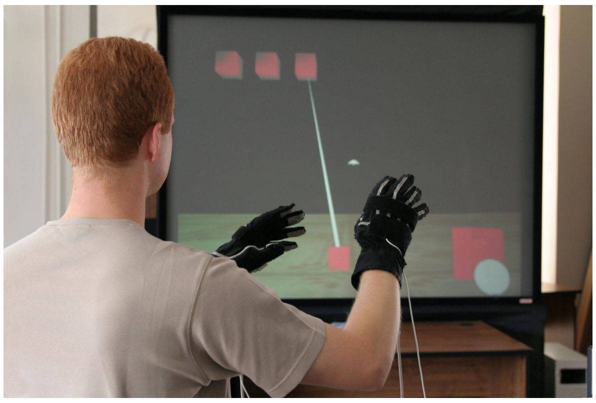
Figure 1: *The virtual environment editor.*

Abstract

We present a virtual reality framework (VRECKO) with an editor that is capable of creating new scenes or applications using this framework. The VRECKO system consists of objects with predefined behaviors that an application designer can dynamically change. With instances of a special object type called Ability, we may extend or change behaviors of objects in a scene. As an example of this approach, we present an editor that we implemented entirely as a set of abilities. The editing is done directly in 3D environment which has several benefits over the 2D editing, particularly the possibility to work with a scene exactly as in the final application.

CR Categories: I.3.7 [Three-Dimensional Graphics and Realism]: Virtual reality; I.3.6 [Methodology and Techniques]: Interaction techniques

Keywords: VR system, editing virtual worlds

1 Introduction

The virtual reality (VR) systems are becoming more common then they were in the previous years. We can walk through an ancient city reconstructed in a computer, test-fly aircraft, arrange a training for a hazardous situation or visualize furniture, all in environments that are non-existent in reality.

It is possible to make a separate application for each of the purposes, but it is more practical to create a generic framework that covers the basic needs. The designer implements the applications as extensions of such framework. Using a component-based approach, a programmer creates new components to extend the framework or they can reuse the existing ones in order to create a new application. We have implemented a framework with such features, that it is both efficient and easy to use.

However, even with the component-based approach it is necessary to prepare a scene, connect the components to interact properly, and perform various additional tasks to set up the environment. While it may be possible to pass the settings to the framework as parameters, carefully preparing these parameters requires considerable effort.

For this reason, researchers developed and published several editing applications. These editors ease the creation and setting up of a target application. Many of the editors serve for domain specific purposes, such as arranging furniture or creating a GUI-like application. With such editors, the users usually work directly in the virtual space with the same devices as in the final application. There are also editors capable of editing larger scenes with the abilities to add objects or make logical connections between them, but in this case, a developer accomplishes the work often in 2D using keyboard and mouse.

We created an editor that works entirely in 3D space and contains tools to manage the objects and connections between them to create interactive applications.

2 Related Work

Nowadays there are many systems having the ability to construct virtual worlds and to interact with them. Some virtual reality systems are designed specially for a given application area, e.g. medical simulation or visualization of scientific data. However, there are only several systems with modular architecture, which designers can adjust according to the requirements of a user and make

*{kovalcik, flasar, sochor}@fi.muni.cz

changes and extensions of the system at run-time.

One of the first developed systems was MAVERICK [Hubbold et al. 2001] which uses an object-oriented design and decouples the system into a modules. Another example of an early work is the VR-Juggler [Bierbaum et al. 2001] which presented an idea to create a virtual platform that hides the technical details of the hardware from the programmer.

In [Tanriverdi and Jacob 2001] the authors describe a design model for developing virtual reality interfaces called VRID. The paper focuses on the methodology of designing the interface components of a VR system. The system has multi-component object architecture. Each object consists of five components: graphic, interaction, behavior, mediator and communicator component. One object can contain several different behavior components, which enable the user to create complex object behavior. The mediator component controls the communication among other components in the object. The communicator provides communication with other objects inside the virtual world using the distributed event model. If any object intends to receive future events from a given object, it registers at the communicator of the sender object. Any event generated is then sent to the target object.

The following projects are more flexible and provide new possibilities. In [Oliveira et al. 2003] the authors present a novel design approach called JADE (Java Adaptive Dynamic Environment). They discuss a challenging problem with regard to the development of VE applications. They point mainly at the non-extensibility, non-interoperability and non-evolution of the previous VE solutions. The JADE system is based on a component design methodology with a layered component framework. The main part of the system is a kernel that is accessible from any place in the system. Another important component is the ModuleManager, which provides functions for the management of the modules such as loading or replacing them in memory in the run time. The ResourceLocator component serves as an access point to the outer resources (harddisk, operating system, network etc.). Inter-module communication employs both direct access as well as event triggering. Two event distribution schemes are available: The first scheme is the distributed event model where each module dynamically registers subscribers that want to receive a generated event. The second scheme is a central event model. In this case, a central event manager component processes every event and dispatches it according to its type, priority and source. The event is then delivered to the objects, which are registered in the event manager. The JADE kernel and the resulting system can be configured at startup via a command line or via a file containing for example an XML description.

Similar problems were recognized in [Kapolka et al. 2002] and led to the development of a unified component framework called NPSNET-V implemented in Java^{TM}. The framework is able to reconfigure, add, remove and upgrade components in the run time. In comparison with the JADE system, it presents a more general solution. The core of the system is the micro-kernel that provides functions for module managing. The used modules define the functionality of a developed VR application. Other components are realized using modules with appropriate functions, similarly as in the JADE. A newer version or another module with a similar interface can replace each component (module). Both JADE and NPSNET-V are implemented in Java, which guarantees wide compatibility on many platforms.

In addition to the core system design, some researchers focused on the editors for the virtual environments. The performance of low-level specialized editors, such as Autodesk® 3ds max®, NewTek^{TM} Lightwave 3D® or Softimage|XSI® is very hard to match, therefore most authors have decided to use 3D models created in one of the existing editors and implement only the positioning and interaction in their editors to compose the final environment.

An example of the low-level editing was presented in [Grossman et al. 2001]. The paper describes a 3D modeler designed for large-scale displays, which allow users to work directly in the 3D space. However, the modeling is primarily aimed at the car design and it lacks the tools to modify large scenes or adjust the object interaction.

Interesting approach to the high-level editing was described in [Holm et al. 2002]. The authors designed an authoring tool for the design and prototyping of scenarios for virtual environment, the Safety Virtual Environment (SAVE). The tool is composed of a VR-simulation part and a desktop application (SAVE Assembly and Constructing Environment – SAVEace) which can be used by two persons at the same time to design a scenario in collaboration. The 3D editor is in the desktop part of the system and it is able to rapidly place the objects to the scene and to create logical dependencies between the objects. The SAVEace represents a sophisticated editor and the environment can be created quickly. The disadvantage is that two persons are necessary to create and test the environment. The user located in the VR has no access to the editing tools and can only give advices to the user using the desktop part of the application.

The SAVEace editor provides complex functionality, but a user accomplishes all editing work on a 2D screen using a mouse and keyboard combination. There have been various attempts to perform the editing directly in the 3D space using data gloves as a spatial input device. Most researchers focused mainly on specific tasks, such as path editing for keyframe animation [Osawa and Asai 2003]. The animation is created in the immersive environment using gloves. To help the user with the modelling a gearbox widget is used [Osawa and Ren 2003]. This widget is a part of the it3d library [Osawa et al. 2002], which allows the user to quickly develop 3D applications consisting of components. It may be thought of as an extension of the 2D GUI into a 3D space. There is a range of widgets that were created to allow the same number of possibilities in 3D space as was available on 2D computer screen. These widgets include for example button, slider or combo box.

3 Motivation

Our motivation was to create a component-based system that will be able to handle the interaction in the virtual reality using a number of input and output devices. To accelerate the development of the applications or scenes an editor was necessary. We have chosen to edit the scene directly in the immersive environment as this offers some advantages over the more traditional 2D approach, mainly the ability to test any change immediately and to see the scene exactly as the final user.

4 System Description

We designed virtual reality engine named VRECKO as a system for managing the virtual environment. The system has the ability dynamically change behavior of objects. It shares the design and principles used in JADE and NPSNET-V systems, but also incorporates new features. The VRECKO system allows us to create virtual worlds composed of visual objects with a specified behavior, which are able to communicate with the other components within the system. In this section, we will describe the system and point out the differences in comparison to the other systems.

The core of the VRECKO system consists of several main component types, which we will describe in the following sections. We will also highlight dependencies between components.

All components have simple interface – channels for input and output events/requests (see figure 2) and also the management functions, such as the initialization or update, functions for events/requests processing and for setting the priority. Update and communication schemes between components will be described later in this paper.

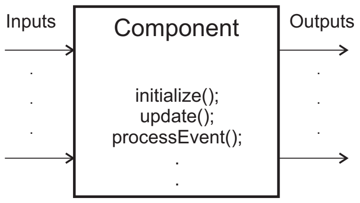

Figure 2: *Core of the component interface*

4.1 World, Scene and EnvironmentObjects

The main component of a virtual environment is the *World*. It is a container for the *Scene*, *DeviceManager*, *EventDispatcher* and the *Scheduler* components. The *World* component is accessible from any function and allows other components to use the services provided by the contained components.

The *Scene* serves as a container for virtual objects that are situated within the virtual world. This component provides the management of objects in space. Our system supports also collision detection for which we have used the Extensible Scene Graph library [Ošlejšek and Sochor 2005]. The objects may consist of many primitives; therefore, we employ additional spatial data structures. Current implementation includes bounding volume hierarchies using Axis-Aligned Bounding Boxes, k-DOPs (discrete oriented polytopes) and others.

The virtual object in the VR space is represented by the *EnvironmentObject* component. It stores the local transformation and geometry together with its collision hierarchy, material and user data. Behavior of these objects is provided by the *Abilities* that have access to the data stored in the EnvironmentObject.

Global rendering architecture is based on graphic patterns which combine local and global illumination models [Ošlejšek and Sochor 2003]. For the rapid rendering of a scene, VRECKO uses OpenSceneGraph [OSG], an open source 3D graphics toolkit. Written entirely in Standard C++ and OpenGL it runs on many platforms, including MS Windows and GNU/Linux.

4.2 Device and DeviceManager

The *Device* component controls a specific physical device or resource. The data from the component can be obtained through the interface. To manage the devices connected to the system, we use the *DeviceManager*. When we want to connect a new device to a virtual world under construction, we send the request for the device connection. If the device is not already present in the system, the DeviceManager installs the device and calibrates it if necessary. Otherwise, the existing instance of the requested device is used.

4.3 Scheduler

The system runs in discrete time mode. Timing frequencies and respective timing slots are controlled by the *Scheduler* component and may differ for individual components. To update any component inside the system with a certain frequency, scheduler must register such components. It is also possible to set the update priority. The scheduler decides the order of the component updates using the priority evaluated in each frame.

If the specified update frequency of a Device component is much higher than the frame rate, the Devices can choose to run in a separate thread. We exploit this possibility e.g. for force-feedback devices. In addition to that, it is possible to synchronize the updates with the frame rate, which we use whenever we want to update some components with the same frequency as the graphics rendering.

4.4 Object Behaviour

A behaviour of a virtual object can be dynamically modified using *Abilities*. We can add a new instance of ability to a given EnvironmentObject, or remove or replace an existing abilities. The Abilities are stored in dynamic libraries (plug-ins). The ability is identified by its name and the name of the plug-in in which it is placed. The component has access to the functions and data of its owner EnvironmentObject.

An Ability can perform tasks in two ways: In the *processEvent* function by responding to a received event or in the *update* function that is periodically called without the necessity of any outer signal. These two approaches we can combine.

4.5 Event and Request Model

The mechanism used for the communication is maintained by the *EventDispatcher*. The components can communicate with each other using events and requests within the system. The difference between the two we will describe in the following subsections.

4.5.1 Event Scheme

The communication mechanism uses a similar approach as in [Oliveira et al. 2003]. The components can communicate directly or generate events for connected components. The *EventDispatcher* queues input events and messages and distributes them to the relevant receiving components.

Invoked events flow through these channels. When a new event is created and sent to the event dispatcher, the dispatcher must find according interconnection and send the event to the receiving component(s).

We allow two ways of communication between components. When sending events through the event dispatcher we can choose the *forwarding output* method or the *activating input* method (Figure 3).

When forwarding output, the event dispatcher receives an event and delivers it to a given component intact. The activating input method means that if a specified event is generated at a given output, it is replaced by a predefined event and sent to a given input. As an example, we can choose a keyboard and a light components. When we press a key on keyboard, this component generates an event with the character of the pressed key. This event is of a string type, but the light component has an input of a Boolean type. Thus if we want to switch light on/off we must design an interconnection that allows the conversion between these two types.

4.5.2 Request Scheme

The second way components can communicate within VRECKO system is the usage of request. The components are able to request

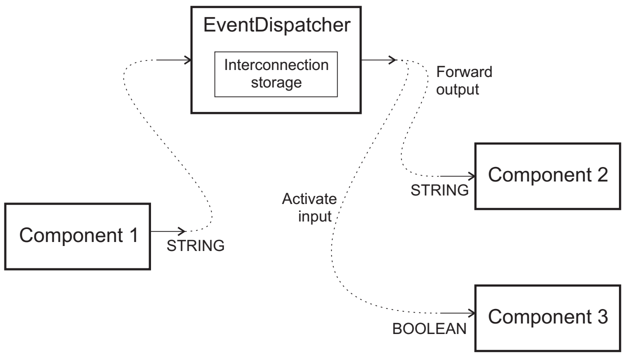

Figure 3: *The scheme of a communication between components; examples of input/output data types.*

data from other components via an interconnection defined between components in the EventDispatcher (Figure 4). In this case, the EventDispatcher only forwards the request. A part of the request is a value that can be used to specify the request.

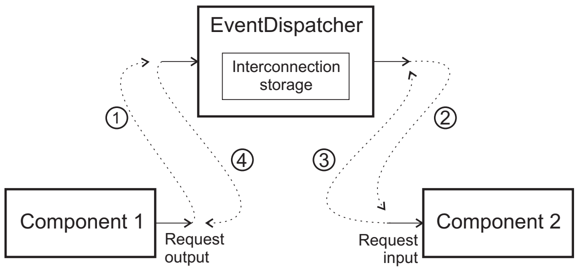

Figure 4: *The communication scheme of a request: four phases are depicted with the 1 and 2 being the asking part and 3 and 4 being the answering part of a request.*

The other benefit of the requests is the ability to specify a *default interconnection* that may be defined in the EventDispatcher. It provides the ability for any component to obtain a specific data without the need to define the exact interconnection. If, for example, one component provides the information about the pointer position, it is sufficient to specify one default request so that any component requesting the position of the pointer will be automatically ”redirected” to this service component. The requests are essentially equivalent to the intermediate call of a function of another component as the result is received immediately.

5 Editing the Virtual Worlds

We have developed an editor application designed entirely as a set of abilities, which allows us easily implement further extensions.

5.1 Abilities Used in the Editor

The abilities used in our system can be arranged according their usage to the following groups:

- Controlling ability
- Editing abilities
- Service abilities
- Placed abilities

The controlling ability group contains only a single ability that controls the main behavior of the application. The editing abilities provide implementation of various editing tasks, such as object movement or animation setting. The service abilities are utilized by other abilities to perform simpler tasks. The placed abilities are assigned to various objects and are intended to work in the resulting application that is being created in the editor. We will describe each group in detail in the following sections.

5.1.1 The Controlling Ability

The core part of the editor is located in the ability named EditorController. This ability tracks the actual state of the editing process and serves as a ”director” who sends messages to other abilities. Every finger press or mouse button click triggers an event message that arrives to the EditorController and the controller immediately forwards the message to the relevant ability.

This ability also includes functions for answering the requests that ask for the currently selected object or the current pointer position. These requests are default requests enabling other abilities to easily query the position or selection, and receive the results. While in theory it may be simpler just to make the default requests operate directly on the device abilities, we made the EditorController to answer the requests because it overcomes the possible different device interfaces and also handles multiple pointers positions distinguished by a single parameter – the pointer identification number (ID). Using the ID in various communication between the abilities offers another advantages, for example, an editing ability can offer only one input for all possible button events and easily handle any such event by examining only the pointer and the button IDs.

We created the EditorController ability solely for the editor application and it contains editor-specific code. Other complex application with context-based behavior will need similar abilities that have the control over other abilities.

5.1.2 Editing Abilities

There are several abilities that were created to work together with the EditorController. This abilities have several inputs and outputs with a specific names that are used to automate the communication. It is sufficient to place such an ability to the configuration file without connecting it to any other ability – the connections will be created automatically by the EditorController or the messages will be sent to the ability directly using the specified inputs.

To detect the abilities that are editor-compatible the EditorController iterates through all the abilities present in the World (that are not owned by any EnvironmentObject) and checks whether a given ability contains the ”GetInfo” request input. In the positive case, it sends the request to the GetInfo and it expects returning of a proper structure. The structure contains the information about the ability behavior, thus the EditorController can activate and deactivate the ability properly.

Other standardized inputs or input requests are designed to inform the ability about button state changes and to activate/deactivate the ability or to request its current state.

A new ability can be created and added into the editor easily. The implementation is straightforward, because developer can derive new abilities from a basic class, which already has the elementary behavior implemented.

The current editing abilities include an ability to move objects across the world, an ability to edit a simple animation of an object or an ability to connect service abilities located in the scene assigned with objects (e.g. to turn on the light when the trigger is pressed).

5.1.3 Service Abilities

The two aforementioned groups of abilities – the controlling ability and the editing abilities –both use service abilities to fulfill simple (and not so simple) tasks. These abilities are highly re-usable and can be used in many other abilities or different applications.

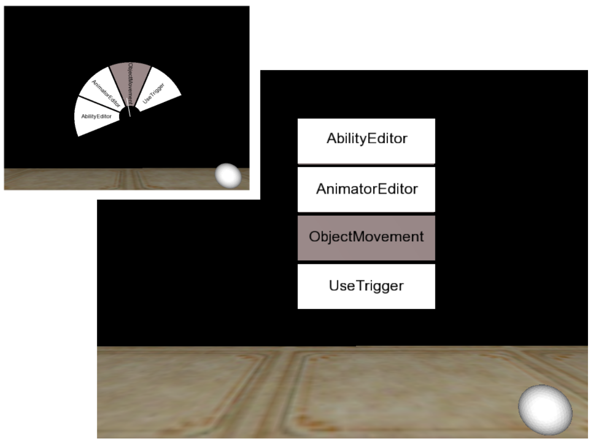

Figure 5: *The DynamicMenu ability, which can be used in any other ability in one of the two variants*

An useful service ability is the DynamicMenu (see Figure 5), which is able to display a specified menu structure, let the user to interact with it and return the item that was selected. Each ability that wants to use the menu has only to create the DynamicMenu ability, specify the menu structure and wait for the event signaling an item was selected.

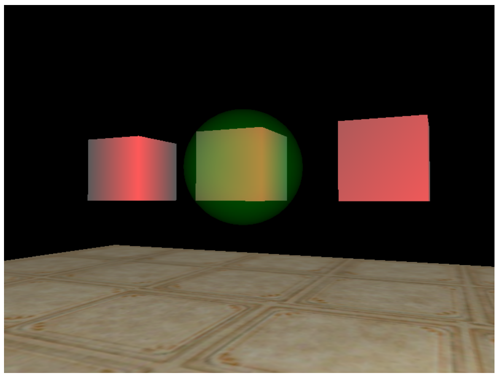

Figure 6: *The ObjectEffect ability adds a graphical effect around the owner object. It is used mainly for the visualization of the selection.*

Another example of a service is the ObjectEffect ability (see Figure 6) that displays an effect around its owner object. This is used to emphasize a selected object by drawing a semi-transparent bounding volume. Similarly, the ConnectionViz ability visualizes all event connections that are connected to its owner object. Working with such abilities is simple as it means only adding its instance to a target object. Add-on functionality is done by the ability.

5.1.4 Placed abilities

The last group consists of the abilities that are assigned to the objects using the editor. They become a part of the final product – the new scene created.

For instance, if the designed behavior is that a trigger will turn on the light, we need to add two different abilities to the trigger and the light object and connect them together. A similar case will be described in more detail in the later section as an example of the usage.

5.2 Connecting the Abilities

The connections between abilities can be categorized by its creator (see the Figure 7 for the illustration). The first group contains connections that represent the heart of the editor. A designer must specify them in the configuration file before executing the editor application. Second group of connections is created automatically by the EditorController ability without the human interaction. The last group of connections contains the connections between abilities in the created scene. These connections are created by some of the editing abilities, usually as a result of a user's action. Besides the listed types there are also service abilities with the connections created by the ability that is using it, but such connections are not significant for the overall picture therefore we will put them aside in this categorization. We will now review the other groups in detail.

The connections created by a human are mainly to route events from the devices' outputs to the EditorController, to enable the controller to track the pointer position and buttons states. These connections are unique to every hardware configuration and it is necessary to adjust them with any change of the input device. There are also another few connections made by a human, which are the default requests for requesting the pointer position or rotation and the current object selection. These are used by other abilities whenever necessary.

After execution of the editor, the EditorController finds all editing abilities through the mechanism described above. The connections between the controller and the editing abilities are then created dynamically according to the current state of the application. Only the ability that is currently activated receives the messages from the controller. The other abilities are unconnected.

The editing abilities are able to change the scene and to add or remove event connections between the placed abilities that are owned by the objects in the scene. It is the equivalent of writing down the connection into the configuration file, but in a much faster, convenient and safer way. To make the editing easier, the editor may visualize these connections.

5.3 Working in the Editor

The editor is primarily focused on editing interactive virtual scenes from the inside using several different input and output devices. This section covers the used hardware devices and the way that a user works in the editor.

5.3.1 Hardware devices

The user can control the application using a number of various devices for the display and navigation. Our preferred combination of the devices, which we will use in the subsequent description of the editor, is the following:

- Large projection screen displaying images from two projectors. The user wears polarized glasses to achieve the 3D view.

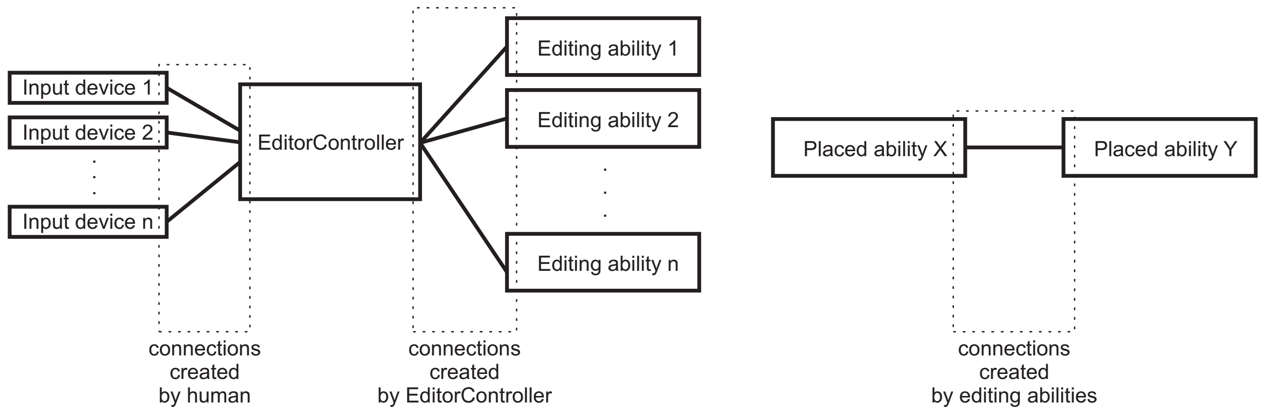

Figure 7: *Main connections between abilities in the editor divided into groups by the creator of the particular connection. Connections between the devices and the EditorController are specified by a human before the application is executed. The EditorController automatically creates the connections to the editing abilities. Placed abilities are connected by the editing abilities usually because of the user's order. We do not show connections to the service abilities, as they do not pose significant contribution to the system overview.*

- FakeSpace Pinch Gloves^{TM} signaling when a user touches finger with another one (see Figure 8).
- Ascension Technologies Nest of Birds^{TM}device tracking the positions of user's hands.

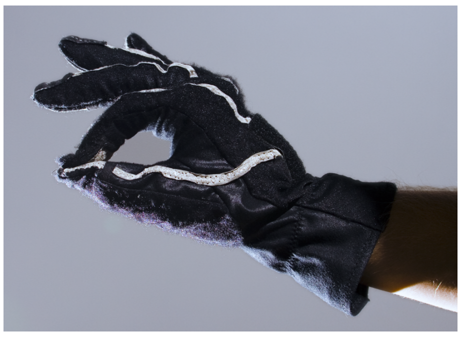

Figure 8: *The FakeSpace Pinch Gloves^{TM}.*

Aside from this equipment, the editor can be just as easily run only on a common desktop PC, using a monitor as the output device and mouse/keyboard combination to control the application. While the ease of use is obviously impaired, we found it very useful option for debugging purposes.

Similarly, we can use different devices mostly only by changing a few lines in the configuration file to re-route the message flow.

5.3.2 The User Experience

At the beginning, the system presents to a user a worldview. Users are wearing the pinch gloves and have eight basic input signals at their disposal. Using the thumb to touch one of the other fingers of the same hand with glove, users invoke signals. It is possible to use other signals such as mutual touch of three and more fingers or touch of fingers of both hands, but currently we restrict ourselves to the basic combinations only.

The user can map some editing ability to any of the eight fingers except the one that is chosen to let the user show the system menu, where the finger-to-ability mapping can be adjusted.

When moving a pointer around the scene the currently selected object is indicated. After touching the fingers in one of the allowed combinations, the editor activates the corresponding editing ability that performs the respective task. After the finger release, the ability is asked to deactivate, but it may refuse if necessary and continue to perform its task (so it may let the user select an item from a complex menu or perform any other action, which requires more than a single press-release activity). The example of the first, simple kind of an ability is the ObjectMovement ability, which moves an object in space. After the activation of the ability, the object starts to move according to the pointer movement and the object movement stops with the ability deactivation. The user may also turn on the snapping to the nearest objects to make positioning simpler. This feature is controlled by the fingers of the other hand.

An example of the latter, more complex editing ability is the ability for the editing of the connection. With the activation of the ability, the first object is selected ("the sender" in the connection). Then the user is expected to move the pointer over the second object, which will act as "the receiver". After the finger release, the second object is selected and the ability remains active. Instead of deactivating, the menu with possible connection options is displayed. The users can select existing connection to delete it or they can create new connection from the list of the available options.

6 Example of Usage

Let us consider a simple real-life example of a task: Two objects in a scene are parts of an interaction. The first object is used to trigger a function of the second object. This might happen when a switch is triggering a light bulb or in a slightly more complex situation when a button causes opening of a car's boot. In the car example if the objects are already in place we only need to add two abilities to the objects, set them up and connect one to another. This could be represented by about 20 lines in the configuration file, but it can be created much more easily in the editor. The schematic view of the used components is presented on the Figure 9.

The first thing that needs to be done is to set up the animation of the boot. This is very simple thanks to the AnimatorEditor editing ability. This ability can set current object position, orientation and scale as a starting or ending transformation of the animation. The Animator ability realizes the animation by interpolating between these extremes to produce a smooth movement over time. The user only needs to position the opening part of the car's body properly and save its state as the starting transformation, then reposition it

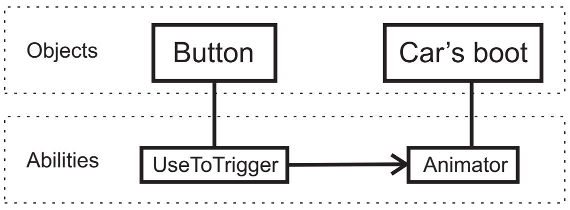

Figure 9: *An example of connecting two objects so that using one object will trigger an animation of the second object.*

and save the transformation again, but as a final transformation.

After setting up the animation, users have to add the UseToTrigger ability to the trigger object and connect its output to the input of the Animator ability. Both of the mentioned actions can user do using the menu evoked by the AbilityEditor editing ability. The UseToTrigger ability transforms the Use signal (emitted when the user interacts with an object) to the Trigger signal (emitted when controlling other objects).

7 Conclusions and Future Work

We have presented the VRECKO virtual reality system that is extensible and allows us to create various interactive applications. To exhibit its extensibility we have described an editor that was created as an ordinary application based on this system. The editor is able to help us with the creation of the new applications or scenes and is easily extensible by writing of new editing abilities.

We designed the editor to be as easy to use as possible to support ordinary tasks. Furthermore, users can edit the application within the virtual environment, which makes the editing more straightforward and allows "novice" users to work in the application only with a short training.

There are a number of possible improvements, mainly in the editor ergonomics, as we would like to make the editing as user-friendly as possible. We intend to implement an advanced snapping algorithms to make the positioning easier (an another approach to constraint-based editing). We would also like to reduce the necessity of the user movements by using object references, which would be available at any time (possibly as analogy of a multiple clipboards).

8 Acknowledgements

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC06008 (Center for Computer Graphics).

References

BIERBAUM, A., JUST, C., HARTLING, P., MEINERT, K., BAKER, A., AND CRUZ-NEIRA, C. 2001. Vr juggler: A virtual platform for virtual reality application development. In *VR '01: Proceedings of the Virtual Reality 2001 Conference (VR'01)*, IEEE Computer Society, Washington, DC, USA, 89.

GROSSMAN, T., BALAKRISHNAN, R., KURTENBACH, G., FITZMAURICE, G., KHAN, A., AND BUXTON, B. 2001. Interaction techniques for 3d modeling on large displays. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, ACM Press, New York, NY, USA, 17–23.

HOLM, R., STAUDER, E., WAGNER, R., PRIGLINGER, M., AND VOLKERT, J. 2002. A combined immersive and desktop authoring tool for virtual environments. In *VR '02: Proceedings of the IEEE Virtual Reality Conference 2002*, IEEE Computer Society, Washington, DC, USA, 93.

HUBBOLD, R., COOK, J., KEATES, M., GIBSON, S., HOWARD, T., MURTA, A., WEST, A., AND PETTIFER, S. 2001. Gnu/maverik: A microkernel for large-scale virtual environments. *Presence: Teleoper. Virtual Environ. 10*, 1, 22–34.

KAPOLKA, A., MCGREGOR, D., AND CAPPS, M. 2002. A unified component framework for dynamically extensible virtual environments. In *CVE '02: Proceedings of the 4th international conference on Collaborative virtual environments*, ACM Press, 64–71.

OLIVEIRA, M., CROWCROFT, J., AND SLATER, M. 2003. An innovative design approach to build virtual environment systems. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, ACM Press, 143–151.

OSAWA, N., AND ASAI, K. 2003. An immersive path editor for keyframe animation using hand direct manipulation and 3d gearbox widgets. In *IV '03: Proceedings of the Seventh International Conference on Information Visualization*, IEEE Computer Society, Washington, DC, USA, 524.

OSAWA, N., AND REN, X. 2003. Gearbox widget for fine adjustments by hand motion. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, ACM Press, New York, NY, USA, 313–314.

OSAWA, N., ASAI, K., AND SAITO, F. 2002. An interactive toolkit library for 3d applications: it3d. In *EGVE '02: Proceedings of the workshop on Virtual environments 2002*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 149–157.

Openscenegraph. www.openscenegraph.org.

OŠLEJŠEK, R., AND SOCHOR, J. 2003. Generic rendering architecture. In *Conference Proceedings 2003 Theory and Practice of Computer Graphics*, IEEE Computer Society.

OŠLEJŠEK, R., AND SOCHOR, J. 2005. A flexible, low-level scene graph traversal with explorers. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics*, ACM Press, New York, NY, USA, 203–210.

TANRIVERDI, V., AND JACOB, R. J. 2001. Vrid: a design model and methodology for developing virtual reality interfaces. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM Press, 175–182.