Components for a virtual environment

Michael Haller, Roland Holm, Markus Priglinger, Jens Volkert, and Roland Wagner

Johannes Kepler University of Linz
Altenbergerstr 69 A-4040 Linz (AUSTRIA)

[mhaller|rholm|rwagner]@faw.uni-linz.ac.at
[mp|jv]@gup.uni-linz.ac.at

Abstract

In this paper we present an approach for the design of a virtual world using a toolkit metaphor, and we show a virtual training environment (SAVE), which makes use of the presented methods. The main idea of the system is a mechanism, which is responsible for the component communication network.

1. Introduction

The design of virtual environments and virtual reality applications is a challenging task. While 2D WIMP interfaces often build on previous experiences with other applications through the use of common interaction elements (widgets) and the building of an appropriate mental model of the application through repeated use, these information sources are often unavailable for 3D interface designers due to the lack of standards. There is little knowledge about *how* virtual environments are designed, what issues need to be addressed, and little guidance about how design should be carried out [Kaur98]. Previous work has mainly focused on technical issues, such as improving runtime performance. By looking for better performance many developers tend to neglect a good design of the application in respect to the reusability.

Essentially, the goal of our project is the development of a generic set of tools that provide users with the means to create virtual environments as efficiently as possible. Our system can be compared with the principle of a toolkit box: the users can pick different components out of a repository and build their own complex environment. We offer a generic definition for a virtual world with our toolkit components. The toolkit metaphor has been used by many authors to suggest parts that fit together and exhibit ease of use [Griss93]. Our description of the virtual environment is based on VRML97, so dependencies can be defined as well. Obviously, the very nature of virtual environments contributes to the difficulty of describing and modelling object dependencies. Typically, a virtual environment consists of static and dynamic components. One of the most important questions of this work arises from the problem of expressing the event driven and visual components.

Several advantages are connected to a component based virtual environment:

- **Flexibility:** All components can be combined in arbitrary order.
- **Reusability:** A component can be used in various configurations.
- **Extensibility:** The system can be extended with new features (e.g. with a moving platform).
- **Communication:** If suitable interfaces are offered, the components can communicate with each other.

A general problem with the construction of virtual environments is the complexity of modelling and implementation. Therefore, different design methods and tools are necessary.

2. SAVE

SAVE (Safety Virtual Environment) is a Virtual Reality based safety training system for dangerous and hazardous facilities. It is a multi purpose virtual reality software system that is mainly intended for employee training and it has been developed for the OMV refinery of Schwechat. SAVE was designed to use HMD technologies and demonstrated the benefits of VR for safety training. The system supports real-time collision detection, simulation of dynamic behaviors of the objects, interactions between the user and the objects. It is controlled by the trainer, who can observe and manipulate objects in the virtual scene, e.g. the trainer changes the state of a valve [Haller99a, Haller99b].

The SAVE system consists of two major parts:

- **The scene simulator:** This application runs on a graphics workstation and creates the user's view of the whole training scene. It also handles all interaction between the trainee and the objects in the scene, like switches, levers,

tools, etc. Since every head movement is tracked, the application renders a new image on every move according to the trainee's viewpoint and orientation, creating a strong immersive effect.

- **The trainer application:** This program is used by the trainer to observe the whole training process and runs on a different machine, linked to the graphics workstation over a network. The trainer controls any part of the scene and reacts to the trainee's actions. Moreover, it allows to manage the trainee's training progress in a database, thereby allowing individual training.

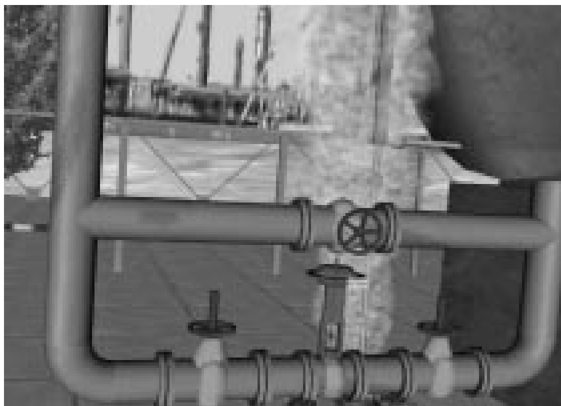

Figure 1: SAVE is a virtual environment for safety training

Each virtual training scenario comprises a scene in which the trainee can move freely and interact with objects like pumps, valves, and other devices. The four main modules of the application are

1. the ***simulation*** (trainee module), which is responsible for the visual VR part and the user input (tracking). This constitutes up the main part of the application and runs on a graphics workstation.

2. the ***trainer control center***, which is used by the trainer to control the scenario and take influence on what happens in the VR world. This is a Java application running on a PC.

3. the ***sound server***, which generates 3D sound depending on the position and orientation of the trainee. This is a DirectX based application using a consumer 3D sound card.

4. the ***HYPF server***, which controls the hydraulic platform. Currently, the platform is under construction.

All four parts can be located on separate computers, although in practice we use only two (see figure 2): the graphics workstation for the simulation and a PC for the remaining modules.

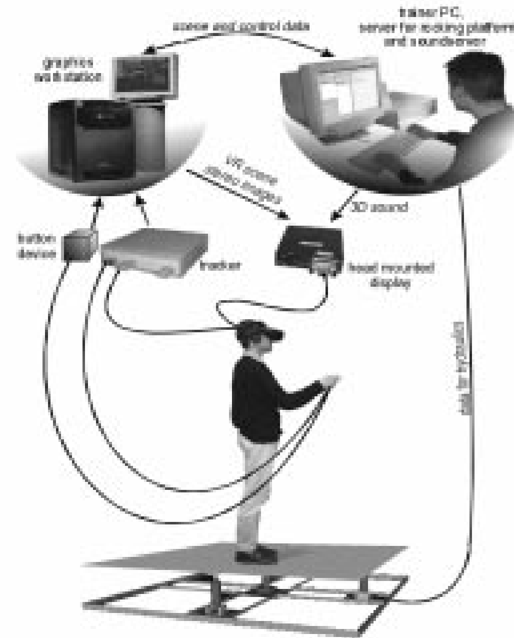

Figure 2: The topology of SAVE

3. Architecture

We used an object-oriented design to make the system more flexible, more understandable, and extensible. The architecture has an event driven design, i.e. components start the information processing after receiving an event. After processing, the components can generate new events and the processing can be done in parallel. The system resulting from this approach is more robust and easier to understand.

Figure 3 shows the different modules of our virtual environment system. Starting from the description files of the repository, users can select objects and build a virtual world. The system consists of the following layers:

- Repository
- Repository Manager
- Editors
- Scenario description files
- Virtual world simulation

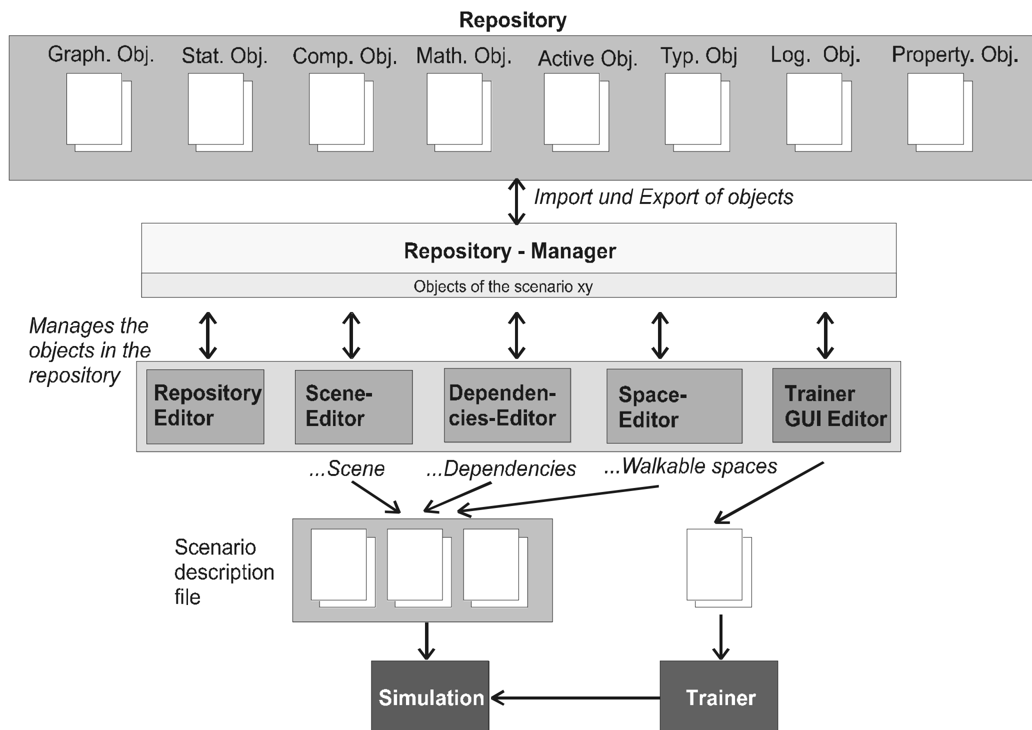

Figure 3: The architecture of SAVE

3.1. Repository

All objects used for describing a scenario are stored in the repository. It contains different types of components: **Graphical components** represent graphical objects in the virtual world, **logical components**, e.g. AND, OR, NOT, etc., allow a connection of the different objects. **Mathematical components**, e.g. sqrt, pow, etc., are used for mathematical functions, **active components**, e.g. clock, timer, etc. are not triggered by other components but rather generate event sources, **property components**, e.g. clickable objects, transportable objects, etc. describe the behavior of objects, and **compound components**, like pumps etc. consist of more basic components.

3.2. Repository Manager

All editors do not access the objects in the repository directly, but through the repository manager. This insures, that the objects do not have to be managed separately in each module.

3.3. Editors

Instead of having one tool to generate a virtual environment, we base our system on several independent tools which however are combined and communicate with each other.

Five different editors are planned for modelling the virtual world:

- Scene Editor
- Dependencies Editor
- Space Editor
- Trainer GUI Editor
- Repository Editor

Authors use these editors to specify the scene description. The **Scene Editor** will make it possible to build new scenarios by placing objects, props, buildings, and pipes in the scene. With the help of this tool the author is able to place objects, rotate and scale them to match the desired layout. After finishing the placement, the objects' dependencies can be defined. With the **Dependencies Editor** dynamic objects can be connected by plugging their slots together. The author can define walkable and non walkable areas, by using the **Space Editor**. Finally, an external module allows controlling the scene. Its graphical user interface and its functionality will be created with **the Trainer GUI Editor**. The **Repository Editor** allows a visualization of the compound objects in a tree widget. These editors provide an easy to use application for modelling virtual environments.

The modular approach allows the exchange of component properties during the modelling task. Time and cost efficiency will be increased through the scalability and flexibility of the concept

presented. However, the biggest increase in efficiency is due to reusability.

3.4. Scenario description files

The result of the modeller for the virtual environment are scenario description files, which describes the objects and their properties for the actual scenario. Starting from these files, our system generates the virtual world, including the simulation and the trainer module.

In our virtual world all the components are described by prototype nodes based on VRML97, which helps the user to manage scene complexity by providing a method for defining higher level objects. All components or nodes have input slots, output slots, and parameters, which allow a closer description of the object. Default values of the prototype are used, if no explicit values have been supplied.

```
DEF pushbutton VRClickable {
   children [
      DEF buttonSwitch VRSwitch {
         children [
            Inline {...}
            Inline {...}
         ]
      }
   ]
}
DEF intCaster VRIntCast {}
DEF theCounter VRCounter {
   limit 1
}

...
ROUTE pushbutton.clicked TO intCaster.in
ROUTE intCaster.out TO theCounter.add
ROUTE theCounter.value TO
         buttonSwitch.setChoice
```

An example for the combination of several smaller objects into a more complex object can be found above: a pushbutton consists of two separate geometries: one for the up-state (not pushed) and one for the down-state (pushed). Only one geometry is visible at any time. If the user clicks (i.e. pushes) the button, it remains in down-state until the user clicks it again. To build this pushbutton, the two geometries (imported as "Inline" files) are grouped under a switch node. The switch node itself is a child of a VRClickable node, which makes it (i.e. its geometries) sensitive to clicks by the user. If the user clicks the button, VRClickable generates a boolean event, which is routed through a VRIntCast node to cast the value to integer. The integer value (0 or 1) is added to a counter value and the accumulated value is finally sent to the switch object which selects the respective subnode. Because the counter has a limit of 1, it only keeps values modulo 1 (0 and 1), which corresponds to the selectable subnodes of the switch node. Therefore, successive clicks switch between 0 and 1 - while an event with a boolean false (generated when a selection ends) is ignored.

Once a component has generated an event, the event is propagated from the eventOut slot along any route to other nodes. These nodes may respond by generating additional events, continuing until all routes have been honored. Event notifications are propagated from sources to listeners by the corresponding method invocations on the target listener objects. Each event source can have multiple listeners registered to it. Conversely, a single listener can register with multiple event sources.

Our system is based on the producer/consumer concept, i.e. the nodes process a message only, when receiving a message of the other nodes. In other words: No component (consumer) starts sending messages and processing information without having received a message from other nodes (producer). The only components, which generate messages are called active components. But they are triggered by system calls and do not start by themselves sending messages. By routing events from the output slots to the input slots of another node, customized functionalities and dependencies can be realized, e.g. if a switch has been switched on, a lamp lights, etc.

For the sake of being independent of a certain graphics library (e.g. Optimizer, Performer, etc.), we encapsulate direct calls to the actual graphics library in certain graphical classes. Instead of creating the scene graph using library-specific calls we build a meta scene graph. Figure 4 shows three graph types. The leftmost graph structure presents the VRML97 data structure generated by the parser. Note the thick black lines connecting the nodes. These lines represent parent/children relationships. Although the nodes **VRIntCast** and **VRCounter** are not involved in any parent/children relationship, they are stored in memory for further processing, but are actually not part of the VRML97 scene graph.

The graph structure in the middle of the illustration constitutes the meta scene graph. This structure is derived from the VRML97 scene graph. Again, thick black lines represent parent/children relationships. Note that **VRIntCast** and **VRCounter** are connected by directed dotted lines. These arcs represent the component communication network. **VRClickable** sends the **clicked**-event to VRIntCast which sends an **add**-event to **VRCounter** after processing the **clicked**-event. Finally, after processing the **add**-event, **VRCounter** sends the **setChoice-**event to **VRSwitch**, which will select the appropriate **VRGeometry**.

The right data structure is the scene graph of the graphics library which contains group-, transformation and geometry nodes. Note that the component communication network is **not** part of this scene graph.

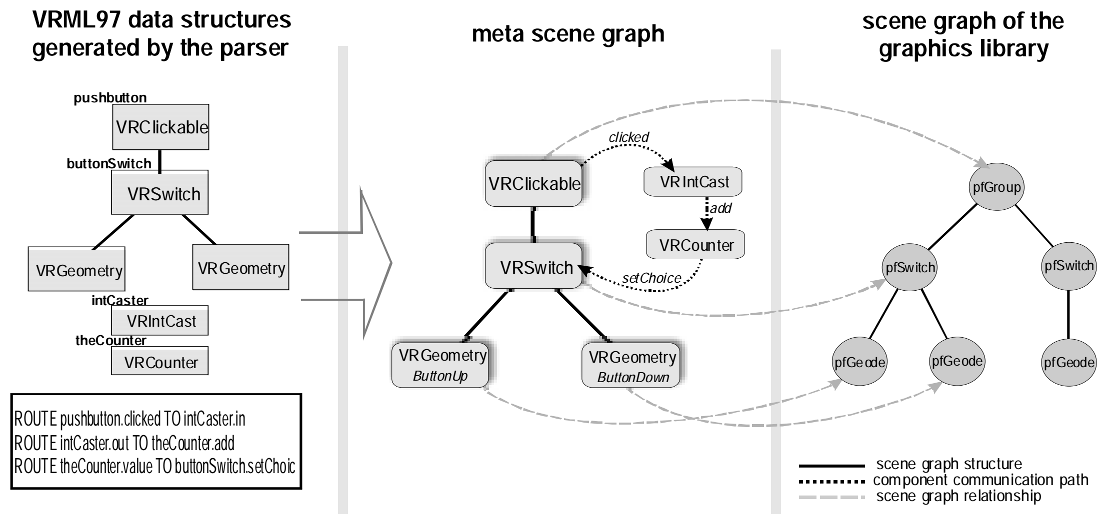

Figure 4: From the data structure through the meta scene graph to the scene graph of the graphics library

Building the meta scene graph does not require any knowledge of the underlying graphics library. Graphical classes can easily be replaced by classes which encapsulate calls to an alternative graphics library. No further changes in the source code are required. The mapping of the VRML information to the actual scene graph structure is performed in the two steps. The parser idientifies VRML nodes and creates a corresponding VRML node representation. Having parsed the last node, the scene description is available for further processing. The VRML node tree is traversed recursively and each node is converted to its corresponding meta scene graph node representation. The result is the meta scene graph. After a second traversal the component communication network is established, and finally, the VRML node tree is deleted.

4. Conclusions

As Smith, Duke, and Massink [Smith99] note, virtual environments are a mix of continuous and discrete components. What has been presented in this paper is an initial research for a framework with components for a virtual environment. The results have been realized in a refinery virtual environment, which used the different components. The next steps of the project is to define tools, which provide a very user friendly interface for assembling the components to a virtual environment. There are planned tools, where the user can choose of a huge pot of graphical and logical components which can be connected together. Doing so, the user creates his own virtual environment, which can be used for a new safety training.

Acknowledgments

Many people have made important contributions to SAVE. The authors wish to thank all the members of the SAVE team, including Anton Dunzendorfer, Uwe Pachler, and Gernot Schaufler. This project was sponsored by OMV, Austria’s biggest oil refinery. Therefore, we wish to thank Alois Mochar and Franz Grion.

References

[Griss93] M. L. Griss, *Software reuse: From library to factory*, IBM Systems Journal, Vol. 32, No. 4, ISSN 18-8670, 1993.

[Haller99a] M. Haller and G. Kurka and J. Volkert and R. Wagner, *omVR - A Safety Training System for a Virtual Refinery*, Proc. of ISMCR'99, Topical Workshop on Virtual Reality and Advanced Human-Robot Systems Vol. X, pp 291-298, Japan, Juni 1999

[Haller99b] M. Haller and R. Holm and J. Volkert and R. Wagner, *A VR based safety training system in a petroleum refinery*, Proc. of Eurographics'99, 20th Annual Conf. of the European Association for Computer Graphics, pp 5-7, Mailand, September 1999

[Kaur98] Kulwinder Kaur, *Designing Virtual Environments for Usability*, Centre for Human-Computer Interface Design, 1998, 241 pages.

[MDLS98] W. Müller, R. Dörner, V. Luckas, A. Schäfer, *An Efficient Object-Oriented Authoring and Presentation System for Virtual Environments*. In Proceedings of GraphiCon '98, pp 111-118, ISBN 5-89209-294-1, Moscow, Russia, September 1998.

[Smith99] Shamus Smith and David Duke and Mieke Massink, *The hybrid world of virtual environments*, Proc. of Eurographics'99, 20th Annual Conf. of the European Association for Computer Graphics, pp 297-307, Mailand, September 1999