A Framework for Realizing Multi-Modal VR and AR User Interfaces

Wolfgang Broll, Irma Lindt, Jan Ohlenburg, Axel Linder

Collaborative Virtual and Augmented Environments Department
Fraunhofer-Institute for Applied Information Technology (FIT)
Schloss Birlinghoven, 53754 Sankt Augustin, Germany
{wolfgang.broll, irma.lindt, jan.ohlenburg, axel.linder}@fit.fraunhofer.de

Abstract

The development of Virtual Reality (VR) and Augmented Reality (AR) applications still is a cumbersome and error prone task. While 2D desktop applications have very clear standards regarding user interfaces and a very limited set of common input devices, no standards for VR/AR applications yet exist. Additionally a wide range of different 3D input devices exists.
In this paper we will present our framework for realizing multi-modal VR and AR user interfaces. This framework does not rely on a single mechanism, but provides a set of three individual mechanisms complementing each other. We use an interface markup language for defining cross-platform user interfaces, object behaviors for rapid VR/AR interaction prototyping, and object-oriented scene graphs for realizing complex application scenarios.

1 Introduction

While we can find a widespread use of Virtual Reality (VR) and Augmented Reality (AR) environments today, compared to 2D environments, there is still a significant lack of common user interface mechanisms or input devices. Thus appropriate tools and frameworks are missing, while user interfaces are rather based on best-practice experiences and individual application requirements (Bierbaum, Hartling, and Cruz-Neira, 2003). This usually results in a large programming effort on realizing application specific solutions, making them actually usable and adapting them to the user needs. Additionally, even minimal further development of existing software or adaptations to similar, but new environments often turns into a nightmare! Thus, applications and the corresponding user interfaces are quite often re-build from scratch rather than evolved from existing solutions.

In this paper we will not provide a general solution to overcome this problem, but present our approach to provide device independent and flexible user interface descriptions, and to deal with the necessity to evolve and to redesign VR/AR user interface mechanisms in multiple design cycles. We will present three different mechanisms provided by our framework to describe and model multi-modal user interactions rather than programming them: a user interface description language, allowing for device independent descriptions of VR/AR user interfaces, a scene-internal interaction prototyping mechanism, for immediate evaluation of alternatives, and a general object modeling language, allowing to model rather than to program application, while hiding system specific aspects. We will show how they support the development and implementation process, and compare their individual application or usage areas. The different approaches are supported by our distributed multi-user VR/AR framework Morgan (Ohlenburg, Herbst, Lindt, Fröhlich and Broll, 2004), which has already proven its suitability for the realization of AR applications and 3D user interfaces in various projects.

The structure of this paper is a follows: in the second section of this paper we will review related work in the area of tools, frameworks, and applications for supporting the realization of user interfaces for VR and AR environments. In the third section we will introduce our approach on a user interface markup language supporting VR/AR environments in addition to traditional 2D desktop user interfaces. In the fourth section we will present our approach on interaction modeling by behavior objects, which is primarily used as an interaction prototyping mechanism. In the fifth section we will present an integrated object-oriented scene graph approach, introducing a completely new approach to VR/AR application and user interface realization. Finally we will discuss the three different approaches in section five before finishing with a conclusion and look into the future work.

2 Approaches for Realizing Multi-Modal VR and AR User Interfaces

A straight-forward approach to realize multi-modal VR and AR user interfaces is to select appropriate interaction devices and interaction techniques for a specific application and to implement the user interface with a standard programming language and an API of the target windowing platform. This allows for using specific functionality of the employed interaction devices and the platform and for controlling every aspect of user interface. A drawback of this approach, though, is that it is rather cost-intense, error prone, requires expert knowledge and does not allow for an early evaluation of the user interface, which may lead to user interfaces that do not sufficiently meet user requirements.

To avoid implementing user interfaces from scratch for every VR or AR application, a variety of software toolkits and frameworks have evolved that facilitate the development of VR and AR applications and their user interfaces (Schmalstieg, 2000; Bauer, 2001). The Studierstube system supports the Personal Interaction Panel – a light weighted panel augmented with 3D widgets. The 3D widgets such as buttons or sliders are described as an Open Inventor scene graph. The DWARF framework supports the description of multi-modal interactions as petri nets (Hilliges, 2004). These approaches reduce the required complexity for realizing VR or AR user interfaces and allow for reuseing user interface code.

Many VR and AR toolkits and frameworks employ device abstraction to allow for a flexible exchange of interaction devices without the need to recompile the application. The OpenTracker library (Reitmayr, 2001) included in the Studierstube system implements a device abstraction layer and includes functionality to process tracking data using geometric transformations, Kalman filters or data fusion. VRPN (Taylor, 2001) implements a network-transparent interface between application programs and physical devices and allows for a dynamic discovery of interaction devices. Device abstraction is not a means for realizing user interfaces, but rather an additional software layer between devices and frameworks. Typically, with abstraction layers devices that offer similar input data or that require similar output data can be exchanged transparent for the VR/AR application and its user interface.

Approaches to a more flexible support of interaction devices beyond device abstraction are based on abstract interaction techniques. One example is SliVR that fits semantic events, such as navigation tasks, to available interaction devices (Renzulli, 2003). STARS allows for device independent formulation of interaction requests (Magerkurth, 2004). Abstract interaction techniques support a flexible exchange of interaction devices even if the employed devices do not provide similar input or output data.

Authoring tools for VR or AR user interfaces focus on an easy and rapid creation of user interfaces. GRAIL allows experienced users to configure and adapt the user interface of the AR environment (Penn, 2004). It is e.g. possible to specify how to manipulate virtual objects using multi-modal interaction techniques such as tangible user interfaces or speech input. DART (MacIntyre, 2004) offers designers means to set up and configure an application using scripts that can be modified by users to include content such as video, sound, and 3D models into the AR application and to program the user interface. The furniture assembly instructor (Zauner, 2003) is an authoring tool that is built upon the AMIRE framework (Dörner, 2002). It uses flow chart diagrams to visualize the steps required to assemble 3D objects. The authoring tool supports 3D widgets and tangible user interface elements.

User interface description languages (UIDLs) describe user interfaces independent from underling devices, platforms and modalities to provide portability of user interfaces and to ensure universal usability. The User Interface Markup Language (UIML) (Abrams, 1999) is an example of an XML-compliant user interface description language. UIML aims at a universal, modality-independent interchange format that can represent potentially any user interface. Further examples for XML-compliant UIDLs are XIML, AUIML, Seescoa XML, XForms, XUL and AIAP. A thorough review of different UIDLs can be found in (Trewin, 2003) and (Souchon, 2003).

Behaviour languages aim at a simplified and rapid development of VR/AR user interfaces. They do not require in-depth programming skills and can be employed by non-programming experts. Examples can be found in (see e.g. Zachmann, 1996) and (Daubrenet, 2000).

Object-oriented modeling languages are based on the object-oriented design paradigm and support code reusage and focus on an easy and rapid creation of user interfaces. An example is VRML++ (Diehl, 1997) that extends VRML'97 with object-oriented features such as classes, inheritance, dynamic binding and polymorphism . In the object-oriented language VRML++ prototypes are used for class definition, nodes are objects, events and script nodes are methods and fields are variables, since VRML'97 lacks inheritance, the syntax is extended and a pre-processor converts the extended syntax into the standard VRML'97 syntax.

3 User Interface Definition Language

User Interface Definition Languages (UIDLs) aim at an automated generation of user interface code, based on a common representation for different interaction devices such as desktop and mobile PCs, PDAs, and cellular phones. UIDLs are meta-languages, which are based on text-based descriptions (UTF 8), e.g. XML. The idea of UIDLs is to provide a simple possibility to define user interfaces by specifying their attributes within a single document. The advantages of using text-based languages (therefore of UIDLs as well) are the platform- and device-independence, that they are easy to use to learn, and their extensibility. UIDLs can be adapted to define user interfaces either for special or for general purposes. A UIDL can be compared to a skeleton, which only represents the main structure of a user interface description. To be used, a UIDL additionally needs a vocabulary specifying its purpose and capabilities. We developed MRIML (Mixed Reality Interface Markup Language), a vocabulary especially created to support VR and AR user interfaces. We further developed a component for an automated generation of specific user interfaces from MRIML-documents.

3.1 Mixed Reality Interface Markup Language (MRIML)

The most important aspect of MRIML is its capability of representing both – common interaction techniques of WIMP-based User interfaces and VR/AR user interfaces. Further the description does not assume any specific information about the platform or the device it is deployed to. Another aspect is, that the way input and output is provided, does not have to be considered within the description. This enables the exchange of input and output devices and the use of multimodal input and output techniques.

The vocabulary is a set of elements (widgets), which can be divided into four logical categories: Containers, Controls, Listeners and Layouts. *Containers* group other elements, while *Controls* are elements for direct interaction. Elements of the category *Listener* are used for user specified behavior and elements of the category *Layout* influence element alignment, or specify their appearance. Table 1 provides an overview of the MRIML elements.

Category	**Container**	**Control**	**Listener**	**Layout**
Elements	Frame Menu Sequence SelectionSet	Button Label Artifact TextInput Slider	FocusListener ChoiceListener PoseListener TextListener	Color Alignment Size TextFormat

Table 1: Overview of MRIML elements

Each element's specification is encapsulated in the two sections attributes and children. The section attributes specifies the appearance and the behavior of each element. The section children specifies which other elements can be contained in the specified element, and represents the guidelines of usage. The control element *Button* for instance specifies the attributes *position, orientation, description, bindingType, boundTo, isVisible, id, media, mayBePhysical, isEnabled*, and *type*, and allows for the children *Label, FocusListener, ChoiceListener, Layout.*

Listing 1 shows the description of the *Button* in a MRIML-document:

```
<part class=”Button” id=”numericalValues”>
      <style>
            <property name=”type”>pushButton</property>
            <property name=”position”>0.0 0.0 0.0</property>
            <property name=”orientation”>0.0 0.0 0.0</property>
            <property name=”isVisible”>false</property>
      </style>
      <part class=”FocusListener” id=”numericalValuesFocusListener”>
      …
      </part>
</part>
```

Listing 1: Example of the specification of a *Button* in MRIML

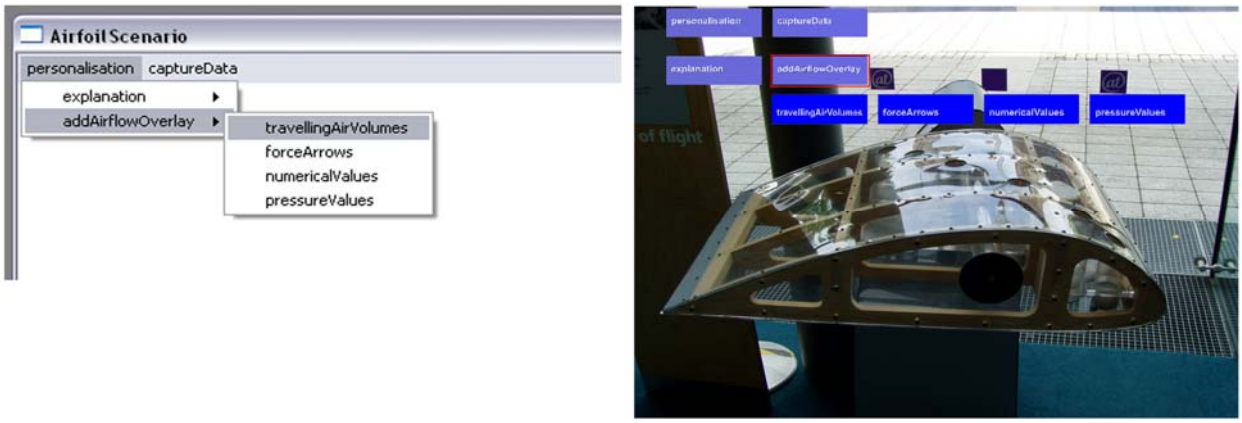

Figure 1: Resulting Graphical- and Augmented Reality user interfaces

The elements and their definitions encapsulate the mechanisms for describing WIMP-based and VR/AR interaction techniques. The attribute *position* e.g. is not limited to two dimensions, but may contain 3D location information. Combined with the attribute *orientation* each element can be spatially aligned. The following examples demonstrate the use of MRIML to implement some well known VR/AR interaction techniques.

Tangible user interfaces can e.g. appear as real placeholders, which are connected to virtual elements. If the user moves the real placeholder, the virtual element is moved as well. In MRIML an element may have a *PoseListener*, which reacts on changes of position and orientation (pose). In case of tangible user interfaces it reacts on changes of external entities, e.g. the pose of the real placeholder. The developer simply has to define the virtual element by specifying its attributes and to add a PoseListener as a child of the element. The runtime environment has to connect the real placeholder to the specified element via its unique id. After this procedure the virtual element reacts on pose changes of the real placeholder.

The attribute *mayBePhysical* is a flag which shows, that a certain element, e.g. a button, may actually be real, if there is a corresponding hardware device. In AR it is quite common to combine real and virtual interaction elements within a single user interface. In this case, the specified element will not be rendered visually and the real interaction element behaves like the specified element.

An *Artifact* is an element used for representing elements, which are not part of the user interface itself, but which are influenced by elements of the user interface. This enables the specification of the behavior of external virtual elements which have to react on actions of the user interface. The developer has then the possibility to define the behavior of an external element within the description of the user interface. The advantage is that the specification is as easy as with the other elements of the user interface.

3.2 Usage of MRIML

Using MRIML requires the definition of the user interface elements within a UIDL document using the MRIML vocabulary. After the document is composed properly (valid XML and MRIML), the document is passed to the rendering unit, together with the vocabulary itself. The rendering unit consists of a set of individual renderers – one for each target platform. After parsing the document, the individual renderers create the user interfaces for the individual platforms according to the overall description (see Figure 2).

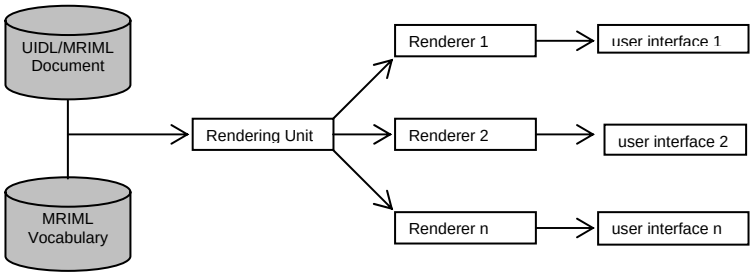

Figure 2: Overview of the usage of a MRIML-Document

The main benefit of MRIML is its ability to use a single user interface description for both graphical- and AR user interfaces without the necessity of programming. Figure 1 shows the two rendering results of the sample scenario.

4 Interaction Prototyping

For easy evaluation of multi-modal user interfaces we use a scene-internal rapid prototyping mechanism. This mechanism allows us to define object or scene behaviors such as user interactions or animations within the actual scene description. Behavior objects represent an object-oriented approach to describe such functionality. They are assembled from a set of basic components used for reacting on user input, querying additional information, providing awareness on input streams or other scene graph objects, or applying changes to the environment. Additional components provide basic programming facilities such as checking conditions, executing statements, time-dependent execution, or even scripting. Thus switching to another input device may simply be realized by changing the registering to the appropriate input event of this device. Scene graph related queries such as picking or collision detection allow for easy modelling of user interactions. Additional support is provided for shared scene graphs in distributed multi-user environments. Figure 3 provides an overview of the behavior component class hierarchy.

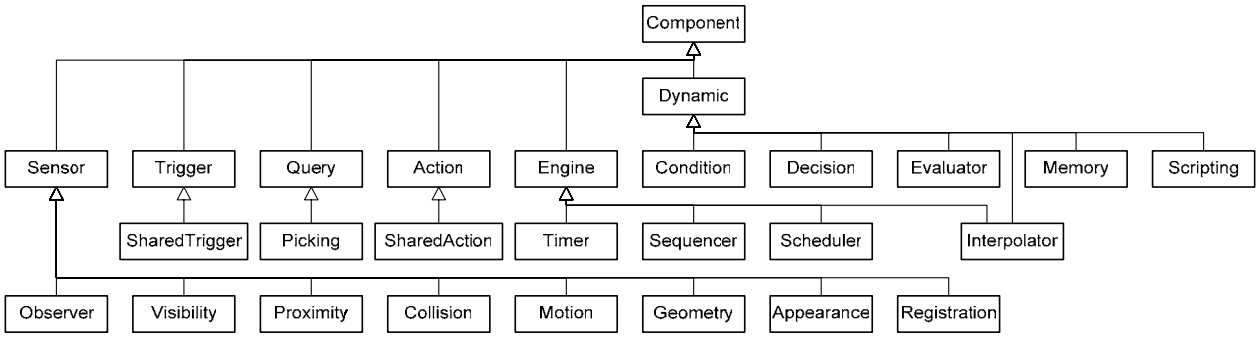

Figure 3: Behavior component class hierarchy

Let us start with a simple example on how to use behaviors. In order to keep it simple we use a desktop VR example, where a user wants to drag and drop a teapot using the mouse pointer. First of all we have to detect the user's action, i.e. selecting and clicking on the 3D object. Events created upon mouse actions already provide an identifier of the object at the mouse pointer's position. We could realize the drag & drop behavior as a global behavior (applying to arbitrary scene objects), or as a specific behavior of a particular 3D object (i.e. a certain part of the scene graph hierarchy). As we do not want to make all objects in the scene graspable, we use the latter alternative. In this case, the behavior object defined will be attached to the teapot object and we may use *Trigger* components to detect and react on mouse events. Trigger components handle events sent to the object the behavior is attached to. In order to drag and drop the object we have to evaluate the mouse button actions. This could be done within two individual Trigger components (one accepting mouse button down events and one for mouse button up events). However, we will catch all mouse events and decide within an additional *Condition* component. In order to do so, we have to define appropriate data fields there and connect the incoming event fields to those fields (see Figure 4). Also the Trigger component has to signal the Condition when to evaluate the conditions. Depending on the result of this evaluation, the teapot is then moved according to the mouse pointer movement or released again. This is realized using an *Action* component. Action components emit events. Here we use the Action component to send the translation values to the appropriate object. This is realized by defining an appropriate output event (transformation), supplying its translation value from the mouse event by connecting both, and finally executing the Action component (i.e. sending out the defined event) after the mouse button was pressed. Now we are almost done. However, we want the object to follow the mouse pointer while the button is pressed. Thus we additionally have to catch mouse motion events. We can do this using an additional Trigger component, which is activated on the mouse button down event and deactivated when releasing the button. Again the mouse position data is forwarded to the emitted event.

4.1 Basic Behavior Mechanisms

As described in the example, behaviors communicate with their environment by events. Thus they represent pretty autonomous objects, which receive and send events. Events may be received by chance, after a previous registration, or as a result of a query. Outgoing events are used to distribute and to apply the results of the behavior to scene graph objects or to system components. Scene graph objects are able to catch specific events. However, if there is at least one behavior object attached to this scene graph object, the event will be forwarded to this behavior first and only be applied to the scene graph object, if not consumed by the behavior. Events, which can not be processed locally, are forwarded to the parent in the scene graph hierarchy.

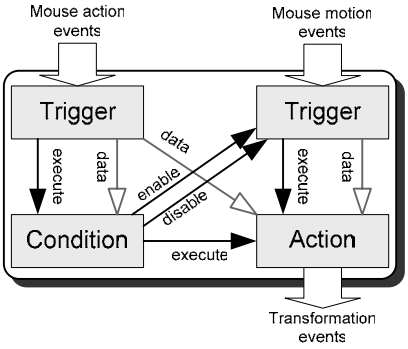

Figure 4: Simple drag & drop behavior

In order to react on event input, we use *Trigger* and *Sensor* components. While Trigger components may only handle events sent to the object the behavior is attached to, Sensor components can be used to register interest for particular events or specific conditions in other scene graph or system components. Thus Trigger components are usually used to handle specific events sent by other behaviors, or to deal with system events, which are sent to particular objects by default (such as the objects at the mouse pointer location). Dedicated Sensors exist to react on particular conditions and changes. Most of these Sensors are used to detect changes in the scene graph representation of an object: *Motion* components detect transformation changes, *Geometry* components react on geometry changes, and *Appearance* components trigger modifications of the visual appearance (color, material, transparency, texturing, etc.). *Observer* components are used to react on specific changes within scene graph objects. Thus it would even be possible to simulate an Appearance or Motion type component behavior by surveying the appropriate data fields within the scene graph. Additionally, *Collision* components allow to detect object collisions , *Proximity* components react on distances between objects and *Registration* components allow the registration of system events (e.g. to receive events from a 6-DOF tracking device).

Action components issue events from a behavior. They always specify a specific event type to be distributed upon their execution. Thus, while this event type is fix for the life time of the Action component, the data sent within the event is varied according to some input or the result of a calculation.

Within behavior objects, components communicate by data and signal connections. Data connections allow to forward data from one component data field to that of another component, whenever the source data field is modified. This includes data from and to data fields of events caught (Trigger, Sensor) or prepared for sending (Action). While data connections are actually independent of the execution of a component, some data fields are typically set only during an execution. Data connections provide implicit type conversation between input and output data fields. Signals are used to specify the execution order of the individual components. Each component provides several input and output signals. All components provide enable and disable input signals to activate and deactivate the component. Trigger and Sensor components for instance issue a signal upon successful reception of an event. In the example above this signal was used to execute the Condition component and evaluate the mouse button state. While the transfer order of data connections is arbitrary, the execution order of signals is significant. Many components for instance disable themselves after successful execution, but will usually issue another signal before this to further process the result. It is obvious that an arbitrary execution order would not guarantee the intended results.

4.2 Data Processing

Quite often the execution of one or several components depends on certain conditions, i.e. on the state or value of one or several data fields. All components provide simple condition checks before execution. These conditions are checked when receiving an appropriate input signal or an external event. If the conditions are not met, the component will not be executed. Additional components such as *Condition* and *Decision* allow to specify individual data fields and to apply conditions on these fields. While the Condition component supports simple conditions on these fields only, the Decision component allows even the evaluation of complex mathematical expressions.

While some object behaviors such as the drag & drop example already receive all information required from the incoming events, other require querying specific information from other scene graph objects or the system. This of

course also requires the communication by appropriate events. *Query* components are used to specify, set-up, and send such query events to outside destinations. Events containing the query results will then also be caught be the Query component, which will issue a signal upon reception. The *Picking* component is a specialized Query component, which allows picking objects from a point a space along a ray into a specified direction.

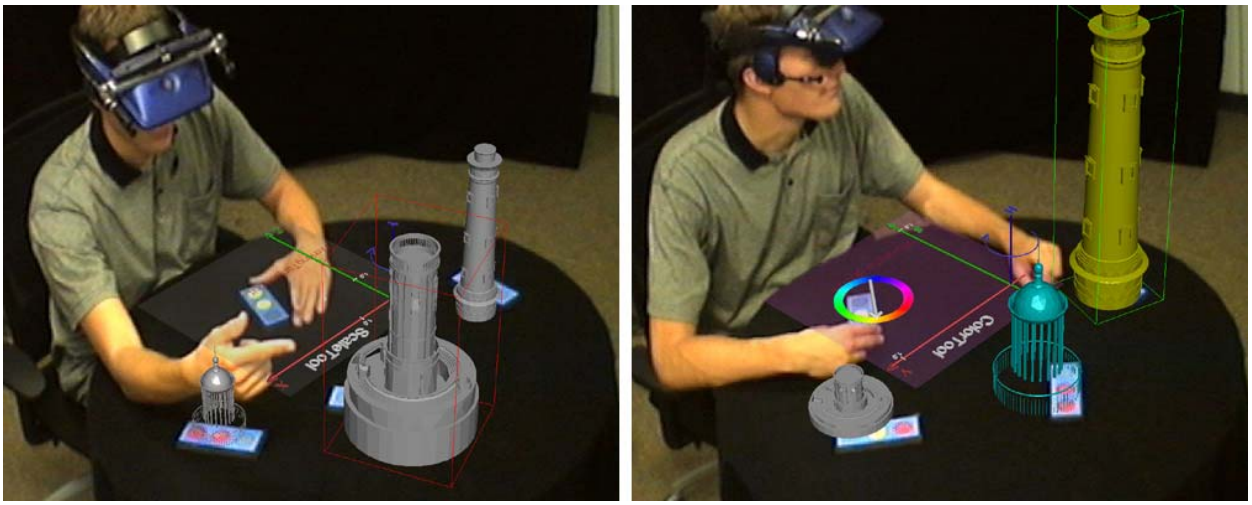

Figure 5: Example of 3D modeling environment realized by behavior objects

Many behaviors also require certain calculations, e.g. to provide a mapping between input and output values. To support this, the *Evaluator* component allows specifying arbitrary mathematical expressions, which are evaluated upon individual signals. *Memory* components allow influencing reading and writing of data fields by signals. This is typically used to reset or restore information under certain conditions. For more complex data processing *Scripting* component allow the execution of JavaScript code within behavior objects. All these components exchange data via user defined data fields available in all dynamic components. Figure 5 shows a user interface for 3D modeling created using these components.

4.3 Time dependent behaviors and animations

Object behavior does not only depend on user input or external states, but is often directly depending on time. *Engine* components are used to support this kind of object behavior. The *Timer* component issues output signals each frame or in specific intervals within a specified period. This for instance provides the basis for animations or time-out behaviors. *Scheduler* and *Sequencer* components support the creation of a series of signals each at a specific time or after a particular period.

Interpolator components are used to map a fraction or time value to a key frame value and by that select a interpolated output value. The type of the output values may be individually specified within the component. Interpolator components can either be triggered by signals issued from a Timer or can use their build-in time dependent execution. The output values generated are primarily used for animations. Interpolation modes supported include linear, discrete, and cubic interpolation.

4.4 Supporting Shared Environments

A distribution status inherited by all events provides the basis for the support of shared environments. This status is also forwarded along signals within behaviors. It ensures that scene graph changes are distributed only once or even never. We distinguish early synchronization (events have been distributed, behaviors are executed locally at each site, scene graph changes are not further distributed) and late synchronization (events have not been distributed, behaviors are executed only at one site, scene graph changes are distributed). Behaviors allow influencing the distribution status when issuing events using *SharedAction* components. However, signals and events based on time dependent components usually will never be synchronized as they are typically executed independently at each site. *SharedTrigger* components allow implementing locking for the execution of behaviors as well as support for multi-user interactions.

5 Object-Oriented Modeling Language

In VR/AR environments widgets are not limited to elements such as buttons, labels, menus or scrollbars, but every virtual object may be part of the user interface. Contrary to classical widgets in a WIMP environment, which usually define interaction based on mouse and keyboard, VR/AR widgets have to deal with many interaction mechanisms and therefore their behavior is much more complex. We argue that virtual environments (VE) and user interfaces in VEs are inherent complex and the demand for object-oriented design and object-oriented programming is high as well for creators and content authors. The realization highly complex VE is already done using object-orientation throughout the whole development process.

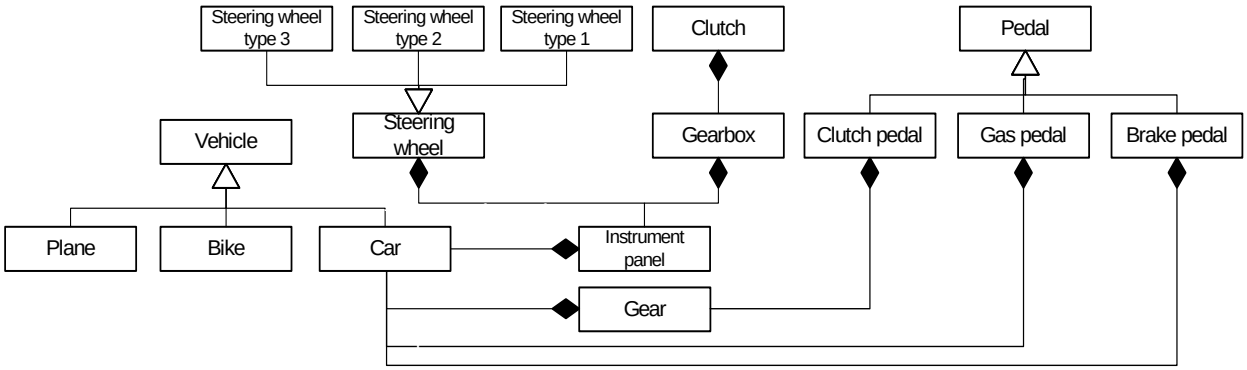

Figure 6: Simplified extract of a class hierarchy of vehicles for the example of a driving simulation

This becomes clearer looking at the following example. In a driving simulation a number of cars can be used for driving around. The user interface of each car consists of the whole instrument panel, including steering wheel, gas pedal and the gearshift, each of them has a very complex behavior and it is very important that it matches real world behavior as good as possible. The same applies for the visual representation of those objects. Of course, every car has a different instrument panel and a different visual appearance. Although the user interfaces of the cars differ, they still have a lot of similarities, e.g. the steering wheel can be rotated to steer the car and the handling of the gas pedal is the same as well. In order to realize such complex behavior of the different user interfaces, object-oriented mechanisms are a natural choice. They allow defining common classes and behavior, e.g. for steering wheels and gas pedal, and allow inheriting from those classes to override properties to define customized behavior or unique appearance. An instrument panel can be designed to accept all steering wheels independently from their behavior or appearance. See Figure 6 for a simplified class hierarchy of the widgets of class Car. As this diagram demonstrates, even very simple examples tend to have quite complex interconnection and relations among the different objects. In order to deal with such complex behavior within a VE object-oriented design methods can be of great benefit, especially if their relations can be realized using object-oriented techniques. Additionally, the interaction with such elements can be performed using different devices or modalities. The user might be sitting in a real driving simulator controlling the virtual car by a real instrument penal, holding a real steering wheel, he maybe in a VR environment using data gloves with force-feedback to grab the steering wheel and turning it. It is also possible to use joysticks, keyboard and similar input devices to interact with the car. The different interaction techniques and devices can also be handled using object-orientation.

This demonstrates the complexity of the behavior of VR/AR user interfaces. In high level programming languages object-orientation is provided to deal with such complex systems. Thus the functionality of this example would be realized within an object-oriented programming language such as Java or C/C++. A large number of applications already exists, utilizing animation and behavior rich VEs, especially for 3D computer games. Those are usually realized on top of a game engine, such as the commercially available CryEngine[1] or Unreal Engine[2]. To change high-level behavior of objects, e.g. the drag of a steering wheel or the gas pedal, usually the authors can only influence certain parameters; otherwise the programming logic has to be changed.

[1] CryEngine, CryTek, http://www.crytek.com
[2] Unreal Developer Network, Epic Games, http://www.epicgames.com

Apart from game engines, a lot of VEs use scene graph description languages such as VRML'97 or X3D, which only provide very basic scripting possibilities These VEs usually use the description languages only to store the scene graph information, high-level behavior is realized by external applications which work on these description languages. Both approaches have in common, that behavior is defined and implemented within the application and changing the behavior - apart from parameterizing - implies changing the source code. Depending on the size of the application and the scope of the change, this can be a very time consuming task. It is desirable to be able to define complex object behavior in an efficient and powerful way directly within the scene graph description. To achieve that, object-oriented concepts seem to be the natural choice, since object-orientation is currently the most powerful methodology for realizing inherent complex systems.

As VRML++ (Diehl, 1997) shows, the need to use object-orientation within description languages is evidently, but still no proposed solution has provided sufficient functionality and flexibility. Especially the syntactic shortcomings of VRML'97 cannot be reduced by the syntax extension of VRML++. Debugging preprocessed code is also very time consuming and error prone.

Therefore, one of our very recent research areas is to design an object-oriented modeling language. It will enable content authors to design highly complex virtual scenes and create their own object behaviors. Since this description language will provide the facilities of object-orientation, common behaviors can be efficiently refined using concepts like abstraction or specialization. Additionally, objects and their behavior can be reused in other applications and modify their behavior or appearance without effecting the original application. We believe that our approach of an object-oriented modeling language will be able to deal with these issues and provide a feasible solution.

6 Discussion

In this section we want compare the different approaches and provide some best-practice information on the use of each of them. Table 2 provides an overview over the strengths and weaknesses of the individual mechanisms used to realize multi-modal user interfaces within our framework. User interface description languages are independent of particular input devices and the target platform. Thus however is true only, if an appropriate renderer exists. Interaction prototyping provides full flexibility and level of control and allows for fast realization. However, its main drawback is scalability (as it was optimized to be used for early testing not for production). The object-oriented modeling language provides the most powerful approach, but the required effort and the expert level both depend very much on the classes and functionality already provided in the application area.

Approaches/ Characteristics	Input device/ modality independence	Target platform/device independence	Flexibility	Level of control	Reduced efforts (refers to development time)	Simplicity (referring to expert level)	Scalability
User interface description languages	+	+	-	-	+/-	+	+/-
Interaction prototyping	+/-	-	+	+	+	+	-
Object-oriented modeling language	+/-	-	+/-	+	+/-	+/-	+

Table 2: The different approaches for the realization of multi-modal VR/AR user interfaces and their characteristics. (+: high +/-: moderate, -: low)

Our experience in previous projects showed that a typical development cycle will start with simple prototypes, which will be replaced by more and more powerful user interface descriptions and object models. As reusability of all mechanisms is very high, new user interfaces are often rather assembled than created from the scratch.

7 Conclusions and Future Work

In this paper we presented our approach on supporting flexible user interface design for VR and AR environments. We presented three different mechanisms provided as part of our VR/AR framework Morgan: the use of a VR/AR-

enabled user interface description language, a rapid prototyping mechanism for user interactions, and an object-oriented VR/AR modeling language, each supporting for individual stages of user interface and application design.

Beside further user tests, our future work we will focus on the development of appropriate authoring tools to simplify the use of these mechanisms, and the provision of sample class libraries for object-oriented scene graphs.

References

D. Abawi, R. Dörner, M. Haller, J. Zauner. Efficient Mixed Reality Application Development. *In Proceedings of the 1st European Conference on Visual Media Production*, London, March, 2004.

M. Abrams, C. Phanouriou, et al. UIML: An Appliance-Independent XML User Interface Language. In *Proceedings of the Eighth International World Wide Web Conference*, May 1999.

M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riß, C. Sandor, M. Wagner. Design of a Component-Based Augmented Reality Framework. *In Proceedings of the Second IEEE and ACM International Symposium on Augmented Reality (ISAR 2001)*, 2001.

A. Bierbaum, P. Hartling, and C. Cruz-Neira. Automated testing of virtual reality application interfaces. In *Proceedings of the workshop on Virtual Environments 2003*, ACM Intl. Conf. Proc. Series, pp.107-114, 2003

D. Bowman, E. Kruijff, J. LaViola, and I. Poupyrev. 3D User Interfaces: Theory and Practice, Addison-Wesley, Boston, 2004.

W. Broll, I. Lindt, J. Ohlenburg et al. ARTHUR: A Collaborative Augmented Environment for Architectural Design and Urban Planning. *Journal of Virtual Reality and Broadcasting*, 1(2004), no. 1, December 2004.

S. Diehl. VRML++: A Language for Object-Oriented Virtual-Reality Models. In *Proceedings of the 24th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Asia'97)*, Beijing, China, 1997.

O. Hilliges, C. Sandor, G. Klinker. A Lightweight Approach for Experimenting with Tangible Interaction Metaphors. *In Proceedings of the International Workshop on Multi-user and Ubiquitous User Interfaces (MU3I) 2004*, Funchal, Madeira.

B. MacIntyre, M. Gandy, S. Dow, J. D. Bolter. DART: A Toolkit for Rapid Design Exploration of Augmented Reality Experiences. *In Proceedings of User Interface Software and Technology (UIST'04)*, October 24-27, 2004, Sante Fe, New Mexico.

C. Magerkurth, M. Memisoglu, T. Engelke, and N.A. Streitz. Towards the next generation of tabletop gaming experiences. *In Proceedings of Graphics Interface 2004*, London, Canada, 2004, pp. 73-80.

J. Ohlenburg, I. Herbst, I. Lindt, T. Fröhlich, W. Broll. The MORGAN Framework: Enabling Dynamic Multi-User AR and VR Projects. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST 2004)*, Hong Kong, Hong Kong, 2004, pp. 166-169.

A. Penn, C. Mottram, A. Fatah gen. Schieck, et al. Augmented reality meeting table: a novel multi-user interface for architectural design. *In Proceedings of the 7th International Conference on Design & Decision Support Systems in Architecture and Urban Planning 2004*. Kluwer Academic Publishers, Eindhoven, pp. 213-220.

G. Reitmayr, D. Schmalstieg. An Open Software Architecture for Virtual Reality Interaction. *In Proceedings of VRST'01*, Banff, Canada, Nov. 15-17, 2001.

P. Renzulli et al.. Towards Higher-level Interaction Paradigms for Virtual Reality. *NSF Collaborative Virtual Reality and Visualization Workshop*, Lake Tahoe, 2003.

R. Taylor, T. Hudson, A. Seeger, H. Weber, J. Juliano, A. Helser. VRPN: A Device-Independent, Network-Transparent VR Peripheral System. *In Proceedings of VRST 2001*. Banff, Canada, Nov. 15-17, 2001.

D. Schmalstieg, A. Fuhrmann, and G. Hesina. Bridging Multiple User Interface Dimensions with Augmented Reality. *In Proceedings of ISAR'00*, 2000, pp. 20-29.

N. Souchon, J. Vanderdonckt. A Review of XML-compliant User Interface Description Languages. *Design, Specification, and Verification of Interactive Systems (DSV-IS 2003)*, pp. 377-391, 2003.

S. Trewin, G. Zimmermann, G. Vanderheiden. Abstract User Interface Representations: How well do they Support Universal Access? In *Proceedings of CHI 2003*, 2003.

J. Zauner, M. Haller, A. Brandl, W. Hartmann. Authoring of a Mixed Reality Assembly Instructor for Hierarchical Structures. *In Proceedings of the Second International Symposium on Mixed and Augmented Reality*, Tokyo, October 2003.