Model-Based Design of Virtual Environment Behavior

Bram Pellens, Frederic Kleinermann, Olga De Troyer, and Wesley Bille

Research Group WISE, Vrije Universiteit Brussel, Pleinlaan 2 - 1050 Brussel, Belgium

Abstract. Designing and building Virtual Environments (VEs) has never been an easy task and it is therefore often performed by experts in the field. This problem arises even more when it comes to the design of behavior. To address this issue, an model-based approach called VR-WISE has been developed. This approach aims at making the design of VEs more accessible to users without or with little background in VE development. This means that the VE is specified by means of high-level models from which the actual VE can be generated. The purpose of this paper is twofold. Firstly, the design environment, supporting the approach, is described. We show how the models can be created in a graphical way. We also discuss the "Verbalizer". This module provides a natural language interface for the models, which enhances the intuitiveness of the specification. Secondly, we explain how behavior models can be specified.

1 Introduction

Today, Virtual Environments (VEs) are used for different purposes. Despite the fact that its use has grown, the design and development of VEs is only performed by a limited number of persons (VR-experts). This is mainly due to the technology being not very accessible to novice users. Although software tools (such as 3D Studio Max [4]) do assist the developer in creating a VE, they require considerable knowledge about VR technology. Authoring tools are most of the time used to create the static part. This static part is then imported in a toolkit where the code of the behavior is added by means of dedicated programming.

We have developed an approach, called VR-WISE, to support the design phase in the development process of a VE. It enables the specification of a VE by means of high-level models. The aim of VR-WISE is to make the design more intuitive, requiring less background and thus making the VR technology available to a broader public.

The approach is supported by a toolbox that allows constructing the models in a graphical way, i.e. by means of diagrams. To model behavior, the diagrams are complemented with a textual language. The diagrams and the textual language are designed in such a way that there is a close relation with natural language. This allows to "read" the diagrams more easily, which increases the understandability and makes them more intuitive. Furthermore, the models are expressed in terms of concepts of the domain of the target application. This helps users to bridge the gap between the problem domain and the solution domain.

In addition, a graphical notation has a number of well-known advantages. It enhances the communication between designers, programmers and other stakeholders. It is more efficient in its use. Tools can be developed that not only prevent errors, but also provide views on the design from different perspectives and on different levels of abstraction. On the other hand, a textual language can be more expressive and is therefore better to cope with higher complexity. However, the disadvantage of a textual language is that in general, it is more difficult to learn and less intuitive to use. In VR-WISE, we combine the graphical language with small textual scripts. These scripts allow specifying more complex behaviors, while maintaining the intuitiveness of the graphical language.

The approach for modeling behavior is *action-oriented*, i.e. it focuses on the different actions that an object must be able to perform rather than on the states in which an object can be in. Specifying behavior in such a way is more intuitive for non-professionals. To provide additional support for novice users, the toolbox contains the *Verbalizer* module. This module generates a textual formulation of the behavior specifications. Because our approach uses domain terminology and intuitive modeling concepts that have a strong relation with natural language, it is well suited for verbalization. The verbalization reduces the time needed to learn the graphical notation and enhances the communication with non-professionals. Furthermore, this Verbalizer is also useful in the context of iterative design and code documentation, as explained later in the paper.

The paper is structured as follows. In the next section, we give a general introduction of the VR-WISE approach. Section 3 describes the toolbox developed to create the models. Next, section 4 gives an overview of the models used to describe object behavior within the VR-WISE approach. Section 5 discusses related work concerning modeling of object behavior. The paper ends with a conclusion and future work.

2 VR-WISE Approach

The development process in the VR-WISE approach is divided into three (mainly) sequential steps, namely the *Specification step*, the *Mapping step* and the *Generation step*. An elaborated overview of the approach can be found in [6].

The Specification step allows the designer to specify the VE at a high level by means of models. To create the models, domain knowledge together with high-level modeling concepts are used. The models define the objects needed in the VE, the relationships between them, their behaviors, the interactions with other the objects and with the user. Due to the use of domain knowledge and intuitive modeling concepts, there is a strong similarity between how one describes its VE in our approach and how it would be done using natural language. The Mapping step involves specifying the mappings from the conceptual level into the implementation level. The purpose of the mapping is to specify how a particular domain concept described in the Specification step (first step) should be represented in the VE. The Mapping step is needed to be able to transform the

models into implementation models and to generate code. This is done in the Generation step.

3 OntoWorld: VR-WISE toolbox

To support the VR-WISE approach, a toolbox called OntoWorld has been developed. This toolbox enables a designer to build the models that form a complete conceptual specification of the VE, and to specify the mappings. Based on this information OntoWorld will generate a number of code files for the VE.

An important part in our toolbox is the *Visual Designer* (figure 1). This is a graphical diagram editor that allows creating the models in a graphical way. The models describing the static structure of the VE as well as the behavior of its objects can be specified using the Visual Designer. The tool has been implemented using Microsoft Visio. A number of stencils, one for each type of diagram, are built containing the graphical representations of the different modeling concepts available in our approach. Examples of graphical elements can be found on the left side of figure 1. A discussion of the different modeling concepts (and their graphical representation) to specify the static structure of the VE is beyond the scope of this paper. We refer the reader to [2] for more details on this. The modeling concepts available to model the behavior are discussed in the following section. The graphical elements can be dragged from the stencils and dropped onto the canvas and proper connections can be made. Properties can be added, displayed and modified by the designer.

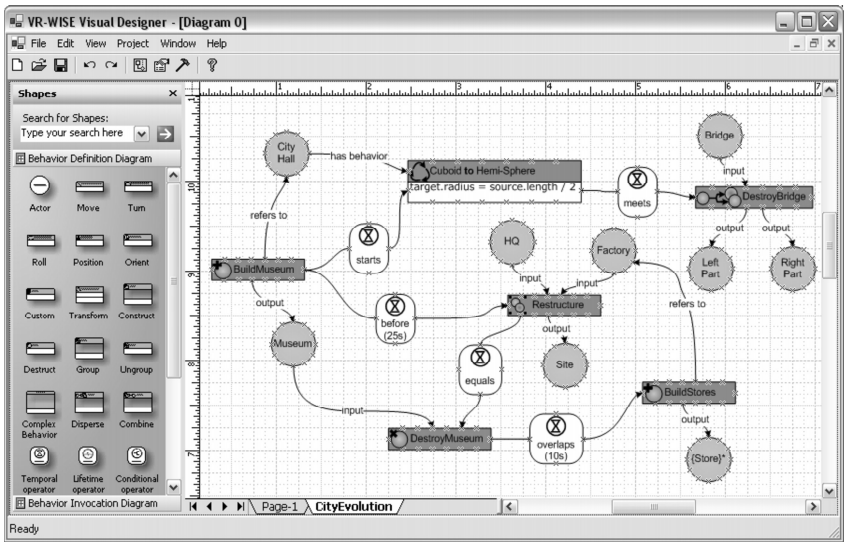

Fig. 1. Screenshot Visual Designer

From the specifications and the mappings, OntoWorld generates VRML/X3D [11] and Java3D [8] (source code which can be compiled and launched either as a local application or on the Web).

An additional interesting tool in the OntoWorld toolbox is the Verbalizer (see figure 2). This module automatically generates a textual formulation for the conceptual specifications. This provides the designer with natural language descriptions of his models and this while making the models. A template-based approach [7] is used to generate the textual formulation. Every semantic representation (e.g., of a behavior) is associated with a (range of) template(s). So, the graphical representation can be converted into a text-like representation. Adding this feature to the design phase has a number of advantages:

- **Interactive Design**. Displaying a natural language-like formulation of the behavior, that has been modeled, provides the designer with a better understanding of what he/she has actually modeled. The automatic generation of textual formulations will allow for an early detection of design errors. The textual formulations will also shorten the learning time.
- **Code Documentation**. After the design process has been completed, the conceptual specifications are used to generate programming code. Generated code is usually not or poorly documented. The code generator can use the formulations generated at design time to document the code, and hereby facilitates the post-modeling phase and possible extensions and customization.

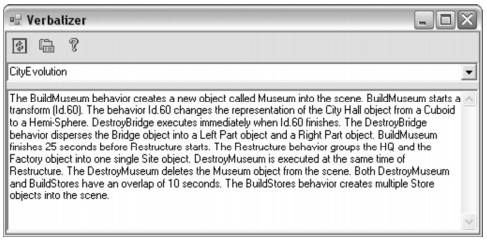

Fig. 2. Screenshot Verbalizer

4 Specifying Behavior by means of Models

Specifying the behaviors in VR-WISE is done in two separate steps: (1) specifying the *Behavior Definition* and (2) specifying the *Behavior Invocation*.

The first step consists of building *Behavior Definition Models*. A Behavior Definition Model allows the designer to define the different behaviors for an object. The behavior is defined independent of the structure of the object, and independent of how the behavior will be triggered. This improves reusability and enhances flexibility, as the same behavior definition can be reused for different

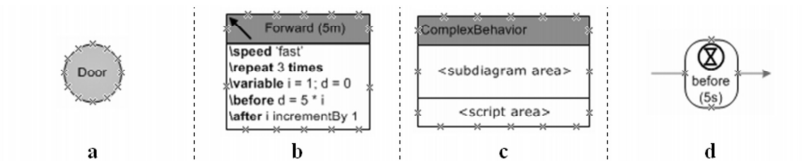

Fig. 3. Actor (a), Behavior (b), Complex Behavior (c), Operator (d)

objects and/or can be triggered in different ways. Because we want the definition of a behavior to be separated from the actual definition of the structure of an object, *actors* are used to specify behavior instead of actual object(s). An actor can be seen as a kind of abstract object and represents an object that is involved in a behavior. An actor is graphically represented by a circle containing the name of the actor (see figure 3a).

We have made a distinction between primitive behavior and complex behavior. A behavior is graphically represented by means of a rectangle. Different types of primitive behaviors are pre-defined. For primitive behavior (see figure 3b), the rectangle carries an icon denoting the type of primitive behavior as well as some additional information (i.e. parameters). For complex behavior, a name is specified and an additional area is used to contain the model that describes the complex behavior (figure 3c).

In order to cope with complex situations, behaviors may have an optional area that holds a (textual) script. The following flags can be specified:

- **/speed** denotes the necessary time for completing a behavior. The possible values for this flag can be *very slow*, *slow*, *normal*, *fast* or *very fast*.
- **/type** sets the type of movement that needs to be made. A movement can be executed in different ways. Possible values here are *smooth*, *lineair*, *slow* or *fast*.
- **/condition** states the conditions that need to be satisfied for the action to be executed. An example could be that a door needs to be unlocked before the OpenDoor behavior can be successfully executed.
- **/repeat** denotes the number of times that the behavior (or action) needs to be executed.
- **/variable** specifies custom variables to be used within the action. These variables are in fact placeholders for values. An un-typed system is used so that variables can hold values of any type.
- **/before** allows specifying the expressions that need to be performed before the actual operation is being executed.
- **/after** allows specifying the expressions that need to be performed once the behavior has been fully executed.

In the expressions, either user-defined custom functions (or algorithms) or a number of pre-defined functions can be used. Since the scripts only belong to a single behavior, they will in general be small. For this reason, they are easy to use, even for non-programmers. The scripts can be composed in the

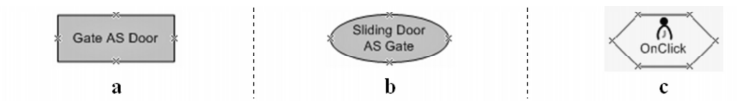

Fig. 4. Concept (a), Instance (b), Event (c)

Visual Designer through the built-in Script Editor. A large number of predefined functions, constants and operators can be selected and easily used.

To specify complex behavior, behaviors (primitive as well as complex) can be combined by means of the *operators*. There are three types of operators: *temporal*, *lifetime* and *conditional* operators. The temporal operators allow synchronizing the behaviors (example operators are before, meets, overlaps, starts, ...). Lifetime operators control the lifetime of a behavior (example operators are enable, disable, ...) and the conditional operator controls the flow of a behavior by means of a condition. In the Visual Designer, operators are graphically represented by rounded rectangles (see figure 3d). The icon within the rectangle indicates the type of operator.

The second step in the behavior modeling process consists of creating a *Behavior Invocation Model* for each Behavior Definition Model. This type of diagram allows attaching the behaviors defined in a Behavior Definition Model to actual objects, and to parameterize them according to the needs. Furthermore, these models also specify the events that will trigger the behaviors attached to the objects. The main modeling concepts for these models are *concept* and *instance*, which can be compared to the concepts Class and Object in object-oriented programming languages. They are represented by a rectangle (figure 4a) and an ellipse (figure 4b) respectively.

By assigning an actor to a concept, the behavior is coupled to the concept, i.e. every instance of that concept will obtain all the behaviors defined for the actor. By assigning an actor to an instance, only that particular instance will obtain the behaviors of the actor.

In our approach, behaviors are triggered by means of *events*. Events are graphically represented by a hexagon with an icon denoting the type of the event and possibly some additional information below the icon (see figure 4c). There are four kinds of events: initEvent, timeEvent, userEvent and objectEvent. The initEvent will be invoked at the moment the VE initializes. A timeEvent allows triggering a behavior at a particular time. A userEvent triggers a behavior upon an (inter)action of the user and an objectEvent does this when there is an interaction with other objects. Details on the steps to model behavior as well as the different concepts (to model animations of the objects) can be found in [5]. In the remaining part of this section, we discuss the modeling concepts for modeling object dynamics, i.e. to describe structural changes in the VE (e.g. adding, modifying or deleting objects) rather than just describing the different poses of the objects. Due to space limitations, the Behavior Invocation Models will be discarded in the following examples and only the Behavior Definition Models are given.

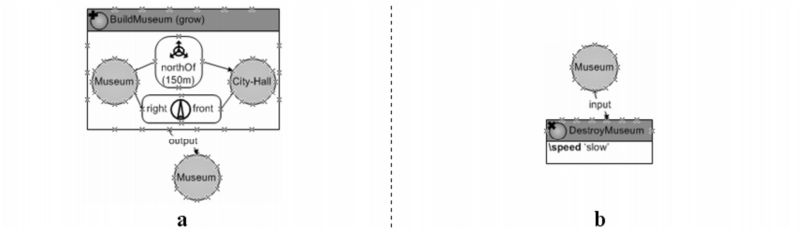

Fig. 5. Creation (a), Removal (b)

4.1 Modeling the Creation and Removal of Objects

A first issue in building dynamic scenes is to cope with new objects. To address this, our approach uses the *construct*-behavior allowing new objects to be added or inserted to the VE at run-time. Similar to the other types of behavior, actors are used in the definition; later on, in the Invocation Models, concepts are associated with the actors. The construct-behavior has at least one output actor, i.e. representing the object that is created. Creating a new object is done by instantiating a concept that has been described in the Domain Model (one of the models created to specify the static structure of the VE). This concept is specified as output actor. The object that is to be created also needs to be positioned and oriented in the VE. This is specified by means of a so-called *Structure Chunk*. This is a small *Structure Model*. Structure Models are used in our approach to specify (at a high level) the structure of the VE. Specifying structure is done by means of spatial relationships and orientation relationships. They are used for placing the objects. Also connection relations can be used for building complex objects. As these relations have been described elsewhere [2], the paper will not discuss their graphical notation in details. For the construct-behavior, the Structure Chunk describes the positioning of the new object (in terms of an actor) at the time of the instantiation. Note that after the creation has been completed, the description given in the Structure Chunk might not be valid anymore, depending on whether the objects involved have performed some other behavior or not. The construct-behavior is graphically represented in a similar way as the other behaviors except that an additional area is used for specifying the Structure Chunk (figure 5a).

To illustrate the construct behavior, an example is provided where a city is expanding with new buildings (see figure 5a). In the Behavior Definition Model, the creation of an actor Museum is defined. This behavior is called BuildMuseum. The Structure Chunk specifies that the newly created object must be north of the City-Hall with a distance of 150 meters and oriented with its right side to the front side of this City-Hall.

Besides creating new objects, building dynamic scenes also includes the removal of objects from the scene. Therefore, the *destruct*-behavior is introduced. Note that destroying an object will not only make the object disappearing from the environment, but it will also delete it from the scene-graph. When the ob-

ject that needs to be destroyed is part of a connectionless complex object, the relationships in which this object was involved will be deleted too; when it is part of a connected complex object, the connections in which the object is involved will be deleted as well. In figure 5b, the destruct-behavior is defined for a Museum actor, stating that any object that will be associated with this actor could possibly be destroyed.

4.2 Modeling Visual Changes of Objects

An issue that still remains open is the modeling of the visual modifications that an object may undergo. In section 2, it was explained that a concept (or an instance) is given a specific representation in the VE using the mappings. When creating animations stretching over a longer period of time, it could happen that the representation of the concepts should change (during simulation). The first type of modification we considered is the *transformation*-behavior. This type of behavior will change the appearance of an object. Note that the concept itself is not changed; only its representation in the VE is changed.

Figure 6a gives an example of a definition where the actor's representation is changed from a cuboid to a hemi-sphere. Changing the representation of an object may also cause changes of the properties of the representation. These changes can be described by means of transformation rules (specified in the middle area of the graphical representation of the behavior). With a transformation rule, the designer can describe for example, that the length in the original representation (source) is being transformed into the radius of new representation (target) (e.g., radius = length/2). When no rules are given, a standard one-to-one transformation is performed for the corresponding properties when possible; otherwise the defaults of the properties are taken.

To create VEs with a high degree of realism, we also have to consider objects breaking or falling apart. This brings us to the second type of modification supported by our approach, namely the *disperse*-behavior. A disperse-behavior subdivides an object into two (or more) pieces. The disperse-behavior has one input actor and two or more output actors. The input actor represents the object that will be subdivided and the output actors represent the pieces. After such a behavior has been executed, the input object is destroyed and the output objects have been created. Like in the construct-behavior, the new objects that result from such a behavior need to be positioned in the environment. Again, a Structure Chunk is used for relating the newly created objects to each other and for relating them to already existing objects. The positioning of the objects can be done by means of spatial and orientation relations. When no relations are expressed between the newly created objects and already existing objects, then (the bounding box of) the new objects are positioned at the same location as the original (input) object.

In the example in figure 6b the definition for the behavior DestroyBridge is given. It specifies that the complete bridge will disperse in two objects, two smaller pieces of the bridge. The Structure Chunk specifies that one of the pieces (Left Part) is positioned left of the other one (Right Part).

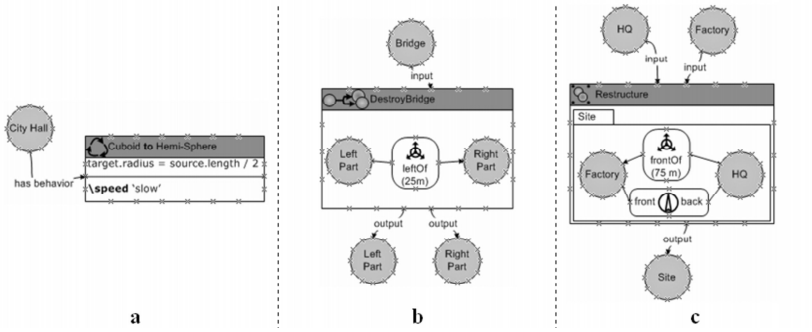

Fig. 6. Transform (a), Disperse (b), Grouping (c)

If a disperse-behavior is invoked on a complex object (connected or unconnected), the disperse behavior will remove all the relations that exist between the (parts of the) complex object. This implies that if the user moves one of the objects, the others will not move accordingly since there is no physical connection anymore. In the disperse-behavior, all the information of the original object can be used for the creation of the new objects.

Opposed to objects breaking or falling apart, modeling VEs with dynamic objects also requires having means of joining objects together. In other words, we need to be able to specify that objects can be created by combining objects or assembling objects at runtime hereby creating either complex connected or complex unconnected objects. To support this, we have the *grouping*-behavior. This kind of behavior allows creating spatial relationships or orientation relationships (for unconnected objects) and making connections (for connected objects) at runtime. Such a behavior definition has two (or more) input actors and one output actor. The output actor represents the newly created (complex) object that is built up of the pieces represented by the input actors. Also here, a Structure Chunk is used to describe the structure of the new object, which should be based on the part-objects. Note that in this case, the Structure Chunk is used to express a complex object and therefore the relationships that will be created at run-time will be fixed relationships. This means that after this behavior has been performed for a number of objects, the newly created object will behave as a complex object and therefore if one of its parts or the object itself is moved for example, the relations will ensure that the other parts are moved as well.

In figure 6c, an example is shown on how some objects are grouped to create a new unconnected complex object. The HQ actor and the Factory actor represent the input objects and are being taken together to form a site (represented by the Site actor).

Both the disperse- and the grouping-behavior have counterparts, namely *combine* and *ungrouping* respectively. The combine-behavior will merge a number of objects together in the same way as the grouping but with this difference that the input objects do not exist anymore once the behavior has been per-

formed. The ungrouping-behavior is the reverse of the grouping and removes connections that were made during this grouping. Here, the difference with the disperse-behavior is that the output objects are not created but already exist. We will not give examples for these behaviors, as they are similar to the ones that were already discussed.

5 Related Work

Most of the work on the specification of object behavior is concerned with textual definitions as in [3]. However, these textual languages are not easily accessible by untrained users.

The design of Virtual Environment behavior has also been addressed in [9]. The Flownet formalism is being used as a graphical notation for specifying the behavior. However, even for simple behaviors, the specification becomes large and difficult to read and is therefore not very suitable for non-experts. The VR-WISE approach allows specifying the behavior also through a graphical language but the terminology and modeling concepts used are closely related to how one would describe behavior in natural language.

In [1], an icon-based approach is presented to specify behaviors of objects in VRML. The designer can drag icons, representing VRML nodes, from a palette onto a workspace and create the connections between outputs and inputs of the nodes. Also here, considerable knowledge about the VRML language is required to build a behavior specification.

A commercial development environment that closely relates to our research is Virtools Dev [10]. Also Virtools allows constructing object behavior graphically by combining a number of primitive building blocks (which are pre-made scripts) together. However, the function-based mechanism (where the designer needs to take into account the frame-to-frame basis way of processing the behaviors by the behavior engine) tends to be less comprehensible for novices.

6 Conclusion

In this paper, we have described a model-based approach to specify behavior for VEs. We also introduced the OntoWorld toolbox that supports the approach. The Visual Designer module of the toolbox allows specifying the models by means of diagrams. The Verbalizer module that generates textual explanations for the models was added to improve the understanding of the models and to make documenting of the code possible.

The modeling language for the behavior has evolved to a kind of hybrid mix between a graphical language and a textual scripting language. The behaviors are mainly specified in a graphically way, but a simple scripting language can be used for more complex cases. This way, we offer the advantages of both worlds; the comprehensiveness of the graphical language and the ability to cope with increasing complexity by means of a textual language. The combination

of a model-based approach with the use of an intuitive graphical language may seriously reduce the complexity of building dynamic and interactive VEs.

At the time of writing, we are implementing a debugger within the Visual Designer which checks the models for errors. Furthermore, our approach uses a formalization of the different relations and concepts used for modeling behavior. One of the benefits of having a formalization is that errors made by the designer can then be checked at design time. This gives the advantage of accelerating the modeling of behaviors as designers can quickly see where the errors are. This formalization is currently being implemented. The future work will consist of conducting a number of experiments in order to evaluate the modeling power and intuitiveness of the developed tools and the different modeling concepts proposed. Next, we will investigate how we can extend our models with patterns coming from game design research in order to come to a richer set of modeling concepts.

7 Acknowledgements

This research is carried out in the context of VR-DeMo project, funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders.

References

1. Arjomandy, S. and Smedley, T.J., "Visual Specification of Behaviours in VRML Worlds", In Proceedings of the ninth International Conference on 3D Web Technology, ACM Press, Monterey, California, USA, 2004, pp.127-133
2. Bille, W., De Troyer, O., Pellens, B., Kleinermann, F., "Conceptual Modeling of Articulated Bodies in Virtual Environments", In Proceedings of the 11th International Conference on Virtual Systems and Multimedia, pp. 17-26, Publ. Archaeolingua, Gent, Belgium, 2005
3. M. Kallmann and D. Thalmann, "Modeling Behaviors of Interactive Objects for Virtual Reality Applications", Journal of Visual Languages and Computing, Vol. 13, 2002, pp. 177-195
4. Murdock K.L. 3DS Max 7 Bible. Wiley Publishing Incorporated, 2005
5. Pellens, B., De Troyer, O., Bille, W., Kleinermann, F., Conceptual Modeling of Object Behavior in a Virtual Environment, In Proceedings of Virtual Concept 2005, pp. 93 - 94 + CD-ROM, Publ. Springer-Verlag, Biarritz, France, 2005.
6. Pellens, B., Bille, W., De Troyer, O., Kleinermann, F., VR-WISE: A Conceptual Modelling Approach For Virtual Environments, In Proceedings of the Methods and Tools for Virtual Reality (MeTo-VR'05) workshop, Ghent, Belgium, 2005
7. Reiter, E. and Dale, R. Building applied natural language generation systems. In Journal of Natural Language Engineering, 3:57 - 87, 1997
8. Selman, D., Java3D Programming, Manning Publications, 2002
9. Willans J.S., Integrating behavioural design into the virtual environment development process, Phd thesis, University of York, UK, 2001
10. –, Virtools Dev, http://www.virtools.com/, Accessed August 2, 2006
11. –, X3D, http://www.web3d.org/x3d/specifications/, Accessed May 31, 2006