Intuitively Specifying Object Dynamics in Virtual Environments using VR-WISE

Bram Pellens
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels
+32 2 629 5713
bram.pellens@vub.ac.be

Frederic Kleinermann
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels
+32 2 629 5713
frederic.kleinermann@vub.ac.be

Olga De Troyer
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels
+32 2 629 3504
olga.detroyer@vub.ac.be

ABSTRACT

Designing and building Virtual Environments is not an easy task, especially when it comes to specifying object behavior where either knowledge about animation techniques or programming skills are required. With our approach, VR-WISE, we try to facilitate the design of VEs and make this more accessible to novice users. In this paper, we present how behavior is specified in VR-WISE, as well as the prototype developed for the approach.

Categories and Subject Descriptors

H.5.1 **[Information Interfaces and Presentation]**: Multimedia Information Systems – Artificial, augmented, and virtual realities, Evaluation/methodology.

General Terms

Design.

Keywords

Virtual Reality, Conceptual Modeling, Design Phase, Behavior

1. INTRODUCTION

These days, Virtual Environments (VEs) are used for numerous purposes. Despite the fact that its use has grown, the design and development of VEs is only performed by experienced people since the technology is not very accessible to novice users. This situation is especially noticeable in the development of the behavior. In the complete design process of a VE, specifying the animations of the objects is usually the most difficult bit for inexperienced people [5].

We have developed an approach, called VR-WISE, to support the design phase in the development process of a VE and to enable the specification of a VE at a high-level, free from any implementation aspects. The aim of VR-WISE is to make the design more intuitive, requiring less background and thus making the technology available to a broader public. In this paper, we show how behavior can be specified at a high level and by means of a graphical language. The language is *action-oriented*; meaning that it focuses on the different actions that an object must be able to perform rather than on the states an object can be in. Specifying behavior in such a way is much more intuitive for non-professionals because the behaviors are specified in a way similar as they would have in natural language.

In most animation models for VEs, the lifetime of the objects and their visual representation are not explicitly specified. Usually, it is assumed that the object already exists when the behavior starts; also often the visual representation of the object remains the same throughout the complete behavior. If one wants to go beyond this and deal with such behavior, one needs to resort to programming in order to do so. The graphical specification language proposed here includes modeling constructs to specify changes in the content of the VE over time (e.g., creating, modifying or deleting objects). These modeling constructs complements the ones for specifying the change of the poses of the objects in a VE as available in most modeling tools.

The work presented here closely relates to [6] where the design of behavior is also addressed using a graphical notation. Unfortunately, for simple behaviors, the specification becomes large and difficult to read and is therefore not very suitable for novices. In [1] an icon-based approach is presented to specify behaviors of objects in VRML. However, considerable knowledge about the VRML language is required. Similar modeling concepts as ours are introduced in [3]. However, these are limited in such a way that no actual code can be generated for them. A commercial development environment that closely relates to our research is Virtools Dev [7]. Also Virtools allows constructing object behavior graphically by combining a number of primitive building blocks together. However, the function-based mechanism tends to be less comprehensible for novices.

2. VR-WISE AND TOOL SUPPORT

The development process in the VR-WISE approach is divided into three (mainly) sequential steps. The *Specification step* allows the designer to specify the VE at a high level using domain knowledge together with high-level modeling concepts. In this step, the domain objects needed in the VE, the relationships between them as well as their behaviors, the interactions with other objects and with the user are specified. There is a strong similarity between how one describes a VE in our approach and how it would be done using natural language. The *Mapping step*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VRST'06, November 1–3, 2006, Limassol, Cyprus.

Copyright 2006 ACM 1-59593-321-2/06/0011...$5.00.

involves specifying the mappings from the higher conceptual level into the lower implementation level. The purpose of the mapping is to specify how a particular domain concept described in the first step should be represented in the VE. The *Generation step* will generate the actual source code for the VE specified in the Specification step using the mappings defined in the Mapping step. A detailed overview of the approach can be found in [4].

To support the VR-WISE approach, we developed a design environment called OntoWorld. This software tool enables a designer to firstly build a complete conceptual specification of the VE, and then specify all the desired mappings after which a number of code files are automatically generated. The prototype tool has been extended with the Conceptual Specification Builder (CSB), a diagram editor supporting the modeling of VEs. The static structure of the VE as well as the behavior of its objects (as will be described in this paper) can be specified using the CSB. The OntoWorld tool has also been extended with a Verbalizer. This module allows the automatically generation of a textual formulation of the specifications. Adding this feature to the modeling environment of the VR-WISE approach has a number of interesting advantages:

- Interactive Design. The generation of textual formulations will allow for an early detection of errors and results in a more iterative design process.
- Code Documentation. In the same way as the textual formulations are generated at design time, the code generator can use these formulations to document the code.

3. MODELLING BEHAVIOR IN VR-WISE

In this section, we will discuss the different behavior modeling concepts of the VR-WISE approach. They are illustrated by means of a more elaborated example. 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 Diagrams, which 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 objects and/or can be triggered in different ways.

Figure 1 shows an extract of the Behavior Definition Diagram defining the behaviors for the objects (buildings) inside an environment representing a city. Basically, a Behavior Definition Diagram consists of a number of actors (represented by a circle e.g., City-Hall, Bridge) attached to behaviors (represented by means of a rectangle and carrying an icon to represent the type of behavior). An actor can be seen as a kind of abstract object. Because we want the definition of a behavior to be separated from the actual definition of the structure of the objects, actors are used to specify behavior instead of actual objects. A behavior can be either primitive or complex. Examples of primitive behavior are move, turn, roll, position, orient,… Complex behaviors have an additional area containing the diagram that describes the complex behavior (see e.g., BuildMuseum, DestroyBridge). In order to cope with complex situations, both the primitive behaviors and the complex behaviors may have an optional area that holds a (textual) script (see e.g., BuildStores). To specify complex behavior, behaviors (primitive as well as complex) can be combined by means of *operators* (represented by means of a rounded rectangle and also carrying an icon to denote its type). 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 by means of a condition.

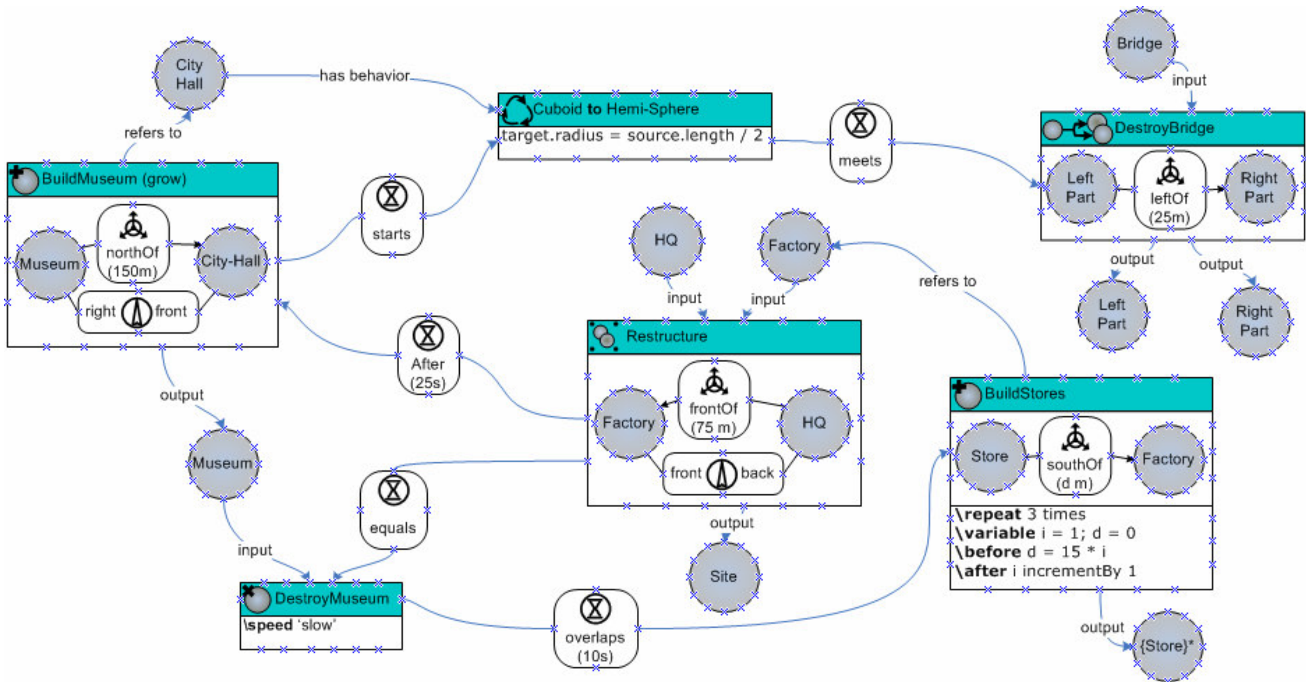

Figure 1. Behavior Definition Diagram

As shown in the example, when the *BuildMuseum* behavior is executed it starts the *Cuboid-to-Hemi-Sphere* behavior of the *City-Hall* actor. This transformation behavior will trigger the *DestroyBridge* behavior as soon as it finishes (expressed by the *meets* operator). The *Restructure* behavior is executed 25 seconds after the *BuildMuseum* behavior. This *Restructure* behavior is running in parallel with the *DestroyMuseum* behavior. The *BuildStores* behavior is running with an overlap of 10 seconds with the *DestroyMuseum* behavior.

The second step in the behavior modeling process consists of creating a *Behavior Invocation Diagram* for each Behavior Definition Diagram. This type of diagram allows attaching the behaviors defined in a Behavior Definition Diagram to the actual objects, and to parameterize them according to the needs. Furthermore, these diagrams also specify the events that may trigger the behaviors of the objects. Due to space limitations, the Behavior Invocation Diagram of the example is not included here. More details on modeling the general behavior can be found in [4]. In the remaining part of this section, a number of additional 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, will be discussed.

3.1 Modeling the Creation and Removal

A first issue in building dynamic scenes is to cope with new objects. The *construct*-behavior allows specifying how at run-time new objects can be created and inserted into the VE. 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 Specification. 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 Diagram*. Structure diagrams are used in our approach to specify the structure of the environment at a high level. Specifying structure is done by means of spatial relationships and orientation relationships. Also connection relations can be used for building complex objects. As these relations have been described elsewhere [2], the paper will not discuss them. Here, the structure chunk describes the positioning of a particular 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 it has additional area to specify the structure chunk.

It is also possible to specify how the objects that should be created need to be introduced in the VE. This is done by means of the optional property *method*. The value for this property can be either *appear, fade-in, grow* and *zoom-in*. When 'appear' is used, the object will just appear as such. 'fade-in' allows the object to gradually become visible. 'grow' and 'zoom-in' make the object entering the VE by gradually expanding from nothing to the actual size from either the ground (grow) or the center (zoom-in).

In the example (figure 1), the creation of an actor Museum is defined in the BuildMuseum behavior. 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. To handle the creation of different objects, the construct-behavior can be repeated as can be seen in the BuildStore behavior. This is done using a scripting language. In this behavior definition, the output is a list of Store actors. This is specified by the notation: {…}*. Such a list of objects can be manipulated as a whole or the separated objects in this list can be manipulated individually.

In addition to creating new objects, building dynamic scene also include the removal of objects from the scene. Therefore, the *destruct*-behavior has been introduced. Note that destroying an object will not only make the object to disappear from the environment but will also delete it from the scene-graph. When the object 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 of the object will also be deleted.

Similarly as for the construction of objects, an optional *method* property can be specified. The possible values for this property are here: *disappear, fade-out, shrink* and *zoom-out*. When 'disappear' is used, the object just disappears at once. When 'fade-out' is taken as value, the object will gradually become invisible. 'Shrink' and 'zoom-out' allows to remove the object by gradually become smaller, either towards the ground or towards the center of the object respectively.

In figure 1, the destruction-behavior called DestroyMuseum is defined for a Museum actor, stating that any object implementing this actor could possibly be destroyed. This \speed argument specifies how fast the object should disappear. A high speed will let the object disappear almost immediately while a low speed will fade-out the object gradually.

3.2 Modeling Changing Objects

An issue that still remains open is the modeling of the structural 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 mappings. When creating animations stretching over a longer period of time, it could happen that also 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. 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) in a certain way (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.

In the example of our city evolution, the City-Hall actor has a transformation behavior that changes the City-Hall's representation from a cuboid to a hemi-sphere.

To create VE 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 static structure chunk is used for relating the newly created objects to each other and for relating them to already existing objects. Again, placing the objects can be done by means of spatial and orientation relations.

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 other objects will not move accordingly since there is no physical connection anymore. Note that the disperse-behavior could be replaced by a behavior that executes the destruction of the input object immediately followed by the construction of the output objects. However, in the disperse-behavior, all the information of the original object can be used for the creation of the new objects.
In the behavior definition in figure 1, the definition for a 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 (the Left Part) is positioned left of the other one (the Right Part).

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 static 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 static 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 the 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 for example moved, the relations will ensure that other parts are also moved.

In the definition in figure 1, the HQ actor and the Factory actor represent the input objects of the Restructure behavior. These actors are being taken together according to the relations that are specified between them to form a site (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 performed. 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 those already discussed.

4. CONCLUSIONS

In this paper, we have introduced our approach for modeling object dynamics in a VE. The approach is based on high level, implementation independent specifications and intuitive modeling concepts. Usually, animation techniques are used to specify how certain aspects of objects should change. More advanced object behavior like the ones described here cannot be specified and are usually implemented using a programming language. In this paper, we discussed how it is possible to specify object dynamics at a high level and using intuitive modeling concepts. A prototype tool has been developed to support the VR-WISE approach. A verbalizer, which automatically generates textual explanations for the diagrams, has been added. This greatly improves the understanding of the diagrams.

5. ACKNOWLEDGMENTS

This work is carried out in the context of the VR-DeMo project (IWT 030284) and is directly funded by the IWT.

6. 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, California, USA, pp.127 – 133, 2004

[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] Campos, J., Hornsby, K., Egenhofer, M., A temporal model of virtual reality objects and their semantics, In *Proceedings of International Conference on Distributed Multimedia Systems*, San Francisco, USA, pp. 581 – 588, 2002

[4] 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, Springer-Verlag, Biarritz, France, 2005

[5] Terra, S. and Metoyer, R. Performance Timing for Keyframe Animation. In *Proceedings of the ACM Symposium on Computer Animation*, France, pp. 253 – 258, 2004

[6] Willans J.S., *Integrating behavioural design into the virtual environment development process*, Phd thesis, University of York, UK, 2001

[7] --, Virtools Dev, http://www.virtools.com/, Accessed August 2, 2006