BEHAVIOR3D: An XML-Based Framework for 3D Graphics Behavior

Raimund Dachselt, Enrico Rukzio
Dresden University of Technology, Department of Computer Science
Heinz-Nixdorf Endowed Chair for Multimedia Technology
01062 Dresden, Germany
{raimund.dachselt, enrico.rukzio}@inf.tu-dresden.de

Abstract
Success of 3D applications on the Web inherently depends on object behavior and interaction. Current Web3D formats often fall short in supporting behavior modeling. This paper introduces a flexible concept for declaratively modeling 3D object behaviors. Based on Extensible 3D (X3D) a node concept is suggested with object-oriented features such as inheritance, strong typing, and polymorphism. An XML-based language *Behavior3DNode* serves the interface definition of new nodes. Their implementation is simplified by automated code generation. A novel grammar generation mechanism collects all existing nodes in a dynamic XML Schema. Thus new behavior nodes can be used along with built-in nodes as first class scene graph elements. A rich set of predefined behaviors is proposed, among them *Animation* and *State Machine* node collections. The concepts were successfully implemented with VRML97/X3D and integrated into a 3D component approach.

Categories and Subject Descriptors
H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems – *Animations.* I.3 [**Computer Graphics**]: I.3.6 Methodology and Techniques – *languages, standards.* I.3.7 Three-Dimensional Graphics and Realism – *animation, virtual reality.* D.2.11 [**Software Engineering**]: Software Architectures – *domain-specific architectures, declarative languages.*

General Terms
Design, Standardization, Languages.

Keywords
Animation, Behavior Language, Object Behaviors, Dynamic Grammar, XML-Schema, Extensible 3D (X3D), SMIL, Contigra

1. Introduction
The availability of 3D technologies on consumer platforms is continuously growing. This is a result of permanent improvements in 3D accelerated graphics hardware and enhancements of 3D software systems. The widespread usage of Internet technologies and the development of a multitude of proprietary Web-based 3D formats consequently resulted in an increasing number of 3D-enhanced Web applications. Though promising results already exist in domains such as electronic commerce, computer assisted learning, sports, avatars, or entertainment, only few success stories about 3D graphics on the World Wide Web can be told. Success and further progress in this field inherently depends on compelling 3D content. It should be a media rich and highly interactive content to actually drive enabling applications. As far as object geometry and media assets are concerned, a variety of excellent modeling and authoring tools and format converters already exist. There are various interesting proprietary Web3D technologies and associated authoring tools to produce such media-rich 3D content. However, most of them are tailored to specific application domains and limited in producing really interactive and dynamic 3D scenes.

Proprietary solutions offer some convincing functionality to author specific behaviors such as animations or simple object interactions. However, they fall short in providing additional behavior types. Among the interesting solutions are tools such as Virtools Dev [Virtools Dev] and Cult3D Designer [Cult3D]. Developers face the disadvantage, that the underlying formats are mostly not disclosed. Therefore necessary behavior extensions are not easy to accomplish, if possible at all. Script languages are basically the only way to achieve application-specific complex behaviors. VRML97 [VRML97] as the standard for 3D graphics on the Web and its successor Extensible 3D (X3D) [X3D Specification] offer more flexibility through built-in behavior-related nodes, script nodes, and extensibility mechanisms. However, creating complex object behaviors and reusing them in other projects is far from being a simple task. Although sophisticated solutions exist for producing and especially reusing dynamic web pages and other media content, projects with interactive 3D graphics are often developed from scratch. They mostly demand programming skills or at least knowledge of scripting languages. Therefore it excludes non-programmers from designing 3D applications, which remains a tedious work.

This problem cannot be attributed to the authoring process alone, but initially depends on the capabilities of the underlying 3D formats. They usually employ scene graphs and focus on geometry, appearance, and simple dynamic behavior. With respect to complex behaviors and reuse of behavior building blocks they often fall short. Taking the mentioned problems into account we conceive our vision of an extensible, flexible and unifying description format for behaviors and interactions. It should be an open format, relate to standards, and it should simply integrate into existing 3D technologies. A rich set of predefined and classified behavior modules should be available. To reduce programming for non-expert users we propose a declarative format, thus being a reasonable basis for authoring tools.

The work presented in this paper is part of the research project CONTIGRA *(Component OrieNted Three-dimensional Interactive GRaphical Applications)* [Dachselt et al. 2002]. A declarative component architecture based on X3D [X3D Specification] was designed for the easy construction of Web-enabled, desktop Virtual Reality applications and 3D scenes. The document-centered approach is founded on XML Schema [XML Schema] languages describing the interface and implementation of 3D components as well as their configuration, assembly, and linking. A component implementation consists of three independent scene graphs containing geometry, audio, and behavior information. A separated link section connects their relevant nodes. The BEHAVIOR3D approach introduced here facilitates the construction of a component's behavior graph. The focus of this work lies on the behavior language itself as a basis for future authoring tools.

This paper is organized as follows. The next section relates our work to existing approaches. This is followed by a critical investigation of X3D behavior concepts and a definition of deduced requirements. The main part introduces BEHAVIOR3D with its node concept, new markup languages, and behavior node collections. Thereafter section 5 describes one possible implementation of BEHAVIOR3D with X3D, the integration into the CONTIGRA project, and an interactive 3D example. The paper is finished by a discussion particularly examining implications of the concept for X3D and an outline of future work.

2. Related Work

Powerful 3D graphics APIs such as Open Inventor and Java3D exist for imperative modeling of 3D object behaviors. Although almost any task can be accomplished with such 3D class libraries, programmer's knowledge is inevitable. Instead, declarative modeling of behavior will be the focus of this work.

The term behavior throughout this paper refers to Roehl's definition, where four levels of behavior are distinguished: direct modification of an entity's attributes defines level 0; the change of an entity's attributes over time constitutes level 1; level 2 comprises a series of calls to level 1 behaviors to perform some task; level 3 after all is characterized as top-level decision-making [Roehl 1995]. All levels should be well supported by a behavior language. The separation of graphs within the CONTIGRA project is based on the idea of an independent behavior graph developed in [Döllner and Hinrichs 1998]. It allows both the geometry-independent modeling of behaviors and their easy adaptation and exchange. Since many relationships are more of a temporal than a spatial nature, mixing of geometry and behavior into one single graph will only scarcely be indicated. It has to be mentioned, that the term behavior graph was derived from the term *scene graph*, though behavior graphs can often be represented in simple or even flat hierarchies.

Various declarative Web3D formats were analyzed for this work. The standardized VRML97 [VRML97] scene graph contains various nodes – which might generate and receive events – and a routing mechanism to propagate scene changes. Beside writing node and route statements, authors can employ script nodes to realize almost arbitrary functionality. To achieve reusability of sub graphs, a set of nodes can be encapsulated with the prototype concept. A few behavior extensions were already proposed to VRML97, such as *WaveInterpolator* and *RolloverSensor* [VRML 2.0 PROTO], as well as prototypes for event manipulation, arithmetic, Boolean logic, and event filters [Seidman 1998]. Proposals of the *VRML Object-Oriented Extensions Working Group* [VRML Object-Oriented] for object-oriented VRML extensions influenced this work. VRML++, developed by [Diehl 1997], was thereby of special interest. A new class concept including abstract classes was proposed along with this object-oriented approach. However, the disadvantage of a third concept beside built-in nodes and prototypes has to be noticed. Inheritance, an improved type concept, and polymorphism greatly enhanced VRML reusability, runtime stability, and maintainability. Unfortunately most of these concepts were not yet integrated into the successor X3D.

With the X3D specification [X3D Specification] a new XML encoding and various extensibility concepts were introduced. New behavior nodes and groups of nodes, e.g. the *Event Utility* component, were suggested. Some object-oriented concepts were introduced with the ongoing development of the X3D XML Schema [X3D-Schema] and Scene Authoring Interface (SAI) [SAI]. A detailed analysis can be found in the next chapter.

With its Binary Format for Scenes (BIFS) [MPEG-4] another standardized format, MPEG-4, provides a scene graph based language comparable to VRML97. Additional nodes allow the definition of character faces and bodies along with the declaration of their animation. Together with the BIFS animation protocol it is a powerful format well suited in particular for character animation and 2D/3D composition. The *Avatar Markup Language* [Kshirsagar et al. 2002] is based on XML and MPEG-4 and likewise oriented towards avatar animations.

One of the proprietary formats, Viewpoint [Viewpoint], uses an XML-based scene description language containing so called *Scene Interactors* for defining behaviors. The state machine paradigm forms the basis of the event handling and influenced a part of our work. *Actions* can be declared, which might be state-dependent or not. Although they can encapsulate scene behavior and can be parameterized, the XML language contains an inconsistent mixture of elements and concepts.

The *Synchronized Multimedia Integration Language* (SMIL) 2.0 [SMIL 2.0] is a declarative, XML-based description language for interactive, animated multimedia applications on the Web. As such it is not tailored to 3D graphics, but offers neat functionality with its intuitive time and animation concepts integrating discrete and continuous media types. Since SMIL 2.0 became quite complex, related elements, attributes, and attribute values were grouped into modules and profiles. The animation and timing & synchronization modules, particularly the synchronization and grouping elements influenced the development of BEHAVIOR3D. The work of [Kemkes 2001] already sketches a possible integration of SMIL concepts into X3D and was considered here too.

Other behavior-related work includes research on the integration of various input devices and their mapping to scene behavior [Althoff et al. 2002, Figueroa et al. 2002]. It was not integrated into BEHAVIOR3D yet. Research on constraints, e.g. by [Diehl and Keller 2000] and [Codognet and Richard 1998], was considered for further language extensions and new behavior collections.

Surveying related work one can observe, that various declarative languages exist for defining Web3D behaviors. They are partly XML-based and offer various interesting concepts. Some formats are specialized for behavior declarations in specific domains (e.g. character animation). There is no single format integrating every behavior concept and offering coherent extensibility mechanisms.

3. Defining Behavior in X3D – Prospects and Shortcomings

3.1 Format Requirements and Choice of X3D

As the basis of this work a number of general requirements was made for a powerful behavior definition concept. The format should be declarative, thus being easy to read and use even for non-experts. Moreover, it should serve as an exchange format for behavior definitions independent from specific 3D technologies and form the basis for authoring tools. It is desirable to separate the behavior graph from other scene definitions with the objective of readability, maintainability, and reusability. A rich and extensible set of behavior modules must be available. That suggests a modular concept based on object-oriented principles. Nodes must be ordered in appropriate hierarchies employing inheritance. Another important goal for developing an expressive 3D behavior concept is the usage of standard formats where possible or at least to interoperate with them. That is one of the reasons why VRML97 / X3D was considered as a suitable basis for this work. It is a powerful and general-purpose format fulfilling some of the mentioned requirements and providing various extensibility mechanisms.

However, why is it not favorable to simply use these mechanisms and create additional behavior nodes with X3D? This chapter attempts to answer this question by looking closer at the X3D capabilities for defining behavior and adding new functionality. In conjunction with this analysis further requirements are defined for an improved behavior concept related to X3D.

3.2 Using Built-In Behavior Nodes

X3D offers various built-in nodes for defining simple object animations and interactions according to behavior levels 0 and 1 as defined in [Roehl 1995]. Among them are nodes such as time, sensors, interpolators, triggers, and sequencers. X3D nodes are divided into so-called *Components*, whereas this term refers to functionally related X3D objects, typically node collections. The behavior nodes are arranged in the components *Environmental Sensor, Key device sensor, Pointing Device Sensor, Interpolation, Event Utilities,* and *Time*. There are many recurring application scenarios, where these pre-defined nodes are not sufficient, particularly for complex animations or state-based modeling of 3D scenes. That means authors have to use the *Script* node described in section 3.3 to realize even common 3D functionality.

With the ongoing development of the X3D XML Schema [X3D-Schema] and the Scene Authoring Interface (SAI) [SAI] existing nodes are arranged in a node hierarchy. These steps towards node classification, stronger typing, and node inheritance facilitate both the usage of nodes and their implementation. Inheritance should be applied consequently, so that newly defined nodes profit from their derivation from existing nodes. Figure 1 depicts a part of the already proposed hierarchy and the X3D behavior components. The behavior node hierarchy needs to be extended. All existing, built-in X3D nodes should be included in the new behavior concept due to adoption of good concepts and backward compatibility.

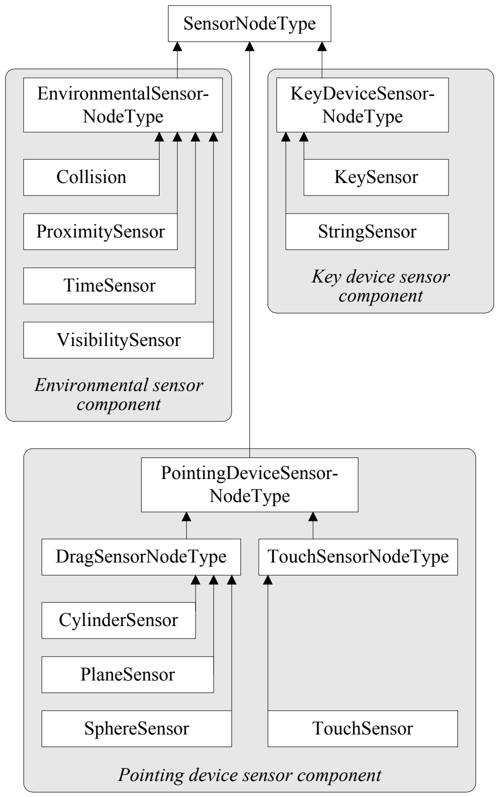

Figure 1. Part of the X3D SAI node hierarchy and grouping of nodes within X3D components

3.3 Adding Behavior via Script Nodes

The *Script* node allows authors to write arbitrary event processing code and perform computations within a 3D scene. It can be seen as a link between the declarative world of nodes and imperative programming. The need for integrating complex, previously not existing behavior via some sort of programming is evident for any non-trivial 3D application. However, the script node should not be needed to code common functionality again and again.

Script nodes have a number of disadvantages. They do not integrate into the object hierarchy and rather constitute an independent concept. Programmers would like to use inheritance for script nodes too. Reusability of scripts is not well supported; it can only be accomplished by wrapping them in prototypes. Debugging of script languages (e.g. JavaScript) is difficult and time-consuming in script nodes. Field definitions in script nodes do not allow safe typing. Assuming a field referencing a node, authors can only specify data types *Node* or *Nodes*, not the particular node types, e.g. *InterpolationNode* or node types derived from it. There is the clear need for polymorphism and strong typing. Another known problem with fields in VRML97 scripts is the prohibition of access type *exposedField* in field declarations. This problem was not completely eliminated in X3D yet. There should be a unique handling of access types throughout the format. A last problem to be mentioned is the confusing mixture of general scene nodes, behavior related nodes, script nodes, and ROUTE statements within a document. Parts should be clearly separated for improved maintenance and readability.

3.4 Creating new X3D Nodes with Prototypes

Using the prototype concept new nodes can be defined in terms of sub graphs of already existing nodes. The X3D specification [X3D Specification] states, that once defined they can be instantiated like built-in nodes. However, this is not entirely true, particularly not for the current X3D XML encoding. Assuming a prototype being defined in a separate document, an author would first need to use an *Externproto* definition before actually using the new node. In the X3D VRML97 encoding this looks like:

```
EXTERNPROTO AnimateRotation [
    field  MFFloat  key
    field  MFRotation  to
    …
] ["File.wrl"]
…
AnimateRotation {
    key  [ 0 1 ]
    to  [ 1 0 0 -1.7, 1 0 0 0 ]
}
```

In XML syntax the new node really becomes a second-class node, since it has always to be wrapped within a *ProtoInstance* element as shown in the following document fragment.

```
<ExternProtoDeclare name="AnimateRotation" url="File.x3d">
    <field accessType="field" name="key" type="Floats"/>
    <field accessType="field" name="to" type="Rotations"/>
    …
</ExternProtoDeclare>
…
<ProtoInstance name="AnimateRotation">
    <fieldValue name="key" value="0 1"/>
    <fieldValue name="to" value="1 0 0 -1.7, 1 0 0 0"/>
</ProtoInstance>
```

Instead of writing a *ProtoInstance* statement, the direct usage of the new node via its name would be desirable. Another disadvantage of both encodings is the lengthy repetition of the *Externproto* field interface before actually using it in the scene. As a consequence new behavior definitions should be liberated from the burden of out-dated VRML concepts and their associated syntax.

As with script nodes, another disadvantage of prototypes is again the missing integration into the X3D XML Schema or SAI object hierarchy. It is yet another concept within X3D and does not homogeneously integrate with nodes and scripts. Moreover, object-oriented features, such as inheritance, polymorphism, a safe type concept etc. would be desirable. They should seamlessly integrate into X3D and do not form an additional concept. Authors shall be able to create new behavior nodes and integrate them into the existing hierarchy. While using a new node in an X3D document, users should not need to reflect, whether this node is built-in, a script node or some other node extension. All nodes should be first class nodes and require the same homogeneous syntax.

4. Behavior3D

After having looked at related work and the behavior and extensibility mechanisms of X3D, this chapter introduces the novel BEHAVIOR3D concept in detail. Figure 2 illustrates the two levels behavior node development and usage with all associated grammars and instance documents. Thus it serves as an overview of the whole BEHAVIOR3D concept. First of all the general node concept for defining behavior is explained in section 4.1. In the following section a new XML Schema grammar *Behavior3DNode* is introduced for describing such behavior nodes at the node development level. Once these nodes are defined, it should be easily possible to use them as first class nodes in a behavior graph. For that purpose another grammar, *Behavior3D*, was designed for the node usage level. It is an automatically generated XML Schema integrating all available behavior nodes, thus providing a repertoire of behavior definitions. This grammar is described in section 4.3. The concept of behavior node collections will be introduced afterwards. In Figure 2 they are sketched at the development level. Collections constitute reasonable behavior modules comparable to X3D components. After all the actual implementation of behavior nodes is described in chapter 5.

All grammars and instance documents of BEHAVIOR3D are coded with XML. The decision for this hierarchical document definition format was made because of its interoperability, easy processing, standardized form, widespread use, and general usability. In particular XML Schema [XML Schema] was chosen because of its partial support of object-oriented features (e.g. substitution groups), namespaces, extensibility (e.g. type extensions) as well as its improved type concept.

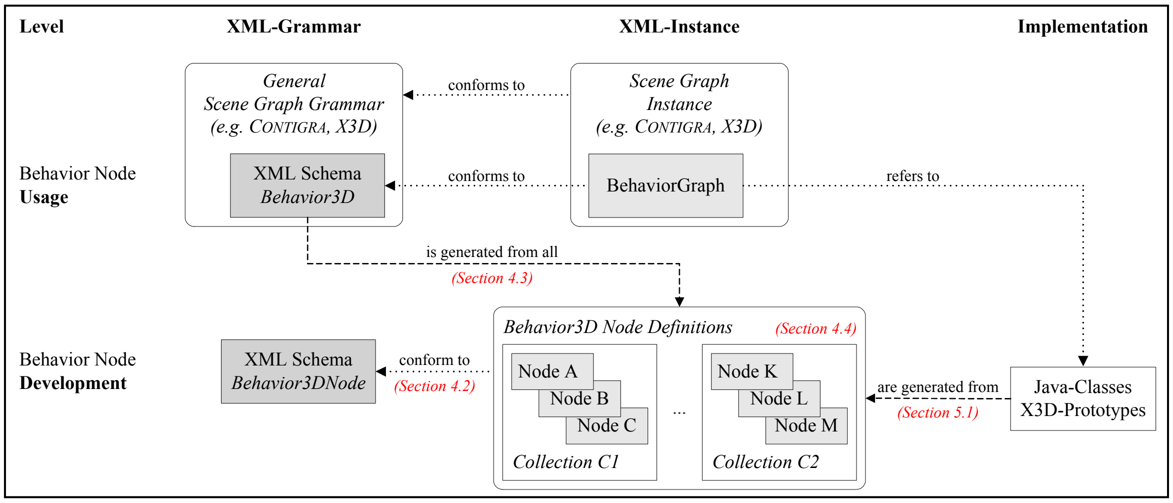

Figure 2. Overview of the different BEHAVIOR3D levels, grammars, and instance documents

4.1 Basic Node Concept

BEHAVIOR3D comprises a generic object-oriented node concept merging built-in nodes, scripts, the prototype concept of X3D, and the class concept of VRML++ [Diehl 1997]. Furthermore an improved field concept was added. A behavior node is a constituting part of the behavior graph and represents certain functionality within a 3D application. Every node might possess one or many typed fields for defining characteristics of this node.

4.1.1 Fields

Every field has a name, a type, a value, three change modes, and a possible default value. All X3D field types and all existing BEHAVIOR3D nodes can be used as field types. The change modes define the access time of a field. Field values can be assigned during authoring time (mode *configurable*), can be changed during runtime receiving an event (mode *receivesEvents*), and can generate events after a change of their value (mode *generatesEvents*). The default value of a field can be interpreted as an initial value and must be set if *configurable* equals true, otherwise it is not used. Table 1 lists all possible change mode combinations of a BEHAVIOR3D field and compares valid value triples to X3D field access types.

Table 1. Change Modes of Behavior3D-Fields

	Combinations			corresponds to X3D field access type
	configurable	***receives Events***	***generates Events***	
1	false	false	false	-
2	false	false	true	*eventOut*
3	false	true	false	*eventIn*
4	false	true	true	-
5	true	false	false	*field*
6	true	false	true	-
7	true	true	false	-
8	true	true	true	*exposedField*

The comparison of the eight combinations to the X3D field access types reveals a richer expressiveness of the proposed change modes. The first combination, where no value can be assigned to a field at any time, has no practical relevance. However, combinations 4, 6, and 7 provide a clear definition for typical application scenarios. Setting 4 can be used for a field, which cannot be configured but only changed at runtime. This could for example be an IP address, which is dynamically assigned during runtime. Setting 6 is typically used for some field reflecting a current state, number, or item etc. It can be set at configuration time, generates events during runtime, but cannot be changed directly at runtime. Setting 7 can be conveniently used for values initialized at configuration time and changed during runtime. A typical example might be a font style field that does not need to generate events.

4.1.2 Inheritance and Composition

Nodes can inherit from other nodes. Only single inheritance is suggested within BEHAVIOR3D to avoid problems of multiple inheritance. Node B derived from parent node A inherits all fields and the implementation of A, respectively the event handling. That means node B possesses at least all fields of A, but could add new fields, add new implementation methods, and overwrite parent node methods. Nodes can be abstract, which prohibits their instantiation. The notion of abstract nodes allows node designers to shift shared fields and implementations of different nodes to a common abstract parent node. Thus implementation stability will be improved because of frequent use in inheriting nodes. Figure 3 depicts a small portion of the proposed BEHAVIOR3D hierarchy to illustrate the concept.

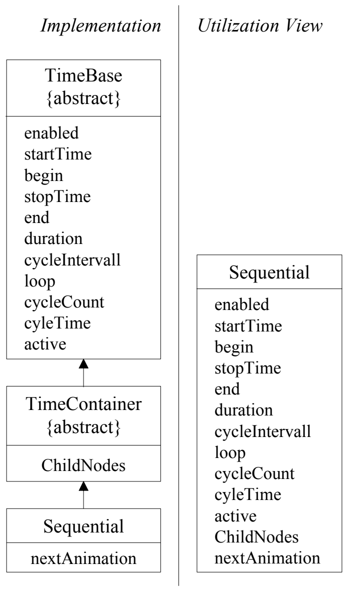

Figure 3. Inheritance in BEHAVIOR3D

The abstract node *TimeBase* provides the basis for all time- and animation-related behavior nodes. Various fields are defined; a basic implementation is provided with them. *TimeContainer* is another abstract node inheriting from *TimeBase*. It adds a new field to accommodate other time-related behavior nodes. With *Sequential* the first actually instantiable node is derived from *TimeContainer*. It adds another field, *nextAnimation*. The user of this node will be offered the complete interface including all inherited fields as shown on the right. This concept allows the creation of new behavior nodes based on already existing nodes. Stability and maintainability of code will be improved, extensibility is guaranteed.

Beside inheritance *node composition* is proposed as another method of reusing nodes. Instances of existing nodes can be referenced and used inside a newly defined node. Thus their functionality is reused, but not their structure and complete set of fields as with inheritance.

4.1.3 Polymorphism and Typing

Every field in BEHAVIOR3D is typed. In comparison to X3D field types can be substituted by all other types inheriting from it. This applies in particular to nodes as field types. One specific behavior node can be demanded as a type of a field. Polymorphism allows all nodes derived from this parent node to be substituted for the field both at configuration and runtime. This stronger typing concept helps preventing runtime errors while instantiating nodes.

4.2 Declaration of new Behavior Nodes

This section introduces the new XML Schema grammar *Behavior3DNode*. It represents the realization of the general node concept and allows the description of a behavior node's interface

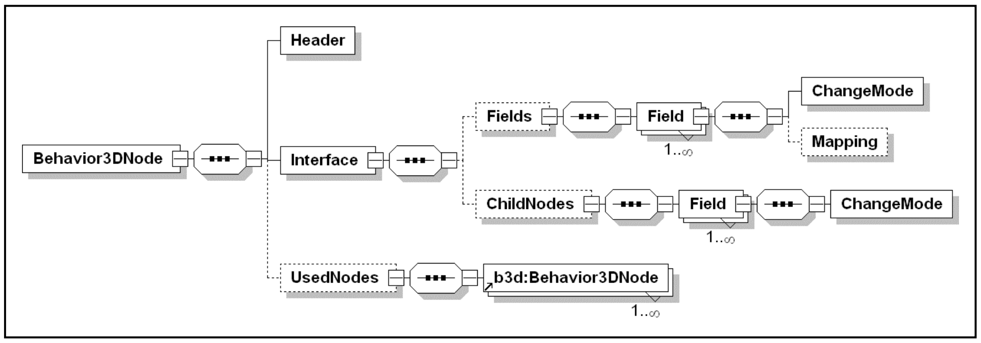

Figure 4. A diagram of the *Behavior3DNode* XML Schema

at the development level. Every new node consists of a valid instance document of this grammar plus some implementation files. The node interface document is used to automatically generate the appropriate implementation files as well as the whole language repertoire of behavior nodes.

4.2.1 Behavior3DNode

Figure 4 depicts an XML Schema diagram of the *Behavior3DNode* grammar. Dashed lines depict optional elements. The symbol with squares in a row denotes a sequence of elements; the arrow stands for an element reference. A behavior node declaration consists of a *Header,* an *Interface,* and a *UsedNodes* part. The *Header* element contains the self-explanatory attributes *name* and *documentation* as well as *collection*. The latter determines this node's affiliation to behavior node collections (see section 4.4).

The fields particularly define the characteristics of a node. They are divided into two groups. The first group *Fields* contains an enumeration of *Field* elements with non-node data types. These are types such as *Color, Rotation* etc. They represent the actual fields of a behavior node, which are later used by setting the corresponding XML attributes. The second group *ChildNodes* contains a definition of all possible behavior child nodes, which are modeled as field declarations of node data types. These node-fields later appear in the compact XML syntax as child elements instead of attributes. Thus they are comparable to children of a grouping node.

According to the field concept described in section 4.1.1 every field contains the attributes *name, dataType, default,* and *description*. The element *ChangeMode* contains the three Boolean attributes *configurable, receivesEvents,* and *generatesEvents*. The second element *Mapping* allows a mapping of this field to an already existing field of another referenced BEHAVIOR3D node. To continue, the *Interface* also contains two attributes to realize inheritance. Attribute *extends* allows the specification of a parent node from which this node inherits. Attribute *nodeType* can contain the values *public* or *abstract* to define, whether it is allowed to create instance of this node or not.

Within the element *UsedNodes* other behavior nodes can be referenced, which are used and needed by this node in terms of node composition. The *Mapping* element establishes the connection from fields of the currently defined node to fields of the used nodes. The indication of used nodes also helps tools to check their availability in terms of interface and implementation documents.

4.2.2 Examples

The following two examples illustrate, how behavior nodes are defined with the previously described grammar. At first the declaration of the behavior node *TimeContainer* is given:

```
<Behavior3DNode>
    <Header name="TimeContainer" collection="Animation"/>
    <Interface nodeType="abstract" extends="TimeBase">
        <ChildNodes>
            <Field dataType="TimeBase" minOccurs="0"
             maxOccurs="unbounded" description="The
             Animations, which should be controlled.">
                <ChangeMode configurable="true"
                 receivesEvents="false"
                 generatesEvents="false"/>
            </Field>
        </ChildNodes>
    </Interface>
</Behavior3DNode>
```

The attribute *collection* declares this node to belong to the behavior node collection *Animation* described in section 4.4.1. The node is of abstract type, as indicated with the *nodeType* value. A *TimeContainer* has no declared attribute fields (i.e. the *Fields* part is missing) yet possible *ChildNodes*. They are defined as fields of a complex data type, i.e. of one of the node data types. By means of polymorphism non-abstract and also derived nodes can be used within instance documents. In this case a *TimeContainer* allows zero to infinite nodes to be used as children, which are derived by the abstract node type *TimeBase*. *ChangeMode* indicates, that child elements can only be added at configuration time and must not be changed during runtime.

In the second example the *Sequential* node is declared, which inherits from *TimeContainer* as declared with the attribute *extends*.

```
<Behavior3DNode>
    <Header name="Sequential" collection="Animation"/>
        <Interface nodeType="public" extends="TimeContainer">
        <Fields>
            <Field name="nextAnimation" dataType="Boolean"
             description="Stops the current animation and starts
             the next animation">
                <ChangeMode configurable="false"
                 receivesEvents="true" generatesEvents="false"/>
            </Field>
        </Fields>
    </Interface>
</Behavior3DNode>
```

In contrast to its abstract parent node this node can be instantiated. No child nodes are declared, but one additional field *nextAnimation* of Boolean type. It cannot be configured before runtime and can only receive events. As a result it stops the currently running animation and triggers the next animation defined in the sequence of actions within a *Sequential* node.

It has to be noticed, that all fields of a basic data type – which are defined within the *Fields* element – will be translated to XML attributes of the grammar described in the next section. All node-typed fields declared within the element *ChildNodes* (as in *TimeContainer*) will be translated to XML content elements of the defined node, i.e. a sequence of elements of the given type.

4.3 Defining the Behavior3D Node Repertoire

The behavior nodes defined with *Behavior3DNode* shall be used within scene graphs on the usage level as shown in Figure 2. To achieve their integration one could use a generic language construct comparable to X3D prototypes. However, once a collection of behavior nodes was defined it would be preferable to use these nodes as first class nodes in a behavior graph. That means, instead of writing statements like `<ProtoInstance name="AnimateRotation"…` as described in section 3.4, it would be better to directly include `<AnimateRotation key="0 1" … />` in a behavior definition. As a consequence all (newly defined) behavior nodes should be specified by a grammar in the same way like already existing nodes of the general scene graph grammar. For that purpose the new XML Schema *Behavior3D* was developed, which represents the complete repertoire of available behavior nodes. The idea is to automatically generate this grammar from a given number of document locations. Within the translation process, all *Behavior3DNode* instance documents – each representing a single behavior node – are collected and transformed to be part of the whole grammar. Since this grammar is dynamically generated, it always includes the whole available repertoire of behavior nodes. An XSLT [XSL] stylesheet was developed to do the actual translation and grammar generation work. This process can be activated e.g. every time an authoring tool will be started. The process is illustrated with a single node definition according to the *Behavior3DNode* grammar. An *AnimateRotation* node is defined at the development level:

```
<Behavior3DNode>
  <Header name="AnimateRotation"/>
  <Interface/>      <!-- Interface part omitted -->
</Behavior3DNode>
```

The following XML fragment shows the result of the translation process. It is the generated definition for the *AnimateRotation* node, which will be one part of the *Behavior3D* grammar.

```
<element name="AnimateRotation" type="AnimateRotationType"
         substitutionGroup="Animation"/>
<complexType name="AnimateRotationType">
    <complexContent>
        <extension base="AnimationType">
            <attribute name="key" type="x3d:Floats"/>
            <attribute name="to" type="x3d:Rotations"/>
            <attribute name="by" type="x3d:Rotations"/>
        </extension>
    </complexContent>
</complexType>
```

Through the import of *Behavior3D* into other scene graph grammars all nodes are immediately available as first class elements. The third example shows part of the actual behavior graph instance document on the usage level. One can notice, how *AnimateRotation* is directly used in comparison to the X3D examples given in section 3.4.

```
<Sequential begin="5.0">
    <AnimateRotation key="0 1" to="1 0 0 0, 1 0 0 -1.5"/>
</Sequential>
```

4.4 Behavior Node Collections

The previous examples already introduced some of the new behavior nodes. However, with BEHAVIOR3D a larger number of nodes were defined including all behavior-related X3D nodes mentioned in section 3.2. The concept of *behavior node collections* is proposed to group functionally and semantically related nodes. The following collections were already defined: *Time, Event Utilities, Interpolation,* and *Environmental Sensor* including behavior nodes from existing X3D components and additional nodes for type conversions, logical operations etc.; general *Device Sensor* collections including X3D's *Pointing Device Sensor* and *Key Device Sensor.* In the future these collections could also include other devices nodes as for example described in [Althoff et al. 2002]. Two other collections were not only completely specified in terms of *Behavior3DNode* instance documents, but also implemented as Java classes: *Animation* and *State Machine.* Since these collections have no equivalent in X3D they shall be introduced in the following sections in detail. We also propose a *Constraints* collection, which was not defined yet.

4.4.1 Animation

This behavior node collection was particularly developed to overcome the shortcomings in defining complex animations in Web3D formats such as X3D, Shockwave3D, and Viewpoint. On the other hand SMIL 2.0 allows the intuitive declaration of basically 2D animations. Therefore important SMIL 2.0 concepts were adapted from the *Animation Modules* and *Timing and Synchronization Module* [SMIL 2.0] to the field of 3D graphics. Three abstract nodes form the basis of this collection, *TimeBase* and the inheriting nodes *Animation* and *TimeContainer*. Figure 5 shows the inheritance diagram for all nodes of the Animation collection.

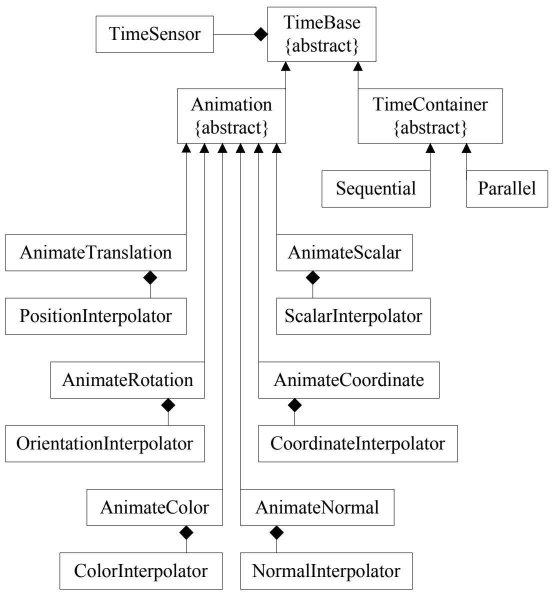

Figure 5. Inheritance diagram of the Animation collection

Arrows indicate node inheritance, diamonds node composition. One can notice the usage of X3D interpolator nodes and the X3D *TimeSensor* node as parts of the new nodes.

Six instantiable nodes are inheriting from abstract node *Animation*: *AnimateTranslation, AnimateRotation, AnimateColor, AnimateScalar, AnimateCoordinate,* and *AnimateNormal*. The behavior node *AnimateTranslation* shall be described as a representative example here. Its interface is shown in Table 2.

Table 2. Interface of Node *AnimateTranslation*[1]

	Field Name	Data Type	Default	
▶	enabled	Boolean	true	▶
▶	startTime	Time	0.0	▶
	begin	Time	0.0	
▶	stopTime	Time	1.0	▶
	end	Time	0.0	
	duration	Time	0.0	
▶	cycleInterval	Time	1.0	▶
▶	loop	Boolean	false	▶
	cycleCount	Float	0.0	
	cycleTime	Time		▶
	active	Boolean		▶
	accumulate	Boolean	false	
	calcMode	String	linear	
	keySplines	Strings	[]	
▶	key	Floats	[]	▶
▶	to	Vector3FloatArray	[]	▶
	by	Vector3FloatArray	[]	
	out	Vector3Float		▶

The upper fields are inherited from *TimeBase*. A node-internal time was introduced in addition to the global system time. With the fields *begin, end,* and *duration* one can assign, when an animation starts, ends and how long it lasts after the node was activated. The following XML fragment shows a possible application of these fields.

```
<AnimateTranslation begin="3.0" end="8.0" cycleInterval="10"
  key="0.0, 0.5, 0.99" to="-1 1 0, 0 0 0, 1 -1 0"/>
```

This simple animation starts 3 seconds after the *startTime* event was received. It ends after 5 seconds. As a consequence the animation cycle with its duration specified by *cycleInterval* is only half done when the animation stops. The next example illustrates an animation starting 3 seconds after receiving the *startTime* event and lasting for 3 seconds as specified in the *duration* field.

```
<AnimateTranslation begin="3.0" duration="3.0" cycleInterval="10"
  key="0.0, 0.5, 0.99" to="-1 1 0, 0 0 0, 1 -1 0"/>
```

The fields *to* and *by* are possible ways to define key values as absolute or relative delta vectors. According to the general time settings *AnimateTranslation* generates *Vector3Float* events with the *out* field, which are typically routed to a *Transform* node to animate the position of all subsequent nodes.

[1] The tables read as follows: Arrows on the left depict fields receiving, arrows on the right generating events. Fields with a given default value implicate change mode *configurable* = true. The thick lines separate fields inherited from different nodes; fields added by the youngest node are shown at the bottom.

The following definition of an *AnimateScalar* node illustrates another feature of all nodes derived from *Animation*. Intervals can be used to cycle through simple animations and even accumulate their results.

```
<AnimateScalar cycleInterval="5.0" cycleCount="3.0"
  accumulate="true" key="0.0, 0.5, 0.99" by="10, 5"/>
```

The single animation lasts 5 seconds and will be repeated three times. Since values are accumulated with *accumulate* set to true, the following results will be generated: starting with 0, after 2.5 seconds 10, then 5 at the end of the first cycle. The initial value used for the next iteration is 5 (not 0). At the end of the second cycle 10 is generated, after the third cycle it is 15. All nodes inheriting from *Animation* also possess the field *calcMode*. This field defines the interpolation mode for values at key points. Possible values are *discrete, linear, paced,* and *spline*. Their meaning equals the definitions of the SMIL Animation Modules.

Two instantiable nodes are derived from abstract node *TimeContainer* within the *Animation* collection: *Sequential* and *Parallel* (see Figure 5). They are grouping and synchronization nodes supporting behavior definitions of level 2 according to [Roehl 1995]. All animations contained in a *Parallel* node will be started at the same time respectively executed in parallel. The *Sequential* node activates all contained actions in sequence, which is illustrated by the following example.

```
<Sequential begin="1.0" cycleInterval="5.0">
    <AnimateTranslation key="0 1" to="0 0 0, 0 0.05 0"
      cycleInterval="2.0"/>
    <AnimateRotation key="0 1" to="1 0 0 0, 1 0 0 -1.5"
      cycleInterval="3.0"/>
</Sequential>
```

Nodes *AnimateTranslation* and *AnimateRotation* are grouped and executed after each other. Attribute *begin* of the *Sequential* node indicates a start of the first animation 1 second after having received the *startTime* event. After 2 seconds *AnimateTranslation* is finished and *AnimateRotation* is started. In addition to the fields inherited from its parent nodes, *Sequential* defines a new field *nextAnimation*, which can only receive events. When set to the Boolean value true, the current animation is stopped and the next animation started as defined in the sequence.

4.4.2 State Machine

This behavior node collection serves the intuitive description of behavior and interaction with connectable state machines. The concept developed in Viewpoint [Viewpoint] served as a basis for these nodes. According to [Roehl 1995] behavior definitions of level 3 can be expressed with the *State Machine* collection. Figure 6 depicts the inheritance diagram for the nodes of this collection.

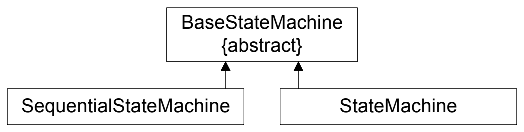

Figure 6. Inheritance diagram of the State Machine collection

The node *SequentialStateMachine* allows the realization of a state machine running sequentially through its states. The second node inheriting from the abstract *BaseStateMachine* is *StateMachine*. With this node flexible state machines can be modeled through explicitly specifying arbitrary states and state transitions. Table 3 shows the interface of this node.

Table 3. Interface of Node *StateMachine*

	Field Name	Data Type	Default	
	stateCount	Integer	1	
	currentState	Integer	1	▶
	transitions	Strings	[]	

The *currentState* field inherited from *BaseStateMachine* was defined with the following change modes: configurable="true" receivesEvents="false" generatesEvents="true". That means, an initial state can be set, but at runtime *currentState* is just changed by the internal state transitions. This is an example for a field access type not directly expressible with X3D and illustrates the advantages of the field concept explained in section 4.1.1.

The important field added by *StateMachine* is *transitions*. Every state transition within this list of *stateCount* transitions will be specified with a value quadruple. The following XML fragment shows an example with 3 defined states as indicated with the attribute *stateCount*.

```
<StateMachine stateCount="3" transitions="
 1 2  LCD_Sensor.touchTime  OpenLaptop.startTime,
 2 1  LCD_Sensor.touchTime  CloseLaptop.startTime,
 2 3  Keyboard_Sensor.touchTime  OpenKeyboard.startTime,
 3 2  Keyboard_Sensor.touchTime  CloseKeyboard.startTime"/>
```

Since states are represented as integer values, the first entries of the quadruple declare the start and end state of a specific transition as integers. The third value indicates the behavior or interaction, which actually triggers the state transition. The last value holds the behavior to be triggered as a result of the state transition. Both values are references to already defined behavior nodes using their DEF attribute. The substring after a dot indicates the field of a specified node, which must be of type *time*.

5. Integration and Implementation

In the previous chapters the basic BEHAVIOR3D concept and its realization with XML Schema were introduced. Though BEHAVIOR3D was inspired by X3D, it is an independent behavior language. However, behavior declarations can be translated to X3D documents in order to be actually run in an appropriate 3D viewer. This chapter explains the realization with X3D, sketches the successful integration into the project CONTIGRA and demonstrates an interactive application example.

5.1 Implementation with X3D

Beside the XML interface document of a behavior node there exist two partly automatically generated Java files for each node. Java was chosen as the implementation language instead of JavaScript to make use of its object-oriented and other language features. Moreover, projects in the context of X3D, such as the XJ3D player [XJ3D], are also being developed in Java and form a possible basis for future integration. Our current implementation is based on the Java platform scripting reference of the VRML97 standard, since the X3D SAI-implementation is not available yet. Take for example a new behavior node called *ExampleNode*. Its interface is declared in *ExampleNode.xml*. With the help of XSLT stylesheets all implementation documents are automatically generated. Figure 7 depicts the transformation process from this *Behavior3DNode* instance document to the corresponding Java and X3D documents.

With the stylesheet *NodeTemplate.xslt* the Java class *ExampleNodeTemplate.java* will be generated. This class contains access methods for the fields of a node and corresponding attributes, which are automatically initialized. For every field with change mode *receivesEvents*=true methods will be generated for receiving events. All other initializations, such as the processing of field mappings to used nodes, are also generated.

The *ExampleNodeTemplate.java* class inherits from the behavior class specified in the attribute *extends* of the node interface declaration. This way the emerging Java class hierarchy exactly represents the declared behavior node hierarchy. The base class for all behavior nodes is *BaseNode.java*, so that all template classes will indirectly inherit from it. This class takes care for correct initialization of nodes, general event handling using Java reflection, as well as managing debugging information. It is itself derived from class *Script.java* of the VRML97 Java platform scripting reference.

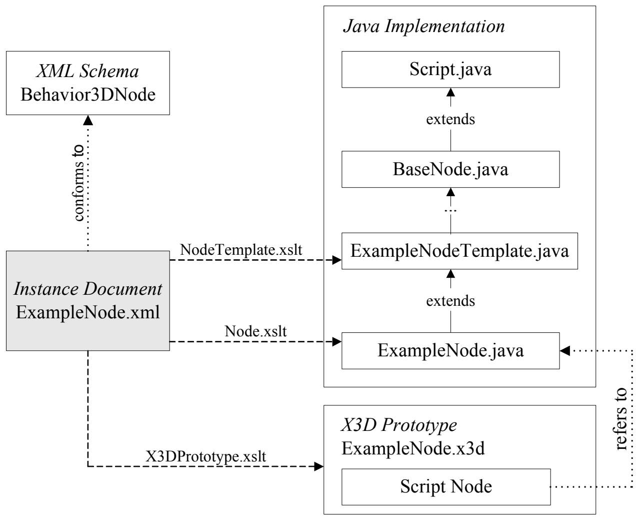

Figure 7. Transformation process for a BEHAVIOR3D node

With the stylesheet *Node.xslt* the Java class *ExampleNode.java* will be generated, extending the parent class *ExampleNode Template.java*. As opposed to its parent class, which should not be changed at all, this class only supplies an initial code frame, where custom code needs to be added to realize the actual behavior. Event processing methods inherited from the parent class can also be extended. Both automatically generated Java classes considerably facilitate the implementation of functionality and reduce programmer's work.

Since an X3D scene graph cannot use node definitions of the proposed XML *Behavior3D* grammar and the associated Java classes per se, an X3D wrapping becomes necessary. The lower part of figure 7 shows the translation of *ExampleNode.xml* to an X3D prototype node with the stylesheet *X3DPrototype.xslt*. The generated prototype basically declares its fields corresponding to the node interface. It also contains an X3D script node, which itself references the Java class *ExampleNode.java* described above. Within the script node's interface three fields are generated for every field declared in *ExampleNode.xml,* i.e. *fieldName, set_fieldName,* and *fieldName_changed*. This allows the easier mapping of BEHAVIOR3D change modes and provides more implementation flexibility. The automatically generated

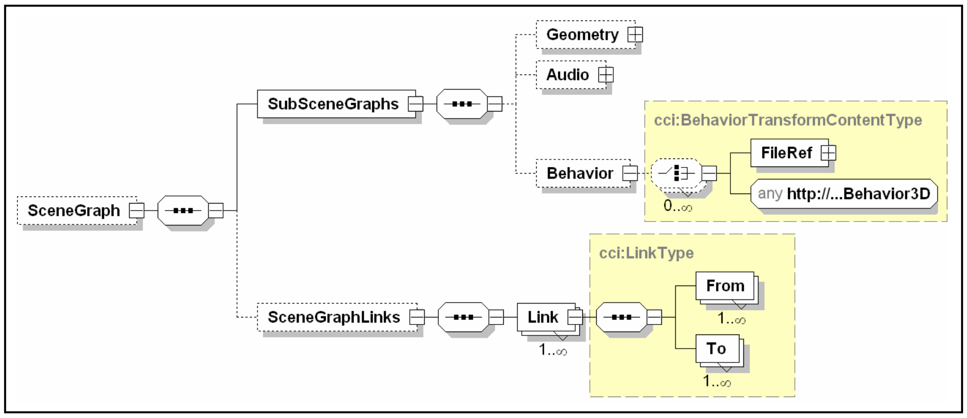

Figure 8. Part of the XML Schema hierarchy for component implementations

X3D behavior prototype can be used in arbitrary X3D scene graphs. It is to be mentioned, that once the Scene Authoring Interface (SAI) will be available, it will significantly reduce the complex translations described in this section. However, it could be shown, that BEHAVIOR3D can be easily implemented on the basis of X3D and delivers functioning results.

5.2 Integration into Contigra

As already mentioned earlier, the BEHAVIOR3D approach was developed as part of the CONTIGRA project [Dachselt et al. 2002, Contigra]. Within that project a 3D component is declaratively described in two separated XML documents, containing its interface declaration and implementation. Components might contain subcomponents, arranged in a component graph as a method of composition. In addition to that a component usually consists of various scene graph parts, particularly in the case of a single component without sub-components. In this section the focus is laid on the *SceneGraph* part. Figure 8 depicts this section of the implementation grammar's structure. The basic diagram symbols are explained in section 4.2.1. A switch indicates a choice of elements.

One can observe a strict separation of the three different scene graph hierarchies *Geometry, Audio,* and *Behavior*. As opposed to many 3D scene graph technologies, with CONTIGRA a clear separation of these graphs is enforced for the reason of better reusability, easier maintainability, exchangeability, and clarity. A component's behavior graph is a transformation hierarchy consisting of behavior nodes made available through the included, automatically generated grammar *Behavior3D* as described in section 4.3. That means behavior nodes such as *Sequential, StateMachine, AnimateTranslation* etc. can be directly used within the implementation document of a 3D component.

As a consequence of the *SubSceneGraphs* division, the separated link section *SceneGraphLinks* will become necessary to actually connect related nodes of these graphs. In order to use more powerful node and field connections than possible with single event-based ROUTE-statements, an extended link concept was developed, where links convey a specific semantics and form n:m relations.

CONTIGRA components can be translated to various 3D formats. Due to the close orientation towards X3D it was natural to first develop a translator to that format. As a proof of concept components with all their sub-components and sub scene graphs can be translated to valid X3D/VRML97 documents using again XSLT stylesheets. In this process the X3D prototypes and Java classes generated for each behavior node are employed.

5.3 Application Example: Interactive Laptop

To illustrate the practical application of the BEHAVIOR3D concept one of the created examples will be introduced in this section. By touching its lid a laptop shall be opened. After touching the keyboard it should fold up. Both actions should be invertible. It is forbidden to close the laptop while the keyboard is folded up. The geometry of a laptop[2] consisting of three parts was the basis for the behavior description of this scenario. The whole laptop was realized as a CONTIGRA component. The following XML code shows the complete behavior description in terms of the behavior graph. In addition to that, scene graph links are necessary to connect behavior to geometry nodes. They are left out here, since they are straightforward and not a direct part of the behavior concept.

```
<TouchSensor DEF="LCD_Sensor"/>
<TouchSensor DEF="Keyboard_Sensor"/>

<StateMachine stateCount="3" transitions="
  1 2  LCD_Sensor.touchTime  OpenLaptop.startTime,
  2 1  LCD_Sensor.touchTime  CloseLaptop.startTime,
  2 3  Keyboard_Sensor.touchTime  OpenKeyboard.startTime,
  3 2  Keyboard_Sensor.touchTime  CloseKeyboard.startTime"/>

<AnimateRotation key="0 1" to="1 0 0 0, 1 0 0 -1.7"
  cycleInterval="2" DEF="Openlaptop "/>
<AnimateRotation key="0 1" to="1 0 0 -1.7, 1 0 0 0"
  cycleInterval="2" DEF="CloseLaptop"/>

<Sequential DEF="OpenKeyboard">
    <AnimateTranslation key="0 1" to="0 0 0, 0 0.05 0"
     cycleInterval="1" />
    <AnimateRotation key="0 1" to="1 0 0 0, 1 0 0 -1.5"
     cycleInterval="1" />
</Sequential>
<Sequential DEF="CloseKeyboard">
    <AnimateRotation key="0 1" to="1 0 0 -1.5, 1 0 0 0"
     cycleInterval="1" />
    <AnimateTranslation key="0 1" to="0 0.05 0, 0 0 0"
     cycleInterval="1" />
</Sequential>
```

[2] Laptop courtesy of Richard Choi (http://www.web3d.co.kr)

First of all two *TouchSensors* are defined, which are associated with the LCD and keyboard geometry of the laptop. This association is usually established in X3D through the insertion of sensors into the appropriate scene graph part. With this concept they are separately defined and inserted later during the translation process with the help of a link statement to the geometry graph.

Afterwards one can notice the modeling of the three states, which are visualized in Figure 9. The laptop can be closed, opened, and opened with the keyboard folded up. State transitions are declared with associated triggers and resulting behavior.

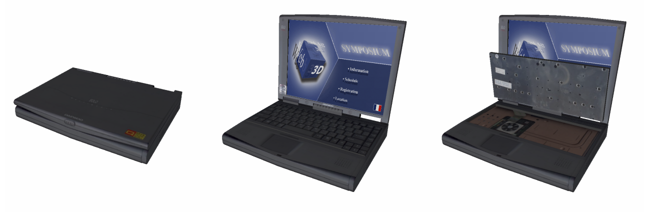

Figure 9. The three states of an interactive laptop

With the help of two *AnimateRotation* nodes the opening and closing of the laptop is realized as a smooth animation. It is sketched in Figure 10 on the left. The two *Sequential* nodes are slightly more complex, since the keyboard is first translated upwards before it is rotated to the back. Folding it down represents the inverse operation. Figure 10 on the right visualizes this animation.

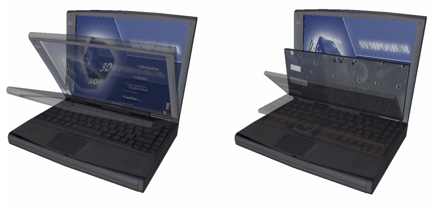

Figure 10. Two animations of the laptop example

6. Discussion and Future Work

In this paper a flexible concept for declaratively modeling 3D object behaviors was introduced. The good potential of X3D and its limitations for declarative behavior definitions were analyzed. Though the BEHAVIOR3D node concept resembles the X3D node model, node inheritance and an improved field concept were added. X3D nodes, parts of the prototype concept and VRML++ were combined to form a coherent concept with object-oriented features for behavior graphs, i.e. inheritance, strong typing, and polymorphism. This results in a considerable increase of node reusability and maintainability.

To be format-independent a new XML-based definition of node interfaces was developed according to the node concept. Implementation templates can be automatically generated from these *Behavior3DNode* instance documents, where custom code additions are reduced to a minimum. The generated classes could also be written in C++ instead of Java. Since the declarative behavior graph is neither dependent on X3D nor CONTIGRA, it could be translated to other 3D technologies, too. However, a technical orientation towards the future Web3D standard naturally implicates best translation results, which are not guaranteed with other technologies.

We also proposed a rich set of predefined behavior nodes categorized in comprehensive collections employing inheritance. Through the consideration of all behavior related X3D nodes a full backward compatibility was achieved. In addition to that the node repertoire was considerably expanded, especially for animation and state machine behavior. Our notion of node collections easily translates to X3D components. In fact, the introduced collections can also be seen as a proposal for new X3D behavior components.

The repertoire of behavior nodes can be made available in an automated fashion through the novel dynamic grammar generation. This way both built-in and new nodes can be syntactically used as first class language elements. The idea of automatically generating a dynamic scene graph grammar could be applied to other X3D extensions as well. Thus the homogeneous usage of all nodes would syntactically improve X3D documents. The proposed concepts are not limited to behavior nodes alone. Node inheritance, improved field access types, automated implementation class generation, and dynamic language extensions could be used as a framework for developing new X3D nodes in general. We hope to stimulate a discussion of these issues with our work.

It was further shown that the suggested behavior definitions are transformable to actual X3D scenes. BEHAVIOR3D was successfully integrated into the CONTIGRA project through defining a separate behavior graph. This demonstrates the practicability of our approach for developing 3D applications. An important limitation of the approach must not be unmentioned. Behavior, interactions, and functionality always need to be implemented *imperatively* at some point. Elegant declarative modeling of behavior must be paid for with more programming “in the back”. Moreover, declarative node connections tend to slow down the execution within a 3D player and shift the responsibility to clever implementations of the player.

As future work more behavior nodes should be defined and implemented. The proposed behavior node collections need to be extended and harmonized with existing X3D components. A visual authoring tool for editing behavior will be built to further ease the creation of interactive scenes for non-experts. Finally, it should be interesting to discuss and to investigate more closely, how the introduced ideas can be integrated into X3D.

7. References

ALTHOFF, F.; STOCKER, H.; MCGLAUN, G.; LANG, M. 2002. A Generic Approach for Interfacing VRML Browsers to Various Input Devices and Creating Customizable 3D Applications. In *Proceeding of the 7th International Conference on 3D Web Technology (Web3D 2002)*, Tempe, Arizona, USA, pp. 67-74.

CODOGNET, P.; RICHARD, N. 1998. Multi-way constraints for describing high-level object behaviours in VRML. In *Proceedings of the Interaction Agents workshop at the AVI'98 conference*, L'Aquila, Italy.

CONTIGRA Project web pages
http://www.contigra.com

Cult3D Designer
http://www.cult3d.com/Cult3D/designer.asp

DACHSELT, R.; HINZ, M.; MEIßNER, K. 2002. CONTIGRA: An XML-Based Architecture for Component-Oriented 3D Applications. In *Proceeding of the 7th International Conference on 3D Web Technology (Web3D 2002)*, Tempe, Arizona, USA, pp. 155-163.

DIEHL, S. 1997. 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)*, Bejing, Asia.

DIEHL, S.; KELLER, J. 2000. VRML with Constraints. In *Proceedings of the Web3D-VRML 2000 fifth symposium on Virtual reality modeling language*, Monterey, California, USA, pp. 81-86.

DÖLLNER, J.; HINRICHS, K. 1998. Interactive, Animated 3D Widgets. In *IEEE Proceedings of Computer Graphics International '98*, Hannover, Germany, pp. 278-286.

FIGUEROA, P.; GREEN, M.; HOOVER, H. 2002. InTml: A Description Language for VR Applications. In *Proceeding of the 7th International Conference on 3D Web Technology (Web3D 2002)*, Tempe, Arizona, USA, pp. 53-58.

KEMKES, A. 2001. X3D and SMIL. http://www.web3d.org/TaskGroups/x3d/perceptronics

KSHIRSAGAR, S.; MAGNENAT-THALMANN, N.; GUYE-VUILLÈME, A.; THALMANN, D.; KAMYAB, K.; MAMDANI, E. 2002. Avatar Markup Language. In *Proceedings of the workshop on Virtual environments (EGVE) 2002*, Barcelona, Spain, pp. 169-177.

MPEG-4: Binary Format for Scenes (BIFS) http://mpeg.telecomitalialab.com/standards/mpeg-4/mpeg-4.htm#10.6

ROEHL, B. 1995. Some Thoughts on Behavior in VR Systems. http://ece.uwaterloo.ca/~broehl/behav.html

SAI (Scene Authoring Interface / Scene Access Interface) http://www.web3d.org/TaskGroups/x3d/sai/SceneAccessInterface.html

SEIDMAN, G. 1998. Cooking With Hotpot: Making Events In VRML Work For You. http://www.cs.brown.edu/~gss/VRML98/paper.rev.html

SMIL 2.0 (Synchronized Multimedia Integration Language): W3C Recommendation 07 August 2001 http://www.w3.org/TR/smil20/

Viewpoint http://www.viewpoint.com

Virtools Dev http://www.virtools.com/solutions/products/virtools_dev.asp

VRML97. 1997. The VRML Consortium Inc.: “The Virtual Reality Modeling Language – International Standard ISO/IEC 14772-1:1997”, http://www.web3d.org/technicalinfo/specifications/vrml97/index.htm

VRML 2.0 PROTO Library http://www.accad.ohio-state.edu/~pgerstma/protolib/protolib/

VRML Object-Oriented Extensions Working Group http://rw4.cs.uni-sb.de/~diehl/ooevrml/

X3D-Schema, Version 0.8 (June 2002) http://www.web3d.org/TaskGroups/x3d/translation/X3dSchemaDraftSpy.xsd

X3D Specification: M4 - Final Working Draft http://www.web3d.org/TaskGroups/x3d/specification-milestone4/

Xj3D Open Source VRML/X3D Toolkit http://www.xj3d.org

XML Schema http://www.w3.org/XML/Schema

XSL (Extensible Stylesheet Language) http://www.w3.org/Style/XSL/