CONTIGRA: An XML-Based Architecture for Component-Oriented 3D Applications

Raimund Dachselt, Michael Hinz, Klaus Meißner
Dresden University of Technology, Department of Computer Science
Heinz-Nixdorf Endowed Chair for Multimedia Technology
D-01062 Dresden, Germany
{raimund.dachselt, michael.hinz, klaus.meissner}@inf.tu-dresden.de

ABSTRACT
Even though numerous Web3D technologies exist, most of them do not support a high-level, multi-disciplinary authoring process. Moreover, concepts of reuse are rarely provided. A component-based approach is introduced with the CONTIGRA architecture to construct interactive, three-dimensional applications, either stand-alone or web-based. The approach is entirely based on declarative XML documents describing the component implementation, its interface, as well as component configuration and composition of 3D user interfaces and virtual environments. Extensible 3D (X3D) is used as the scene graph basis. However, the resulting applications can be translated to other 3D technologies, too. Another advantage of the approach is reuse both at the implementation level and the higher abstract component level. This paper introduces the overall architecture and the XML schemas used for the component documents. It finally outlines the associated authoring process and tools involved.

Categories and Subject Descriptors
H.5.2 [**Information Interfaces**]: User Interfaces – *graphical user interfaces (GUI), prototyping, standardization.* I.3 [**Computer Graphics**]: I.3.6 Methodology and Techniques – *languages.* I.3.7 Three-Dimensional Graphics – *virtual reality.* D.2 [**Software Engineering**]: D.2.2 Design Tools and Techniques – *user interfaces.* D.2.11 Software Architectures – *domain-specific architectures, declarative languages.*

Keywords
Component-based development, 3D Components, 3D Widgets, 3D User Interfaces, Virtual Environments, Extensible 3D (X3D), XML Schema, CONTIGRA.

1. INTRODUCTION
The acronym CONTIGRA stands for *Component OrieNted Three-dimensional Interactive GRaphical Applications*. In this ongoing research project [2,5] metaphors and interaction principles for three-dimensional user interfaces are investigated. A declarative, component-based architecture on the basis of X3D (Extensible 3D) [18] and XML (Extensible Markup Language) [19] was designed for the construction of web-enabled, desktop Virtual Reality applications and 3D virtual environments (VE).

Due to considerable improvements in 3D graphics hardware and fast-evolving Internet technologies, the availability of 3D technologies for consumer platforms increases. As a result, the number of web-based three-dimensional applications in domains such as e-commerce, product presentations, entertainment, and virtual actors grows. By now a multitude of proprietary web 3D formats is available. This impedes standardization, particularly since most of the technologies are tailored to special application domains. VRML [16] as the established Web3D graphics standard is widely used. However, due to various problems it is rarely used for enabling applications such as e-commerce sites. X3D as the VRML successor promises to be more successful due to the flexible XML-encoding, modularization through profiles and smarter 3D browsers. However, no single 3D technology is expected to dominate the Web3D arena in the near future.

Despite the difficulties in selecting an appropriate technology, another major problem is the lack of guidelines, interaction paradigms, and design standards for three-dimensional user interfaces. Moreover, only few authoring tools exist, which are often tailored to a specific Web3D format and application domain. Concepts of reuse are mostly missing, which makes it extremely difficult and tedious to author non-trivial interactive 3D applications. Projects are often developed from scratch and involve a great deal of programming without reusing building blocks. This excludes non-programmers from designing 3D applications. Graphic designers, sound designers, and other experts should contribute to authoring them, since such systems inherently contain rich media elements. That is why rapid prototyping and high-level visual authoring tools need to be developed.

With regard to the current situation and the mentioned problems we conceive our vision of an available repertoire of 3D components, such as standardized 3D widgets. These interface controls and VE building blocks should be easy to construct, to configure, and to reuse in various projects in a component-oriented fashion. The underlying component architecture should not be code-centered, but instead employ a declarative, document-centered approach, which facilitates readability, rapid prototyping, and visual tool support. Authoring tools should provide high-level views as an abstraction from scene graph details, thus promoting multi-disciplinary development. Applications should not depend on a specific 3D format to be

***Web3D'02,* February 24-28, 2002, Tempe, Arizona, USA.**

portable and adaptable to various platforms. Web3D and other Internet media standards must be supported.

This paper is organized as follows: The next section relates our work to existing approaches, followed by an introduction to the overall architecture and CONTIGRA layers. The main part is devoted to the XML-based implementation and the presentation of the CONTIGRA markup languages. Thereafter the authoring process is described in section 5 along with tools associated with it. This is followed by a conclusion and outline of future work.

2. RELATED WORK

Until now there are only a few component approaches to constructing interactive 3D applications, even though the usage of component concepts for 3D graphics seems to be promising. An overview of this comparatively new field of research and a detailed classification of existing 3D component approaches is given in [3]. Current component technologies are inherently oriented towards code construction using programming languages. None of the competing technologies such as CORBA, DCOM or Enterprise Java Beans (EJB) is tailored to 3D applications on the web. Up to now component technologies are only rarely used in VE systems [1].

According to [3] 3D component approaches can be coarsely divided into code-centered and document-centered solutions. Among the first is the NPSNET-V system [1]. It primarily supports the development of scalable, distributed VE. Java components can be dynamically loaded across a network at runtime. In addition to that, the proprietary component system Bamboo [17] was designed, where code modules operate in a cross-platform and cross-language manner [1]. The i4D architecture [11, 12] is a dedicated 3D component solution. It introduces so called actors, high-level attributes, and actions to modify them, which all together create an abstraction to scene graphs. Components are realized as DLL/DSO, the layered architecture runs on various platforms and 3D APIs. Smart Virtual Prototypes [15] are based on VRML and Java classes to define simulation components consisting of user interface objects (realized as VRML prototypes), interactor components (on the client side) and virtual components (on the server side), both implemented as Java classes. Since coding in one or the other language is essential for all these approaches, they are not suitable for non-programmers. The Three-dimensional Beans approach [9] combining JavaBeans and Java3D takes that into account. It provides the 3D Beanbox editor, used for visual authoring and assembling of 3D Beans. Meta Beans [8] supplement this concept. They are associated with 3D Beans and encapsulate customizable interaction metaphors in components. Abstractions to specific 3D toolkits and component technologies are not provided with this approach. Recent developments within this project also include an XML description language for 3D Beans.

This shows a trend towards a declarative description of 3D components. The Jamal declarative component framework [13] is based on a flexible and expandable Component Interface Model. The Bean Markup Language (BML) is being used for declarative description of component connections. The isomorphisms between VRML-Prototypes, X3D-documents, Java Beans, and IDL described in [14] show the possibility to define component interfaces and connections in an abstract way, which can be translated to a specific implementation at a later point. This idea influenced the CONTIGRA architecture. With 3dml, a higher-level markup language for 3D interaction techniques and VE applications was recently introduced [10]. 3dml supports readability and rapid development. Since it is well suited for encapsulating interaction techniques, it could probably be used in addition to the CONTIGRA behavior graph (see section 4.1).

Among the declarative approaches one can also find early extensibility mechanisms such as Open Inventor Nodekits and VRML Prototypes, being limited to the respective format and restricted in terms of configuration and distribution. The recent X3D developments include advanced extensibility concepts. Beside VRML prototypes the concept of profiles allows scene graph extensibility and the component mechanism modularization within X3D players. X3D profiles are nevertheless only subsets of nodes, mainly for a certain application domain, but not components in the sense of reusable building blocks. X3D does not provide abstractions to scene graph level nor explicit definition of three-dimensional component interfaces nor high-level configuration and assembly. Since X3D is based on a well established scene graph concept and extensible set of nodes, its XML-encoding can be translated to other scene graph based formats. The other reason for choosing this format as the basis for the CONTIGRA scene graph encoding is the expected standardization of X3D.

3. THE OVERALL ARCHITECTURE

The CONTIGRA approach comprises a 3D component concept that is largely independent of implementation issues and allows easy, declarative, and multi-disciplinary authoring of 3D applications. It is based on structured documents describing the component implementation, their interfaces, as well as the assembly and configuration of 3D components to more complex interactive applications.

3.1 Example of a Web3D Application

A short introduction of an application example from the information visualization domain shall illustrate the different development stages within the CONTIGRA architecture. Fig. 1 depicts a navigation technique for medium-sized trees, called Collapsible Cylindrical Trees (CCT) [4]. Child nodes are mapped on rotating cylinders, which will be dynamically displayed or hidden according to the current path or sub-hierarchy. A fast interaction technique allows a single-click navigation to reach every node and open the associated web page or perform some other action on it.

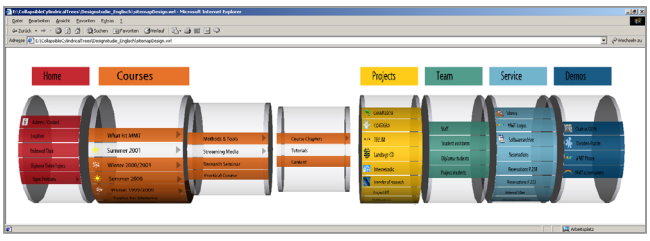

Figure 1. Web sitemap of the MMT research group using the CCT visualization & navigation technique

3.2 Component Development Stages

A CONTIGRA component is not a binary executable, but defined as an encapsulated component implementation based on declarative scene graph documents plus a separated component interface document. The development of a component-based three-dimensional application as sketched above will demand activities on different abstraction layers and development stages. Fig. 2 depicts the various stages and their associated tasks, resulting files, and tools.

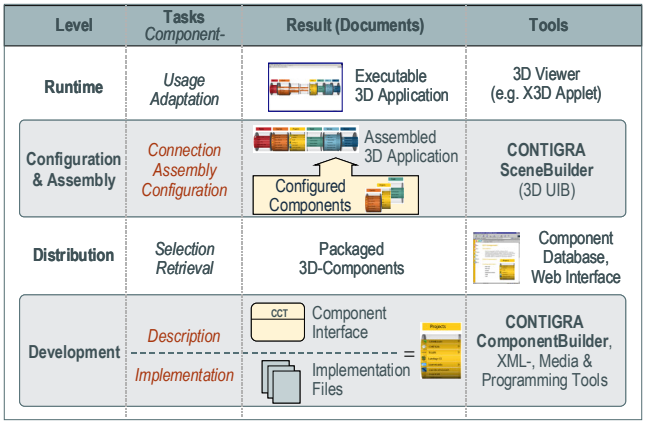

Figure 2. CONTIGRA levels with associated tasks and tools

At the *development level* simple or compound 3D components have to be implemented. Scene graph files, media assets, and script files belong to the implementation of a component. People involved at this stage are scene graph experts, programmers, and 3D designers. The same people create the component interface using a component description language. Component functionality, configurable parts, and parameters as well as rich meta information are coded in this document. Thus the results of this level are deliverable, functioning standard components in terms of non-executable document files, such as a CCT wheel component. Tools used are common XML editors, the Contigra ComponentBuilder (see section 5), as well as scripting and media authoring tools.

The next level, *component distribution*, assumes a set of packaged components, where only the component interface is visible and the implementation remains encapsulated. The components can be distributed over the World Wide Web and stored in databases. The associated task at this level is the retrieval of suitable components. Project managers or web engineers have to find the appropriate components for their projects. A web portal provides query functionality and helps to access desired components. Once the components are selected, they can be deployed at the *configuration and assembly level*. Typical tasks are the configuration, assembly, and connection of components to build more complex 3D applications. People involved at this stage can be non-3D-experts, for example project managers, web engineers, and designers. The CONTIGRA SceneBuilder tool (see section 5) is used to adapt the components to special needs and to connect them. The results of this stage are declarative documents describing the entire 3D application. Until this point a neutral, essentially format-independent representation should be used, which can now be exported to a specific 3D technology, such as Java3D or X3D. The resulting application can be viewed at *runtime level* within a 3D viewer. Adaptations to current system performance or user preferences can be done at runtime, too.

4. THE XML-BASED IMPLEMENTATION

After having introduced the basic architecture, this section illustrates the implementation of the CONTIGRA concept using markup languages coded with XML Schema [20]. Since a homogeneous encoding is provided on all levels, this is a consistent approach for all abstraction stages starting from scene graph level up to the description of complex component assemblies. In [3] advantages of using XML are described in detail. Among them are platform independence, standardization, interoperability with other Internet standards, availability of tools, and document processing using the Document Object Model [6] or XSL Transformations [21]. The hierarchical structure predestines XML for scene graph encodings, as one can see with X3D. XML Schema in particular makes it easy to integrate the various markup languages on all CONTIGRA levels through namespaces. In addition to that, data types, derived types, substitution groups, and abstract types make it easier to code high-level parameters and to employ object-oriented concepts, e.g. for component classes.

4.1 Level 1 – CONTIGRA SceneGraph

To illustrate the different levels, the focus of the following sections lies on the documents involved in the development process along with their underlying grammars.[1] Fig. 3 contains the identified tasks of the development stages and shows the associated XML documents with their schemas as well as other connected resources. The documents within the shaded part constitute a single 3D component.

The Contigra SceneGraph schema serves as a component implementation language and integrates various scene graph, media, and script files. The most important documents of the implementation level are the scene graph documents. The X3D XML encoding is used for describing the component geometry. Though X3D (respective VRML) contains some behavior and audio nodes, the scene graph is divided into different, separated parts. These are basically specialized collections of nodes. They are coded with separate grammars to solve problems currently not addressed in X3D. In [7] Döllner and Hinrichs developed the idea of separating geometry and interaction behavior in various graphs. This concept is expanded to a separated 3D audio graph, since spatial audio effects are not sufficiently provided by VRML/X3D. Take for example the description of church acoustics, including reverb, sound reflections etc. Using the newly developed *Audio3D* Schema, this can be described separately and could be reused for various similar church geometries. The *Behavior3D* schema currently being developed integrates existing X3D behavior nodes, adds new interaction nodes and provides uniform access to external methods and functions coded in script files, JARs etc. Both behavior and audio

[1] The CONTIGRA Schemas can be found online under http://www.contigra.com.

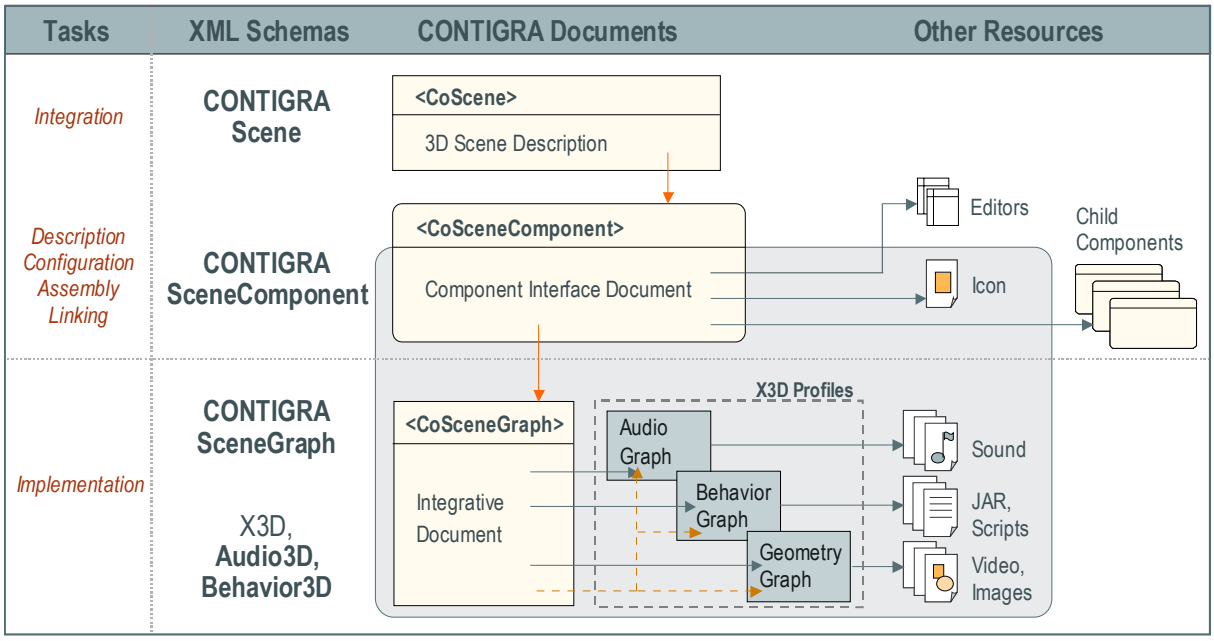

Figure 3. XML schemas and documents of the different CONTIGRA levels

graph are planned to become X3D profiles as soon as the X3D Schema has reached a mature status. References to external video, sound, and script files are embodied in the respective nodes.

The scene graph structures can either reside in a large document containing all parts of a component or in separate documents, which is preferable in terms of reuse on the implementation level. All nodes to be used later as configurable parameters of a component should be named using the DEF-mechanism provided by X3D. To integrate all implementation parts and to reference only the required sub scene graphs, an XML document coded with the CoSceneGraph schema is used. It serves as a flexible wrapper to the pieces of an implementation.

Fig. 4 shows the structure of this grammar[2]. A CoSceneGraph instance document contains a *Header* with meta information about the developer, company, version etc. The following optional parts *Geometry*, *Audio*, and *Behavior* contain references to different sub scene graphs of the respective type. The following extract from the CCT example application illustrates, how parts of the scene graph are referenced using XLink with XPointer syntax.

```
<Geometry>
    <CoSceneGraphFile xlink:href="dial.xml#xpointer
      (//Group[@DEF='DialGeometry'])" xlink:label="DialGeometry"/>
    <CoSceneGraphFile xlink:href="item.xml#xpointer
      (//Group[@DEF='ItemGeometry'])" xlink:label="ItemGeometry"/>
    ...
</Geometry>
```

[2] In the diagrams some parts are left out for space reasons. Dashed lines depict optional elements, attributes are not included. The symbol with squares in a row denotes a sequence of elements, squares in a column stand for the content model *All*, whereas the switch indicates a choice of elements.

Alternatively the appropriate group node could be included as an element. The following section *GraphLinks* establishes connections between the different scene graph parts. To stick to the church example, it is necessary to associate the simplified audio geometry of a church (e.g. some simple boxes) to the real detailed geometry. To avoid hard coding this association in either of the documents, links are stored under the *GraphLink* element. *AttributeLinks* are similar to VRML routes, since they connect fields. *DependenceLinks* establish links between nodes. If the source node is copied, deleted, or not rendered at all, the same actions are performed on the linked node. During the processing of a CoSceneGraph instance document all scene graph parts will be collected and internally stored as only one X3D scene graph. For this component implementation an interface document is written on the next level. Using enhanced X3D, the CONTIGRA SceneGraph level provides an abstraction to proprietary 3D formats, since documents can be translated to specific Web3D formats as needed.

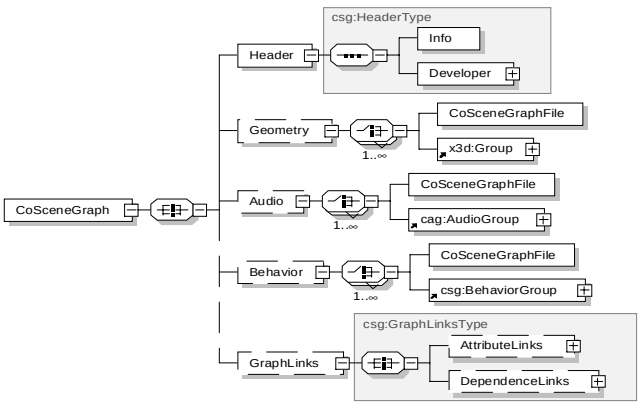

Figure 4. Structure of XML schema CoSceneGraph

4.2 Level 2 – CONTIGRA SceneComponent

Contigra SceneComponent is the core markup language of the proposed architecture. It is a component description language used to define interfaces of a component separated from its scene graph implementation. As such, it provides an abstraction level and is well suited for distribution, search and retrieval as well as component deployment. Each 3D component has one associated CoSceneComponent document. A number of tasks shown in Fig. 3 are performed in the various sections of the interface document. Thus a CoSceneComponent instance contains the component's documentation, its application purpose, license and deployment information, the description of configurable geometry and other parameters, offered methods of the component, and references to child components. Fig. 5 depicts the most important elements of the CoSceneComponent schema.

4.2.1 Component Classes and the Concept of Document-Based Inheritance

Before explaining the elements of the grammar, the idea of document-based inheritance is introduced. Whereas typical interface description languages only describe methods and parameters of a component, the CoSceneComponent grammar combines interface declarations with a specific component configuration. Accessible parts, parameters, and methods are not only listed, but values are already assigned to them. That means, one CoSceneComponent document describes one specific instance of a component as an instance of this class of components. Consequently, a document serves as a prototype, which can be copied and partly changed to produce another component of that class. Take for example the wheels of the CCT application, where each wheel is derived from another one, just

Figure 5. Structure of XML schema CoSceneComponent

with a changed material parameter and item hierarchy. The CoSceneComponent grammar provides a kind of generic super class, whereas instance documents implement specific sub classes. The following header extract from a blue CCT wheel shows, how the prototypic document serving as a basis is being referenced using the *referenceDocument* attribute.

```
<Info id="CCCT3" name="CCT-Wheel_blue" lastRevision=
 "2001-09-12" version="1.0">
    <ComponentClass xsi:type="CoCCTComponent"
     referenceDocument="CCT-WheelComponent_Proto"/>
</Info>
```

To avoid arbitrary component classes and an uncontrolled growth of documents, the element *ComponentClass* was introduced with a typed component hierarchy. This extensible hierarchy contains classes such as CoComponent, CoWidgetComponent, CoSlider-Component, CoAvatarComponent etc. It was developed to bring forward standardization efforts of three-dimensional user interfaces.

4.2.2 Component Description

The *Header* and *Documentation* elements describe a component textually. They are used for search and retrieval. Within the header the *Info* element contains basic information such as component *id*, *name*, *lastRevision*, *version*, *certificate*, and the component *class*. The class concept is explained later. The *Developer* section provides information about the *author*, *company* etc. Within the element *Deployment* information is given on how to use this component and how to author it. *AuthoringSupport* contains links to specialized editors for this component as well as an iconic representation for use in visual authoring tools. The *LicenseModel* element is used for business aspects and billing issues for commercial components. *Semantics* contains *Hints* with attributes such as *mayContain*, *suitedFor*, *combinedWith*, and *complements*. Though difficult to formalize, these hints serve as decision guidance, for example which kind of 3D menu to use in an application.

4.2.3 Component Interface and Configuration

The *Implementation* part establishes the connection to component implementations. It is basically a pointer to a *CoSceneGraph* element, which will usually reside in a separate file. References to named scene graph parts (e.g. exchangeable geometry) will always be resolved via this link to a CoSceneGraph document. The description of component functionality and configurable high-level parameters is included in the following three parts *GeometryList*, *ParameterList*, and *MethodList* of the interface document. All visible and exchangeable geometry parts of the component are listed as *Geometry* elements within the *GeometryList*.

```
<GeometryList>
    <Geometry name="CylinderGeometry" changeMode=
      "configurationTime" authorRole="design"
      description="Geometry of a rotating cylinder"
      nodeRef="DialGeometry"/>
    <Geometry name="ItemGeoemtry" changeMode=
      "runtime" authorRole="design" description="Geometry of
       a menu item" nodeRef="ItemGeometry"/>
    …
</GeometryList>
```

This excerpt from a CCT-Wheel instance references two named parts of the scene graph. Notice the abstraction from scene graph details at this point. The link to the real sub scene graphs representing this geometry is only established via the CoSceneGraph document. It contains scene graph references and consequently finds the matching node names, defined with the DEF attribute. Attributes *name* and *description* are used in authoring tools. The attribute *changeMode* may contain the values *never*, *configurationTime*, and *runtime*. They indicate, that a part cannot be changed at all, can only be modified at the configuration level, or can be changed even at runtime. A value set to *configurationTime* or *runtime* is comparable to an eventIn setting in VRML. With the help of the *authorRole* attribute it can be described, which person can modify which part or parameter of the component. This supports a multi-disciplinary authoring process. Possible values are *view*, *design*, *program*, and *doEverything*, which represent different access rights.

With the *ParameterList* containing an arbitrary number of *Parameter* elements, all other exposed high-level parameters are described, providing an abstraction to the scene graph level. The excerpt shows typical parameter definitions. Note, that the attributes of element *Parameter* are the same as of the element *Geometry* with the addition of the *bindable* and *fieldRef* attributes explained below.

```
<ParameterList>
    <Parameter name="WheelMaterial" changeMode=
      "configurationTime" authorRole="design" bindable="false"
      description="material of the cylinder" nodeRef="CylinderMat"/>
    …
    <Parameter name="ItemFont" changeMode="configurationTime"
      authorRole="design" bindable="false" description="text font of
      menu items" nodeRef="ItemTextFont" fieldRef="family">
        <cpt:CoString>Arial Narrow</cpt:CoString>
    </Parameter>
    <Parameter name="Width" changeMode="runtime"
      authorRole="program" bindable="true" description="calculated
      width of the cylinders">
        <cpt:CoFloat>12.0</cpt:CoFloat>
    </Parameter>
    …
</ParameterList>
```

The parameter “WheelMaterial” references a material node in the scene graph. The provided new material settings are left out for space reasons. Parameter “ItemFont” is an example for a parameter referencing a field within a scene graph, in this case the *family* field of a named *FontStyle* node, which will be substituted by the new given value. The attribute *fieldRef* was introduced for such references. Parameter “Width” is an example for an exposed parameter not associated with parts of a scene graph at all. The width of a CCT wheel is calculated automatically. Since the new attribute *bindable* is set to true, this parameter can be linked to others, which will be notified after a change has happened. It resembles a VRML eventOut. The content model for a *Parameter* is defined using the XML Schema *any* element, which allows the flexible insertion of various types. As one can see with the elements *CoString* and *CoFloat*, basic parameter types are already provided.

Finally the element *MethodList* contains all offered methods of a component. *Method* elements contain the attribute *name*, which

is matched with corresponding parts of the behavior graph, a list of typed *Parameters* for that method, and a possible return value *Result*. In the CCT example methods such as *CollapseWheel* and *ExpandWheel* are defined, which can be called from the parent component. It is to be mentioned, that for parameters with *changeMode* set to *configurationTime* or *runtime* get- and set-methods are automatically being generated.

4.2.4 Component Assembly and Linking

The last part of a CoSceneComponent document is used to describe *Subcomponents* of a compound component. Since this element is optional, it can be left out for simple components such as the CCT wheels. However, a compound component contains a transformation hierarchy of sub components. Every *Transform* element contains typical transformation attributes and a reference to a CoSceneComponent instance file, as seen in this example.

```
<Subcomponents>
    <Transform translation="1.0 0.0 0.0">
        <CoSceneComponentFile>
            CCT-WheelCoRed.xml
        </CoSceneComponentFile>
    </Transform>
    <Transform translation="11.0 0.0 0.0">
        <CoSceneComponentFile>
            CCT-WheelCoBlue.xml
        </CoSceneComponentFile>
    </Transform>
    …
    <ComponentLinkList>…</ComponentLinkList>
</Subcomponents>
```

While processing a compound CoSceneComponent instance, the parameter declarations of all sub components are collected and added to the parameter list of the container component. To avoid this behavior, parameters of sub component can be explicitly prevented from being accessible from the parent component interface using the *BlockedParameters* element. The *ComponentLinkList* establishes links between exposed parameters of different components, possibly involving a method call.

4.3 Level 3 – CONTIGRA Scene

CONTIGRA Scene serves as a high-level component integration language. An instance document of this grammar represents a declarative description of an interactive VE or 3D application, which is ready for translation into an executable 3D format. It contains a link to a compound component and general scene parameters. Fig. 6 shows a schema diagram of the CoScene grammar. A scene consists of a *Header* element with typical meta information. The *Documentation* part contains not only the description of a 3D scene, but also a *Help* element for application help information. Up to the SceneComponent level all coding was basically format independent. The *SystemRequirements* section establishes the connection of the compound scene component to a specific runtime environment. This includes hardware requirements such as *Processor*, *Memory*, and *InputDevices* / *OutputDevices*. Estimated *PerformanceCosts* and the required minimal *FrameRate* are also coded in this section. In addition to that, *WindowSize* demands and *Player* hints for the integration into web pages are described here. This section can be easily extended to include parameters such as required network connection etc.

The element *SceneParameters* describes typical scene characteristics, such as *Camera*, *ViewpointList*, and *LightList*. In addition to that, the element *AudioScene* defines global audio parameters for the *Environment* or the *Listener*'s initial position. The element *RootComponent* eventually contains a reference to a CoSceneComponent instance. Usually this instance will contain subcomponents. That is to say, the compound component represents the whole 3D application without the context of a specific runtime environment or 3D format.

5. AUTHORING PROCESS AND TOOLS TO PRODUCE CONTIGRA DOCUMENTS

In sections 3 and 4 the different levels of the overall architecture were explained along with the developed XML schemas and documents. This section illustrates, how the authoring process and the tools involved are conceived. The CONTIGRA Component-Builder and SceneBuilder tools are currently under development.

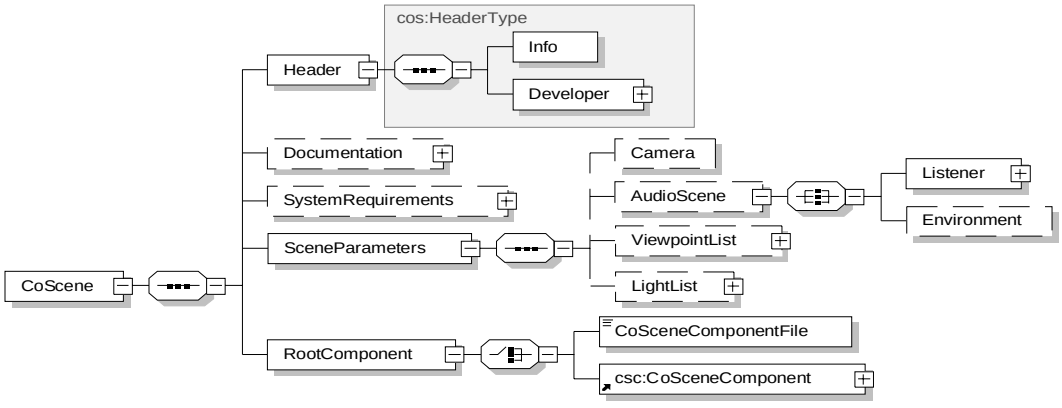

Figure 6. Structure of XML schema CoScene

Since detailed results are not yet available, this section only explains their conception. Both tools are part of a complex authoring environment, a 3D user interface builder. A deliverable component consists of XML scene graph documents, media and script files, the integrating CoSceneGraph instance document as well as the component interface declaration. To create it, component developers are using external editors for X3D and media authoring, as well as scripting tools.

In addition to that the CONTIGRA ComponentBuilder tool is used to produce the entire component. It is responsible for integrating all implementation parts. X3D documents can be loaded and displayed in a 3D window, parts can be selected and combined to be included in the component. Associations between nodes and fields of scene graph parts can be established, as described in the audio geometry example in section 4. The CoSceneGraph document will be automatically generated for the desired component implementation. Editors allow the component description according to the CoSceneComponent schema. The interface document will be generated automatically, too. For creating compound components CONTIGRA SceneBuilder functionality can be used to arrange and wire various components. Eventually, all component documents can be packaged for distribution.

The CONTIGRA SceneBuilder tool provides functionality for selecting 3D components and previewing them in a 3D window. They can be transformed and arranged to produce the desired VE or 3D application. Internally a compound CoSceneComponent instance is constructed, which stores the component transformation hierarchy. Subcomponents can be configured using parameter editors for standard parameter types. Additional specialized editors for a component are referenced within the *Deployment* part of the component interface and can be loaded as they are needed. Geometry parts can be visually selected and exchanged. Event wiring and component connections are established with a graphical link editor, which creates connections between parameters and methods. CoSceneComponent descriptions will be merged to one single CoSceneComponent document. The same applies to the implementation parts, which are assembled to form a complex X3D scene graph. For that purpose, references and conflicts are resolved and distributed parts integrated. The resulting scene can be either packaged into a new, compound component, using ComponentBuilder functionality or a 3D application can be produced. In this case, the user may adjust typical scene parameters and define lights, viewpoints etc. In addition to the compound CoSceneComponent document, the internally constructed CoScene instance document also serves as an exchange format for the CONTIGRA SceneBuilder.

Until this point there exists a declarative scene description including the compound component, one X3D scene graph, and external media and script files. Using transformation modules, these format independent descriptions can be translated to specific 3D formats such as Java3D or Shockwave3D. It is possible to produce different implementations from one high-level description of an application. For that purpose Extensible Stylesheet Language transformations and the Document Object Model (DOM) interface are used. The XML descriptions are parsed and transformed to either a declarative 3D format such as VRML or to program code such as Java3D. The resulting document or application can be viewed within the appropriate 3D viewer, plug-in or 3D browser. When truly componentized X3D browsers will be available, it should be possible to directly render a X3D file and to load code modules for unknown profiles or nodes (such as the audio3D extension). Developing translation modules of the CONTIGRA SceneBuilder will be difficult especially for proprietary formats. First experiences with 3D technologies such as Viewpoint and Shockwave3D have shown, that a functionally equivalent one-to-one translation will not be possible in every case.

6. CONCLUSION AND FUTURE WORK

In this paper an architecture for the component-based development of VE and 3D user interfaces was introduced. The proposed multi-layered architecture is entirely based on declarative documents coded with XML Schema. The document-based approach allows high-level descriptions, a visual, multi-disciplinary authoring process, and the translation of the resulting documents to various 3D technologies. The CONTIGRA XML descriptions at least work on the specification level, thus contributing towards the standardization of interface elements and 3D world components. The separated description of geometry, behavior, and audio at scene graph level facilitates reuse of a component's implementation. Through providing a level of abstraction at scene component level, high-level reuse is a major improvement in comparison with common scene graph based 3D technologies.

Future work includes refinement of the CONTIGRA XML schemas. Behavior and audio scene graph extensions have to be realized as X3D profiles. The main activity will be the further development of the user interface builder tools along with translation modules to various, also proprietary 3D technologies. Performance issues also need to be solved. More 3D components and applications have to be built in order to evaluate the efficiency of the authoring process. Experts from other fields such as graphic and audio design should be involved, too.

7. REFERENCES

[1] Capps, M.; McGregor, D.; Brutzman, D.; Zyda, M.: "NPSNET-V". *In IEEE Computer Graphics and Applications*, Vol. 20, No. 5, 2000, 12-15.

[2] CONTIGRA Project web pages: http://www.contigra.com

[3] Dachselt, R.: "Contigra - Towards a Document-based Approach to 3D Components", *Workshop proceedings "Structured Design of Virtual Environments and 3D-Components" of the ACM Web3D 2001 Symposium*, Paderborn, 2001.

[4] Dachselt, R.; Ebert, J.: "Collapsible Cylindrical Trees: A Fast Hierarchical Navigation Technique". To appear in: *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2001)*, San Diego, October 2001.

[5] Dachselt, R.: "CONTIGRA: A High-Level XML-Based Approach to Interactive 3D Components", *SIGGRAPH 2001 Conference Abstracts and Applications*, Los Angeles, August 2001, 163.

[6] Document Object Model (DOM): http://www.w3.org/DOM/

[7] Döllner, J.; Hinrichs, K.: “Interactive, Animated 3D Widgets”. *In IEEE Proceedings of CGI '98*, 1998, 278-286.

[8] Dörner, R.; Grimm, P.: “Customizable Interactions in 3D Web Applications with Meta Beans”. *In Proceedings of the Web3D 2001 Symposium*, Paderborn, Germany, 2001, 127-134.

[9] Dörner, R.; Grimm, P.: “Three-dimensional Beans – Creating Web Content Using 3D Components in a 3D Authoring Environment”. *In Proceedings of the Web3D-VRML 2000 Symposium*, Monterey, USA, 2000, 69-74.

[10] Figueroa, P.; Green, M.; Hoover, H. J.: “3dml: A Language for 3D Interaction Techniques Specification.” *Short presentation at Eurographics 2001*, Manchester, United Kingdom, September 2001.

[11] Geiger, C.; Paelke, V.; Reimann, C; Rosenbach, W.: “A Framework for the Structured Design of VR/AR Content”, *In Proceedings of VRST 2000*, October 2000.

[12] Geiger, C.; Reiman, C.; Rosenbach, W.: “Design of Reusable Components for Interactive 3D Environments“. *In Proceedings of the Workshop on Guiding Users through Interactive Experiences*, Paderborn, Germany, April 2000.

[13] Rudolph, M.: “Jamal: Components Frameworks and Extensibility”. URL: http://www.web3d.org/TaskGroups/x3d/lucidActual/jamal/Jamal.html, 1999.

[14] Rudolph, M.: “X3D Components”. URL: http://www.web3d.org/TaskGroups/x3d/lucidActual/X3DComponents/X3DComponents.html, 1999.

[15] Salmela, M.; Kyllönen, H.: “Smart Virtual Prototypes: Distributed 3D Product Simulations for Web based Environments”. *In Proceedings of the Web3D-VRML 2000 Symposium*, Monterey, USA, 2000, 87-93.

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

[17] Watsen, K.; Zyda, M.: “Bamboo - A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments”. *In Proceedings of the IEEE VRAIS’98*, Atlanta, Georgia, 1998, 252-259.

[18] Web3D Consortium: “X3D: The Virtual Reality Modeling Language - International Standard ISO/IEC 14772:200x”, URL: http://www.web3D.org/TaskGroups/x3d/specification/

[19] Extensible Markup Language (XML): http://www.w3.org/XML/

[20] XML-Schema: http://www.w3.org/XML/Schema

[21] XSL Transformations (XSLT): http://www.w3.org/TR/xslt11