

SSIML/Components: A Visual Language for the Abstract Specification of 3D Components

Arnd Vitzthum*
University of Munich

Abstract

3D components can facilitate the development of interactive 3D content. However, the specification of 3D components with sophisticated behaviours and complex inner structures can become a tedious and error-prone work, since support for this task primarily exists on code level. Thus, we present SSIML/Components, a visual language for the abstract pre-implementation specification of 3D components. Our language helps to decompose complicated 3D scene structures into easily manageable, maintainable and reusable parts. By automatically generating code from the visual models, a seamless transition from the design level to the implementation level is possible.

CR Categories: D.2.2 [Software Engineering]: Design Tools and Techniques—Computer-aided software engineering (CASE), Object-oriented design methods; I.3.6 [Computer Graphics]: Methodology and Techniques—Languages

Keywords: 3D components, SSIML, visual design, abstract design, visual modeling language

1 Introduction

An approach to ease the development of interactive 3D scenes investigated in several existing research works (see section 2) is to apply the concept of components to the field of 3D graphics. 3D components can enable a more rapid scene development by using predefined reusable and reliable software building blocks. They allow the decomposition of complex scene structures and the encapsulation of scene subgraphs. This permits to exchange parts of a scene easily and improves the maintainability of 3D content. In addition, 3D components offer well defined interfaces which hide the complexity of the component implementation and provide only access to selected attributes of scene elements. Three-dimensional components can be composed to create new and more complex components. Behaviours can also be encapsulated in components. Connections between geometries and behaviours can be defined in a more structured manner with the use of a component concept.

Furthermore, 3D component architectures can help to facilitate the integration of interactive 3D content into applications. The integration of interactive 3D content can increase the attractiveness and enhance the usability of stand-alone and web applications, e.g. in the fields of product visualization and configuration (refer to [Segura et al. 2005] for an example), e-commerce and virtual exhibitions.

*e-mail: arnd.vitzthum@ifi.lmu.de

Copyright © 2006 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481 or e-mail permissions@acm.org.

Web3D 2006, Columbia, Maryland, 18–21 April 2006.
© 2006 ACM 1-59593-336-0/06/0004 \$5.00

Well defined 3D component interfaces can improve the clarity and manageability of the application code which is required to modify a scene. In addition, the possibility to parameterise 3D geometries enables an easy adaptation of the visual appearance of a component.

Nonetheless, apart from the advantages of 3D components some problems remain. The first problem is that the 3D components themselves have to be most often specified on the implementation level using a textual description. This complicates the specification in particular of components with a large number of properties, complex inner structures and sophisticated behaviours. The second problem refers to the integration of 3D content into applications as well as behaviour inclusion into 3D components. While application logic and non-trivial behaviours are commonly created by programming experts, the 3D geometry is created by 3D content developers such as designers. Moreover, programmers and designers utilize often different kinds of tools to solve their tasks. This increases the danger that 3D content and program code become inconsistent. Especially for the task of integrating 3D components seamlessly into applications a lack of advanced concepts and tools can be noticed.

In order to address the problems mentioned above, we combine the concept of 3D components with an abstract pre-implementation design using a visual language. In our opinion, an abstract design is crucial for reducing implementation errors and costs. At the same time the component concept helps to manage complex 3D scenes and to define concrete integration relations between an application and the 3D world.

This paper builds up on two previous works. In [Vitzthum and Pleuss 2005] the *Scene Structure and Integration Modelling Language* (SSIML) is presented. SSIML is a visual design language and focuses on the (semi-)formal pre-implementation specification of 3D scenes and their integrations into applications. SSIML is an extension of the Unified Modeling Language (UML). This allows specifying integration relations between UML classes and 3D scene elements. A second work [Vitzthum 2005] describes the integration of behaviours into SSIML. In this paper we extend SSIML with the concept of 3D components. By generating code from SSIML models, a seamless transition from the design level to the implementation level is possible.

In section 2 we present an overview of related work. Section 3 contains a brief description of the visual language SSIML. Section 4 introduces our new approach for 3D component specification, SSIML/Components. Section 5 describes the mapping from SSIML/Components models to code. Section 6 contains conclusion and outlook.

2 Related Work

The use of components in the field of interactive 3D applications was already extensively investigated in some previous research projects. Some works such as NPSNET-V [Capps et al. 2000] and MOVE [García et al. 2002] focus on component-based architectures for building extensible collaborative and distributed virtual environments. Such environments typically contain components

which support network communication between different users inside a virtual environment.

However, we concentrate on 3D components, i.e. components which encapsulate 3D geometry and behaviour. We consider 3D components analogous to two-dimensional visual components such as Java Swing-components or visual Delphi components.

Some classical work addresses the modularization of behaviour and 3D code. For instance, in the Actor/Scriptor Animation System (ASAS) [Reynolds 1982] there exist so called actors which encapsulate animation and behaviour code. In a paper of Conner et al. [Conner et al. 1992] 3D components were seen as analogue to two-dimensional widgets (2D user interface components). Approaches such as CONTIGRA [Dachselt et al. 2002] and 3D Beans [Dörner and Grimm 2000] allow defining such 3D components on the implementation level. In detail, CONTIGRA is an XML-dialect for specifying three dimensional components and applications. In contrast to CONTIGRA the 3D Beans approach is based on the Java Beans architecture [Sun]. Also the prototype mechanism of VRML [Web3D-Consortium 2003] and X3D [Web3D-Consortium 2005] can be considered as a possibility to define component-like structures on a textual level or by using graphical authoring tools. For example, the Internet Scene Assembler [Parallelgraphics] is a 3D authoring tool which allows assembling 3D scenes using so called *objects* which are encoded as VRML prototypes. Created scenes can be saved as new objects and reused later.

Nonetheless, component specification on the implementation level can become a tedious task, in particular if the components are complex, contain sophisticated behaviours and are composed of many subcomponents. Thus, in contrast to the works mentioned above our approach supports an abstract pre-implementation visual component specification. Visual software design has a great success in the field of traditional software development as the Unified Modeling Language (UML) [OMG 2005] and several visual domain-specific languages show. In our approach the component specification is based on a MOF [OMG 2002] - compliant meta-model.

3 SSIML

In this section we briefly introduce SSIML, since our modelling approach for 3D components is based on this language. As mentioned in section 1, SSIML allows specifying the integration of 3D content into applications on a pre-implementation design level. SSIML is an extension of the Unified Modeling Language (UML). Most parts of SSIML are realized as elements of a meta-model as well as UML stereotypes to enable an easier integration of SSIML into UML case tools. Indeed, SSIML was integrated into NoMagic MagicDraw [NoMagic] and Rational Rose [Rational].

The structure of the 3D content is described in SSIML by a scene graph-oriented notation (i.e. by an acyclic directed graph). This specification is called *SSIML scene model*. SSIML permits to describe objects and scene in a simple and compact manner. SSIML models are independent of a specific platform and therewith enable a description on a very generic level. Implementation details are not contained in the models.

The scene model contains nodes and directed edges between the nodes representing parent-child relations. Nodes have different types. In detail, there are *scene* nodes, *group* nodes, *object* nodes, *light* nodes and *composed nodes* (figure 1). The scene node is the root of the SSIML scene graph and represents the origin of the 3D world. Group, light and object nodes are so called *located nodes*. This means that these nodes have a defined location in the 3D world.

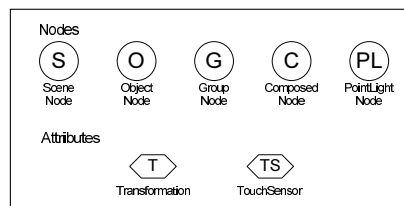


Figure 1: Basic SSIML elements.

Thus, every located node has a *transformation* attribute (figure 1). Beside transformation attributes, there exist a number of other attributes such as different types of sensors. However, these attributes will not be considered further here. Attributes are only visible in a SSIML scene model if they are explicitly modelled. Attributes are connected with their corresponding nodes via directed edges starting from the node.

Located nodes also have a *content*. The content of a light node is the definition of the light. The content of the object node is the object's appearance and geometry but also audio content. The content of a group node are simply the children of the node.

A special type of a node is the composed node. A composed node encapsulates a reusable subgraph.

SSIML elements such as nodes and attributes also have names. Names are placed near their corresponding elements. Some node types such as objects and composed nodes also have *content types*. The content type of an object defines which 3D model the node contains (i.e. the 3D model which describes the object appearance and geometry) while the content type of a composed node describes the subgraph structure inside the node. The content type name of a node follows the node name (syntax: `<node name>:<content type>`).

The integration of the scene into the application is specified in the *SSIML interrelation model*. This model contains the SSIML scene model as well as relevant classes of the application represented by UML classes. Different types of relations between the classes and scene elements can be specified, e.g. relations describing how a class can change the scene structure or modify scene attribute values. For a more detailed description of the SSIML models, please refer to [Vitzthum and Pleuss 2005].

In [Vitzthum 2005] an extension of SSIML was presented which extends SSIML with behaviour specification capabilities. This extension is called SSIML/Behaviour. Special classes were introduced (*behaviours*) which contain behaviour descriptions based on UML2 statecharts. Behaviours are connected with the attributes of scene nodes which shall be modified by the behaviour. Behaviours also interact with application classes. Therefore relations between application classes and behaviours can be specified. A more profound discussion of SSIML/Behaviour can be found in [Vitzthum 2005].

4 SSIML/Components

In this section our approach for the abstract specification of 3D components is introduced by an example. The presented concepts are based on the visual modelling language SSIML and its extension SSIML/Behaviour which were described in the previous section and lead to a new language called SSIML/Components.

In order to enable the integration of SSIML/Components models

into UML tools, the question was investigated if the component notation provided by the UML can be adapted for 3D components. Unfortunately, the UML component notation is suboptimal for 3D components. For example, a UML component can expose a set of interfaces which are required by other components. For a 3D component it is often more useful to provide a set of properties (e.g. colour properties) which can be written to influence the visual 3D representation of the component. Interfaces may contain a large number of methods while properties usually only offer mechanisms for reading and/or writing the property values.

In addition, since the extension mechanism built into UML primarily provides the graphical adjustment of classes, it is relatively difficult to adapt the presentation of UML components to the requirements of a visual design notation for 3D components inside UML case tools. As a result, we have defined a customized notation for 3D components on the basis of a MOF [OMG 2002] - compliant meta-model which extends the meta-model of the UML. This enables interoperability between the newly defined component model elements and required UML model elements such as classes. 3D components can be specified using a separate tool. This tool contains a graphical editor which supports the SSIML component notation. Classes from models created with UML case tools can be loaded into the component tool. Relations between classes and 3D components can also be specified using the tool.

Some concepts in SSIML/Components are comparable to the concepts of VRML/X3D prototypes. However, the component concepts introduced in this paper go a step further, for example by allowing a component to inherit properties from several different supercomponents.

Note that SSIML components are not comparable to X3D components, since X3D components are collections of functionally related X3D objects and services (usually collections of X3D nodes).

4.1 Running Example

In order to demonstrate our approach for modelling 3D components, we use a simple example from the domain of product visualization. We have chosen a web application which visualizes a motor scooter model for the customers. In the figures 2 and 3 the user interface of the application is shown. Besides rotating the whole scooter, the customer has different possibilities to interact with the model. First, he or she can rotate the handlebar unit within a given angle. Second, he or she can switch *on* and *off* the front and rear lamps and the left or right winker lamps. Third, it is possible to change the colour of the scooter model. The model of the motor scooter was divided into several components to realize the described interactions and to make the implementation of interaction code manageable. In detail, these components are presented in the following sections.

4.2 Simple and Complex Data Types

In SSIML/Components there exist several data types. Data types can be a *simple types* or *complex types* or arrays which store simple or complex type values. Simple data types in SSIML are *String*, *Int*, *Float*, *Boolean* and *Node* (for SSIML scene nodes). The complex type concept allows software designers to create their own data types. A complex type consists of a set of complex type *elements*. An element of a complex type has a *name*, a *simple type* (or an array of a simple type), an optional *values range* (enclosed in angle brackets) and an optional *default value* (enclosed in curly brackets). If no default value is given, a base value will be assigned

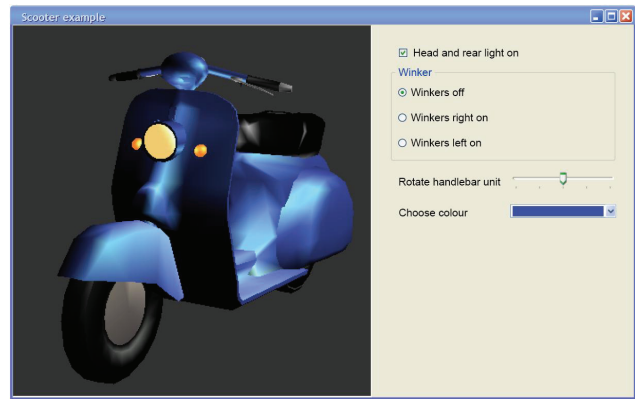


Figure 2: Application with front view of the motor scooter.

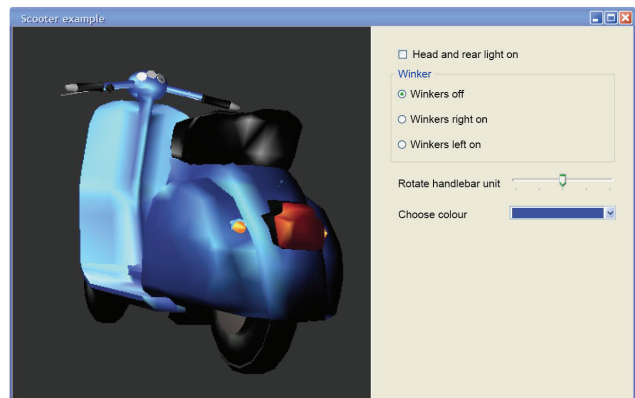


Figure 3: Application with back view of the motor scooter.

to the element. The base value is zero for numeric types, the empty string for the *String* type and *NULL* for nodes. The values range can be defined by a list of *allowed values* or a *minimal and a maximal value*.

A complex type also enables the assignment of semantics to data. This is important when the model is mapped to platform specific code. For example, when mapping a model to X3D code, a simple *Float* value in the model may be assigned to different node fields, such as the radius of a sphere or the transparency of a material. But if a complex type *SphereRadius* is defined which contains the float value, it is clear that the float value has to be assigned to the radius field of an X3D sphere node.

In figure 4 the complex type *Hue* is shown. Data of the type *Hue* describes a hue. *Hue* has two elements: *interval* and *fine_tuning*. *interval* is a *String* element which enables the definition of a hue interval. Allowed *String* values are *red-orange*, *orange-yellow*, *yellow-green*, *green-blue*, *blue-violet* and *violet-red*. The default value of *interval* is *red-orange*. The float element *fine_tuning* (values range is zero to one) expresses where exactly in the interval defined by *interval* the hue is located. For instance, if the interval is *red-orange* and *fine_tuning* is 0 (default), then the hue described is *red*. If the interval is *red-orange* and *fine_tuning* is 1 then the hue is orange. And if the interval is *red-orange* and *fine_tuning* is 0.5 then the hue is located exactly in the middle between red and orange. This allows describing a hue in an intuitive manner compared to a specification of a hue by a number. The specification of the hue is also more exactly than by defining it only by a name such as *orange*.

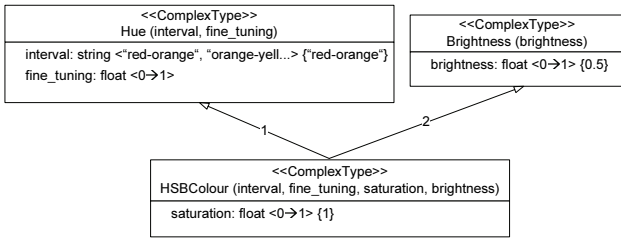


Figure 4: The complex types Hue, Brightness and HSBColour.

The *initialization pattern* (noted in parentheses below the complex type name) describes the sequence in which the elements of Hue have to be initialized. It is explained later in section 4.3 how the initialization pattern can be used to assign an initial value to a component property of a complex type.

In figure 4 a second complex type Brightness is defined. This type allows describing brightness values. Brightness has an element brightness. brightness is a float value between zero and one which describes the brightness of a material or colour (zero means no brightness and one means maximum brightness). Since no default value is given, brightness is initialized with the base value zero.

Complex types can be extended. In figure 4 HSBColour extends Hue and Brightness. HSBColour is also called a *subtype* of Hue and Brightness while Hue and Brightness are *supertypes* of HSBColour. HSBColour inherits all elements of its supertypes. As the example shows, multiple inheritance is allowed when extending complex types. If two or more supertypes have elements with equal names and a subtype inherits from all, the subtype inherits the element of the supertype with the highest inheritance priority. The inheritance priority is represented by a number which is noted on the inheritance arrow (lower numbers mean higher priority). The initialization pattern of Hue is extended in HSBColour with the brightness element from Brightness. If no initialization pattern is given in a subtype, the pattern of the supertype with the highest inheritance priority will be adopted. It is possible that not all elements occur in the initialization pattern. Values of elements which are not part of the initialization pattern can not be changed later.

It is also possible to extend the set of inherited elements by defining new elements or to overwrite the values range or the default values of supertype elements. For example, in figure 4 HSBColour is extended with the element saturation. saturation describes the saturation of a colour. A saturation value of zero means no saturation while a saturation value of one means full saturation. The default value is one.

With HSBColour the developer is able to specify a colour in the HSB colour space. The HSB components correspond to the way humans perceive colour. Thus, defining a colour in the HSB space is often considered to be more intuitive than defining it in the RGB space. There are also some 3D APIs such as *OpenSG* [OpenSG-Forum] and 3D animation scripting systems such as *ASAS* [Reynolds 1982] and *Fran* [Elliott 1999] which support HSB or related colour models.

4.3 Components

In contrast to complex types, components in SSIML/Components (in the following called *SSIML components* or only *components*) encapsulate a subgraph of the scenegraph and can be therewith in-

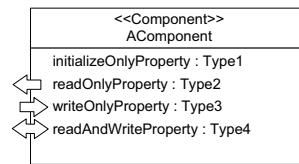


Figure 5: Access methods of component properties.

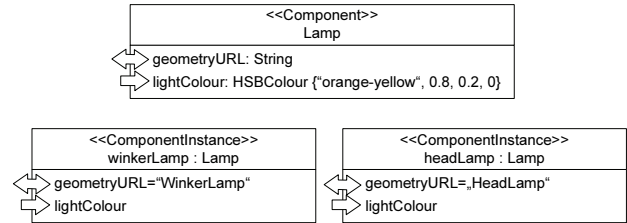


Figure 6: The component Lamp and the two instances winkerLamp a headLamp

tegrated in the scene hierarchy. A component has a *name* and a *set of properties*. A property has a *name*, a specific *data type* and a *initial value*. In addition, an *access method* can be defined for every property. Access methods are *Read*, *Write*, *ReadWrite* and *Init* (figure 5). *Read* means that a property value can only be read, *Write* means that the value can only be written and *ReadWrite* means that the value can be read and written. *Init* allows initialising the value but does not allow subsequent read or write operations on the value.

Like complex types components can be extended using inheritance mechanisms. This concept is analogue to the extension concept of complex types. The component extension mechanism additionally enables the extension of access methods. Table 1 shows how access methods can be extended.

4.3.1 A Simple Component

In figure 6 the component Lamp is defined. Lamp has two properties: geometryURL and lightColour. geometryURL is a string which should contain the URL to the resource file which describes the lamp geometry. For example, a scooter's rear light would have another geometry than its front light. The property lightColor is required to simulate switching the light *on* and *off*. In the example, no initial value is given for geometryURL. This means that geometryURL is automatically initialized with the base value of the String type as long as no other value is defined for it, i.e. the empty string. For lightColour the initial value *orange-yellow, 0.8, 0.2, 0* (a low saturated yellow with zero brightness - i.e. black) is defined. By manipulating the brightness value the light can be switched from *off* (zero brightness) to *on* (full brightness).

Table 1: Allowed extensions of property access methods.

Access method →	Init	Read	Write	ReadWrite
Extendable by ↓				
Init	yes	no	no	no
Read	yes	yes	no	no
Write	yes	no	yes	no
ReadWrite	yes	yes	yes	yes

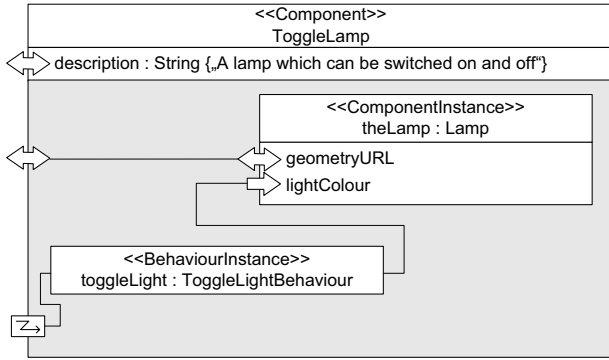


Figure 7: The component ToggleLamp

4.3.2 Component Instances

A component can be instantiated. In figure 6 two instances of Lamp are shown. One instance defines a winker lamp with and the other defines a headlamp. `winkerLamp` and `headLamp` have different geometries. For example, the geometry of `winkerLamp` is defined in a resource `WinkerLamp`.

4.3.3 Composed Components and Behaviour Integration

It would be desirable to toggle the light state via sending *switch on* and *switch off* signals to a lamp object instead of changing the brightness information of the light via program instructions. This means that the description of the behaviour which switches the lamp on and off has to be encapsulated in the lamp component.

In figure 7 the definition of the `ToggleLamp` component is shown. Beside the property `description` which allows describing the component using a `String` the inner structure of the component is modelled. The `ToggleLamp` component contains an instance of the `Lamp` component and an instance of a *behaviour* object, i.e. an instance of the behaviour class `ToggleLightBehaviour`. `toggleLight` is associated with the property `lightColour` of the `Lamp` component instance. In addition, `toggleLight` is connected with a so called *behaviour port* which allows exposing the interface of the behaviour object for external access. This permits `toggleLight` to manipulate the `lightColour` value of the `Lamp` depending on *message events* received from external objects and according to the behaviour description defined in the class `ToggleLightBehaviour`.

Properties of an component which are mapped to a component instance property or a node inside a component are denoted on the component border in the visual component model. If the outer property has no name then the name of the inner property is adopted. In figure 7 the property `geometryURL` of `ToggleLamp` is mapped to `geometryURL` of `theLamp`. It is necessary that the access method of the outer property (i. e. the component property) can be mapped to the access method of the inner property (i.e. the property of the inner component instance). Table 2 shows which properties can be connected depending on their access methods.

4.3.4 Arranging Component Instances in the Scene Graph Structure

As already mentioned, components encapsulate subgraphs of the scene. Furthermore, component instances are leafs of

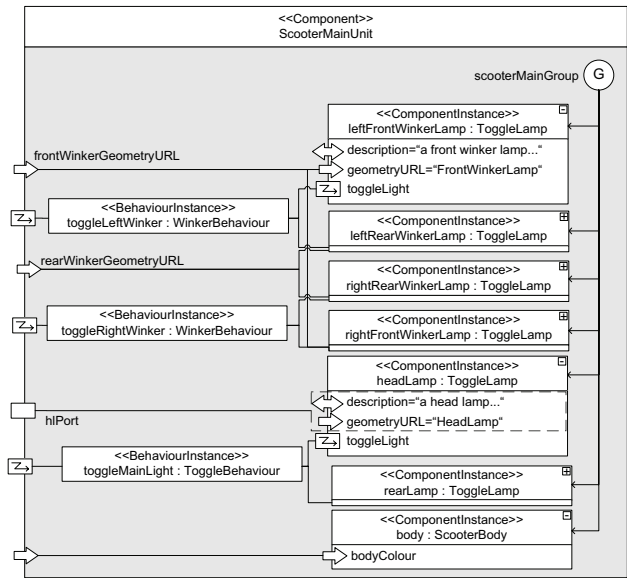


Figure 8: The component ScooterMainUnit

(sub)graphs in the SSIML scene model. In figure 8, the composition structure of the component `ScooterMainUnit` is presented. `ScooterMainUnit` contains a group `scooterMainGroup` which comprises the required component instances. These are two front winker lamps, two rear winker lamps, one rear lamp, one head lamp and an instance of the `ScooterBody` component with a `bodyColour` property for changing the colour of the scooter's body. In our example, `body` also contains the rear wheel of the scooter. Moreover, `ScooterMainUnit` also contains one behaviour object which controls the simultaneous switching of the head and rear lamp and two other behaviour objects which control the simultaneous blinking of the left rear and front winker lamps and the right rear and front winker lamps.

In figure 8, both properties of `headLamp` form a group (dashed rectangle). The whole group is connected with the port `hiPort` (drawn as a square on the border of `ScooterMainUnit`). A port enables the exposition of a whole group of properties or even all properties of an inner component instance under a new name. The name of an outer property is then assembled from the port's name concatenated with an underscore and the name of the inner property (e.g. `hiPort.description`). The properties of a component instance may also be hidden in the visual model. For example, in figure 8 the properties of `leftRearWinkerLamp` are not visible.

As also shown in figure 8, it is possible that properties are blocked by the outer component. For instance, the property `description` of `leftFrontWinkerLamp` is blocked.

A property may be initialized more than one time in the model. The rule which determines finally the initial value of a property is as follows: the base values are replaced by the default values of complex types, these default values are replaced by the initial values of

Table 2: Allowed mappings between access methods.

External property	Internal property
Write	Write, ReadWrite
Read	Read, ReadWrite
ReadWrite	ReadWrite

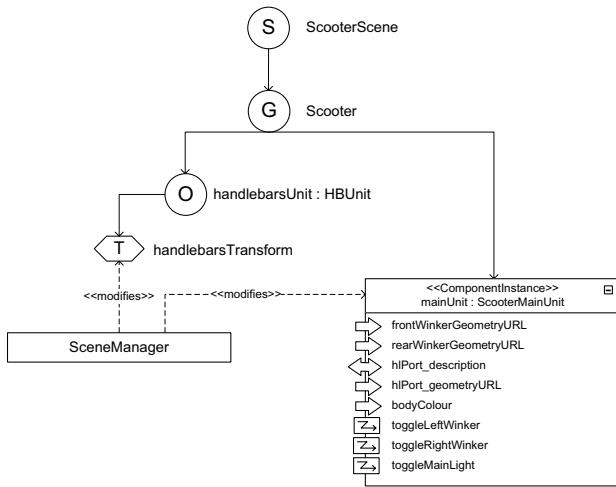


Figure 9: SSIML interrelation model of the scooter scene and application-scene interrelations

properties in a component definition, the initial property values of the component are replaced by the initial values of the component instance and the initial property values of the component instance are replaced by initial property values of an outer component instance which are mapped to property values of the inner component instance. For example, the initial value of the property description of headLamp in figure 8 replaces the initial property value of description in Lamp (the empty string).

4.3.5 The Overall Scene

In figure 9, the whole scene representing the motor scooter and the relations between application and scene are illustrated. The scooter is divided in the handlebars unit and the main unit. The handlebars unit comprises the handlebars, the front fork and the front wheel which are encapsulated in the object node `handlebarsUnit`. The component instance `mainUnit` is defined by the component `ScooterMainUnit` which was presented before (section 4.3.4). The `SceneManager` class has access to the transformation attribute of the handlebars unit to simulate steering motions. In addition, `SceneManager` is associated directly with the `mainUnit` component instance (arrow to the border of `mainUnit`). Thus, it has also access to all properties and behaviour ports of `mainUnit`.

5 Model-Code Mapping

In order to evaluate the feasibility of our approach we map the SSIML/Components models to code using tools for automatic code generation. As target language for the generated code we have chosen X3D (classic VRML encoding). However, because of the level of abstraction of the SSIML models it would be also possible to generate code for other platforms and formats. In this paper we focus on code generation from SSIML components. A description of code generation from class-node interrelations, basic SSIML scene elements and SSIML behaviours can be found in [Vitzthum and Pleuss 2005] and [Vitzthum 2005].

Since our models are based on a MOF-conform meta-model, they can be encoded in the XML Metadata Interchange Format (XMI)

[OMG 2003]. This allows applying XSLT [W3C 1999] stylesheets to generate X3D code from the XMI-encoded model descriptions.

5.1 Translation of SSIML data types

SSIML simple types can be mapped easily to the corresponding X3D data types. For example, the SSIML `String` type can be translated into an X3D `SFString`. Analogous, a string array can be transformed into an X3D `MFString` type. SSIML complex types can be mapped to X3D prototypes.

However, especially for complex types the mapping is not always straightforward. Some complex types can be mapped directly to their counterparts in the target language while others can not. For example, a complex type `RGBColour` could be mapped directly to the `SFColor` type in X3D. The `HSBColour` type can not be translated directly into an X3D type.

Thus, the developer has the possibility to adapt the code generation mechanism for every complex type individually. If the code generation routines are not specifically adapted for a certain complex type, then a default code is created for this type automatically. In the following the default code generated for the complex type `HSBColour` (see figure 4) is presented.

Listing 1 shows the mapping of the complex type `HSBColour` to X3D code. In X3D, `HSBColour` is represented by an X3D prototype. Since prototypes can not extend other prototypes also the elements which `HSBColour` inherits from other complex types such as `Brightness` (figure 4) are integrated into the X3D prototype interface. Elements of complex types are translated to appropriate fields of X3D prototypes. For instance, SSIML `Float` elements are translated to `SFFloat` fields in the X3D prototype interface declaration. These fields are initialized with the values defined in the SSIML model. Since values ranges are defined for the elements of `HSBColour`, it has to be checked if values assigned to element related prototype fields are in the given range. Therefore scripts are defined inside the prototype's body which realize the required validation. As an example, in listing 1 the validation of the `brightness` value is illustrated. If the `brightness` field is initialized incorrect, its value will be set back to the base value (function `initialize`). If the field shall be assigned with an invalid value, this value will not be set. Instead, the last valid value is kept (function `set_brightness`). If `brightness` was modified, an appropriate event is generated by the `brightness_changed` output field. Since a complex type contains no access restrictions, beside an X3D `initializeOnly` field for every element an `inputOnly` and `outputOnly` field is generated. For `brightness` these are the fields `set_brightness` and `brightness_changed`. The generation of three fields instead of generating one X3D `inputOutput` field is necessary, because the fields have to be connected with the corresponding fields of the validation script via IS statements (listing 1).

5.2 Translation of SSIML components

Like complex types SSIML components are also mapped to X3D prototypes. Listing 2 shows the interface of the X3D prototype generated from the component `Lamp`. `Lamp` has a simple type property (`geometryURL`). Since this property has no access restriction it can be mapped to an `inputOutput` field in the X3D prototype interface declaration. The SSIML property `lightColour` has the complex type `HSBColour`. For the property `lightColour` an X3D field `lightColour` is created. This field is initialized with the initial value of `lightColour` defined in the SSIML

Listing 1: Mapping of the SSIML complex type HSBColour to X3D code

```

PROTO HSBColour [
  #Fields for hue and saturation
  ...

  #Fields for brightness
  initializeOnly SFFloat brightness 0.2
  inputOnly SFFloat set_brightness
  outputOnly SFFloat brightness_changed
] {
  #Scripts for the validation of hue and
  #saturation values
  ...

  #Script for the validation of the brightness value
  Script
  {
    initializeOnly SFFloat _brightness
    IS brightness

    inputOnly SFFloat set__brightness
    IS set_brightness

    outputOnly SFFloat _brightness_changed
    IS brightness_changed

    url ["ecmascript:

      function initialize () {
        if (!(0<=_brightness<=1)) _brightness = 0;
        _brightness_changed = _brightness;
      }

      function set__brightness (val) {
        if (0<=val<=1) {
          _brightness = val;
          _brightness_changed = _brightness;
        }
      }

    "]
  }
  ...

```

model. Since `lightColour` can be written, an additional input field `set_lightColour` is created.

The body of the X3D prototype contains a group node which again contains further nodes. The group node is required as root node of the prototype. For every element of the prototype interface (except simple type properties with no access restriction such as `geomtetryURL`) an additional `Script` node is generated into the X3D prototype body. Depending on the access methods of a SSIML property this script realizes the code for read and write operations to the corresponding X3D fields. In listing 2 the script for the SSIML property `lightColour` is presented. Since `lightColour` is a write-only property, the script contains an X3D `inputOnly` field (`set__lightColour`) which is associated with the corresponding `inputOnly` field (`set_lightColour`) of the X3D prototype interface. The field `_lightColour` of the script is required for initialization purposes and connected with the initialization field `lightColour` of the X3D prototype interface.

If `lightColour` was a property with not only write access but also read access, additional output fields for the prototype interface and the `lightColourScript` node would be generated. Table 3 clarifies the mapping of SSIML component properties to X3D prototype fields.

Listing 2: Mapping of the SSIML component Lamp to X3D code

```

PROTO Lamp [
  inputOutput SFString geometryURL ""

  initializeOnly SFNode lightColour HSBColour
  { ... brightness 0.0 }

  inputOnly SFNode set_lightColour
]
{
  Group {
    children [

      DEF lightColourScript Script {
        directOutput TRUE

        initializeOnly SFNode _lightColour
        IS lightColour

        inputOnly SFNode set__lightColour
        IS set_lightColour

        #output format specified
        #by the programmer
        outputOnly SFCOLOR colour_changed

        url ["ecmascript:
          function initialize() {
            generateOutput(_lightColour);
          }

          function set__lightColour(v) {
            if (v!=null) _lightColour = v;
            generateOutput(_lightColour);
          }

          //has to be implemented
          //by the programmer
          function generateOutput(v) {

          }

        "]
      }
    ]
  }
  ...

```

The field `url` of the script contains necessary operations. The function `initialize` is used to send the initial values of `lightColour` via the function `generateOutput` to nodes connected with the script. The function `set__lightColour` has the same purpose with the difference that it calls `generateOutput` every time the values of `lightColour` were changed *after* initialization time.

The programmer can define various output fields and implement the `generateOutput` function of the script. This allows generating output data in a format which is suitable for further processing. In this case the programmer would write the code to convert the colour stored in the `_lightColour` field from the HSB into the RGB format. The result could then be stored in a `SFCOLOR` object and sent via an output port to color fields of connected nodes. The code for the `generateOutput` function can also be embedded into the code generation routines. This has the advantage that the code is generated automatically whenever a `HSBColour` property occurs in a SSIML model. Therewith implementation costs can be kept at a moderate level.

The geometry of the lamp object can be integrated into the prototype body using appropriate authoring tools. Output fields of the `lightColourScript` node can be connected via `ROUTE` state-

Table 3: Mapping of SSIML component properties to X3D fields

	Simple Type	Complex Type
Init	<i>initializeOnly</i>	<i>initializeOnly</i>
Read	<i>initializeOnly</i> , <i>outputOnly</i>	<i>initializeOnly</i> , <i>outputOnly</i>
Write	<i>initializeOnly</i> , <i>inputOnly</i>	<i>initializeOnly</i> , <i>inputOnly</i>
ReadWrite	<i>inputOutput</i>	<i>inputOnly</i> , <i>initializeOnly</i> , <i>outputOnly</i>

ments with colour fields of light sources or material nodes.

SSIML component instances are mapped to instances of the corresponding X3D prototypes. For example, `headLamp` is mapped to an instance of the prototype `Lamp`. Composed SSIML components are translated to X3D prototypes which contain instances of other prototypes. For example, the `ScouterMainUnit` prototype contains some instances of the `ToggleLamp` prototype.

Behaviours are also stored in specialized X3D prototypes. Behaviour prototypes contain script nodes which reference Java classes. The behaviour is specified by Java code inside the referenced file. The most important fields of a behaviour prototype interface are fields for starting the behaviour execution and receiving and sending message strings. A prototype which contains a behaviour instance contains also interface fields which are directly linked with the corresponding fields of the behaviour instance. For example, the `ScouterMainUnit` prototype has an input field `toggleMainLight.start` which is connected with the corresponding input field `start` of the `toggleMainLight` behaviour instance. A more detailed overview about the generation of behaviour code from SSIML models can be found in [Vitzthum 2005].

6 Conclusion and Outlook

In this paper we argue that, on one hand, 3D components can speed up the development of interactive 3D content and enhance the integration of 3D content into applications, but, on the other hand, there is a lack of concepts and tools which support 3D component concepts above the implementation level. Thus, especially for components with complex inner structures and a large number of properties component specification on the code level can become an time-consuming task. To address this problem we introduce SSIML/Components, a visual language for the pre-implementation design of interactive 3D components. SSIML/Components also allows composing 3D components to create new components. Besides, it is possible to decompose complicated scene structures into manageable, maintainable and reusable parts. In addition, by using the SSIML/Components language the integration of 3D components into applications can be specified easily. In order to enable a seamless transition to the implementation level, we have investigated the transformation of SSIML/Components models into X3D code. First results indicate that implementation costs can be reduced by generating code automatically from the models. Furthermore, SSIML/Components models can be used for documentation purposes and for discussing the structural design of a 3D scene.

We plan to enhance and complete the code generation routines to achieve a further reduction of the implementation efforts. Moreover, in order to examine the platform independence of the SSIML/Component models we want to generate code for other target languages than X3D. Finally, we plan to evaluate our approach

on more complex real world examples.

References

- CAPPS, M., MCGREGOR, D., BRUTZMAN, D., AND ZYDA, M. 2000. Npsnet-v: A new beginning for dynamically extensible virtual environments. *IEEE Comput. Graph. Appl.* 20, 5, 12–15.
- CONNER, B. D., SNIBBE, S. S., HERNDON, K. P., ROBBINS, D. C., ZELEZNIK, R. C., AND VAN DAM, A. 1992. Three-dimensional widgets. In *SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics*, ACM Press, New York, NY, USA, 183–188.
- DACHSELT, R., HINZ, M., AND MEISSNER, K. 2002. Contigra: an xml-based architecture for component-oriented 3d applications. In *Web3D '02: Proceeding of the seventh international conference on 3D Web technology*, ACM Press, New York, NY, USA, 155–163.
- DÖRNER, R., AND GRIMM, P. 2000. Three-dimensional beans - creating web content using 3d components in a 3d authoring environment. In *VRML '00: Proceedings of the fifth symposium on Virtual reality modeling language (Web3D-VRML)*, ACM Press, New York, NY, USA, 69–74.
- ELLIOTT, C. 1999. An embedded modeling language approach to interactive 3d and multimedia animation. *Software Engineering* 25, 3, 291–308.
- GARCÍA, P., MONTALÀ, O., PAIROT, C., RALLO, R., AND SKARMETA, A. G. 2002. MOVE: component groupware foundations for collaborative virtual environments. In *CVE '02: Proceedings of the 4th international conference on Collaborative virtual environments*, ACM Press, New York, NY, USA, 55–62.
- NOMAGIC. *Magic Draw*. <http://www.magicdraw.com/>.
- OMG. 2002. *Meta Object Facility (MOF) Specification, Version 1.4*.
- OMG. 2003. *XML Metadata Interchange (XMI) Specification, Version 2.0*.
- OMG. 2005. *UML Superstructure Specification, Version 2.0*.
- OPENSF-FORUM. *OpenSG*. <http://www.opensf.org/>.
- PARALLELGRAPHICS. *Internet Scene Assembler*. <http://www.parallelgraphics.com>.
- RATIONAL. *ROSE*. <http://www-306.ibm.com/software/rational/>.
- REYNOLDS, C. W. 1982. Computer animation with scripts and actors. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 289–296.
- SEGURA, A., ARIZKUREN, I., ARANBURU, I., AND TELLERIA, I. 2005. High quality parametric visual product configuration systems over the web. In *Web3D '05: Proceedings of the tenth international conference on 3D Web technology*, ACM Press, New York, NY, USA, 159–167.
- SUN MICROSYSTEMS. *Java Beans*. <http://java.sun.com/products/javabeans/>.
- VITZTHUM, A., AND PLEUSS, A. 2005. SSIML: Designing structure and application integration of 3d scenes. In *Web3D '05: Proceedings of the tenth international conference on 3D Web technology*, ACM Press, New York, NY, USA, 9–17.

VITZTHUM, A. 2005. SSIML/behaviour: Designing behaviour and animation of graphical objects in virtual reality and multimedia applications. In *ISM '05: Proceedings of the IEEE International Symposium on Multimedia 2005*, IEEE Computer Society Press, 159–167.

W3C. 1999. *XSL Transformations (XSLT), Version 1.0*.

WEB3D-CONSORTIUM. 2003. *ISO/IEC 14772:1997 Virtual Reality Modeling Language (VRML)*.

WEB3D-CONSORTIUM. 2005. *ISO/IEC 19775:2004 Extensible 3D (X3D)*.

