

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/254463473>

Model-driven iterative development of 3D web-applications using SSIML, X3D and JavaScript

Article · August 2012

DOI: 10.1145/2338714.2338742

CITATIONS

4

READS

160

Some of the authors of this publication are also working on these related projects:



ARIDuA - Autonomous robots and Internet of Things in underground mining [View project](#)



Mining-RoX: Autonomous Robots in Underground Mining [View project](#)

Model-Driven Iterative Development of 3D Web-Applications Using SSIML, X3D and JavaScript

Matthias Lenk*
TU Bergakademie Freiberg
Virtual Reality and Multimedia Group
<http://vr.tu-freiberg.de>

Arnd Vitzthum†
University of Cooperative Education
<http://www.ba-dresden.de>

Bernhard Jung‡
TU Bergakademie Freiberg
Virtual Reality and Multimedia Group
<http://vr.tu-freiberg.de>

Abstract

In traditional software engineering domains, Model-Driven Development (MDD) using UML or domain-specific languages (DSL) is successfully established. Although MDD is a particularly promising approach to avoid implementation errors due to miscommunication between heterogeneous developer groups, only a few MDD approaches for 3D-development have been proposed so far. In this paper, we describe how one such MDD approach, SSIML, can be extended to a full round-trip engineering approach to structured 3D-development. Round-trip engineering combines a forward phase, where code is automatically generated from an abstract model of the application, with a reverse phase, where manual code edits are merged back into the abstract model. The proposed approach is demonstrated with the development of 3D web applications based on X3D/X3DOM and JavaScript.

Keywords: 3D development, round-trip engineering, code generation, reverse engineering, model merging, X3D, X3DOM, JavaScript

1 Introduction

3D development is an interdisciplinary process. Basically, two important groups of developers who use completely different tools, data formats, and terminologies can be distinguished: 3D content developers and programmers. Misunderstandings between the developer groups can lead to an inconsistent system implementation [Vitzthum 2008]. An example could be the implementation of a 3D product configuration system. First, the 3D content developer models the scene containing the 3D product representation with a 3D authoring tool. Afterwards, the programmer loads the scene using the appropriate functions of a scene graph API. If the 3D content developer does not follow the conventions for the naming of 3D objects (in this case parts of the product), the programmer cannot address these objects properly via program code. Also, like most software, 3D applications are usually developed in an iterative fashion. More objects may be added to the scene, existing objects may be modified, or some other objects may be deleted. If an early ver-

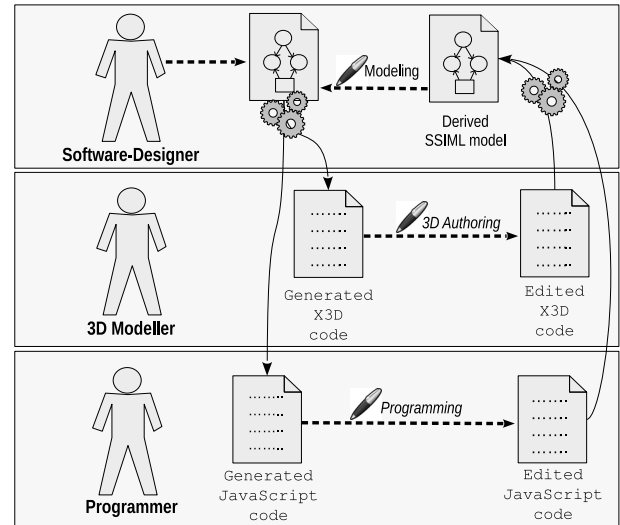


Figure 1: Model-driven development process with roundtrip engineering, applied to 3D web application development.

sion of the software properly works, errors still may be introduced to the software in later phases of development.

As argued in [Vitzthum and Jung 2010], the above issues call for a model-driven, iterative development process with round-trip engineering. In model-driven development (MDD), program code is partially generated from an abstract model, usually specified in UML or a domain-specific language (DSL). *Round-trip engineering (RTE)* [Aßmann 2003; Sendall and Kster 2004; Van Paesschen et al. 2005; Henriksson and Larsson 2003; Antkiewicz 2007] is a model-driven software development methodology that combines forward engineering (model-to-code transformations) with reverse engineering (code-to-model transformations) to support an iterative development process.

1.1 Related work

Compared to traditional software, 3D software is often developed in a code-centric and ad-hoc manner. A design phase, which precedes the implementation phase in traditional software engineering, is seldom present in the 3D domain and only a few MDD approaches have been proposed so far for 3D development [Vitzthum and Jung 2010]. The work presented in this paper builds on one such MDD approach, SSIML, a DSL for specification of 3D applications at a high level of abstraction. In previous work on SSIML, X3D and application logic code were partially generated from an abstract

*e-mail: lenk@informatik.tu-freiberg.de

†arnd.vitzthum@ba-dresden.de

‡e-mail: jung@informatik.tu-freiberg.de

SSIML model (forward engineering) [Vitzthum and Pleuß2005]. A reverse engineering step, where an initial SSIML model is derived from a X3D model has been proposed in [Pleuss et al. 2007]. However, prior work on SSIML does not cover full iterative round-trip engineering.

Other approaches that aim at supporting (special aspects of) high-level 3D development processes include CONTIGRA [Dachselt et al. 2002] that focuses on 3D components and the Interaction Techniques Markup Language (InTML) [Figueroa et al. 2002] that focuses on the integration of 3D interaction techniques into VR applications. CONTIGRA and InTML documents can be automatically transformed into executable formats (e.g. X3D, Java or C++), but a full round-trip is not supported.

Further research to improve 3D application development addresses 3D designers and aims at simplifying the programming task or enabling prototyping and rapid design [Conway et al. 1994; MacIntyre et al. 2004; Abawi et al. 2004]. Although such tools can be used to create interactive 3D applications, without the need for manually programming source code, resulting applications are limited in their range.

In contrast, several tools for round-trip engineering are available for development of traditional (non-3D) software, such as *UML LAB*¹ or *Together*². These tools allow for the *simultaneous* editing of models (rendered as UML class or sequence diagrams) and source code in the same IDE, while changes to either model or code are directly applied to its counterpart. Such tools are however not applicable to the development of 3D applications as 3D development involves different developer groups, i.e. 3D content designers and programmers, who cannot use the same tool / IDE at the same time. Instead, the concurrent development workflow of 3D designers and programmers calls for a *non-simultaneous* synchronization of high-level application models, 3D models and application logic code.

1.2 Contribution and outline

In this paper we describe how round-trip engineering techniques can be applied to the model-driven, iterative development of 3D web applications based on SSIML, X3D [Daly and Brutzman 2008] / X3DOM [Behr et al. 2009] and JavaScript. In our approach, X3D and JavaScript code skeletons are automatically generated from an initial SSIML model (forward engineering). Also, elaborations and modifications at the code level are merged back into the model (reverse engineering). This step is repeated in later iterations, realizing a full round-trip engineering development process (Figure 1).

This paper is organized as follows: In Section 2, we give a short introduction to SSIML. A detailed example of our round-trip approach is presented in Section 3. Then, Section 4 outlines the implementation of the forward and reverse engineering transformations of the proposed approach before we finally conclude in Section 5.

2 SSIML

SSIML, the *Scene Structure and Integration Modeling Language*, is a domain-specific language (DSL) for specifying 3D applications [Vitzthum and Pleuß2005]. SSIML allows for modeling 3D scene graphs at a high level of abstraction.

A SSIML model is comprised of the *scene model* and the *interrelationship model*. The scene model contains different kinds of nodes which represent less or more complex 3D objects. All nodes are















Scene Model Element	Notation	Example
Scene	 <Name>	 robotScene
Object	 <Name> (*:<ContentType>)?	 base: BaseGeom
Group	 <Name>	 robotGroup
Transformation Attribute	 A <Name>	 transform0
TouchSensor	 SA <Name>	 touchSensor
Parent-Child-Relationship	 (<Multiplicity>)?	 controlPanel 4 button
Node-Attribute-Relationship		 segment0 transform0

Figure 2: Notation examples for SSIML scene model elements.

interconnected by *parent-child relationships*. Child nodes can be thought of as parts of their more complex parents. This also means that, if the parent 3D object is moved within the scene, all child objects move with their parent, too. For instance, if a parent object *robot base* is moved, the segments of the robot attached to the base also move. However, the motion of a parent object does not depend on the motion of its children. For example, the last segment of the robot can be rotated without moving the first segment.

All nodes can act as child nodes as well as parent nodes except the *scene root* node. All nodes are reachable from the scene root by traversing parent-child relationships.

Beside the scene root node there are two other important types of nodes: *objects* and *groups*. An object represents a specific geometric 3D object (e. g., a console), although the concrete geometry




Interrelationship Model Element	Notation	Example
Application Component	<Name>	RobotControlClass
Event Relationship	<Event Type>	 touchSensor CLICKED RobotControl
Action Relationship	<Action Type>	 transform0 MODIFIES RobotControl
Representation Relationship	'REPRESENTS'	 robot REPRESENTS RobotInfo

Figure 3: Notation examples for SSIML interrelationship model elements.

¹ <http://www.uml-lab.com>

² <http://www.borland.com/us/products/together/>

is not further defined within the scene model. However, the user can specify the *content type* of an object. The content type acts as reference to a detailed 3D geometry description (e. g., a link to an X3D-encoded 3D-Model). Since an object can act as parent node, it can have child objects. For instance, there could be a parent object *console* containing child objects such as different buttons.

Group nodes are very similar to object nodes, but with the difference that they do not encapsulate geometry. A common application of group nodes is the formation of logical groups of objects, such as the group of all objects on a table. A special type of nodes are *composed nodes*, which enable the encapsulation and reuse of model sub graphs in order to reduce overall complexity and to enhance manageability.

The SSIML scene model provides many additional features, such as *child multiplicities*. Child multiplicities facilitate the definition of sets of objects which own the same geometry and appearance. For example, a certain button can have multiple instances within a 3D visualization of a console.

Objects and groups are so called *located nodes*. A located node has a *transformation attribute* that models the position, orientation and scale of the 3D object represented by the node. Beside the transformation attribute, there are other node specific attributes. For instance, the scene root node can have attributes which express scene illumination, viewpoint and viewing parameters. Nodes and node attributes can be identified by their *names*.

Special node attributes are *sensors*. A runtime instance of a platform-specific sensor implementation (e. g., in X3D) can generate events. There exist different sensor types. For instance, a *touch sensor* instance generates an event if the 3D object which owns the sensor was selected with a pointing device (e. g., a computer mouse). The visual notation of some SSIML elements described above is depicted in Figure 2.

The scene model is complemented by interrelationship model. In interactive 3D applications, there typically exist interrelationships between elements of the 3D scene and *application components* that are responsible for flow control and that hold additional data. These components may also handle events generated by sensors. A relationship between a sensor and an application component that models an event travel route is called *event relationship*. Frequently, actions triggered by event messages modify the 3D scene, e. g., for animation purposes. The ability of an application component to modify an associated scene element can be modeled using *action relationships*. There exist different subtypes of action relationships, such as relationships for removing, adding or replacing parts of the scene and relationships for modifying properties of 3D objects, such as position and orientation (i. e., the transformation attribute's value). A special relationship between a scene element and an application component is the *representation relationship*. Typically, an application component which is wired with a 3D object via a representation relationship contains metadata concerning the 3D object, such as a detailed description of the object and other application- specific data. For instance, the robot object could be associated with technical data of the robot. The graphical notation of SSIML interrelationship model elements is shown in Figure 3.

3 Round-trip example: From SSIML to X3D/JavaScript and back

In the proposed 3D web development process, development begins with the specification of a SSIML model through the software designer. Then, during the forward engineering phase of the development process, the SSIML model is transformed to X3D and JavaScript code skeletons. 3D modelers and programmers then

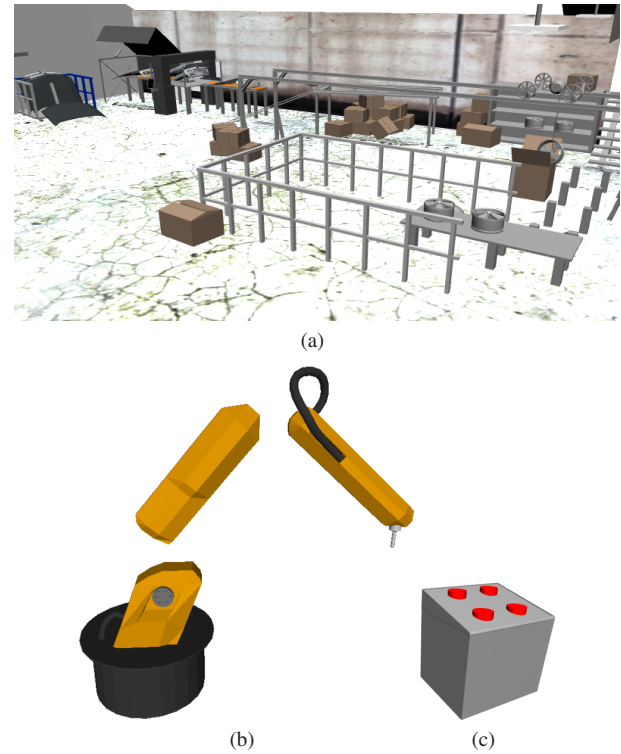


Figure 4: (a) The 3D factory hall includes the structure and accessories. (b) The single parts of the industrial robot. (c) A simple console with buttons, meant to rotate the segments of the robot.

elaborate the auto-generated code. In a reverse engineering phase, the code changes are then played back to update the original SSIML model, and the next iteration begins.

For a better illustration of the proposed 3D web development process, we introduce a small example that is a simulation of a robot in a factory. The 3D scene is composed of the following 3D objects: A factory hall (Figure 4(a)) that already contains accessories like boxes, a simplified model of an industrial robot (Figure 4(b)) that is explicitly made up of its single components and a panel with four control buttons (Figure 4(c)). The 3D designers' task is to model the individual 3D objects and their composition in a single scene. Furthermore, the application shall provide an interactive part that allows the user to control the robot directly within the 3D scene, by pressing buttons on the panel using the mouse. This interaction and animation part has to be programmed in JavaScript. A common SSIML model serves as specification for both the 3D designers' and JavaScript programmers' tasks.

3.1 SSIML model

Figure 5 shows the graphical representation of the SSIML model for the robot scene example. The scene model contains several group nodes to structure the 3D scene as well as the single objects. The object nodes contain relevant attributes, e. g., each instance of the four panel buttons is connected to a respective touch sensor attribute. When a button is pressed by a user, the associated sensor will notify the `RobotControlClass` in the interrelationship model. This application component, which will later be realized as a set of JavaScript functions, is able to access the transformation attributes within the 3D scene and therefore can animate the robot.

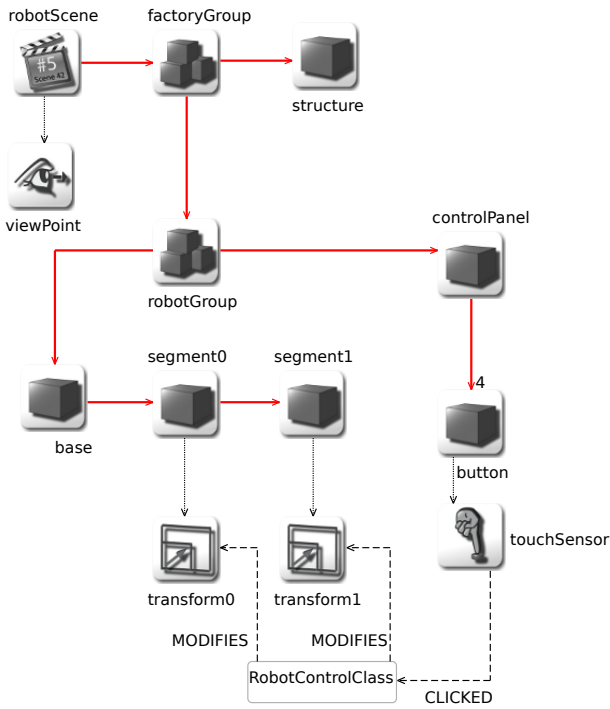


Figure 5: The robot example in the graphical SSIML editor.

```
<!-- id=1218cda5-... -->
<X3D version="3.0" profile="Immersive">
  <!-- id=a52a010e-... -->
  <Scene>
    <!-- id=9de0e6b0-... -->
    <Transform DEF="generatedTransform1"
      translation="0 0 0" rotation="0 0 1 0"
      scale="1 1 1" center="0 0 0" ...>
    <!-- id=4044ed10-... -->
    <Group DEF="factoryGroup">
      <!-- id=7fdbb932-... -->
      <Transform DEF="generatedTransform2" ...>
      <!-- id=4a931273-... -->
      <Inline DEF="structure" url="factory.x3d">
      </Inline>
    </Transform>
    <!-- id=b6444f60-... -->
    <Transform DEF="generatedTransform3" ...>
    <!-- id=8bf539b1-... -->
    <Group DEF="robotGroup">
      <!-- id=00311681-... -->
      <Transform DEF="generatedTransform4" ...>
      <!-- id=3e162bea-... -->
      <Inline DEF="base" url="base.x3d">
      </Inline>
      <!-- id=c142450a-... -->
      <Transform DEF="transform0"
        translation="0 0 0" rotation="0 0 1 0" ...>
      <!-- id=4e3af322-... -->
      <Inline DEF="segment0" url="seg0.x3d">
      </Inline>
    </Group>
  </Scene>
</X3D>
```

Listing 1: Generated X3D code skeleton for the robot example.

3.2 Code generation from SSIML model

In the forward phase, the SSIML model is transformed to X3D and JavaScript code. More concretely, the elements of the SSIML scene model are converted to X3DOM code fragments, and elements of the SSIML interrelationship model to JavaScript code fragments.

3.2.1 Generation of X3D code from SSIML models

SSIML model elements are mapped to X3D nodes in the following way: The SSIML model and the scene root node are transformed to an X3D tag and a scene tag, respectively. Each SSIML group node, e.g. the `factoryGroup`, is mapped to a corresponding group tag in X3D. SSIML object nodes are transformed to X3D inline nodes. A generated X3D inline node links to the X3D file specified in the `encapsulatedContent` attribute of the SSIML object, or, if that attribute is undefined, to a newly generated URL. Additionally, an X3D transform node is generated for these inline nodes in order to support scene composition activities such as translating and rotating the 3D objects to their final positions in the complete 3D scene.

Furthermore, a unique ID – with which each SSIML element is associated – is assigned to each X3D node generated from the SSIML model. This is necessary to track changes made to an element during the edit phase. The ID is bound to its element during the complete development process and may not be modified or removed. Elements without ID will later be treated as newly added elements. To store IDs in the X3D code we use XML comments, each referring to the following X3D node.

Listing 1 shows fragments of the X3D code that is automatically generated from the SSIML model of Figure 5.

3.2.2 Generation of JavaScript code from SSIML models

Similar to X3D code, a corresponding JavaScript code skeleton is generated to access the 3D scene. This code consists of a set of JavaScript functions, which emulate the structure of object-oriented programming languages, such as Java or C++, to simplify the adaptation to other target languages in future work. Listing 2 depicts the generated JavaScript code.

JavaScript code fragments are related to the original SSIML model of Figure 5 in the following way: The SSIML application component `RobotControlClass` is translated to a JavaScript function of the same name that creates a suitable JavaScript object. The application component's action relationships to the original SSIML attributes `transform0` and `transform1` result in member variables of the JavaScript object. These two variables address X3D transform nodes which can be used to modify the robot's segments. To realize touch sensors in X3DOM, appropriate event listeners (conforming to the HTML event model) have to be added to the corresponding X3D nodes [Behr et al. 2009]. The four buttons nodes on the panel, each connected to a touch sensor in the SSIML model, are selected and event listeners are attached in the `init()` function. The `CLICKED` event from the SSIML example is mapped to an html `click` event. Further application logic, e.g. to calculate the rotation values for a robot animation, will be manually programmed by filling in code stubs or in additional functions.

Generated JavaScript elements are also tracked by assigning IDs. The mapping from SSIML elements to corresponding JavaScript elements is more complex than to X3D nodes, since multiple SSIML elements may result in one JavaScript element. E.g., the event handler assignment within the `init()` function is comprised of the concerned object (i.e. `button0`), the attached sensor and the relationships which connect the related objects. To store these IDs we use comments with a *JavaDoc*-like format (Listing 2).


```

/**
 * @id: JavaScriptFile 12...
 */
window.onload = init;

function init( ) {

  /**
   * @id: Assignment 38...
   */
  var robotControlClass = new RobotControlClass( );

  /**
   * @id: Object 1b...
   * @id: NodeAttributeRelationship 1f...
   * @id: TouchSensor db...
   * @id: EventRelationship c4...
   */
  document.querySelector( ' [DEF="button0"]' )
    .addEventListener( "click" ,
      robotControlClass.button0_CLICKED );

  ...
}

/**
 * @id: JavaScriptClass 38...
 */
function RobotControlClass( ) {

  /**
   * @id: ValueAccessRelationship bf...
   * @id: Object c1...
   */
  var transform0
    = document.querySelector( ' [DEF="transform0"]' );

  /**
   * @id: ValueAccessRelationship 24...
   * @id: Object 83...
   */
  var transform1
    = document.querySelector( ' [DEF="transform1"]' );

  /**
   * @id: UserFunction c4...
   */
  this.button0_CLICKED = function( obj ) {
    // TODO Auto-generated method stub
  }

  ...
}

```

Listing 2: Generated JavaScript code skeleton to access the 3D scene.

3.3 Code refinement and extension

In the next phase of development, the auto-generated X3D and JavaScript code is refined concurrently by 3D modelers and JavaScript programmers. To implement the SSIML model, 3D designers have to provide the various inlined X3D models of Listing 1 and position them in the scene (by setting the attributes of the Transform nodes in Listing 1). Similarly, programmers have to implement the generated JavaScript method stubs. Listing 3 shows the event handler implementation through which the robot's first segment is rotated a little bit when the user presses a certain button on the panel.

In iterative model-driven software development, models and code are alternately refined. Usually, the programmer implements the current model but then will have to wait with further activities until the next version of the model is released by the software designer. Round-trip engineering, however, adds a reverse engineering phase where model updates can be partially derived from the code. There-

```

...
function button0_CLICKED (obj){
  var rot = transform0.getAttribute("rotation");
  var angle = parseFloat(rot.split(" ")[3], 10);
  angle += 0.04;
  transform0
    .setAttribute("rotation","1 0 0 " + angle);
};
...

```

Listing 3: Implementation of the event handler function to rotate the robot segment0 when the corresponding button is clicked.

fore, 3D designers and programmers may make modifications to the X3D models and, resp., JavaScript code that not just implement the current SSIML model but rather represent useful features that might be integrated into next iteration of the SSIML model. In general, code level changes are allowed that result in the update or deletion of existing nodes as well as the creation of new nodes in the SSIML model. For our running example, we assume the following model-relevant changes to the code:

1. (Update operation) The JavaScript programmer prefers a different naming convention for X3D nodes. In the code shown in Listing 2, instead of addressing the X3D transform nodes defining the robot's joints by their old names `transform0` and `transform1` she uses the new names `segment0Transf` and `segment1Transf`, respectively. This leads to an invalid application since the SSIML model and X3D code still rely on the original naming.
2. (Insert operation) The 3D modeler notices that the four buttons of the control panel suffice only to control 2 degrees of freedom of the robot in the application. Anticipating that a third degree of freedom should be controllable through the user interface, he adds two more buttons to the control panel in the X3D model. So far, of course, the new buttons have neither been added to either the SSIML model, nor has JavaScript code been written to animate the robot upon touching the buttons in the 3D web application.
3. (Delete operation) The 3D modeler feels the time pressure of an approaching deadline. Rather than taking the time to find a suitable view, he deletes the `Viewpoint` node from the X3D model (this deletion, however, will later be rejected by the software designer, so that the `Viewpoint` will still be part of the SSIML model and the 3D modeler will have to define a suitable view in the next iteration of development).

3.4 Reverse engineering the SSIML model from X3D and JavaScript code

In the reverse engineering step, an updated SSIML model is derived from the modified X3D and JavaScript code. The automatic detection of code modifications relies on the generated IDs in the source code. When a difference between the old model and the reverse engineered model is detected,³ the question arises whether this deviation should result in an update of the SSIML model or whether the implementation should be considered as incorrect. In our example, it is assumed that the above changes (1) and (2) are accepted while the deletion (3) is rejected. Now, an updated SSIML model is automatically derived. In the updated model, the transform nodes are renamed and two more buttons are added to the control panel.

³Change detection is actually performed on an intermediate level between the SSIML model and the source code. Intermediate models are however invisible to the involved developers, see Section 4.

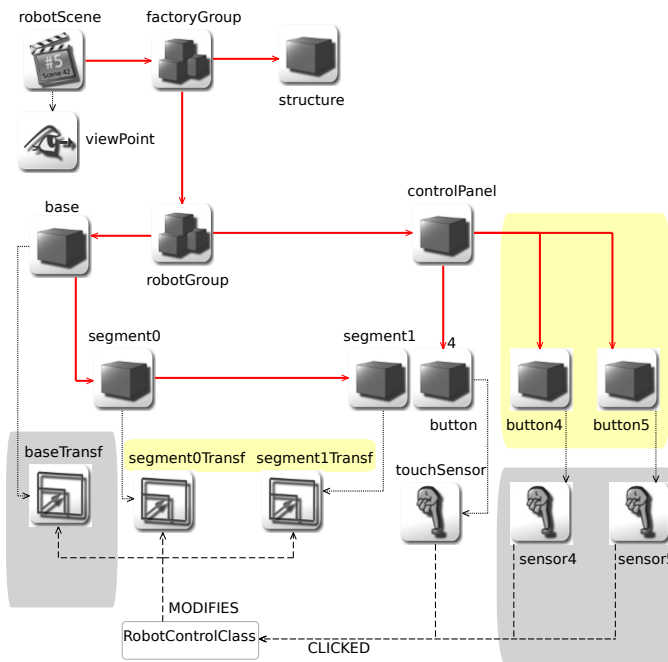


Figure 6: The SSIML model of the final application. Yellow highlighted elements are derived from modified source code, while gray highlighted elements are newly added by the software designer.

The new buttons are however not yet wired to a touch sensor. Also, the SSIML model still contains the original viewpoint node (that the 3D modeler wanted to delete).

3.5 Iterative development: Modifying the SSIML model and re-generating code

After an updated SSIML model has been derived in the reverse engineering step, the software designer may edit the SSIML model further. In our example, the derived SSIML scene model contains two additional buttons on the control panel. Still, the SSIML interrelationship model has to be defined for these new buttons. The software designer accomplishes this by wiring the new buttons to new touch sensors, analogously to the four previously defined buttons. Further, a `CLICKED` event relationship is established for each of the new touch sensors with the `RobotControlClass`. Further, a `MODIFIES` action relationship is added between the `RobotControlClass` and the transformation attribute of the robot's base. Through these changes, the new SSIML model now specifies that clicking the new buttons will result in a rotation of the robot's base.⁴ Figure 6 depicts the updated SSIML model with renamed transform nodes and now six instead of four buttons on the control panel.

So far, however, the SSIML model, the X3D scene and the JavaScript code are not consistent with each other. E.g. the transform nodes of the robot segments are already renamed in the SSIML model and the JavaScript code, but the X3D code still con-

⁴Note that the new desired robot behavior could have been modeled differently in SSIML. E.g., more in the style of the original SSIML model, the multiplicity of the parent-child relation between the SSIML nodes `controlPanel` and `button` could have been changed from 4 to 6 while removing the automatically inserted SSIML nodes `button4` and `button5`. We chose the shown SSIML model in order to demonstrate the different possibilities of modeling 3D applications in SSIML.



Figure 7: The updated web application after the second (and final) iteration.

tains the old names. Therefore, X3D and JavaScript code must be generated again from the SSIML model. During code generation of the second iteration, manually edited code from the previous iteration must not be forgotten. E.g., the JavaScript implementation of the event handler shown in Listing 3 is re-generated in the new JavaScript code.

Afterwards, the auto-generated X3D and JavaScript code needs to be manually refined in order to completely implement the updated SSIML model. In our example, the JavaScript programmer needs to implement the event handler code to animate the robot when the new buttons are touched by the user. The 3D modeler still needs to define a suitable viewpoint.

A final reverse engineering step (code-to-model) verifies that SSIML model and code bases are consistent with each other. The X3D and JavaScript code can now be deployed. Figure 7 shows the resulting web application.⁵

4 Implementation overview

Round-trip engineering (RTE) combines model-driven forward engineering (code generation from models) with reverse engineering (model generation from code). Today, several RTE tools exist that support the *simultaneous* editing of UML diagrams and program code. However, these tools are not suitable for development of 3D applications as 3D development involves different developer groups, i.e. 3D content designers and programmers, who cannot use the same tool at the same time. Thus, a *non-simultaneous* means of synchronizing the X3D and JavaScript code bases with the common SSIML model is required. To address the challenges of non-simultaneous RTE for 3D development, we developed a multi-tiered approach where the SSIML model is first transformed to an *intermediate model (IM)*. The intermediate model is then converted to language-specific *abstract syntax trees (ASTs)* [Jones 2003] for X3D and JavaScript, respectively. Finally, concrete source code in

⁵Available at: <http://elrond.informatik.tu-freiberg.de/roundtrip3d/robot>

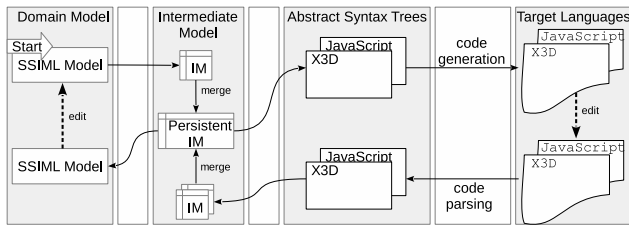


Figure 8: Implementation of the round-trip process. An initial SSIML model is transformed into its intermediate representation, which is then transformed into two abstract syntax trees for X3D and JavaScript, respectively. After code generation from these abstract syntax trees, the code skeletons can be concurrently refined. In the reverse phase, intermediate models derived from X3D and JavaScript code are merged into the central persistent model, from which an updated SSIML model is derived. The new SSIML model can be edited by the software designer, before the next iteration begins.

JavaScript and X3D is generated from the respective abstract syntax trees. After editing of the auto-generated code skeleton, a reverse engineering step is performed where source code is transformed first to abstract syntax trees and then to intermediate models. Merging of manual changes to the X3D and JavaScript occurs at the level of intermediate models. From the merged intermediate model, an updated SSIML model is derived. Now, the next iteration of the development can begin. Figure 8 summarizes this round-trip process at the implementation level.

The intermediate model (IM) serves as central data structure in our round-trip approach. The SSIML model as well as JavaScript and X3D code can be translated to corresponding IM representations (the latter two via their abstract syntax tree representations). A *persistent intermediate model*, into which the IMs of the current SSIML model, JavaScript code and X3D code are merged, contains at any given time a complete description of the web application. UUIDs associated with the respective language elements are used to relate corresponding elements. After the SSIML model, JavaScript or X3D code have been edited manually, the changes are merged into the persistent IM. Then, from the updated persistent IM, updated versions of the SSIML model, JavaScript and X3D code are automatically generated in order to synchronize the various code bases.

The intermediate model describes the SSIML, JavaScript and X3D contents of the web application in a generic fashion (much like an object-oriented database would). Objects in the intermediate model correspond SSIML nodes or X3D / JavaScript code that can be generated from SSIML nodes. Other X3D and JavaScript content that is not represented in the SSIML model, e.g. manually implemented function bodies in the JavaScript code, is stored as plain text in attribute values of generic objects. Intermediate models are not intended to be viewed or edited by programmers. Listing 4 shows a short excerpt of an intermediate model that comprises content derived from SSIML, X3D, and JavaScript to give a brief impression of the XML-based storage format.

The conversions between the various models and code bases is implemented through different transformation frameworks. Translations between SSIML and intermediate models are achieved by using the reflective API provided by EMF [Steinberg et al. 2009]. Conversions between intermediate models (IMs) and abstract syntax trees (ASTs) are achieved via ETL [Kolovos et al. 2008], a hybrid transformation language that allows for declarative mappings and imperative constructs. Figure 9 shows part of the mapping

- ✦ Mapping Model SSIML to X3D
- ✦ Source Element SSIMLModel
- ✦ Target Element X3D
- ✦ Source Element Scene
- ✦ Target Element Scene
- ✦ Source Element Viewpoint
- ✦ Target Element Viewpoint
- ✦ Source Element Group
- ✦ Target Element Group
- ✦ Source Element Transformation
- ✦ Target Element Transform
- ✦ Source Element Object
- ✦ Target Element Inline
- ✦ Source Element DirectionalLight
- ✦ Target Element DirectionalLight
- ⋮

Figure 9: The mapping from SSIML elements to X3D nodes is declaratively defined in a mapping model, i.e. Metamodel Mappings [Miller and Mukerji 2003].

model used for the conversion of SSIML elements (in their intermediate representation) to an AST for X3D code. For the conversion between the X3D and JavaScript code and their respective ASTs, we use Xtext [Efftinge and Völter 2006]. Xtext allows to automatically derive serializers for AST-to-code conversions (forward phase) and as well as parsers for code-to-AST conversion (reverse phase) from a language metamodel and a grammar.

4.1 Discussion of implementation

Our implementation of RTE for 3D applications differs from other RTE approaches by introducing an intermediate model that contains all information from the high-level model (in our case: the SSIML model) and the respective code bases (here so far: X3D and JavaScript). This design choice was motivated by special characteristics of 3D development, including the need for code generation in two relatively different target languages (declarative X3D and procedural / object-oriented JavaScript) and the need for non-simultaneous synchronization due to the concurrent development of 3D designers and programmers. A further goal is the support of multi-platform development, where the same SSIML model could be used for applications on the web, over mobile devices to high-end VR installations such as the CAVE.

Our first implementation of RTE was to directly generate code from a SSIML model with MOFScript. This straightforward model-to-text transformation, however, turned out to be hard to manage and extend. Instead, our multi-tiered approach aims at manageable transformations, which can be achieved best by using different transformation languages. Reflection-like transformations in Java convert between the SSIML model and its generic intermediate representation. Combining hybrid ETL with an declarative and extendable mapping model allows for well structured conversions between the IM and ASTs. Through Xtext, model-to-text as well as text-to-model transformations are obtained automatically from the language descriptions without the need of implementing them manually. By dividing the model-to-code transformation into several smaller steps, also the extension to further programming languages is simplified. For example, in future work we plan to support CAVE settings with C++ as programming language. Here, the SSIML-to-IM transformation can be completely reused. Similarly, code generation from the AST as well as code parsing for the reverse phase can be achieved with Xtext with relatively little effort.

A further advantage of the multi-tiered approach using an IM is


```

<?xml version="1.0" encoding="ASCII"?>
<intermediate:IntermediateModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" ... >

  <intermediateObjects ... featureType="Object" ...>
    <languages>SSIML</languages>
    <attributes featureName="name" attributeValue="button0" ...>...</attributes>
    <attributes featureName="encapsulatedContent" attributeValue="scene/button.x3d" ...>...</attributes>
    <attributes featureType="NodeAttributeRelationship" attributeValue="1f803949-..." ...>...</attributes>
    ...
  </intermediateObjects>

  <intermediateObjects featureType="Transformation" ...>
    <languages>X3D</languages>
    <attributes featureName="name" attributeValue="transform0" ...>...</attributes>
    <attributes featureName="translation" attributeValue="(-1, 3.6, 1)" ...>...</attributes>
    <attributes featureName="bboxCenter" attributeValue="0 0 0" ...>...</attributes>
    <attributes featureName="encapsulatedContent" attributeValue="scene/segment0.x3d" ...>...</attributes>
    ...
  </intermediateObjects>

  <intermediateObjects featureType="UserFunction" ...>
    <languages>JAVASCRIPT</languages>
    <attributes featureName="name" attributeValue="button0_CLICKED" ...>...</attributes>
    <attributes featureName="arguments" attributeValue="obj">...</attributes>
    <attributes featureName="customCode"
      attributeValue="var rot=transform0.getAttribute(&quot;rotation&quot;); ... var angle=parseFloat...">
    </attributes>
    ...
  </intermediateObjects>
</intermediate:IntermediateModel>

```

Listing 4: The merged intermediate model contains SSIML elements (button0), refined X3D code (transform0) and the JavaScript implementation code (button0_CLICKED).

the simplification of model merging. Whereas other approaches require new merging algorithms for each target-language considered, in our approach only one algorithm for merging IMs is needed. This reduces the merging complexity drastically.

A potential drawback of our approach results from the use of IDs for tracing elements through the different models and source codes. While the use of IDs is a common method for this problem, it will also result in the *pollution* of models and source code with IDs. In future work, we plan to experiment with tree-diff algorithms in order to trace elements without the need for IDs.

To synchronize different platform variants of code (e.g. X3D and VRML), the metamodels and grammars must be specified in detail. Implementing the complete X3D specification is also very extensive. For our round-trip scenario, we mainly cover the basic X3D *Interchange profile*.

For a detailed description of the proposed framework for round-trip engineering of 3D applications see e.g. [Lenk et al. 2012].

5 Conclusion

We presented a new approach to the structured development of 3D applications based on round-trip engineering. 3D and program code are generated from a common model specified in SSIML, a domain-specific language for modeling of 3D applications. Reverse engineering techniques serve to synchronize the SSIML model with manual edits of the 3D and program code, thereby supporting an iterative development process. To the best of our knowledge, the presented approach is first to apply round-trip engineering techniques to the development of 3D applications.

General benefits of the presented approach include that SSIML models can serve as communication aid for the involved developer groups and may be used for documentation purposes. Further, au-

tomatic code generation provides a basic code structure and frees the programmer from repetitive implementation tasks. As particular advantage of round-trip engineering, an iterative development process is supported through the automatic derivation of updated SSIML models from changes in the X3D and JavaScript code. The presented approach was designed in such a way, that it honors the concurrent development process involving 3D designers on the one side and programmers on the other. To this end, a novel method for the non-simultaneous synchronization of SSIML models, X3D and JavaScript code was developed.

In the current state of our implementation, SSIML models can be specified in a visual editor. Forward and reverse transformations, including the generation of code skeletons as well as the non-simultaneous merging of intermediate models, are implemented for X3D and JavaScript. Building on the X3DOM framework, functional 3D applications for the web using X3D and JavaScript can be developed.

In future work, the presented approach for round-trip engineering of 3D applications will be extended to further programming languages and multi-platform development. In particular, support for model-driven development of immersive Virtual Reality applications with SSIML will be investigated.

Acknowledgements

This research was supported by the Deutsche Forschungsgemeinschaft (DFG).

References

- ABAWI, D. F., DÖRNER, R., HALLER, M., AND ZAUNER, J. 2004. Efficient mixed reality application development. In *1st European Conference on Visual Media Production*.

- ANTKIEWICZ, M. 2007. Round-trip engineering using framework-specific modeling languages. In *OOPSLA Companion*, ACM, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds., 927–928.
- ASSMANN, U. 2003. Automatic roundtrip engineering. *Electr. Notes Theor. Comput. Sci.* 82, 5. Proceedings of the Fifth Workshop on Quantitative Aspects of Programming Languages (QAPL 2007).
- BEHR, J., ESCHLER, P., JUNG, Y., AND ZLLNER, M. 2009. X3dom: a dom-based html5/x3d integration model. In *Web3D*, ACM, S. N. Spencer, D. W. Fellner, J. Behr, and K. Walczak, Eds., 127–135.
- CONWAY, M., PAUSCH, R., GOSSWEILER, R., AND BURNETTE, T. 1994. Alice: a rapid prototyping system for building virtual environments. In *CHI Conference Companion*, ACM, C. Plaisant, Ed., 295–296.
- DACHSELT, R., HINZ, M., AND MEISSNER, K. 2002. Contigra: an xml-based architecture for component-oriented 3d applications. In *Proceedings of the seventh international conference on 3D Web technology*, ACM, New York, NY, USA, Web3D '02, 155–163.
- DALY, L., AND BRUTZMAN, D. 2008. X3d: extensible 3d graphics standard. In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*, ACM, New York, NY, USA, 1–6.
- EFFTINGE, S., AND VÖLTER, M. 2006. oAW xText: A framework for textual DSLs. In *Eclipsecon Summit Europe 2006*.
- FIGUEROA, P., GREEN, M., AND HOOVER, H. J. 2002. Intl: a description language for vr applications. In *Web3D '02: Proceedings of the seventh international conference on 3D Web technology*, ACM, New York, NY, USA, 53–58.
- HENRIKSSON, A., AND LARSSON, H. 2003. A definition of round-trip engineering. Tech. rep., Linköping University, Sweden.
- JONES, J. 2003. Abstract syntax tree implementation idioms. *Pattern Languages of Program Design*. Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003).
- KOLOVOS, D. S., PAIGE, R. F., AND POLACK, F. 2008. The epsilon transformation language. In *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zurich, Switzerland, July 1-2, 2008, Proceedings*, Springer, A. Vallecillo, J. Gray, and A. Pierantonio, Eds., vol. 5063 of *Lecture Notes in Computer Science*, 46–60.
- LENK, M., VITZTHUM, A., AND JUNG, B. 2012. Non-simultaneous round-trip engineering for 3D applications. In *Proceedings of the 2012 International Conference on Software Engineering Research & Practice, SERP 2012*.
- MACINTYRE, B., GANDY, M., DOW, S., AND BOLTER, J. D. 2004. Dart: a toolkit for rapid design exploration of augmented reality experiences. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, ACM, New York, NY, USA, 197–206.
- MILLER, J., AND MUKERJI, J. 2003. Mda guide version 1.0.1. Tech. rep., Object Management Group (OMG).
- PLEUSS, A., VITZTHUM, A., AND HUSSMANN, H. 2007. Integrating heterogeneous tools into model-centric development of interactive applications. In *MoDELS*, Springer, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735 of *Lecture Notes in Computer Science*, 241–255.
- SENDALL, S., AND KSTER, J. 2004. Taming model round-trip engineering. In *Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications)*.
- STEINBERG, D., BUDINSKY, F., PATERNOSTRO, M., AND MERKS, E. 2009. *EMF: Eclipse Modeling Framework*, 2. ed. Addison-Wesley, Boston, MA.
- VAN PAESSCHEN, E., DE MEUTER, W., AND D'HONDT, M. 2005. Selfsync: a dynamic round-trip engineering environment. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM, New York, NY, USA, OOPSLA '05, 146–147.
- VITZTHUM, A., AND JUNG, B. 2010. Iterative model driven VR and AR development with round trip engineering. In *Proc. SEARIS Workshop at the IEEE Virtual Reality 2010 Conference*, Shaker.
- VITZTHUM, A., AND PLEUSS, A. 2005. SSIML: Designing structure and application integration of 3D scenes. In *Proceedings of the tenth international conference on 3D Web technology*, ACM, New York, NY, USA, Web3D '05, 9–17.
- VITZTHUM, A. 2008. *Entwicklungsunterstützung für interaktive 3D-Anwendungen. Ein modellgetriebener Ansatz*. Dissertation, Ludwigs-Maximilians-University München. In German.

