Model-Driven Multi-Platform Development of 3D Applications with Round-Trip Engineering

Bernhard Jung, Matthias Lenk

Virtual Reality and Multimedia
TU Bergakademie Freiberg
Freiberg, Germany
{jung,lenk}@informatik.tu-freiberg.de

Arnd Vitzthum

University of Cooperative Education
Berufsakademie Sachsen
Dresden, Germany
arnd.vitzthum@ba-dresden.de

Abstract: While model-driven approaches are nowadays common-place in the development of many kinds of software, 3D applications are often still developed in an ad-hoc and code-centric manner. This state of affairs is somewhat surprising, as there are obvious benefits to a more structured 3D development process. E.g., model-based techniques could help to ensure the mutual consistency of the code bases produced by the heterogeneous development groups, i.e. 3D designers and programmers. Further, 3D applications are often developed for multiple platforms in different programming environments for which some support for synchronization during development iterations is desirable. This paper presents a model-driven approach for the structured development of multi-platform 3D applications based on round-trip engineering. Abstract models of the application are specified in SSIML, a DSL tailored for the development of 3D applications. In a forward phase, consistent 3D scene descriptions and program code are generated from the SSIML model. In a reverse phase, code refinements are abstracted and synchronized to result in an updated SSIML model. And so on in subsequent iterations. In particular, our approach supports the synchronization of multiple target platforms, such as WebGL-enabled web applications with JavaScript and immersive Virtual Reality software using VRML and C++.

1 Introduction

Interactive 3D applications including Virtual Reality (VR) and Augmented Reality (AR) play a central role in many domains, such as product visualization, entertainment, scientific visualization, training and education. So far, however, the application of model-driven development (MDD) approaches and visual modeling languages such as UML is by far not as common as in the development of other kinds of software. Arguably, this lack of acceptance of MDD approaches can be attributed to the specifics of the 3D application development process.

First, 3D development is an interdisciplinary process. Essentially, two groups of developers are involved who use completely different tools and terminologies: 3D content developers and programmers. Misunderstandings between the developer groups can lead to an inconsistent system implementation [VP05]. E. g., if the 3D content developer does not follow the conventions for the naming of 3D objects, the programmer cannot address these objects properly via program code. While MDD approaches may be instrumental in avoiding such misunderstandings, general purpose tools based e. g. on UML may be appropriate for programmers, but are certainly inadequate for the often "creatively oriented" 3D developers. More promising seem domain-specific languages (DSL) specifically geared towards 3D development.

Second, 3D applications are usually developed in a highly iterative fashion. Although generally desirable for all kinds of software, support for an iterative development process is particularly relevant in the 3D domain. *Round-trip engineering (RTE)* [Ant07, Aßm03, VPDMD05] is a model-driven software development methodology that combines forward engineering (model-to-code transformations) with reverse engineering (code-to-model transformations) and synchronizations between code and model to support iterative development. RTE has proven useful in the development of "conventional" software as exemplified by several existing integrated tools supporting the *simultaneous* editing and synchronization of UML diagrams and program code. However, due to the concurrent development process in conjunction with the preference for very different modeling/programming tools between 3D designers and programmers (and software designers), the use of a (yet to be developed) integrated tool for the various tasks seems not advisable for 3D development. Instead, an approach is preferable where the distinct developer groups each can employ their tools of choice.

Third, 3D applications are often implemented for multiple platforms, from ordinary PCs, over mobile devices, up to immersive Virtual Reality installations. A possible approach could be the use of cross-platform 3D engines. A disadvantage is that this requires the installation of a 3D engine at the user's site. In web environments, however, users may not have administrative rights to install the necessary plug-in. In highly specialized environments, such as CAVEs, a suitable version of the 3D engine may not be available. Therefore, 3D applications often need to be developed w.r.t. different programming environments, using different programming languages. During the iterative development cycle, cross-platform synchronization should be supported.

In this paper we describe a round-trip engineering approach for the model-driven, iterative development of multi-platform 3D applications. Section 2 introduces this approach at a conceptual level. In Section 3, a longer example of model-driven, round-trip development for 3D applications is given, with WebGL-enabled web browsers and immersive Virtual Reality as deployment platforms. Section 4 gives an overview of the implementation of our round-trip engineering process. We discuss our approach in Section 5 before we finally conclude in Section 6.

2 3D Development with Round-Trip Engineering

This section gives a conceptual overview of the the proposed round-trip engineering approach for multi-platform 3D development (see Figure 1). The involved developer groups are: software designers, who design an abstract model of the 3D application; 3D design-

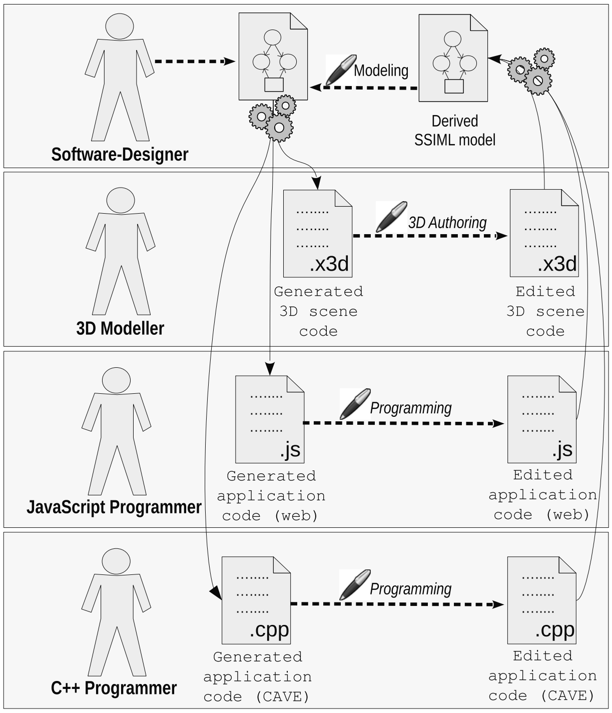

Figure 1: *Model-driven development process with roundtrip engineering, applied to multi-platform 3D application development.*

ers responsible for 3D modeling; and programmers who implement the application logic for the specific target platforms. As example target platforms for the 3D application, we assume WebGL-enabled web browsers with JavaScript as programming language and an immersive Virtual Reality CAVE programmed in C++. Details on the DSL used for application modeling, forward and reverse transformations, as well as other implementation aspects are given in later sections.

Development begins with the specification of an initial abstract model of the 3D application through the software designer. As modeling language, we use the *Scene Structure and Integration Modeling Language* (SSIML) [VP05], a DSL tailored for 3D applications. SSIML models comprise both an abstract description of the 3D scene and a specification of the application logic. Cross-references between the 3D scene description and the application components specify e. g. the events that may be triggered by user interaction with a 3D object and need to be handled by the application logic, or how the application classes modify the 3D scene at runtime. Note that as an iterative development process is assumed, it suffices that the initial SSIML model specifies only a rough first version of the application that can be refined in later iterations.

During the forward engineering phase of the development process, the initial SSIML model is transformed to JavaScript, C++ and X3D code skeletons. The 3D designers' task is to model the individual 3D objects and their composition in a single X3D scene while programmers elaborate the auto-generated JavaScript and C++ code. In the proposed development process, 3D designers and programmers may work concurrently, where both are using different tools appropriate to their task. E. g. 3D designers could model the 3D objects using tools like 3DS Max or Blender, while the programmers use an IDE or a simple code editor.

However, as software development is a non-trivial process, the implementation of 3D and program code may be inconsistent with the current SSIML model in various ways: The implementation could be incomplete or incorrectly reflect the model, e. g. by using different names for 3D objects, application classes or their attributes. Moreover, the implementation may also add parts to the application that may reasonably be reflected in future versions of the SSIML model.

In order to consolidate the various artifacts, the implementation code is then reverse engineered and synchronized with the current SSIML model. For each inconsistency between the SSIML model and the current state of the implementation it has to be decided whether the implementation is incorrect or incomplete or whether the implementation improves the overall application in some way and should therefore be reflected in the next version of the SSIML model. E. g., on the one hand, say, the 3D designer may not care too much about attribute names as only the visual appearance of the 3D objects is usually important for his task. Here, the mismatch should be considered as incorrectness of the implementation. On the other hand, e. g., a programmer may feel that an attribute name in the SSIML model is inconsistent with the conventions of her programming language and uses a different attribute name instead. Here, the changed attribute name may be adopted in the next iteration of the SSIML model. Note that the synchronization step can occur ”on-demand”, at any time of the development. I. e. it is not necessary e. g. that the web application developers wait for the CAVE developers to complete their implementation.

Once the implementation is synchronized with the SSIML model, possibly resulting in updates to the model, the software developer may now further expand the SSIML model. This completes the first cycle of the round-trip engineering process. From the updated SSIML model, again, code is generated in a forward step. Of course, during the forward step it is ensured that the newly generated code still contains the implementation details of the previous iteration, i. e. no code is lost during re-generation. This round-trip process may be repeated for several iterations, until a final version of the 3D application is reached.

3 Example: 3D Development for Web and CAVE

We introduce a small example to better illustrate the proposed 3D development process. The 3D scene is composed of the following 3D objects: A car chassis (Figure 2(a)), separated windows (Figure 2(b)) and different rim types (Figure 2(c)). The application shall provide an interactive part to configure the car, i. e. changing the car’s color, replacing the rims and toggling the visibility of the windows. Interaction is achieved by directly clicking on the respective 3D objects within the 3D scene or by using external GUI elements.

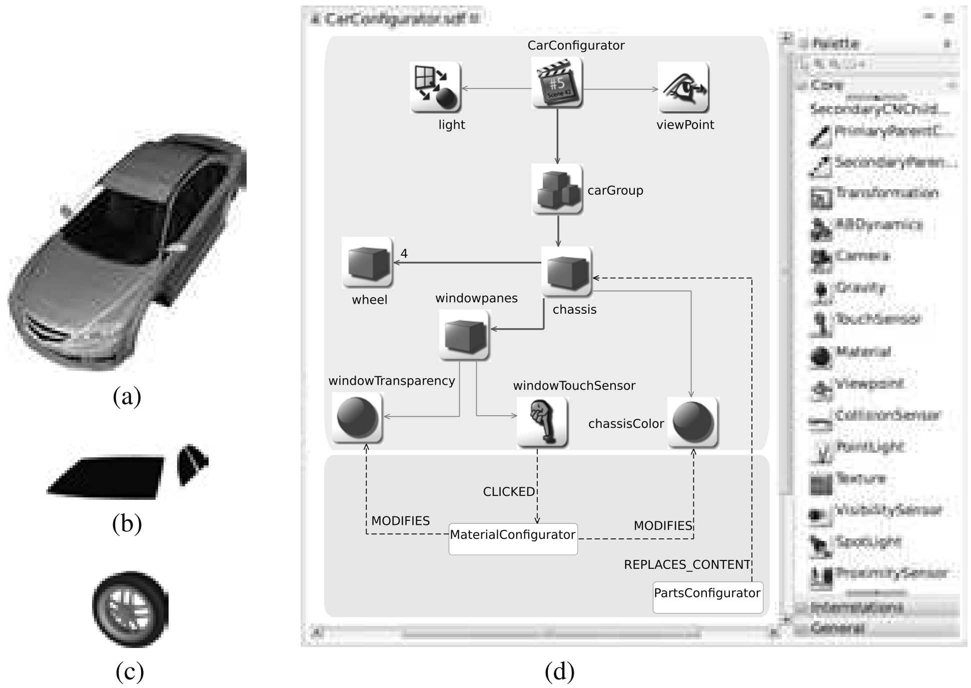

Figure 2: *Left: (a) The stand-alone car chassis with (b) separated windows, (c) one of several wheel types. Right: The SSIML model for the car configuration application in our graphical model editor. Scene model elements are located on the blue (upper) area while elements from the SSIML interrelationship model are positioned on the yellow (lower) area.*

3.1 Modeling 3D applications with SSIML

In the proposed 3D development process, development begins with the specification of an abstract model through software designers. The application model is specified in the DSL SSIML [VP05]. A SSIML model is composed of a *scene model* which allows for modeling abstract 3D scene graphs and an *interrelationship model* to create associations of scene graph elements with application components. The 3D scene graph is composed of parent and child nodes, with some important ones being the *scene root* node, *group* nodes to structure the scene graph, *object* nodes which represent a specific geometric 3D object (e. g., a car chassis) and *attribute* nodes, which can be used e. g. to specify a *transformation* or *material* of an object. Group and object nodes can act as child and parent nodes for other nodes, whereas attribute nodes have no further children. Special attribute nodes, such as *sensor* nodes, are used to trigger events within the 3D scene to notify application components within the interrelationship model. E. g., when a SSIML object node is connected to a touch sensor, this sensor attribute can notify an associated application class about an event, e. g. a mouse click.

```
<!-- id=938c3fb8-b229-40d2-9044-c5a26727bc09 -->
<X3D version="3.0" profile="Interaction">
  <!-- id=a1b5540e-4f9f-4674-8e01-1877a37213b1 -->
  <Scene>
    <!-- id=cb7960db-0141-4f60-ae3d-552ab613decd -->
    <WorldInfo title="CarConfigurator" info="" DEF="..." ></WorldInfo>
    <Transform DEF="generatedTransform1" translation="0 0 0" ...>
      <Group DEF="carGroup">
        <Transform DEF="generatedTransform2" translation="0 0 0" ...>
          <Inline DEF="chassis" url="chassis.x3d"></Inline>
          <Transform DEF="generatedTransform4" ...>
            <Inline DEF="wheel_0" url="wheels.x3d"></Inline>
          </Transform>
          <Transform DEF="generatedTransform5" ...>
            <Inline DEF="wheel_1" USE="wheel_0"></Inline>
          </Transform>
          ...
        </Transform>
      </Group>
    </Transform>
    <DirectionalLight DEF="light" color="1 1 1" ...></DirectionalLight>
    <Viewpoint DEF="viewPoint" position="0 0 10" ...></Viewpoint>
  </Scene>
</X3D>
```

Listing 1: *X3D code skeleton that has been generated from the SSIML model (Figure 2 (d)). Inline nodes are used to include further 3D geometries, e. g. the chassis object. Some IDs and node attributes have been removed.*

The car configuration application can be modelled using SSIML as depicted in Figure 2 (d). Scene model elements which will mainly result in the X3D code are shown on a blue (upper) area, for better illustration. The scene root node has a light and viewpoint attribute and contains the group node "carGroup". This group node contains a "chassis" object which in turn contains four "wheel" objects and the "windowpanes" object. Elements of the SSIML interrelationship model will later be translated to application code and are displayed on a yellow (lower) area. The application class "MaterialConfigurator" is associated with the respective material attributes of the chassis and windowpanes in order to change their color and transparency values. The touch sensor attribute "windowTouchSensor" is connected to the windowpanes object and triggers a "CLICKED" event to the application class when the user clicks on the windowpanes. The second application class "PartsConfigurator" can access the chassis object in order to replace its content i. e. the four wheels with a different kind of wheels.

3.2 Developing the 3D Web Application

Web deployment of the car configuration tool makes use of X3D as scene description language and JavaScript as programming language. The *X3DOM* framework is used for integration of X3D content into HTML pages [BEJZ09]. Modern, WebGL-enabled web browsers can display X3DOM applications natively, without the need for installing a plug-in.

```
function MaterialConfigurator( ) {
  var chassisColor = document
    .getElementById( 'chassis__chassisColor' );
  var windowTransparency = document
    .getElementById( 'windowpanes__windowTransparency' );
  this.windowpanes_CLICKED = function( obj ) {
  };
  //! Insert further application code below !//

  //! Insert further application code above !//
}

function PartsConfigurator( ) {

  var chassis = return document.getElementById( 'chassis' );
  ...
}

function init( ) {
  var materialConfigurator = new MaterialConfigurator( );
  var partsConfigurator = new PartsConfigurator( );
  document.getElementById('windowpanes').addEventListener("click",
      materialConfigurator.windowpanes_CLICKED);
  ...
}
```

Listing 2: *Generated JavaScript code skeleton with functions and member variables to access the 3D scene. Comments, where IDs are stored, have been removed.*

3.2.1 Generation of X3D code from SSIML models

SSIML *scene model* elements (Figure 2, upper blue area) 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 carGroup, 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 (Listing 1).

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. Using the X3DOM framework, the generated and refined X3D scene can be included in HTML documents and rendered in modern web browsers (although no functionalities are implemented yet for the application).

3.2.2 Generation of JavaScript code from SSIML models

Similar to X3D code, a corresponding JavaScript code skeleton (Listing 2) is generated to access the 3D scene. The generated JavaScript functions emulate the class-oriented structure of programming languages such as Java or C++, in order to enable consistent multi-

```
class MaterialConfigurator: public ssiml::ApplicationClass {
  public:
    MaterialConfigurator(ssiml::Scene *instance) :
      ApplicationClass(instance),
      chassisColor((SoVRMLMaterial *)scene->getNodeByName("chassisColor")),
      windowTransparency((SoVRMLMaterial *)scene->getNodeByName("..."))
    {}
    bool windowTouchSensor_CLICKED(ssiml::EventHandlerArguments *ea) {
      return false;
    }
  private:
    SoVRMLMaterial *chassisColor;
    SoVRMLMaterial *windowTransparency;
};

// ...

class CarConfigurator : public ssiml::Plugin {
  public:
    CarConfigurator(xs::Host & host, std::string pluginName, ...) :
      Plugin(host, pluginName, sceneFile, enableSmoothing, highlightType) {
        registerApplicationClass<PartsConfigurator>();
        registerApplicationClass<MaterialConfigurator>();

        registerEventHandler<MaterialConfigurator>(ssiml::EventType::CLICKED,
            "windowpanes", &MaterialConfigurator::windowTouchSensor_CLICKED);
    }
  // ...
};
```

Listing 3: *Generated C++ code skeleton with the MaterialConfigurator class and the CarConfigurator scene class. Methods and member variables are generated to access the 3D scene. Comments, where IDs are stored, have been removed.*

platform adaption. JavaScript code fragments are related to the original SSIML model of Figure 2 in the following way: The SSIML application components MaterialConfigurator and PartsConfigurator are translated to JavaScript functions of the same names which create suitable JavaScript objects. The MaterialConfigurator's action relationships to the original SSIML attributes windowTransparency and chassisColor result in member variables of the JavaScript object. These two variables address X3D material nodes which can be used to modify the appearance of the windows and the chassis, respectively. To realize touch sensors with X3DOM, appropriate event listeners (conforming to the HTML event model) are attached to the respective X3D nodes [BEJZ09]. This event handling mechanism is common on many platforms and therefore can easily be adapted. For the windowpanes object, which is connected to a touch sensor in the SSIML model, a corresponding event listener is generated in the init() function. The CLICKED event from the SSIML example is mapped to an HTML click event. Further application logic, e. g. to set the transparency value of the material, will be manually programmed by filling in code stubs or in additional functions. Like generated X3D elements, generated JavaScript elements are also assigned IDs for tracking. 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. windowpanes), the attached sensor and the relationships which connect the related objects (Listing 2).

3.3 Development of CAVE application

The second platform supported by our round-trip environment is an ultra high-resolution CAVE consisting of 25 projectors (24 full HD plus 1 SXGA+) that offers a much more immersive experience than the web platform. The user can interactively configure the real sized car by using GUI elements on an iPad and by using a flystick for 3D interaction. The 3D scene (Section 3.2.1) is converted from X3D to the VRML format. Functionalities to load, access and display the 3D scene or to realize the event handling mechanism are implemented w.r.t. corresponding libraries of a proprietary CAVE framework. The structure of the generated C++ application (Listing 3) is similar to the JavaScript code (Section 3.2.2). SSIML application classes result in respective C++ classes. Additionally, the class CarConfigurator is generated, in order to initialize the application and to register events. Member variables and stubs for all methods required for the application, such as the "windows_touch" event handler, are also generated in the forward step.

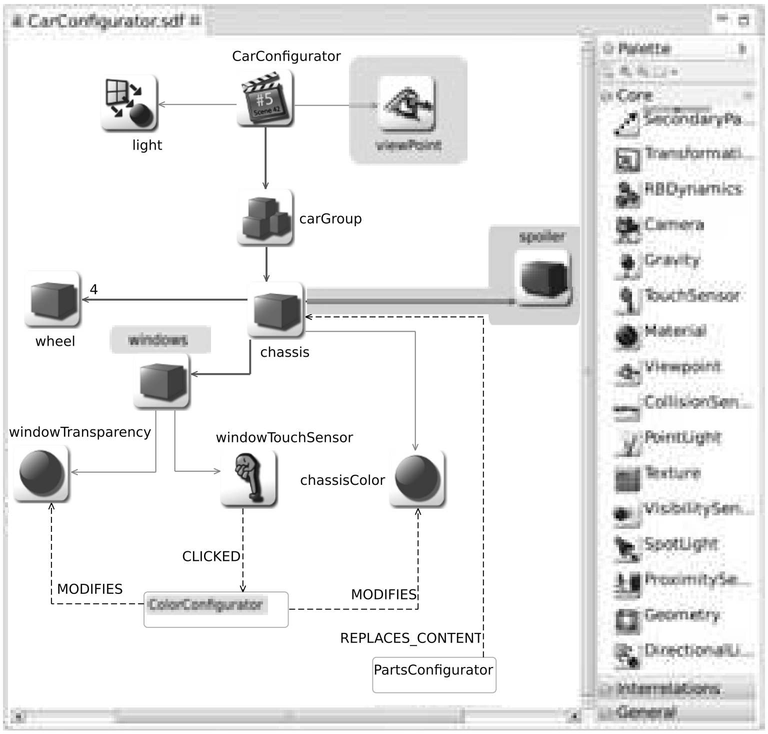

Figure 3: *The second iteration SSIML model for the car configuration application. The "windowpanes" object has been renamed to "windows" and the application class "MaterialConfigurator" to "ColorConfigurator". Furthermore, a "spoiler" object has been added to the new version of the SSIML model.*

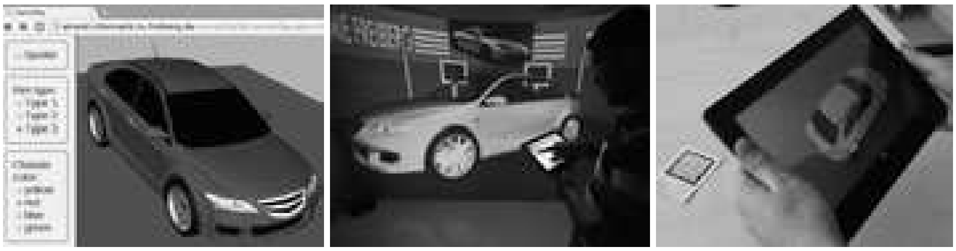
(a) (b) (c)

Figure 4: *(a) 3D application in Web Browser; (b) CAVE-version of the application; (c) Prototype of an AR port of the application.*

3.4 Multi-Platform Round-Trip Engineering

During the refinement of the generated code skeletons, model relevant changes to the code can result in an inconsistent 3D application. In order to detect these inconsistencies and to derive an updated SSIML model, the different code bases are synchronized in the reverse phase of the RTE process. For our running example we assume the following three model-relevant changes to the code, which represent the basic operations *update*, *insert* and *delete*: First, 3D designers rename the 3D object “windowpanes” to “windows”. Since JavaScript and C++ programmers rely on the original naming, the windowpanes object is no longer accessible from the program code, which results in an inconsistent application that might crash. Second, 3D designers, having the CAVE application in mind, add a high-detail background to the 3D scene which is however not suitable for the web application due to the smaller graphics processing power in web browsers. And third, 3D designers forget to define the required light source (which could be interpreted, without further information, as intentional deletion). Model relevant changes, of course, may stem from JavaScript programmers, too. For instance, a JavaScript developer renames the class element “MaterialConfigurator” to “ColorConfigurator” since in her opinion only color values are addressed and modified. When transforming the code bases back to the model in the reverse phase, the software designer, ideally in a group decision process with the other developers, may accept or reject such changes. W.r.t. the example, the deletion of the viewpoint (by not defining it) and the addition of the high-detail background are rejected[1] while renaming the windows 3D object and the application class MaterialConfigurator is accepted for inclusion in the next iteration of the SSIML model. Software engineers can now apply further changes to the consolidated SSIML model (Figure 3), e. g. by adding a spoiler object. During the next forward iteration, manually developed source code and changes from the SSIML model will be re-generated to a consistent application. Figure 4 shows the final application, in the web[2] and CAVE variants, as well as a yet experimental AR application on a tablet computer.

[1]CAVE developers may still programmatically add a high-detail background, but the background object will not be part of the platform-independent SSIML model.

[2]Available at: http://elrond.informatik.tu-freiberg.de/roundtrip3d/carconfigurator.xhtml

4 Implementation overview

In this section we briefly describe the implementation of our Eclipse-based Round-trip environment which includes model to code and code to model transformations, model merging and conflict handling. A detailed description of the participating models and transformations is given in [LSVJ12].

4.1 Transformations between model and code

Round-trip engineering combines model-driven forward engineering with reverse engineering as well as a synchronization phase (see Section 4.2). In our implementation, the overall, complex conversion between model and code is realized through a sequence of simpler transformations (see Figure 5). A SSIML model is first converted into its intermediate representation by *model to model* transformations (M2M). The newly created intermediate model (IM) serves as a persistent storage for data and meta data from the complete application, including all model and code artifacts. SSIML to IM transformations (forward phase) and IM to SSIML transformations (reverse phase) are performed with Java using EMF's reflection API [SBPM09]. In the next step *abstract syntax trees* (ASTs) are created from the IM for each platform. We use the hybrid, rule-based transformation language ETL [KPP08] to achieve complex mappings between SSIML elements (in their intermediate representation) and AST elements of the target platform during the forward phase and the reverse phase, respectively. Code serializers and parsers to convert between the ASTs and their concrete syntactic representation are generated through Xtext [EV06] based on a grammar and a meta model of the target platform. Since serializers and parsers only need to recognize model relevant language artifacts, it is not necessary to semantically distinguish between every construct of the target language. Therefore, it suffices that the meta models and grammars cover relevant subsets of the target platform [Jon03]. Due to the similar structure of the generated JavaScript and C++ files, parts of the meta models and IM and ASTs conversions can be re-used.

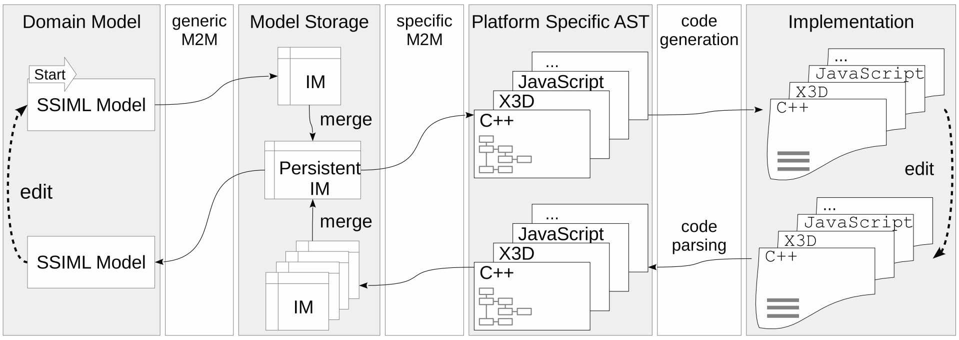

Figure 5: *RTE implementation with participating models and transformations.*

4.2 Synchronization: Model merging and conflict handling

3D modelers and programmers elaborate the generated code bases with different tools appropriate to their task. Therefore, modifications to the code cannot be tracked in real time and thus need to be synchronized with the model in a *non-simultaneous* manner. In order to avoid the need for different merge algorithms for each platform, synchronization is performed on the intermediate layer [LVJ12]. All intermediate models, whether generated from SSIML models in the forward phase or derived from code in the reverse phase, are merged into the persistent intermediate model. Conflicts arise when an element has been modified in different code bases, e. g. an application class has been renamed both in the C++ code and in the JavaScript code. Conflicts and changes to model-relevant elements in the code, such the renaming of the windows object in the X3D code, are automatically detected during merging. Conflict resolution and decision making whether model-relevant changes at the code level should be applied to the next iteration of the SSIML model occurs in an interactive fashion, e.g. by the software engineer in coordination with the different developer teams. When no conflicts are detected during merging, a consistent version of the 3D application has been developed.

5 Discussion

Many existing RTE tools, such as *UML LAB* or *Together*, offer support for software development with focus on *simultaneous* synchronizations of model and code. UML LAB uses templates to synchronize between uniform modeling and general-purpose programming languages, e. g. UML2 class diagrams and Java or C# code. In contrast, the proposed RTE process addresses the special domain of 3D development. 3D and application code is generated from a graphical DSL and elaborated by different developer groups, each using their own specialized development tools. Synchronization of the various artifacts occurs ”on-demand”, in a *non-simultaneous* fashion. The concurrent work style also improves on the typically sequential 3D development process where 3D modeling often strictly precedes the implementation of the application logic.

Our multi-tiered RTE implementation achieves complex model-to-code transformations through a sequence of simpler transformations. In this way, task specific transformation languages can be employed and the process of defining transformations becomes more manageable overall. Furthermore, subdividing the model-to-code transformation chain simplifies the extension to further platforms. Due to the similar structure of the generated code for the different platforms, meta models and IM-to-AST transformations can be reused to some extent. In addition, SSIML-to-IM transformations can be completely reused. A single model-merging algorithm is used instead of *weaving* code of different languages.

Besides SSIML, several other approaches to the structured development of 3D applications have been proposed. The Interaction Techniques Markup Language (InTML) integrates 3D interaction techniques into VR applications, by specifying an abstract XML-based model that can be executed in a runtime environment [FGH02]. However, in this approach the functionality of the program is limited to the scope of its modeling language and multi-platform development depends on the port of the runtime. CONTIGRA [DHM02] aims at high-level and multi-disciplinary development processes of 3D components, that can be transformed to 3D applications. However, a reverse step to achieve iterative development is not possible. APRIL [LS05] can be used to create textual models of AR presentations. These models can be transformed to code for two platforms, while a reverse phase is not possible. SSIML can also be used for modeling AR applications [VH06]. In current work, we are extending our RTE framework to Android-based AR applications (Figure 4(c)).

In the current state of our implementation, a complete round-trip is possible for 3D web applications. I. e. changes to the SSIML model, X3D, and JavaScript code can be fully synchronized with each other and also be applied to the C++ CAVE code. The reverse engineering step for C++ code is currently only partially implemented, such that changes to the C++ code cannot yet be merged into other artifacts. Layout and formatting information of model and source code is not preserved, except in protected regions in JavaScript and C++. Formatting functionalities are provided by programming tools and our model editor.

6 Conclusion

We presented a round-trip engineering approach supporting the multi-platform development of 3D applications. From a common model specified in SSIML, a domain specific language for modeling 3D applications, code skeletons for programs running in modern web browsers and a Virtual Reality CAVE are generated. In a reverse phase, code level changes are synchronized with the SSIML model to support an iterative and concurrent development process. A multi-tiered transformation pipeline is used to implement the RTE process. Splitting up the forward and reverse phases into smaller steps simplifies the extension of our RTE process to further platforms, as only platform-specific transformations need to be implemented for each additional platform. Similarly, as merging occurs between the platform-independent intermediate models, no additional merge algorithm must be implemented when further platforms are considered. The multi-tiered RTE implementation thus scales well with the number of target platforms in 3D application development.

Acknowledgements

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

References

[Ant07] Michal Antkiewicz. Round-trip engineering using framework-specific modeling languages. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy

L. Steele Jr., editors, *OOPSLA Companion*, pages 927–928. ACM, 2007.

[Aßm03] Uwe Aßmann. Automatic Roundtrip Engineering. *Electr. Notes Theor. Comput. Sci.*, 82(5), 2003. Proceedings of the Fifth Workshop on Quantitative Aspects of Programming Languages (QAPL 2007).

[BEJZ09] Johannes Behr, Peter Eschler, Yvonne Jung, and Michael Zöllner. X3DOM: a DOM-based HTML5/X3D integration model. In Stephen N. Spencer, Dieter W. Fellner, Johannes Behr, and Krzysztof Walczak, editors, *Web3D*, pages 127–135. ACM, 2009.

[DHM02] Raimund Dachselt, Michael Hinz, and Klaus Meissner. Contigra: an XML-based architecture for component-oriented 3D applications. In *Proceedings of the seventh international conference on 3D Web technology*, Web3D '02, pages 155–163. ACM, 2002.

[EV06] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Eclipsecon Summit Europe 2006*, November 2006.

[FGH02] Pablo Figueroa, Mark Green, and H. James Hoover. InTml: a description language for VR applications. In *Web3D '02: Proceedings of the seventh international conference on 3D Web technology*, pages 53–58, New York, NY, USA, 2002. ACM.

[Jon03] Joel Jones. Abstract Syntax Tree Implementation Idioms. *Pattern Languages of Program Design*, 2003. Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003).

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zrich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.

[LS05] Florian Ledermann and Dieter Schmalstieg. APRIL A High-Level Framework for Creating Augmented Reality Presentations. In *Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality*, VR '05, pages 187–194. IEEE, 2005.

[LSVJ12] Matthias Lenk, Christian Schlegel, Arnd Vitzthum, and Bernhard Jung. Round-trip Engineering for 3D Applications: Models and Transformations. Preprint 6/2012, Faculty of Mathematics and Informatics, TU Bergakademie Freiberg, 2012.

[LVJ12] Matthias Lenk, Arnd Vitzthum, and Bernhard Jung. Non-Simultaneous Round-Trip Engineering for 3D Applications. In *Proceedings of the 2012 International Conference on Software Engineering Research & Practice, SERP 2012*, 2012.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[VH06] Arnd Vitzthum and Heinrich Hussmann. Modeling Augmented Reality User Interfaces with SSIML/AR. *Journal of Multimedia*, 1(3):13–22, 2006.

[VP05] Arnd Vitzthum and Andreas Pleuß. SSIML: Designing structure and application integration of 3D scenes. In *Proceedings of the tenth international conference on 3D Web technology*, Web3D '05, pages 9–17, New York, NY, USA, 2005. ACM.

[VPDMD05] Ellen Van Paesschen, Wolfgang De Meuter, and Maja D'Hondt. SelfSync: a dynamic round-trip engineering environment. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 146–147, New York, NY, USA, 2005. ACM.