# Entwicklungsunterstützung für interaktive 3D-Anwendungen

## Ein modellgetriebener Ansatz

**Arnd Vitzthum**


München 2008

# Entwicklungsunterstützung für interaktive 3D-Anwendungen

## Ein modellgetriebener Ansatz

**Arnd Vitzthum**

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Arnd Vitzthum
aus Dresden

am 2. Juni 2008

Erstgutachter: Prof. Dr. Heinrich Hußmann
Zweitgutachter: Prof. Dr. Morten Fjeld (TU Chalmers, Schweden)
Tag der mündlichen Prüfung: 7. Juli 2008

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Entwicklung interaktiver 3D-Anwendungen. Interaktive 3D-Grafik wird heutzutage in den verschiedensten Domänen eingesetzt, z. B. im e-Commerce-, Unterhaltungs- und Ausbildungsbereich.

Dennoch stellt die Entwicklung einer umfangreicheren 3D-Anwendung nach wie vor eine Herausforderung dar. Programmcode und 3D-Inhalte werden i. d. R. von verschiedenen Entwicklern erstellt, die unterschiedliches Fachwissen besitzen und mit völlig verschiedenartigen Werkzeugen arbeiten. Diese Situation führt häufig zu Problemen bei der Integration der erstellten Anwendungskomponenten in ein komplexes interaktives 3D-Gesamtsystem. So können etwa Inkonsistenzen auftreten, die einen korrekten Zugriff des Programms auf die 3D-Inhalte zur Laufzeit verhindern. Zudem fehlen Konzepte und Werkzeuge zur Unterstützung einer strukturierten interdisziplinären 3D-Entwicklung.

In der vorliegenden Arbeit wird ein neuartiger Lösungsansatz für die genannten Probleme vorgestellt. Es handelt sich dabei um eine Familie domänenspezifischer Sprachen, die für einen Einsatz in der Entwurfsphase vor der Implementierung im 3D-Entwicklungsprozess konzipiert wurden. Basissprache der Familie, die durch weitere Sprachkomponenten ergänzt wird, ist die Scene Structure and Integration Modelling Language (kurz SSIML). Mittels visueller Modelle lassen sich – werkzeuggestützt – Verknüpfungen zwischen Programmkomponenten und 3D-Inhalten spezifizieren. Durch die automatische Erzeugung von Codeskeletten aus einem Modell, die sowohl dem 3D-Designer als auch dem Programmierer als zu vervollständigende Vorlagen dienen, kann die Konsistenz zwischen den einzelnen Anwendungsbestandteilen sichergestellt werden. Neben der Verknüpfung von Programmcode und 3D-Inhalten sind die Strukturierung und Modularisierung von 3D-Inhalten, die aufgabenorientierte 3D-Visualisierung, 3D-Verhalten und Animation und Augmented Realitiy-Anwendungen weitere wichtige Aspekte, die durch die Mitglieder der SSIML-Sprachfamilie abgedeckt werden. Außerdem wird in der vorliegenden Arbeit ein 3D-Entwicklungsprozess skizziert, der einen sinnvollen Einbezug der vorgestellten Konzepte und Werkzeuge erlaubt.

# Summary

Interactive 3D graphics are currently applied in many domains, such as e-Commerce, entertainment and education. However, the creation of complex 3D applications still poses a challenge to the developers. Application parts, in particular program code and 3D content, are usually created by different developers having different knowledge and working with completely different tools. In addition, there is a lack of concepts and tools which support a structured interdisciplinary 3D application development process. Frequently, this situation leads to difficulties during the integration of the developed software parts into a working overall system. For example, the manipulation of 3D objects during runtime can fail due to inconsistencies between the application components.

In this thesis, a novel approach is presented to address the mentioned problems. A family of domain specific languages is introduced which is intended to support the design phase of the 3D development process. The base language of the family is the Scene Structure and Integration Modelling Language (SSIML). SSIML enables the specification of interrelationships between program components and 3D content components using visual modelling tools. In order to ensure consistency between different application parts and to achieve a seamless transition to the implementation level, SSIML models can be translated into code skeletons automatically. Two kinds of code skeletons can be generated from a single model: 3D templates and program code skeletons. While 3D templates are completed by 3D designers with appropriate 3D authoring tools, gaps in the program code are filled out by programmers. Further main aspects covered by members of the SSIML language family are 3D object modularization, task-focused 3D visualization, 3D behaviour and animation and Augmented Reality (AR) applications. Moreover, the integration of the proposed concepts and tools into an overall 3D development process is outlined in this dissertation.

# Kapitel 1

# Einführung

Benutzungsschnittstellen auf der Grundlage interaktiver 3D-Grafik werden heutzutage in vielen Anwendungsbereichen eingesetzt, bspw. im Bereich e-Commerce bei Produktvisualisierungen und -konfiguratoren, im Entertainmentbereich bei Spielen, in der Informationsvisualisierung, bei Trainingssimulatoren und im Servicetechnikbereich in Form von interaktiven Produktmanualen. Eine spezielle Klasse interaktiver 3D-Anwendungen stellen Augmented Reality (AR)-Anwendungen dar, die derzeit Gegenstand zahlreicher Forschungsprojekte sind und einen der Schwerpunkte dieser Arbeit bilden.

Allerdings stellt die Realisierung komplexerer interaktiver Anwendungen mit 3D-Benutzungsschnittstellen (nachfolgend kurz 3D-Anwendungen genannt) noch immer hohe Anforderungen an die Entwickler. 3D-Inhalte und Animationen müssen spezifiziert, das Verhalten der 3D-Objekte und die Anwendungslogik programmiert und schließlich müssen alle Komponenten in ein funktionierendes Gesamtsystem integriert werden. Jim Blinn, ein Pionier im Bereich der Computergrafik, der vielfach für seine Leistungen ausgezeichnet wurde, formuliert dieses Problem in seinem Artikel *Ten More Unsolved Problems in Computer Graphics* folgendermaßen:

> *Systems Integration: This is the problem of keeping all the balls in the air at once, that is, how to use all the tricks in one production. Just because one researcher can do cloth, one can do faces, and one can do hair doesn't mean that all animation systems can suddenly put them all together.* [Bli98]

Die erfolgreiche Integration aller Software-Bestandteile in ein Gesamtsystem erfordert in größeren Projekten eine enge Zusammenarbeit von unterschiedlichen Entwicklern, wie Designern und Programmierern, die völlig verschiedenartige Entwicklungswerkzeuge, wie 3D-Autorenwerkzeuge und Programmierumgebungen, verwenden und die zudem unterschiedliches Fachwissen besitzen.

Gerade die interdisziplinäre Zusammenarbeit zwischen Designern (nachfolgend auch als *3D-Designer* oder *3D-Entwickler* bezeichnet), welche 3D-Inhalte erstellen, und Programmierern stellt eine große Herausforderung dar, wie u. a. aus einer im Rahmen der vorliegenden Arbeit durchgeführten Befragung von Experten aus dem Bereich der 3D-Anwendungsentwicklung hervorging (eine detaillierte Beschreibung des Experteninterviews

ist in Abschnitt 11.1 zu finden). Ein häufig fehlendes, gemeinsames konzeptuelles Modell der am Entwicklungsprozess beteiligten Personen zieht oft Kommunikationsprobleme nach sich (vgl. a. [GPRR00]). Folglich kann es zu Inkonsistenzen zwischen den entwickelten Systembestandteilen kommen. Marrin et al. [MMSC97] verdeutlichen dieses Problem am Beispiel des 3D-APIs (API: *Application Programming Interface*) Java3D [J3D]:

> *Java3D will use traditional programming techniques to create and control a 3D presentation. Using Java3D, a programmer would [...] read static objects from a file. Code to control the behavior of these objects would then be written. This is a very powerful model of 3D content creation, but it is also extremely tedious. It requires the programmer to understand the structure of any object imported, in order to apply behavioral control in the proper places. This would require a programmer with sophisticated graphic design abilities, a graphic designer with highly technical programming skills, or extremely tight communication between a programmer and a graphic designer.* [MMSC97]

Wie sich u. a. ebenfalls aus der bereits erwähnten Expertenbefragung ergab, ist der Prozess der Entwicklung von 3D-Applikationen oft weniger stark strukturiert und basiert teilweise nur auf informellen Absprachen, die etwa in Meetings getroffen werden. Konzepte und Werkzeuge, die eine formalere Spezifikation der 3D-Anwendung im Vorfeld der Implementierung ermöglichen, fehlen ebenso wie Werkzeuge zur Prüfung der Einhaltung einer Spezifikation und Werkzeuge zur Sicherung der Konsistenz zwischen den einzelnen Codebestandteilen. Hält sich bspw. der Designer bei der Benennung von 3D-Objekten nicht an vorgegebene Namenskonventionen, ist es dem Programmierer nicht möglich, diese Objekte im Code zu adressieren, woraus eine zeitaufwendige Fehlersuche resultieren kann. So könnten z. B. Werkzeuge, die eine automatische Prüfung der Einhaltung von Namenskonventionen erlauben, zur frühzeitigen Vermeidung von Fehlbenennungen beitragen.

Ein weiterer kostentreibender Faktor bei der Entwicklung von 3D-Anwendungen ist die unzureichende Parallelisierung der Arbeitsabläufe. So kann mit der Programmierung z. T. erst begonnen werden, wenn die vom Designer erstellten Inhalte vorliegen [Pol07]. Verbesserungspotenzial würden hier Ansätze zur stärkeren Modularisierung von Geometrien, Animationen und Programmkomponenten bieten. Somit könnte ein Programmierer etwa bei der Erstellung einer Programmablaufsteuerung zu Testzwecken in zunehmendem Maße auf eine entsprechende Bibliothek wiederverwendbarer Grundbausteine und Dummy-Objekte zurückgreifen, bevor die finalen 3D-Inhalte integriert werden.

Ferner ist nicht zu vernachlässigen, dass die Erstellung von 3D-Objekten, ihre Orientierung und Positionierung im dreidimensionalen Raum und ihre Kompositionen zu 3D-Szenen i. d. R. weitaus weniger trivial sind als die Erstellung, Ausrichtung und Komposition der grafischen Elemente von 2D-Benutzungsoberflächen [BKLP04], was nicht zuletzt in der höheren Anzahl an Freiheitsgraden begründet liegt (drei rotatorische und drei translatorische Freiheitsgrade in 3D im Vergleich zu einem rotatorischen und zwei translatorischen Freiheitsgraden in 2D). Als adäquate Hilfsmittel zur Erstellung der 3D-Inhalte dienen 3D-Autorenwerkzeuge.

Zusätzliche Anforderungen müssen bei der Entwicklung von AR-Anwendungen berücksichtigt werden. AR bereichert die reale Welt durch das Hinzufügen virtueller Informationen, die in das Sichtfeld des Benutzers eingeblendet werden. Die Integration virtueller Objekte in die reale Welt kann z. B. über so genannte See-Through-Head-Mounted-Displays (See-Through-HMDs) erreicht werden. Reale und virtuelle Objekte müssen miteinander derart in Beziehung gesetzt werden, dass virtuelle Informationen an der korrekten Position innerhalb der realen Welt dargestellt werden [Azu97]. In die Entwicklung von AR-Anwendungen und speziell AR-Benutzungsschnittstellen müssen also neben virtuellen 3D-Inhalten auch reale Objekte einbezogen werden.

## 1.1 Zielstellungen der Arbeit

Ziel dieser Arbeit ist es, Lösungsansätze für die eingangs skizzierten Probleme bei der Entwicklung von 3D-Anwendungen anzubieten und damit die Erstellung interaktiver 3D-Anwendungen zu erleichtern und Kosten zu reduzieren. Im Einzelnen sollen Konzepte und Hilfsmittel bereitgestellt werden für

- eine effizientere Kommunikation zwischen 3D-Designern und Programmierern,
- eine Verbesserung der Strukturierung des 3D-Entwicklungsprozesses,
- die Unterstützung einer fehlerfreien Verknüpfung von 3D-Inhalten und Programmcode,
- die Modularisierung von Anwendungsbestandteilen wie Animationsbeschreibungen, Ablaufsteuerung und autonomen 3D-Objekten mit eigenem Verhalten,
- eine stärkere Parallelisierung der Arbeitsabläufe Design und Programmierung und
- die Unterstützung der Entwicklung von AR-Anwendungen durch Hilfe bei der Integration realer Objekte in die Benutzungsschnittstelle.

Um diese Ziele zur erreichen, wurde ein modellgetriebener, domänenspezifischer Ansatz gewählt. Das grundlegende Konzept des Ansatzes stellt die im Rahmen dieser Arbeit entwickelte visuelle Modellierungssprache SSIML (SSIML steht für *Scene Structure and Integration Modelling Language*) dar, die eine plattformunabhängige Spezifikation von hierarchisch strukturierten 3D-Inhalten und deren Verknüpfung mit Komponenten einer Rahmenanwendung auf einem vergleichsweise hohem Abstraktionsniveau erlaubt. Durch automatische Codegenerierung aus SSIML-Modellen wird ein nahtloser Übergang zur Implementierung erreicht.

## 1.2 Aufbau der Arbeit

Kapitel 2.6 beschäftigt sich mit 3D-Anwendungen im Allgemeinen. Zunächst werden wichtige Begriffe aus dem Bereich interaktiver 3D-Grafik erklärt, die in dieser Arbeit gebraucht

werden. Anschließend wird anhand eines einfachen Klassifikationsschemas ein Überblick über Arten existierender 3D-Anwendungen gegeben. Es folgt eine Vorstellung wichtiger 3D-Formate und eine Einführung in den traditionellen Entwicklungsprozess von 3D-Anwendungen. Ebenso werden Techniken zur Integration von 3D-Inhalten und Programmcode angesprochen.

Kapitel 3 befasst sich mit modellgetriebener Software-Entwicklung. Kapitel 4 beinhaltet einen Überblick über den SSIML-Ansatz, seine einzelnen Bestandteile, Konzepte und ihre Beziehungen untereinander.

Kapitel 5 führt dann in die SSIML-Kernkomponente ein, welche Elemente zur Modellierung von 3D-Szenen und zur Spezifikation von Beziehungen zwischen der Szene und weiteren Anwendungsbestandteilen enthält. Kapitel 6 stellt die SSIML-Erweiterung SSIML/Tasks vor, die insbesondere die Entwicklung aufgabenzentrierter Anwendungen unterstützt. Kapitel 7 präsentiert SSIML/Behaviour, welches SSIML um die Möglichkeiten der Spezifikation von Verhalten und Animationen von 3D-Objekten ergänzt. SSIML/Components (Kapitel 8) dient der Beschreibung von 3D-Komponenten. Als letzte SSIML-Komponente folgt in Kapitel 9 SSIML/AR, das für die Spezifizierung von AR-Benutzungsschnittstellen vorgesehen ist. Zu jedem der Kapitel, die einen Aspekt des SSIML-Ansatzes vorstellen, existiert ein Abschnitt, in dem verwandte Arbeiten präsentiert werden, die im Kontext des Kapitels von Bedeutung sind.

In Kapitel 10 wird illustriert, wie die Konzepte des SSIML-Ansatzes sinnvoll in einen 3D-Entwicklungsprozess integriert werden können.

Kapitel 11 präsentiert informelle Evaluierungen des SSIML-Ansatzes durch Experteninterviews, durch die Modellierung zweier umfangreicher 3D-Anwendungen in studentischen Arbeiten und durch die teambasierte Entwicklung einer kompletten AR-Anwendung in einem studentischen Projekt. Ebenso werden in diesem Kapitel die als Resultat der Evaluierung umgesetzten Erweiterungen der SSIML-Notation und Verbesserungen der SSIML-Werkzeugunterstützung angesprochen.

Kapitel 12 schließt die Dissertation mit einer Zusammenfassung des wissenschaftlichen Beitrags der Arbeit und einer Diskussion der sich daraus ergebenden, für zukünftige Forschungen interessanten Fragestellungen ab.

# Kapitel 2

# Interaktive 3D-Anwendungen

## 2.1 Begriffsfestlegungen

In diesem Abschnitt werden wichtige Begriffe definiert, die in der vorliegenden Arbeit Verwendung finden und in Bezug zu Anwendungen stehen, die interaktive 3D-Grafik beinhalten.

**3D-Awendung** Eine 3D-Anwendung wird nach Pierce [Pie01] als ein Programm definiert, das dem Benutzer eine 3D-Welt präsentiert und ihm erlaubt, mit dieser Welt in Echtzeit zu interagieren. Eine 3D-Anwendung wird also *nicht* in Abhängigkeit von der Verwendung bestimmter Ein- und Ausgabegeräte, wie Datenhandschuhen oder Head Mounted Displays, betrachtet. Durch diese eher allgemeine Definition kann ein umfassendes Anwendungsspektrum abgedeckt werden.

**3D-Benutzungsschnittstelle** Eine 3D-Benutzungsschnittstelle (auch *3D-User Interface*, *3D-UI*) ist eine Benutzungsschnittstelle, die *3D-Interaktionen* einbezieht [BKLP04]. Eine *3D-Interaktion* ist dabei eine *Mensch-Computer-Interaktion*[1], in der die Aufgaben des Benutzers direkt in einem dreidimensionalen räumlichen Kontext verrichtet werden. Festzustellen ist, dass interaktive 3D-Grafik nicht unbedingt 3D-Interaktionen einschließt. Ein Benutzer kann Änderungen in einer virtuellen Welt auch über 2D-Benutzungsschnittstellen hervorrufen, z. B. indem er über ein 2D-Menü einen Betrachterstandpunkt in der virtuellen Welt auswählt. Eine 3D-Interaktion hingegen ist es beispielsweise, wenn der Benutzer ein dargestelltes 3D-Objekt direkt mit der Maus selektiert, um zu diesem Objekt zu navigieren. Die Integration von *3D-Benutzungsschnittstellen* in Anwendungen stellt daher nur *einen Teilaspekt dieser Arbeit* dar. Allgemeiner wird eine *Anwendungsintegration* von *interaktiver 3D-Grafik* betrachtet.

[1] *Mensch-Computer-Interaktion* bezeichnet den Prozess der Kommunikation zwischen menschlichen Benutzern und Computern. [BKLP04]

**Interaktive 3D-Grafik** Bei interaktiver 3D-Grafik wird ein computerinternes Modell der 3D-Welt in Echtzeit dargestellt. Der Benutzer hat die Möglichkeit, seinen Blickpunkt innerhalb der 3D-Welt zu beeinflussen oder die Darstellung der Welt auf andere Weise interaktiv zu verändern [BKLP04, Dac04].

**3D-Modell** Ein 3D-Modell besteht aus einer Ansammlung von meist geometrischen, dreidimensionalen Objekten. Zwischen den Objekten können Beziehungen bestehen, bspw. strukturelle Beziehungen. Weiterhin kann ein 3D-Modell Informationen über die räumliche Anordnung der Objekte sowie ihr Aussehen (z. B. Materialbeschreibungen und Texturen) beinhalten. Außerdem ist es möglich, dass ein 3D-Modell Metadaten (etwa in Form von Objektbeschreibungen) enthält (vgl. a. die Definition *geometrischer Modelle* in Foley et al. [FvFH95]).

**3D-Szene, Virtuelle 3D-Welt** Eine 3D-Szene wird in der vorliegenden Arbeit als ein 3D-Modell verstanden, das festgelegte Blickpunkt- bzw. Kameraeinstellungen besitzt und i. d. R. auch Lichtquellendefinitionen beinhaltet. Die der Szene zugrunde liegende Datenstruktur ist zumeist der Szenengraph. Dieser wird zur Laufzeit traversiert, wobei Informationen für das Rendering der Szene, d.h. die Umwandlung der computerinternen 3D-Szene in eine 2D-Rastergrafik, gesammelt werden. Die virtuelle 3D-Welt, welche dem Benutzer präsentiert wird, stellt dann das Ergebnis des Renderingvorganges dar.

**3D-Inhalte** Als 3D-Inhalte werden in dieser Arbeit alle Arten von 3D-Modellen bezeichnet.

**Szenengraph** Eine Definition für den Begriff Szenengraph wird in der Arbeit von Dachselt [Dac04] gegeben.

> *Ein Szenengraph ist ein gerichteter azyklischer Graph (Directed Acyclic Graph - DAG) zur effizienten hierarchischen Beschreibung einer dreidimensionalen Szene.* [Dac04]

Dabei dienen Knoten im Graph u. a. der Gruppierung von 3D-Objekten oder der Spezifizierung von Geometrien, Materialien und Transformationen.

## 2.2 Anwendungsdomänen

Dieser Abschnitt gibt einen Überblick über Einsatzbereiche von interaktiven 3D-Programmen (vgl. [BKLP04, Dac04, Alz07]) und entfaltet damit das Spektrum verfügbarer 3D-Awendungen. Zu jeder Anwendungsdomäne werden typische Anwendungsarten und -beispiele aufgeführt. Zudem werden sowohl fachspezifische (z. B. Architektur und Konstruktion) als auch fachübergreifende Domänen (z. B. Informationsvisualisierung) betrachtet, da manche Anwendungsarten in verschiedenen fachlichen Domänen vorkommen

können. Bildschirmfotos einiger Anwendungsbeispiele sind in den Abbildungen 2.1-2.5 zu sehen. Eine feiner granulierte Aufstellung von Anwendungsdomänen speziell für den Bereich Augmented Reality (AR) ist in Abschnitt 9.1.1 zu finden.

### 2.2.1 Fachliche Domänen

**Unterhaltung** Die Domäne, welche die Weiterentwicklung von 3D-Anwendungen wohl am meisten vorangetrieben hat, ist der Unterhaltungssektor. Interaktive 3D-Spiele, die i. d. R. einen hohen Grad an Interaktivität aufweisen, darunter First Person-Shooter wie id Softwares *Doom 3* [DOO] (Abbildung 2.1(a)), *Massive Multiplayer Online Games* (MMOGs) wie Linden Labs *Second Life* [Sec] (Abbildung 2.1(b)) und Blizzard Entertainments *World of Warcraft* [WoW] (Abbildung 2.1(c)), Adventures wie Lucas Arts' *Escape from Monkey Island* [Mon] (Abbildung 2.1(d)) sowie Rennspiele wie Electronic Arts' *Need for Speed ProStreet* [Nee] (Abbildung 2.1(e)), sind hier vorrangig zu finden.

**Kunst und Design** In kreativ orientierten Bereichen werden vor allem 3D-Autorenwerkzeuge wie *Autodesk 3ds max* [3ds] oder das Open-Source-Werkzeug *Blender* [Ble] (Abbildung 2.1(f)) genutzt, um offlinegerenderte Bilder, Animationsfilme oder Designstudien, etwa von Fahrzeugen, zu erstellen. Andere Beispiele sind virtuelle Graffitis [Virb] oder virtuelle Webgalerien [Virc] (Abbildung 2.1(g)).

**Architektur und Produktion** In Bereichen wie Architektur und Maschinenbau genießen 3D-CAD-Programme wie Graphisoft *ARCHICAD* [Arc] (Abbildung 2.1(h)) und *Autodesk Inventor* [Aut] (Abbildung 2.2(a)) zur Konstruktion, Bearbeitung und 3D-Darstellung von Gebäuden und Produkten eine große Verbreitung. Besonderes Augenmerk liegt hier weniger auf einem hohen Interaktionsgrad, sondern eher auf der Genauigkeit der Daten. Ebenfalls verbreitet in der Architektur sind Anwendungen zur Visualisierung von Gebäuden und virtuelle Gebäuderundgänge [Virf] (Abbildung 2.2(b)).

**Montage und Wartung** Interaktive 3D-Handbücher und Schritt-für-Schritt-Anleitungen können eingesetzt werden, um die Montage, Demontage und Wartung von technischen Geräten und Maschinen, aber auch den Zusammenbau von Möbeln zu unterstützen. Beispielanwendungen hierfür sind die Arbeit von Pollner [Pol07], in der ein interaktives 3D-Manual entwickelt wurde, ein AR-Instruktor für das Zusammenfügen von Kabelbäumen im Flugzeugbau [CMGJ99], die *Disassembly Instructor*-Webdemo von ParallelGraphics [Virb] (Abbildung 2.2(c)) und der *AR-Furniture Assembly Instructor* [ZHBH03]. Ebenso gibt es 3D-Autorenwerkzeuge, wie das Werkzeug *RapidManual* der Firma ParallelGraphics [Rap] (Abbildung 2.2(d)), welche auf die Erstellung von interaktiven 3D-Handbüchern spezialisiert sind.

**Medizin und Psychiatrie** Interaktive 3D-Grafik ist ein wichtiges Hilfsmittel bspw. in der Telemedizin [ZZZM05], bei der Visualisierung von Volumendaten aus Magnet-


Abbildung 2.1: Screenshots interaktiver 3D-Anwendungen (I) – *Doom 3* (a) [DOO], *Second Life* (b) [Sec], *World of Warcraft* (c) [WoW], *Escape from Monkey Island* (d) [Mon], *Need for Speed ProStreet* (e) [Nee], *Blender* (f) [Ble], Virtuelle Galerie (g) [Virc], *ARCHICAD* (h) [Arc]

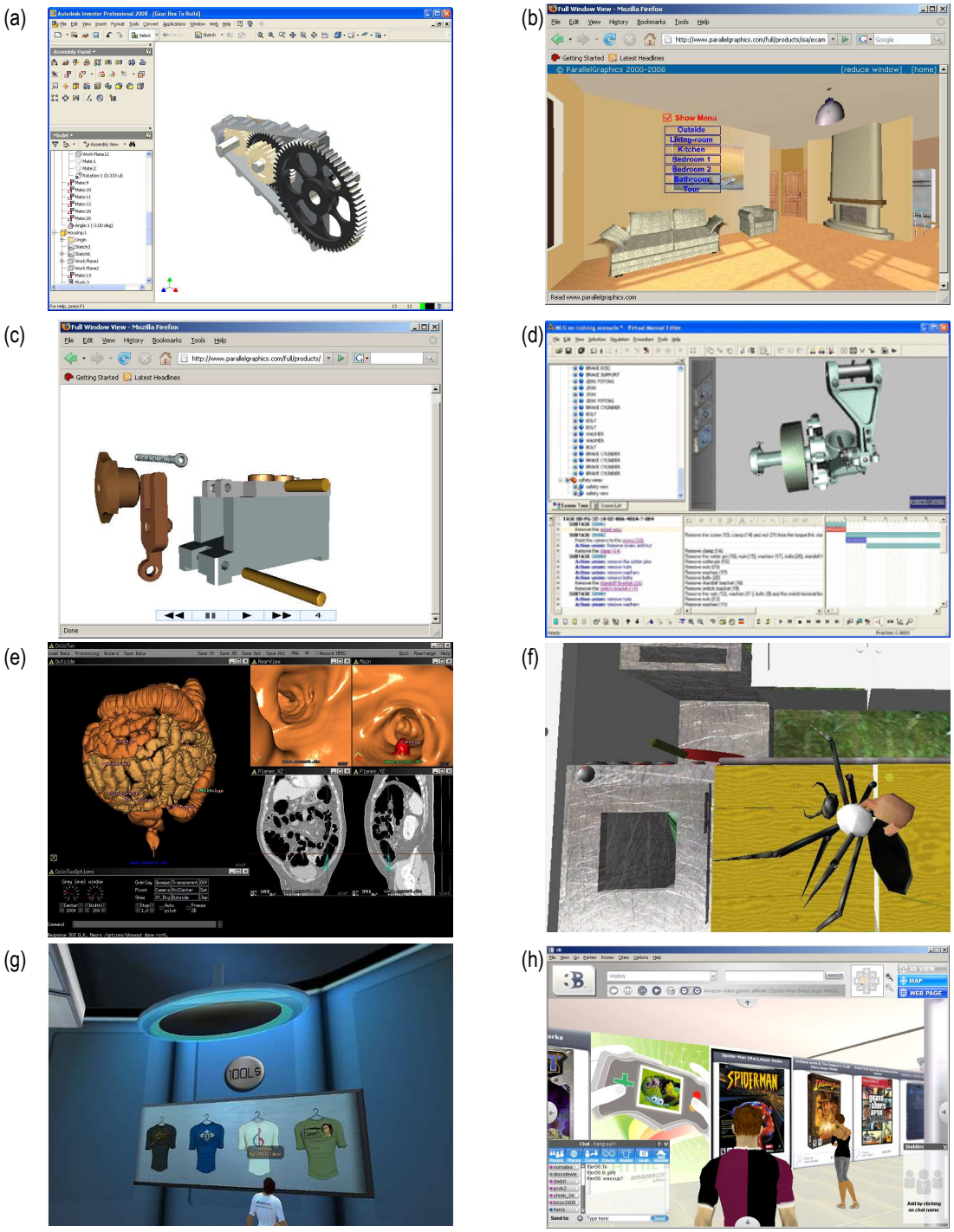



Abbildung 2.2: Screenshots interaktiver 3D-Anwendungen (II) – *Autodesk Inventor* (a) [Aut], Virtueller Gebäuderundgang (b) [Virf], *Disassembly Instructor* (c) [Vird], *Rapid Manual* (d) [Rap], *ColoTux* (e) [Col], VR-Therapie für Spinnenphobien (f) [VRS], Sony BMG-Shop in *Second Life* (g) [Sec], Amazon-Shop im Browser *3B* (h) [3B]

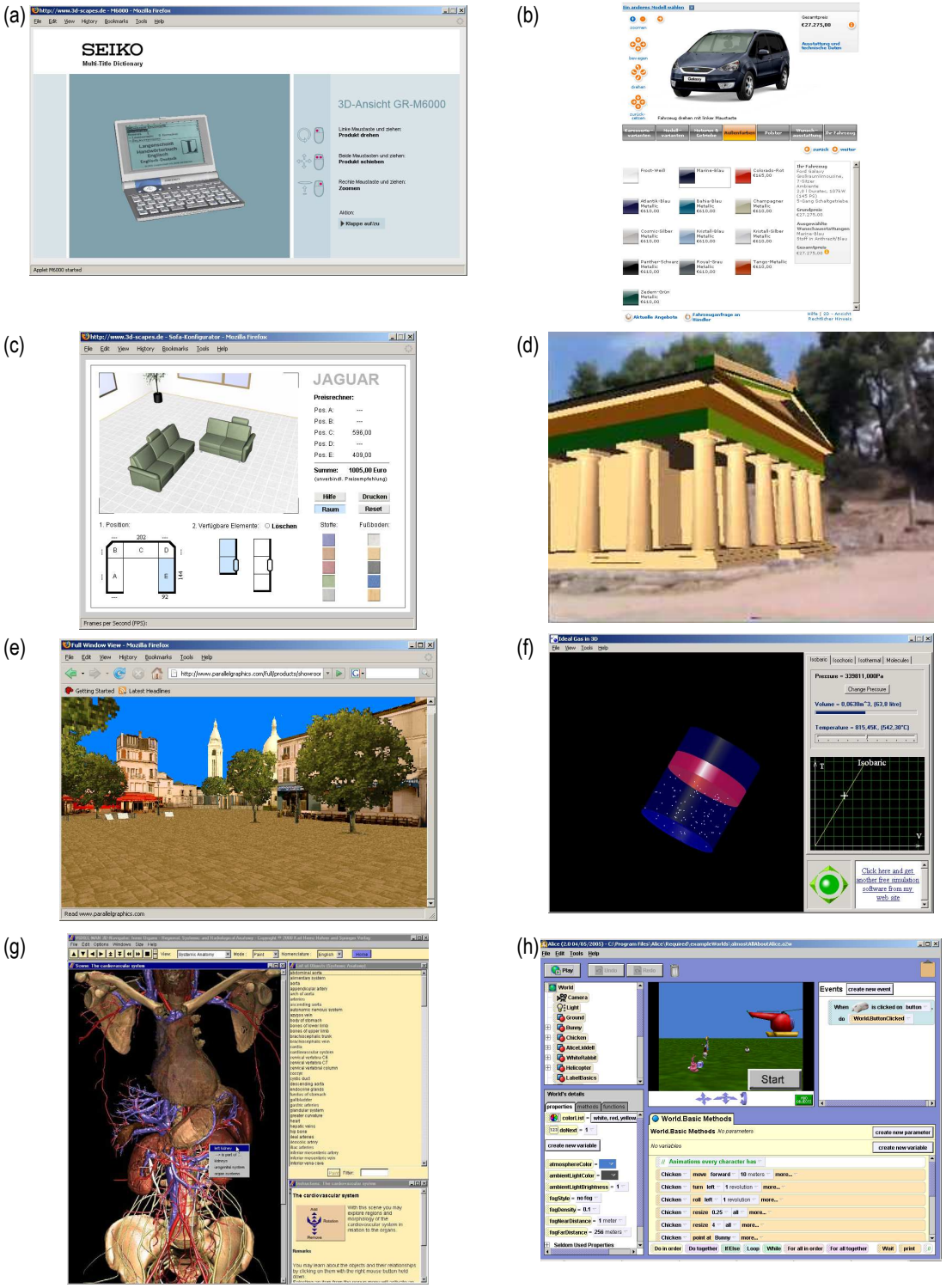



Abbildung 2.3: Screenshots interaktiver 3D-Anwendungen (III) – Produktpräsentation (a) [Vira], Ford-Konfigurator (b) [For], Sofakonfigurator (c) [Sofa], *ARCHEOGUIDE* AR-Visualisierung (d) [VIK+02], Virtueller Stadtrundgang (e) [Vire], Physiksimulation *Ideal Gas in 3D* (f) [Phy], *Voxel Man 3D Navigator: Inner Organs* (g) [Voxa], *Alice* (h) [Ali]


Abbildung 2.4: Screenshots interaktiver 3D-Anwendungen (IV) – *Microsoft Flight Simulator* (a) [MSF], *Voxel Man TempoSurg* (b) [Voxb], *MATADOR* (c) [MAT], *Operational Views in 3D* (d) [OV3], *Fire Safety* (e) [Fir], *BALLView* (f) [Bal], *Google Earth* (g) [Goo], 3D-Browser *3B* (h) [3B]

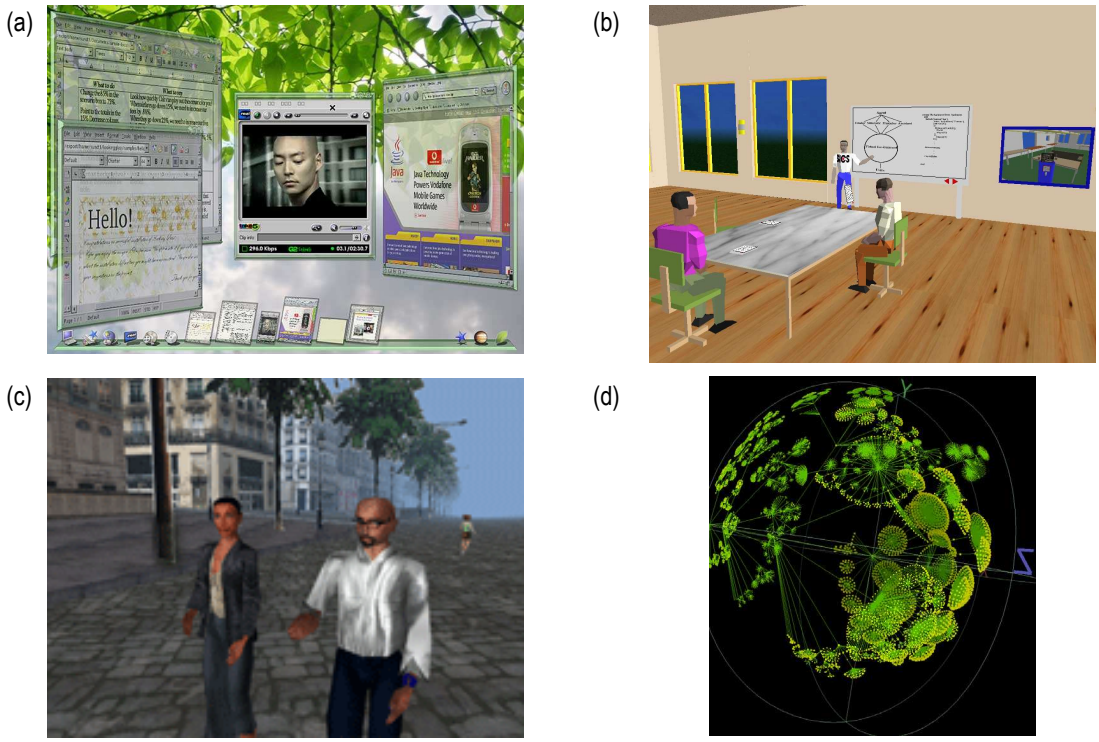



Abbildung 2.5: Screenshots interaktiver 3D-Anwendungen (V) – *Looking Glass* 3D-Desktop (a) [Loo], *DIVE* (b) [DIV], *Blaxxun Contact* (c) [bla], Informationsvisualisierung aus *Walrus* (d) [Wal]

Resonanz-Tomographie und Computer-Thomographie-Scans (etwa mit dem Programmpaket *ColoTux* [Col], s. Abbildung 2.2(e)) und der Therapierung von Phobien [VRS] (Abbildung 2.2(f)).

**e-Commerce** Anwendungsarten in diesem Bereich sind z. B. 3D-Webshops, wie einige Shops in Linden Labs *Second Life* [Sec] (Abbildung 2.2(g)) oder spezielle *3B Rooms* [3B] (Abbildung 2.2(h)), 3D-Produktvisualisierungen [Vira] (Abbildung 2.3(a)), und interaktive Produktkonfiguratoren, etwa für Fahrzeuge [For] (Abbildung 2.3(b)) oder Inneneinrichtungen [Sofa] (Abbildung 2.3(c)). Interaktive Konfiguratoren ermöglichen die Anpassung der Produktvisualisierung z. B. durch den Austausch von Teilgeometrien oder die Änderungen von Farben und Materialien.

**Kultur und Tourismus** Mittels Erweiterter Realität (Augmented Reality – *AR*) wird es möglich, teilweise oder ganz zerstörte historische Gebäude in ihrem Originalzustand zu visualisieren, wie bspw. im Projekt ARCHEOGUIDE [VIK+02] (Abbildung 2.3(d)). Außerdem können mittels AR- und VR-Tourguides Führungen absolviert werden, etwa durch Städte [Vire] (Abbildung 2.3(e)) oder Museen.

**Militär** Auch im Militärbereich findet interaktive 3D-Grafik ihre Anwendung. Häufig benutzt werden VR- und AR-Technologien zum Trainieren militärischer Operationen (s. im nächsten Abschnitt unter *Simulation und Training*) oder zur Versorgung von Soldaten mit wichtigen Zusatzinformationen in kritischen Situationen wie Kampfhandlungen. Dem letztgenannten Zweck dient z. B. das Battlefield Augmented Reality System [JBL+00].

### 2.2.2 Fachübergreifende Domänen

**Lehren und Lernen** In Lehr- und Lernanwendungen tragen interaktive 3D-Grafik und 3D-Animationen zum besseren Verständnis von Strukturen und Abläufen bei, z. B. zum Verständnis physikalischer Gesetze (Simulationen wie *Ideal Gas in 3D* [Phy], s. Abbildung 2.3(f)), anatomischer Strukturen und biologischer Prozesse (Medizinische Lernprogramme wie *Voxel Man 3D Navigator – Inner Organs* [Voxa], s. Abbildung 2.3(g)), oder unterstützen Kinder sogar konstruktiv beim Erlernen der Computerprogrammierung (Applikationen wie Alice [Ali], s. Abbildung 2.3(h)).

**Simulation und Training** Eine wichtige 3D-Anwendungsdomäne ist der Bereich Simulation und Training. Verbreitete Anwendungsarten sind hier u. a. Flug- und Fahrsimulatoren wie der *Microsoft Flight Simulator* [MSF] (Abbildung 2.4(a)), Anwendungen zur Planung und zum Training von medizinischen Eingriffen wie die Simulatoren *Voxel Man TempoSurg* [Voxb] (Abbildung 2.4(b)) und MATADOR [MAT] (Abbildung 2.4(c)), Systeme zur Planung und zum Training von militärischen Operationen wie das Simulationswerkzeug *Operational Views in 3D* von 3Dsolve [OV3] (Abbildung 2.4(d)), Software zur Schulung des Verhaltens in Notfallsituationen, wie die Komponente *Fire Safety* des *Incredible Sims Toolkit* [Fir] (Abbildung 2.4(e)), und Applikationen zur Unterstützung der Durchführung von Montage- und Wartungsarbeiten wie die Disassembly Instructor Demo [Virb] (Abbildung 2.3(c)).

**Informationsvisualisierung und Navigation in großen Datenmengen** Interaktive 3D-Grafik erlaubt die visuelle Exploration von und effiziente Navigation in großen Datenmengen, bspw. bei Molekülstrukturen (etwa mit dem Programm BALLView [Bal], s. Abbildung 2.4(f)) oder Geodaten (vgl. Ying et al. [YYGL04]; s. a. *Google Earth* [Goo] in Abbildung 2.4(g)). Auch 3D-Webbrowser wie *3B* [3B] (Abbildung 2.4(h)) und z. T. auch 3D-Oberflächen für Betriebsysteme wie der Sun *Looking Glass* Desktop [Loo] (Abbildung 2.5(a)) können in diese Kategorie eingeordnet werden.

**Kommunikation und Kollaboration** Verteilte virtuelle Umgebungen können als Kommunikationsplattform, zur verteilten Zusammenarbeit oder als Mehrspielerplattform dienen. Populäre Beispiele sind das *Distributed Interactive Virtual Environment* [DIV] (*DIVE*, Abbildung 2.5(b)), die *Blaxxun*-Multi-User-Plattform [bla] (Abbildung 2.5(c)) und Second Life [Sec].

## 2.3 Interaktivität in 3D-Anwendungen

In der Literatur, die sich mit Interaktivität in e-Learning und multimedialen Lehr- und Lernumgebungen beschäftigt, findet sich häufig eine Einteilung von interaktiven Systemen nach *Interaktionsgraden*. Oft werden in diesem Bereich 5 bis 10 verschiedene Stufen der Interaktivität unterschieden. Schulmeister [Sch05] nimmt bspw. eine 6-stufige Unterscheidung vor. Die Übertragung der Interaktionsgrade aus dem Bereich multimedialer Lehr- und Lernsysteme auf 3D-Anwendungen bietet sich aber nur bedingt an, da die für Lehr- und Lernsysteme aufgestellten Interaktionsgrade i. d. R. Bezüge zu bestimmten Lerntheorien aufweisen. Zudem wird häufig das Ziel verfolgt, Beziehungen zwischen einem Interaktionsgrad und dem „didaktischen Wert“ eines Lernprogramms dieses Interaktionsgrades herstellen zu können. 3D-Anwendungen im Allgemeinen sind jedoch nur in besonderen Fällen auf Basis einer Lerntheorie konzipiert und verfolgen oft andere Ziele als die effiziente Vermittlung von Wissen.

An dieser Stelle wird daher eine einfache, aber auf beliebige 3D-Applikationen anwendbare Unterscheidung zwischen Systemen niedrigerer und höherer Interaktivität getroffen. Eine *Interaktion* zwischen dem Benutzer und einem 3D-System (Benutzer-System-Interaktion) besteht aus einer Aktion des Benutzers und einer entsprechenden Reaktion des Systems. Eine Interaktion kann dabei z. B. ein Navigationsschritt des Benutzers in einer 3D-Welt, die Selektion oder die Manipulation eines 3D-Objektes sein. *Interaktivität* wird hier in Zusammenhang mit der *Interaktionsrate* betrachtet, also der Zahl an Interaktionen, die ein Benutzer innerhalb einer festgelegten Zeitspanne durchschnittlich benötigt, um mit dem System ein angestrebtes Ziel zu erreichen (bspw. das Erreichen des nächsten Levels in einem 3D-Spiel oder die Konfiguration eines Fahrzeuges mit einem 3D-Konfigurator). Je höher die Interaktionsrate ist, umso höher ist auch die Interaktivität des Systems einzustufen. Die Interaktivität einer 3D-Anwendung ist dabei von einer Vielzahl von Faktoren abhängig, wie der Zahl der verschiedenen Interaktionsmöglichkeiten, die das System dem Benutzer bietet, dem Verhalten der 3D-Objekte (passiv, aktiv, intelligent) sowie der Möglichkeit von Kollisionen zwischen Objekten und dem Avatar des Benutzers (der virtuellen Verkörperung des Benutzers in der 3D-Welt), um nur einige zu nennen.

Ein 3D-Ego-Shooter etwa besitzt im Normalfall ein höheres Maß an Interaktivität als ein 3D-Konfigurator. Die 3D-Objekte in einem 3D-Shooter sind – im Gegensatz zu den 3D-Objekten eines 3D-Konfigurators – meist mit einer künstlichen Intelligenz versehen, die sie nicht nur dazu befähigt, autonom zu agieren, sondern auch, auf entsprechende Aktionen des Benutzers strategisch zu reagieren. Das System ist dadurch hoch dynamisch, die 3D-Szene unterliegt ständigen Veränderungen. Die Anwendung ist mit einem spielerischen Ziel verbunden, das den Benutzer zu agilem Handeln motiviert.

Um speziell die Entwicklung von Anwendungen mit einem höheren Grad an Interaktivität zu unterstützen, wird in dieser Arbeit die Sprache SSIML/Behaviour (vgl. Kapitel 7) eingeführt, welche die Spezifikation von Verhalten und Animationen von 3D-Objekten erlaubt. Da ein höherer Grad an Interaktivität oft auch mit einer höheren Komplexität der 3D-Szene einhergeht, empfiehlt sich in diesem Zusammenhang auch der Einsatz der

in Kapitel 8 vorgestellten Sprache SSIML/Components, die fortgeschrittene Mechanismen zur Modularisierung komplexerer Szenen bereitstellt.

## 2.4 Einbindung von 3D-Inhalten

3D-Anwendungen lassen sich auch nach der Art der Einbindung von 3D-Inhalten unterscheiden. Die Anwendungsgruppe der 3D-Autorenwerkzeuge dient vorrangig der manuellen Erstellung von 3D-Inhalten. Weiterhin existieren Anwendungen, bei denen zuvor – z. B. in einem Autorenwerkzeug – erstellte 3D-Modelle eingebunden werden. 3D-Inhalte werden in diesen Applikationen lediglich *benutzt*; die Anwendungen selbst sind nicht explizit als Werkzeuge zur Erstellung von 3D-Inhalten vorgesehen.

Ebenso liegen Anwendungen vor, die dynamisch aus Mengen von Daten interaktiv explorierbare 3D-Visualisierungen erzeugen. Letztgenannte Anwendungen sind vor allem im Bereich Informationsvisualisierung zu finden. Abbildung 2.5(d) [Wal] zeigt eine solche 3D-Visualisierung.

**Abgrenzung** In dieser Arbeit soll ausschließlich die zweitgenannte Anwendungsgruppe betrachtet werden, also Applikationen, welche vorwiegend bereits erstellte 3D-Inhalte benutzen.

## 2.5 Standard-3D-Formate

Neben einer Vielzahl werkzeugspezifischer Formate (z. B. *.XSI* für *Softimage|XSI* [Sofb], *.BLEND* für *Blender* [Ble]), deren genauere Beschreibung den Rahmen dieser Arbeit sprengen würde, existieren einige werkzeugübergreifende, z. T. standardisierte 3D-Formate, von denen in diesem Abschnitt eine aktuelle Auswahl kurz vorgestellt werden soll. Vorrangig werden textbasierte Formate betrachtet, da diese sich für die Generierung von 3D-Daten aus abstrakteren Modellen besonders eignen (vgl. Abschnitt 3.6).

**Collada** Das 3D-Format *Collada* (*COLLAborative Design Activity*) ist ein XML-basiertes Format [Khr06]. Es dient der Speicherung und dem Austausch von 3D-Inhalten verschiedener Arten in hoher Qualität, etwa von Materialien, 3D-Modellen und Texturen. Collada ist also primär ein Austauschformat für 3D-Werkzeuge. Dabei soll die gesamte Werkzeugkette von der Erstellung der 3D-Inhalte bis zur Generierung der echtzeitfähigen 3D-Grafik abgedeckt werden. Collada wurde insbesondere mit dem Ziel entwickelt, die Erstellung von 3D-Spielen zu unterstützen. Es existieren Konvertierungswerkzeuge, die die Übersetzung von Collada-Dateien bspw. in X3D-Dateien ermöglichen [C2X]. Collada befindet sich in der Verantwortung des Khronos Group – Industrie Konsortiums. Die Version 1.0 von Collada erschien 2004. Collada konnte sich bereits als offenes Austauschformat etablieren. Viele 3D-Werkzeuge, darunter auch 3ds max [3ds] und Softimage|XSI [Sofb], unterstützen das Format.

**U3D** Das Format *Universal 3D* (*U3D*) ist aus einer Initiative von Intel, Adobe, Boeing und anderen Mitgliedern des 3D-Industry Forums hervorgegangen. Es handelt sich dabei zwar um ein Binärformat, allerdings soll es aufgrund seines Stellenwertes als Standardformat (U3D wurde 2005 als Ecma 363 standardisiert, inzwischen liegt die 4. Edition vor [ECM07]) und Konkurrent zu X3D nicht unerwähnt bleiben. U3D ist ein offenes und erweiterbares Format. Motiviert wurde die Schaffung von U3D durch die Überlegung, dass eine Vielzahl von CAD-Modellen existiert, die beim Design eines Produktes – etwa eines Fahrzeugs in der Automobilindustrie – aufwendig erstellt, danach aber nicht weiter verwertet werden. U3D stellt einen Versuch dar, einem großen Personenkreis – einschließlich Endkunden – Zugang zu diesen 3D-Modellen zu verschaffen, indem die entsprechenden 3D-Inhalte z. B. für Produktpräsentationen im Internet eingesetzt werden. Dafür werden die CAD-Modelle werkzeuggestützt in das U3D-Format übertragen, wobei eine Kompression stattfindet. U3D wurde für den Einsatz im Web optimiert. So eignet sich U3D etwa zum Web-Streaming und unterstützt die schrittweise Verfeinerung des Detaillierungsgrades bei der Darstellung von 3D-Szenen. Die Integration von festkörper- und skelettbasierten Animationen ist ebenfalls möglich. Das Format wird durch eine virtuelle Maschine interpretiert, womit die Plattformunabhängigkeit sichergestellt werden soll, und bietet eine Schnittstelle zum Laufzeitzugriff. Insbesondere wurde die Einbettung von U3D in elektronische Dokumente, z. B. im Portable Document Format (PDF) angestrebt und auch realisiert [Maj05]. Obwohl U3D bisher wohl noch nicht die angestrebte Verbreitung erreicht hat, unterstützen einige 3D-Werkzeuge, darunter auch 3ds max, den Import und Export von U3D.

**XAML** *XAML* (*eXtensible Application Markup Language*) ist eine XML-basierte Sprache zur Beschreibung von Benutzungsoberflächen. XAML ist Bestandteil des Windows Presentation Foundation - APIs der Microsoft .NET 3.0-Plattform [Dot]. Damit ist XAML vorrangig für den Einsatz unter Microsoft Windows vorgesehen. XAML ist zwar kein „reines“ 3D-Format, es verfügt allerdings über leistungsfähige Möglichkeiten zur Einbettung interaktiver 3D-Szenen. Ein wichtiges Ziel von XAML ist die Unterstützung von Designern und Programmierern durch strikte Trennung von grafischer Oberfläche und Anwendungslogik. Zu diesem Zweck kann die in XAML definierte grafische Benutzeroberfläche mit separat gespeichertem Programmcode hinterlegt werden, der die Anwendungslogik enthält. Die Verknüpfung von Logik und XAML-Oberfläche erfolgt z.B. durch Ereignisbehandlungsmethoden im Programmcode, die über ihre Namen mit Steuerelementen der in XAML definierten Benutzeroberfläche assoziiert werden können. XAML erlaubt die Einbettung von deklarativ beschriebenen 3D-Szenen in die Benutzungsoberflächendefinition. Um auf Objekte innerhalb der 3D-Szene aus dem Programmcode heraus zuzugreifen, müssen die 3D-Objekte auf traditionellem Wege über ihren Namen adressiert werden, wodurch auch hier die in Kapitel 1 angesprochenen Probleme auftreten. Bisher gibt es nur eine geringe – aber wachsende – Anzahl von 3D-Werkzeugen, die das XAML-Format unterstützen.

**VRML97** *VRML97* (VRML steht für *Virtual Reality Modeling Language*) wurde 1997 als ISO/IEC 14772-1 [Int97] standardisiert. Es handelt sich bei VRML97 (nachfolgend kurz

als VRML bezeichnet) um ein textbasiertes Format zur deklarativen Beschreibung von interaktiven 3D-Szenen. VRML verwendet dabei eine eigene, nicht XML-konforme Syntax. VRML hat einige seiner Wurzeln im Speicherformat des Szenengraph-APIs Open Inventor [SC92]. Die Interpretation einer mit VRML beschriebenen Szene erfolgt durch einen so genannten Browser. Es existieren VRML-Browser-Implementierungen als Webbrowser-Plugins oder in Form von Stand-Alone-Anwendungen. Der Standard spezifiziert, wie die Szenen zu interpretieren und auszuführen sind. Neben der Möglichkeit, über eine definierte Schnittstelle (das so genannte External Authoring Interface – EAI [Int04a]) zur Laufzeit aus einer externen Anwendung heraus auf die VRML-Szene zuzugreifen, können die bereits durch den Standard vordefinierten Funktionalitäten durch in die Szene integrierbaren Skriptcode erweitert werden. Trotz Ergänzungen zum Standard (z. B. *Humanoid Animation* [Int05a]), die durch das Web3D-Konsortium [W3D] erarbeitet wurden, ist VRML als Laufzeitformat für die Anforderungen heutiger 3D-Anwendungen oft nicht ausreichend. Allerdings wird es z. T. noch als Austauschformat verwendet, da viele populäre 3D-Autorenwerkzeuge die Möglichkeit des VRML-Imports und –Exports anbieten. Nachteile von VRML, wie zwar vorhandene, jedoch unzureichende Erweiterungsmechanismen und das Fehlen einer binären Codierung zur effizienten und speicherplatzsparenden Übertragung, sollen vom VRML-Nachfolgeformat X3D beseitigt werden.

**X3D** *X3D* (*Extensible 3D*) ist der Nachfolger des VRML97-Formats. X3D bietet neben der klassischen VRML-Syntax insbesondere eine XML-Syntax und eine binäre Codierung an. X3D setzt sich aus mehreren Standards zusammen, wie der 2004 standardisierten abstrakten funktionalen Spezifikation (ISO/IEC 19775 [Int04b]), den Kodierungsspezifikationen (ISO/IEC 19776 [Int05b]) und den Sprachanbindungsspezifikationen für Java und Ecmascript (ISO/IEC 19777 [Int05c, Int05d]). Der Funktionsumfang von VRML wurde in X3D deutlich erweitert, z. B. um Möglichkeiten der Definition von Shadern. Zudem besitzt X3D im Gegensatz zu VRML eine deutlich modularere Struktur. Dies wurde vor allem durch die Einführung der Konzepte *Profiles*, *Components* und *Levels* erreicht. Ein *Profile* repräsentiert eine Menge von Funktionen, die ein X3D-Browser, welcher dieses Profile unterstützt, implementieren muss. Derzeit existieren 6 Profiles, die sich in ihrem Funktionsumfang unterscheiden. Das Profile CORE enthält bspw. den geringsten Funktionsumfang, das Profile FULL hingegen den höchsten. *Components* fassen Elemente wie Szenenknotentypen, die verwandte Funktionalitäten besitzen, zusammen. Die Defintion zusätzlicher Components und Profiles ist möglich. So wurde z. B. in der ersten Revision des X3D-Standards eine Component für physikbasierte Animationen starrer Körper definiert [Int08b]. Ein Browser kann zudem die X3D-Spezifikation auf verschiedenen *Levels* unterstützen. Ein höherer Level kann etwa durch die Realisierung von Funktionalitäten erreicht werden, die in der Spezifikation als optional ausgewiesenen sind. Szenenzugriff zur Laufzeit kann entweder extern (durch externe Applikationen) oder intern (durch Skriptknoten) mittels des Scene Access Interfaces (SAI) erfolgen. Wie auch schon VRML97 und im Gegensatz zu bspw. Collada hat X3D den Anspruch, ein Laufzeitformat zu sein. Daher wird im X3D-Standard wie bei VRML ein Ausführungsmodell für X3D-Szenen definiert. Neben

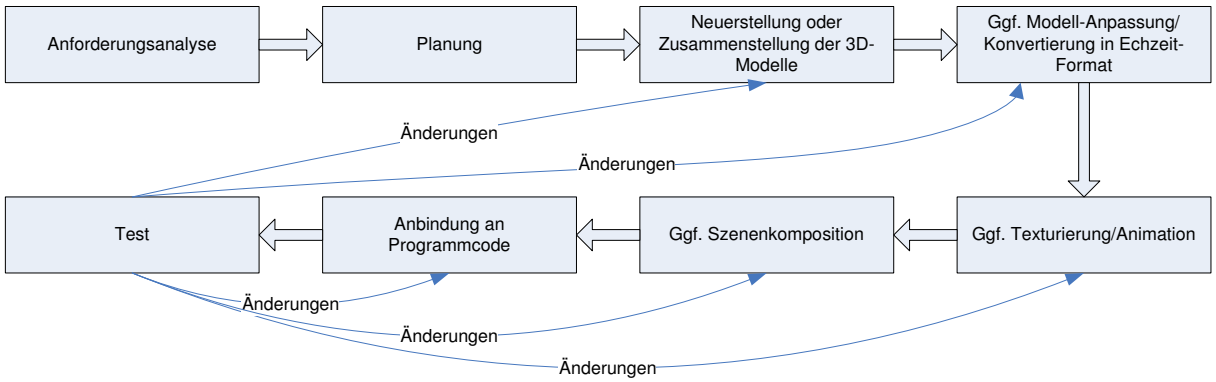



Abbildung 2.6: Der grundlegende 3D-Entwicklungsprozess

einigen Werkzeugen, die auf X3D spezialisiert sind, existieren inzwischen auch Plugins zum X3D-Export für weit verbreitete 3D-Werkzeuge, wie 3ds max [3ds].

## 2.6 Entwicklung von 3D-Anwendungen

Der hier vorgestellte Prozess der Entwicklung von 3D-Anwendungen orientiert sich an einer Befragung von 6 Experten, von denen 5 Experten über Erfahrungen in der 3D-Entwicklung im industriellen Umfeld verfügten und ein Experte aus dem Forschungsumfeld stammte (vgl. a. Abschnitt 11.1). Ferner enthält die Arbeit von Pollner [Pol07] Ausführungen zum 3D-Entwicklungsprozess, die hier ebenfalls mit berücksichtigt wurden.

Natürlich variiert ein konkreter Entwicklungsprozess abhängig von der Art der zu erstellenden Software. Beispielsweise unterscheidet sich der Prozess für die Erstellung eines 3D-Spieles von dem Prozess der Erstellung einer interaktiven 3D-Produktpräsentation für das Web in vielen Details. An dieser Stelle soll der grundlegende Ablauf der Entwicklung einer 3D-Anwendung dargestellt werden (Abbildung 2.6).

Im Entwicklungsprozess werden zunächst die Anforderungen analysiert und erfasst, die an die zu entwickelnde Software gestellt werden. Diese werden hauptsächlich in einer eher informellen Form in Textdokumenten festgehalten. Auf Basis der aufgestellten Anforderungen wird die umzusetzende Anwendung geplant.

Als erster Schritt der Umsetzung erfolgt die Erstellung der 3D-Inhalte, sofern keine (Wieder)verwendung vorhandener 3D-Daten (z. B. die Verwendung von CAD-Modellen) möglich ist. 3D-Daten, die – wie CAD-Modelle – in einer mathematischen Beschreibungsform (z.B. als NURBS oder Bezier-Kurven) vorliegen, müssen in ein echtzeitfähiges Format, d.h. eine polygonale Darstellung, konvertiert werden. Dies geht mit einer Tesselierung einher, bei der komplexere Polygone in so genannte primitive Flächen, wie Dreiecke, zerlegt werden. Außerdem kann es notwendig sein, bereits in Polygonform vorliegende Modelle durch eine Reduktion der Polygonzahl zu vereinfachen. Werden vorhandene Modelle verwendet, müssen diese u. U. zusätzlich an die vorgesehene Nutzung angepasst bzw. mit Geometrien erweitert werden. Anschließend erfolgt ggf. eine Texturierung,

bei der den 3D-Objekten Texturkoordinaten zugewiesen werden. Einfachere 3D-Objekte können weiterhin im 3D-Autorenwerkzeug hierarchisch zu komplexeren 3D-Modellen (Objektkomposition) oder fertigen Szenen, die auch Beleuchtungs- und Kamerainformationen enthalten, zusammengesetzt werden. Animationen der 3D-Modelle werden entweder im 3D-Autorenwerkzeug definiert oder später programmtechnisch umgesetzt.

Nach der Fertigstellung der 3D-Modelle muss Programmcode geschrieben werden, der es erlaubt, das Verhalten der 3D-Objekte zu steuern und die 3D-Szene in gewünschter Weise zu modifizieren. Meist erfolgt dies unter Benutzung einer vorhandenen Codebasis, wie einer 3D-Engine oder eines Szenengraph-APIs. Die als Dateien persistent vorliegenden 3D-Modelle werden zur Laufzeit über Funktionalitäten der Codebasis eingelesen, wobei ein Objektmodell in Form eines (Teil)szenengraphen erzeugt wird, das zur Laufzeit modifiziert werden kann. Es kann entweder ein kompletter Szenengraph, der bereits alle 3D-Modelle und Beleuchtungsinformationen enthält, eingelesen werden (vorausgesetzt, dieser wurde zuvor mit einem Autorenwerkzeug komplett erstellt), oder es werden nur einzelne Modelle geladen und als Teilgraphen in den zuvor anderweitig – z. B. durch Programmierung – konstruierten Gesamtszenengraphen eingefügt.

Es folgt der Test der 3D-Anwendung und ggf. eine Präsentation vor dem Kunden, woraus u. U. eine Überarbeitung von Teilen der Anwendung resultiert.

Während des Entwicklungsprozesses treten – neben technischen Problemen, die häufig auf eine unzureichende Werkzeugunterstützung beim Austausch und der Konvertierung von 3D-Daten zurückzuführen sind – die bereits in Kapitel 1 skizzierten Probleme auf, wie ungenügend parallelisierte Arbeitsabläufe, unzureichende Möglichkeiten zur Modularisierung und Wiederverwendung einzelner Anwendungbestandteile (z. B. Animationsbeschreibungen), und die durch die interdisziplinäre Zusammenarbeit von Designern und Programmierern bedingten Probleme, wie Kommunikationsschwierigkeiten zwischen den Entwicklern und den oft daraus resultierenden Inkonsistenzen zwischen einzelnen Software-Bestandteilen. Weitere Schwierigkeiten, die speziell bei der Entwicklung von Augmented Reality-Systemen auftreten, werden in Kapitel 9 genauer erläutert.

### 2.6.1 Rollen im 3D-Entwicklungsprozess

Im 3D-Entwicklungsprozess können verschiedene *Entwicklerrollen* unterschieden werden. Die Anzahl der Rollen hängt von der Größe, der Art und den Anforderungen eines entsprechenden Projektes ab. Generell kann zwischen kreativ und technisch orientierten Entwicklern unterschieden werden. Kreativ orientierte Entwickler sind z. B. *3D-Modellierer* (manchmal auch als *3D-Artist* bezeichnet), *Texturierer*, *Animator* und *Sound-Designer*. Technisch orientierte Entwickler können z. B. *(Interaktions-)Programmierer* und *3D-Framework-/Engine-Entwickler* sein. Hinzu kommen weitere am Entwicklungsprozess beteiligte Personen, die jedoch nicht unmittelbar unter den Begriff *Entwickler* fallen. Dies sind z. B. Personen, die organisatorische Funktionen besitzen, wie der *Projektleiter*, Personen wie Kunden, welche die Entwicklung des Produktes in Auftrag gegeben haben, und nicht zuletzt Personen, die das Produkt testen (*Tester*) oder benutzen (*Benutzer*).

In der vorliegenden Arbeit werden vorrangig die Aufgaben der *Entwickler* betrachtet. Bei den kreativ orientierten Entwicklern stehen vorwiegend die *3D-Modellierer* und z. T. auch die *Animatoren* im Fokus, die hier zusammenfassend als *3D-Entwickler* oder *3D-Designer* bezeichnet werden, während bei technisch orientierten Entwicklern *Programmierer*, *Framework-Entwickler*, *Software-Designer* und *Transformationsdesigner* unterschieden werden. *Software-Designer* und *Transformationsdesigner* stellen dabei zusätzlich eingeführte Rollen dar. In Kapitel 10 wird skizziert, wie die in dieser Arbeit vorgestellten Konzepte unter Einbezug der vorgestellten Rollen sinnvoll in einen 3D-Entwicklungsprozess integriert werden können.

## 2.7 Integration von Programmcode und 3D-Inhalten

Prinzipiell lassen sich *zwei Arten der Verknüpfung von Programmcode und 3D-Inhalten* unterscheiden: Programmcode kann in 3D-Inhalte eingebettet werden, um Eigenschaften von 3D-Objekten zu modifizieren (*interner Code*), oder er kann sich in Anwendungskomponenten befinden, die die 3D-Szene *von außen* manipulieren bzw. die Zustände von 3D-Objekten lesen, etwa um Kollisionen zu prüfen (*externer Code*). Oft ist eine Kombination von szeneninternem und extern gehaltenem Code sinnvoll.

Interner Code wird häufig in die Knoten eines Szenengraphen integriert. Beispiele dafür sind die so genannten *Behavior*-Knoten in Java3D oder *Script*-Knoten in VRML/X3D. Derartige Knoten dienen meist dem Zweck, 3D-Objekte zu animieren und ihnen Verhalten zuzuordnen, das bspw. ereignisbasiert gesteuert werden kann.

Ein externer Zugriff auf die Szene kann in Form eines Zugriffs auf das Objektmodell der Szene zur Laufzeit erfolgen. Das Objektmodell kann z.B. ein *Open Inventor* [SC92]- oder Java3D [J3D]-Szenengraph sein. VRML97 und X3D stellen für den externen Szenenzugriff spezielle APIs zur Verfügung, nämlich das *External Authoring Interface* (*EAI*) bzw. das *Scene Access Interface* (*SAI*, wird auch für interne Zugriffe verwendet). Einige typische Fälle, bei denen hingegen ein externer Zugriff auf eine Szene sinnvoll ist, sind (vgl. a. [SMM98]).:

- Es handelt sich um eine Anwendung, die über eine *hybride* Benutzungsschnittstelle verfügt, die also sowohl 2D- als auch 3D-Oberflächenkomponenten enthält. Objekte der 3D-Szene, die in einer 3D-Ansicht angezeigt werden, können über 2D-Benutzungsschnittstellenelemente modifiziert werden. Bspw. kann ein Schieberegler benutzt werden, um Objekte der 3D-Szene zu skalieren.

- Es handelt sich um eine verteilte virtuelle Welt, bei der Informationen über ein Netzwerk gesendet werden, etwa Koordinaten von 3D-Objekten oder Textnachrichten von Benutzern.

- Es sollen benutzerspezifische Navigationstechniken realisiert werden.

- Eine Anwendung steuert von außen das Verhalten von 3D-Objekten der 3D-Szene.

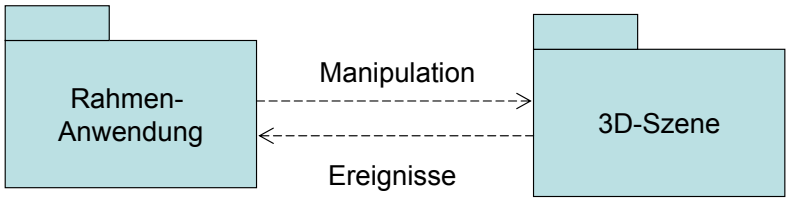



Abbildung 2.7: Kommunikationsbeziehungen zwischen Rahmenanwendung und Szene

Oft bestehen Beziehungen zwischen Applikationsbestandteilen und Szene in beiden Richtungen. Es findet also eine Kommunikation zwischen der *Rahmenanwendung* und *Szene* statt (Abbildung 2.7). *Kommunikation* bedeutet in diesem Fall, dass externe Komponenten sowohl die Szene verändern und Attributwerte von Szenenelementen lesen können, als auch Ereignisse, die innerhalb der Szene auftreten, abhören können. Eine hybride Benutzungsschnittstelle kann es z. B. erlauben, dass Informationen über ein in der 3D-Szene selektiertes Objekt in einem Textfeld angezeigt werden und umgekehrt Objekte der Szene über eine 2D-Benutzungsschnittstelle manipuliert werden. 3D-Autorenwerkzeuge bieten derartige Möglichkeiten.

Das externe Abhören von Ereignissen setzt voraus, dass Ereignisse innerhalb der Szene generiert werden, was durch den in der Szene eingebetteten Code geschieht. Einige Szenengraph-APIs definieren bereits Knoten vor, die Ereignisbotschaften erzeugen können. In VRML/X3D existieren bspw. so genannte Sensorknoten. Ein `TouchSensor` etwa ist mit einem geometrischen Objekt assoziiert und erzeugt eine Ereignisbotschaft, wenn das Objekt vom Benutzer selektiert wurde. Eine mögliche Realisierung der entsprechenden Abhörmechanismen besteht in der Anwendung des Observer-Patterns [GHJV94]: Anwendungsobjekte, die über Veränderungen in der Szene informiert werden wollen, registrieren sich dafür bei den jeweiligen Knoten als Observer.

Die in dieser Arbeit vorgestellte Sprache SSIML erlaubt es, die Kommunikationsbeziehungen zwischen externer Anwendung und der 3D-Szene bzw. 3D-Objekten zu modellieren (vgl. Kapitel 5 und 6). Konzepte zur Integration von Verhaltensbeschreibungen von 3D-Objekten in die Szene werden in den Kapiteln 7 und 8 präsentiert. Die Verhaltensbeschreibungen können auf szeneninternen Code abgebildet werden. SSIML unterstützt also beide der oben skizzierten Varianten der Verknüpfung von 3D-Inhalten und Programmcode.

# Kapitel 3

# Modellgetriebene Entwicklung mit domänenspezifischen Sprachen

## 3.1 Modelle

Für den Begriff Modell im Kontext von (Software-)Systemen finden sich in der Literatur verschiedene Definitionen (z. B. [KBJV06, KWB03, Obj03]). In der vorliegenden Arbeit soll die Definition nach Kleppe et al. [KWB03] verwendet werden:

> *A model is a description of (part of) a system written in a well-defined language. A well defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automatic interpretation by a computer.* [KWB03]

Damit wird der Modellbegriff zunächst unabhängig von einer bestimmten visuellen Repräsentationsform (also einer textlichen oder grafischen Syntax) oder einer bestimmten Technologie definiert. Auch ein Java-Programm ist beispielsweise nach dieser Definition ein Modell[1].

Ein *System* kann etwa ein Software-System sein oder eine bestimmte fachliche Domäne, aber auch ein *abstraktes* System in Form eines weiteren Modells [FN05].

> *[The modeled] system may include anything.* [KWB03]

Bei den in dieser Arbeit verwendeten Modellen handelt es sich hauptsächlich um *Software-Modelle*, also Modelle von Software-Systemen, und Modelle von Modellen (s. dazu Abschnitt 3.3 – Metamodelle).

[1] Soll ein in einer konkreten Textsyntax beschriebenes Modell explizit von einem in einer grafischen Syntax spezifizierten Modell unterschieden werden, wird nachfolgend der Begriff *Code* statt *Modell* für das textcodierte Modell gebraucht.

## 3.2 Visuelle Modellierungssprachen

Eine *Modellierungssprache* ist eine Sprache, die der Beschreibung von Modellen dient. Anzumerken ist, dass nach obiger Definition eines Modells z. B. auch eine Programmiersprache wie Java eine bestimmte Art einer Modellierungssprache ist. Eine Modellierungssprache kann eine General Purpose Language (GPL) oder eine domänenspezifische Sprache (Domain Specific Language, DSL) sein (s. Abschnitt 3.5). Eine Modellierungssprache legt die zu verwendenden Modellelemente und deren Beziehungen zueinander (*abstrakte Syntax*) sowie ein oder mehrere *konkrete Syntaxen* in Form von Notationen für die zu erstellenden Modelle fest. Zudem besitzt eine Modellierungssprache eine implizit oder explizit definierte *Semantik*. Eine *graphische* oder auch *visuelle Modellierungssprache* besitzt eine graphische Notation (im Gegensatz zu einer Textnotation). Ein *visuelles Modell* ist demnach ein Modell, das in einer visuellen Modellierungssprache geschrieben ist. UML [Obj07c] und die im Kapitel 5 vorgestellte Sprache SSIML sind z. B. visuelle Modellierungssprachen. Auf die Bedeutung der *abstrakten Syntax*, der *konkreten Syntax* und der *Semantik* einer Modellierungssprache wird im Abschnitt 3.5 noch genauer eingegangen.

## 3.3 Metamodelle

Die *abstrakte Syntax* einer Modellierungssprache ist ein *Modell*; sie ist in einer wohldefinierten Sprache geschrieben. Die abstrakte Syntax einer Modellierungssprache ist zudem ihr charakterisierendes Merkmal; die Modellierungssprache wird *durch ihre abstrakte Syntax definiert*. Statt des Begriffes *Modellierungssprache* soll nachfolgend kurz der Begriff *Sprache* gebraucht werden.

Ein Modell ist in einer Sprache geschrieben. Eine *Sprache* wird selbst durch ein *Modell* (ihre *abstrakte Syntax*) definiert, welches in einer (weiteren) Sprache, einer *Metasprache*, geschrieben ist. Ein Modell, welches in einer Metasprache geschrieben ist, wird auch als *Metamodell* bezeichnet. Da eine Modellierungssprache vollständig durch ihre abstrakte Syntax definiert ist, wird in der Praxis häufig keine explizite Unterscheidung zwischen der Sprache und dem dazugehörigen Metamodell vorgenommen [KWB03].

Eine Metasprache selbst ist wiederum durch ein weiteres Metamodell, das *Metametamodell* (die abstrakte Syntax der Metasprache) definiert.

Führt man das obenstehende Prinzip fort, können weitere Meta-Sprach-bzw. Metamodell-Ebenen definiert werden. In der Praxis hat sich bisher allerdings ein Maximum von drei Modellebenen (Modell, Metamodell, Metametamodell) als sinnvoll erwiesen. Hinzu kommt die so genannte Instanz-Ebene, wie im nächsten Abschnitt verdeutlicht wird.

Ein in einer bestimmten Sprache $S$ geschriebenes Modell ist *eine Instanz* desjenigen Metamodells, das die abstrakte Syntax von $S$ verkörpert. Ein UML-Modell ist beispielsweise eine Instanz des UML-Metamodells. Das UML-Metamodell ist wiederum eine Instanz des MOF-Metametamodells. Man sagt auch, dass ein Modell *konform zu* seinem Metamodell ist. Ein Metamodell gibt die Regeln für die Strukturierung der zu ihm konformen Modelle vor [KBJV06].

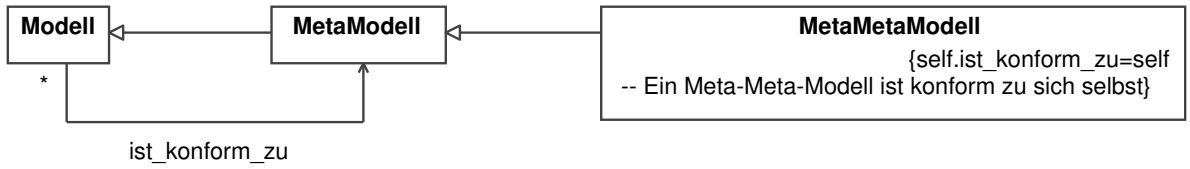



Abbildung 3.1: Die Zusammenhänge zwischen Modellen, Metamodellen und Metametamodellen (nach Kurtev et al. [KBJV06])

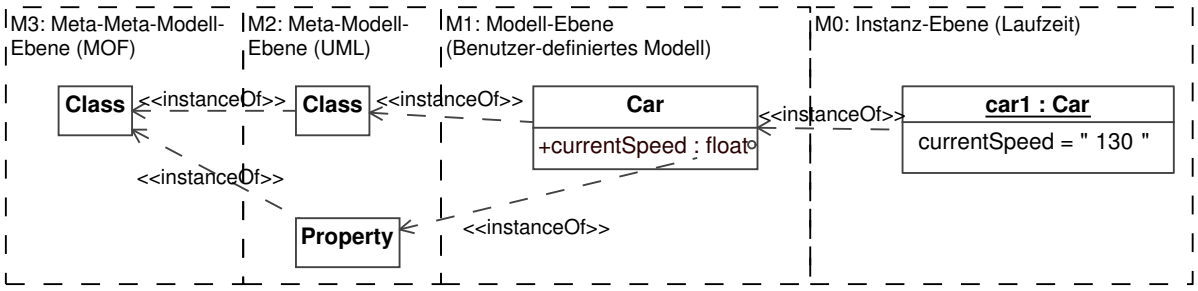



Abbildung 3.2: Die vier Ebenen der Metamodell-Hierarchie nach der OMG

Abbildung 3.1 illustriert noch einmal die Zusammenhänge zwischen Modellen, Metamodellen und Metametamodellen.

### 3.3.1 Die vier Ebenen der Metamodell-Hierarchie nach der OMG

Die Object Management Group (OMG[2]) unterscheidet vier (Meta-)Modellierungsebenen bzw. Schichten: M0, M1, M2, M3 (Abbildung 3.2). Die Ebene M0 nimmt dabei eine Sonderstellung ein, da sie das modellierte System zur Laufzeit repräsentiert.

**M0: Instanz-Ebene** Die Ebene M0 enthält die konkreten Daten (Objekte) eines laufenden Systems, die Instanzen von Elementen der Ebene M1 sind. Bspw. könnte es auf dieser Ebene ein Objekt `car1` vom Typ `Car` mit einem mit dem Wert `130` belegten Attribut `currentSpeed` geben.

**M1: Modell-Ebene** Die Ebene M1 enthält Modelle, wie das Modell eines Software-Systems. Die Elemente eines Modells haben Instanzen auf der Ebene M0. Modelle sind in Modellierungssprachen geschrieben, die durch Metamodelle der Ebene M2 definiert sind.

[2]Die OMG (Object Management Group) ist ein internationales Konsortium der Computer-Industrie, dessen Fokus nach der Gründung 1989 auf der Entwicklung von Standards für objektorientierte verteilte Systeme lag. Inzwischen wurden eine Vielzahl von Standards für verschiedene Technologien entwickelt, u. a. die UML. In den letzten Jahren hat sich das Konsortium vor allem der modellgetriebenen Software-Entwicklung gewidmet.

Ein Modell der Ebene M1 könnte bspw. durch ein UML-Klassendiagramm repräsentiert werden. Ein Modellelement dieses Klassendiagramms wäre etwa eine Klasse `Car` mit einem Attribut `currentSpeed`.

**M2: Metamodell-Ebene** Die Ebene M2 enthält Metamodelle bzw. Modellierungssprachen. Instanzen eines Metamodells sind Modelle der Ebene M1. Instanzen der Elemente eines Metamodells (Metamodell-Elemente) sind Modell-Elemente auf der Ebene M1. Das UML-Element `Class` kann z. B. auf der Ebene M1 eine Instanz `Car` haben.

**M3: Metametamodell-Ebene** Diese Ebene enthält das Metametamodell. Im Fall der OMG-Metamodell-Architektur ist das Metametamodell die *Meta Object Facility* (MOF). Modellierungssprachen der Ebene M2 sind Instanzen des Metametamodells. Die UML ist bspw. eine Instanz der MOF. Instanzen der Modellelemente des Metametamodells sind Elemente der Metamodelle auf Ebene 2. So ist das UML-`Class`-Element eine Instanz des gleichnamigen MOF-Elements. Eine Ebene M4 existiert in der Vier-Schichten-Architektur der OMG nicht. Elemente der Ebene M3 sind Instanzen von Konzepten derselben Ebene.

## 3.4 Technical Spaces

Ein weiteres wichtiges Konzept ist dasjenige der *Technical Spaces* (*TS*s). Ein TS ist wie folgt definiert:

> *A technical space is a model management framework with a set of tools that operate on the models definable within the framework.* [KBJV06]

Verschiedene TSs sind in Abbildung 3.3 dargestellt. So gibt es einen EBNF-TS, einen XML-TS und einen MOF-TS. Im XML-TS etwa existieren Werkzeuge wie XML-Parser, Schema-Validatoren, spezialisierte XML-Editoren usw. Eine wichtige Erkenntnis besteht darin, dass in jedem TS die Dreiteilung der Meta-Ebenen (M1, M2, M3) wieder auftritt.

Von besonderer Bedeutung sind Techniken, die es erlauben, Artefakte von einem TS in einen anderen zu überführen, z. B. ein UML-Modell in ein XMI-Dokument oder Java-Code. Eine derartige Überführung bezeichnet man auch als *Überbrückung* (*Bridging*). Transformations-Werkzeuge, die Bridging ermöglichen, werden *Technische Projektoren* genannt [KBJV06]. Bridging spielt z. B. bei der Abbildung der abstrakten Syntax einer Modellierungssprache auf eine konkrete Syntax oder der Überführung eines Modells in ein ausführbares Format eine wichtige Rolle.

## 3.5 Domänenspezifische Sprachen

Im Gegensatz zu General Purpose-Sprachen (General Purpose Languages, GPLs), wie Java, C++, UML oder gar Maschinensprache, sind domänenspezifische Sprachen (Domain

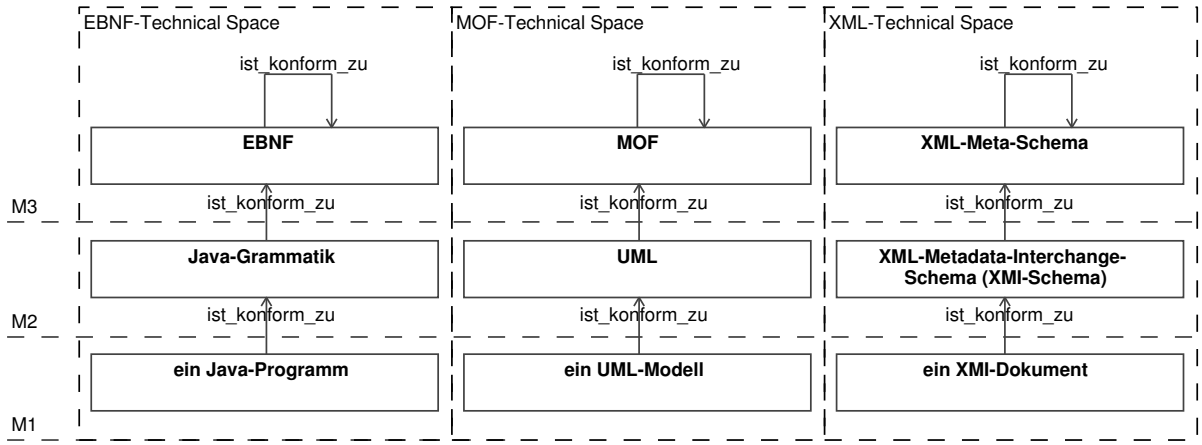



Abbildung 3.3: Beispiele für die Dreiteilung der Meta-Ebenen in verschiedenen Technical Spaces

Specific Languages, DSLs), z. B. SQL und VRML97, jeweils für den Einsatz in einer bestimmten Domäne vorgesehen (vgl. [KBJV06, CJKW07, LS06]). DSLs sind für die effiziente Erfüllung bestimmter Aufgaben konzipiert, während GPLs den Anspruch haben, für die Bearbeitung eines möglichst breiten, domänenübergreifenden Aufgabenspektrums geeignet zu sein. Der Vorteil einer DSL ist also, dass die Aufgabe, für welche die DSL spezialisiert ist, in der Regel mit dieser DSL erheblich effizienter bearbeitet werden kann als mit einer GPL.

Entsprechend der vorangegangenen Betrachtungen ist eine DSL eine Modellierungssprache. Eine DSL selbst kann – wie auch eine GPL – betrachtet werden als *eine Menge miteinander in Beziehung stehender und aufeinander abgestimmter Modelle* (vgl. [KBJV06]). Diese Modelle sind typischerweise die *abstrakte Syntax*, eine oder mehrere *konkrete Syntaxen* und die *Semantik* der DSL.

### 3.5.1 Abstrakte und konkrete Syntax

Domänenspezifische Sprachen dienen der Definition von domänenspezifischen Modellen. Die Modelle können mittels einer textlichen oder grafischen *konkreten Syntax* definiert werden, die intern auf eine „unsichtbare“, darstellungsunabhänige *abstrakte Syntax* abgebildet wird. Eine domänenspezifische Sprache ist nicht unbedingt auf eine einzige konkrete Syntax beschränkt. Beispielsweise besitzt die Sprache X3D [Int04b] eine *XML*- und eine *VRML*-Syntax. UML und die in dieser Arbeit vorgestellte Sprache SSIML besitzen neben einer grafischen auch eine XML-Syntax.

In dieser Arbeit wird skizziert, wie verschiedene grafische konkrete Syntaxen einer DSL dazu genutzt werden können, verschiedene Benutzergruppen der DSL zu unterstützen (vgl. Abschnitt 11.1.3). Konkrete Syntaxen können verschiedenen Zwecken dienen, etwa der

Speicherung oder Übertragung eines Modells (z. B. in XML) oder dessen Visualisierung und effizienten Bearbeitung.

Die abstrakte Syntax einer DSL legt die grundlegenden Elemente der zu modellierenden Domäne sowie ihre Beziehungen fest und spiegelt somit die Konzepte der Domäne wieder. Die abstrakt Syntax ist ein Metamodell, das im Kontext einer domänenspezifischen Sprache auch als *Domain Definition Metamodel* (DDMM) bezeichnet wird [KBJV06]. Eine *konkrete Syntax* einer DSL *ist definiert durch eine (bidirektionale) Abbildung* zwischen dem DDMM und einem weiterem Metamodell, das aus Sicht des DDMMs die Rolle eines *Display Surface Metamodels* (DSMM) verkörpert. Ein DSMM kann z. B. SVG [Wor03] (für konkrete grafische Syntaxen) sein, aber auch eine Sprache, die nur eine reine Textsyntax besitzt. DDMM und DSMM sind in Metasprachen wie in MOF [Obj06a] oder KM3 [KBJV06] definiert.

Als Beispiel soll eine DSL $X$ betrachtet werden, die die Beschreibung von Szenengraphen mittels der Elemente Knoten und Kante erlaubt. Ein Konzept *Knoten* der DSL $X$ etwa kann durch eine konkrete Syntax der DSL auf ein Konzept *Kreis* einer weiteren DSL $Y$ abgebildet werden, die durch ein entsprechendes DSMM definiert ist.

Die Abbildung zwischen dem DDMM und einem DSMM ist selbst ein Modell; sie ist in einer wohldefinierten Sprache, z.B. TCS [JBK06], geschrieben.

### 3.5.2 Semantik

Ein häufig auftretender Aspekt der Semantik einer DSL ist – neben anderen Aspekten – ihre Ausführungssemantik. Diese Semantik wird oft durch eine Transformation definiert, die Elemente der DSL auf Elemente einer weiteren Sprache (DSL oder GPL) abbildet, die ihrerseits präzise Ausführungsvorschriften festlegt. Mit Hilfe dieses Transformationsmodells kann ein zu einem DDMM konformes Modell in eine ausführbare Form (z. B. eine in einem Browser *interpretierbare* VRML-Welt oder ein *kompilierbares* und ausführbares Java-Programm) übertragen werden.

Die Transformation selbst kann wiederum in einer anderen (domänenspezifischen) Sprache verfasst sein, beispielsweise in ATL [JAB+06] oder XSLT [Wor07b], wobei die letztgenannte Sprache erlaubt, XML-codierte Modelle in ein ausführbares Format (z. B. Java-Programm-Quellcode) zu übertragen.

Bei der Übersetzung eines in einer DSL geschriebenen Modells in eine ausführbare Form wird zwischen der *Generierung* und der *Kompilierung* unterschieden. Bei der *Kompilierung* ist das Transformationsmodell fest in einen Compiler integriert. Es kann dabei aus dem Modell sofort ein lauffähiges Programm erzeugt werden (z. B. Java-Bytecode). Bei der *Generierung* wird das Transformationsmodell durch ein (oder mehrere miteinander verknüpfte) *Templates* realisiert (z. B. XSLT-Stylesheets), die von einem Generator zur Generierung des Zielformats verwendet werden. Die Generierung ist damit zwar flexibler als die Kompilierung, da die Templates ausgetauscht werden können, erfordert aber i. d. R. eine zusätzliche Kompilierung des erzeugten Zielformats. Mit der letztgenannten Methode sind also im Normalfall mehrere Schritte notwendig, um ein Quellmodell in ein ausführbares Programm zu transformieren.

Die prinzipiellen Möglichkeiten der Modell-zu-*Code*-Abbildung sind Thema des folgenden Abschnitts, weil diese Abbildung im Kontext der vorliegenden Arbeit eine wichtige Rolle einnimmt (vgl. z. B. Abschnitt 5.6).

## 3.6 Modell-zu-Code-Transformationen

Basierend auf einem Transformationsmodell *MT* (kurz: einer Transformation MT), wird ein Quell-Modell *MQ* in ein Ziel-Modell *MZ* übersetzt. Die Transformation MT ist dabei in einer GPL oder einer Transformationssprache (einer DSL, wie XSLT oder ATL, die der Definition von Transformationen dient) geschrieben. Eine besondere Art der Modell-zu-Modell-Transformation stellt die Model-zu-Code-Transformation dar [CH03]. Bei dieser Transformation wird das Zielmodell in einer konkreten Text[3]- oder sogar Binärcode-repräsentation persistent abgelegt. Die Modell-zu-Code-Transformation verfolgt – wie oben bereits angedeutet – häufig das Ziel, das Modell in eine ausführbare Form zu übersetzen.

Eine Möglichkeit, Code aus einem Modell zu generieren, sind so genannte besucherbasierte Ansätze, die auf dem Besucher-Entwurfsmuster (Visitor Pattern, vgl. [GHJV94]) fußen. Bei einem solchen Ansatz wird die interne Repräsentation eines Modells (z. B. ein abstrakter Syntaxbaum bei einem Modell aus dem EBNF-TS) traversiert, wobei Code in einen Ausgabestrom geschrieben wird. Diese Vorgehensweise wird auch bei Compilern angewendet. Sie eignet sich somit im Gegensatz zu dem nachfolgend vorgestellten Ansatz auch für die Erzeugung von Binärcode.

Bei einer templatebasierten Methode spezifizieren Templates die Abbildungsregeln für Elemente der Quellsprache auf der so genannten linken Seite (*Left-Hand-Side*, kurz *LHS*) auf Elemente der Zielsprache auf der so genannten rechten Seite (*Right-Hand-Side*, kurz *RHS*). Auf der LHS enthält ein Template eine ausführbare Logik, um auf Elemente des Quellmodells zuzugreifen, wie Java-Code, der den Zugriff auf die interne Repräsentation des Quellmodells realisiert, oder deklarative Abfragen, z. B. in OCL [Obj06b] oder XPath [Wor07a]. Auf der RHS enthält das Template den zu einem Quellelement korrespondierenden Code der Zielsprache, der Markierungen enthält, die durch Informationen aus dem Quellmodell ersetzt werden. Die Markierungen selbst beinhalten ausführbare Anweisungen, etwa um gezielt bestimmte Daten des Quellelements zu selektieren oder den Zielcode iterativ zu erweitern. Template-Ansätze erlauben es i. d. R. mehrere Templates miteinander zu verknüpfen, so dass Abbildungsregeln sich über verschiedene Templates verteilen können.

Im Rahmen der vorliegenden Arbeit wurden insbesondere templatebasierte Methoden der Code-Generierung verwendet (vgl. bspw. Abschnitt 5.6).

## 3.7 Domänenspezifische Entwicklung von Software

Allgemein lässt sich von modellgetriebener Softwareentwicklung (*Model Driven Software Development – MDSD*) oder kurz modellgetriebener Entwicklung (*Model Driven Develop-*

[3]Bei einer solchen Transformation spricht man auch von einer Modell-zu-Text-Transformation

*ment – MDD*) sprechen, wenn Software teilweise oder vollständig aus Modellen generiert wird [SV05]. Domänenspezifische Software-Entwicklung (*Domain Specific Development – DSD*, auch *Domain Specific Modeling – DSM*) stellt einen speziellen Ansatz der modellgetriebenen Entwicklung dar.

Domänenspezifische Entwicklung ist eine Methode, Probleme zu lösen, die innerhalb der Zieldomäne immer wieder auftreten [CJKW07]. Oft ist es sinnvoll, nur bestimmte Teile einer Ziel-Domäne mittels einer DSL zu beschreiben. Probleme, die in verschiedenen Situationen in der Domäne auftreten, besitzen i. d. R. Aspekte, die sich voneinander unterscheiden und Aspekte, die immer wiederkehren. Letztere Aspekte stellen den unveränderbaren Teil der Lösung eines Problems dar. Dieser unveränderbare Teil, der *Basisteil*, kann je nach Art und Umfang der zu lösenden Probleme z. B. *Plattform*, *Framework* (vgl. etwa das *IntegratorAR*-Framework in Abschnitt 9.5.2), *API* oder *Interpreter* (wie der in Abschnitt 6.4 vorgestellte Interpreter) genannt werden [CJKW07]. Er wird häufig mit Hilfe traditioneller Software-Entwicklungsmethoden erstellt. Mittels der DSL kann dann der veränderliche Teil einer Problemlösung beschrieben werden. Dieser Teil wird auch als so genannter *Konfigurationscode* bezeichnet, der auf dem Basisteil aufsetzt. Der Konfigurationscode kann aus einem in einer DSL geschriebenen Modell generiert werden. Java-Klassen, die – wie in Abschnitt 5.6.2 beschrieben – Komponenten eines entsprechenden Basisteils nutzen, oder XML-codierte Graphen, die durch einen Interpreter traversiert und ausgewertet werden (s. Abschnitt 6.4), sind Beispiele für Konfigurationscode.

Ein wichtiger Grundsatz der domänenspezifischen Entwicklung ist es also, bereits frühzeitig Ressourcen in die Festlegung einer DSL zu investieren (z. B. durch die Definition eines XML-Schemas für eine XML-basierte DSL), von deren Nutzung man später profitieren kann.

Es kann vorkommen, dass eine einzelne DSL nicht ausreichend oder ungeeignet ist, um verschiedene Probleme einer Ziel-Domäne vollständig abzudecken. Eine Lösung ist es, mehrere DSLs zu definieren, die miteinander kombiniert werden können. Um dies zu gewährleisten, ist es sinnvoll, die DSLs in einer gemeinsamen Metasprache zu definieren. Beispielsweise sind die in den Kapiteln 5 bis 9 dieser Arbeit vorgestellten Sprachen miteinander kombinierbar, da sie in der MOF-Sprache definiert wurden.

## 3.8 Wichtige Initiativen und Werkzeuge zur modellgetriebenen und domänenspezifischen Entwicklung

### 3.8.1 Model Driven Architecture

Die *Model Driven Architecture* (*MDA*) ist eine Initiative der OMG. Die drei Hauptziele der MDA sind Portabilität, Interoperabilität und Wiederverwendbarkeit von Software durch die architektonische Trennung von Aspekten. Modelle stellen dabei die zentralen Elemente im Entwicklungsprozess dar. Typische Zielplattformen der MDA sind Java EE [Sun06], Microsoft .NET [Dot] und die Common Object Request Broker Architecture (CORBA) [Obj08]. Um die genannten Ziele zu erreichen, kombiniert die MDA verschiedene

OMG-Standards, wie MOF [Obj06a], QVT [Obj07a], UML [Obj07c], UML Profiles[4], OCL [Obj06b]und XMI [Obj07b].

Im Kontext der MDA gibt es unterschiedliche Arten von Modellen: Neben den fachlichen Domänen-Modellen (*Computation Independent Models – CIM*s) und Plattform-Modellen spielen so genannte plattformunabhängige Modelle (*Plattform Independent Models – PIM*s) und plattformspezifische Modelle (*Platform Specific Models – PSM*s) wichtige Rollen.

Die Anforderungen für das zu bauende System werden in einem CIM auf fachlicher Ebene modelliert. Dieses Modell enthält i. d. R. keinerlei Details, die sich auf die Art und Weise der Implementierung des Systems beziehen.

Das PIM repräsentiert das zu erstellende System auf einem hohen Abstraktionsniveau. Es beinhaltet keine Informationen über die Plattform, mit der das System realisiert werden soll. Das PIM stellt ein entscheidendes Konzept dar, um die Ziele Portabilität und Wiederverwendbarkeit zu erreichen.

Ein Plattformmodell bildet eine Plattform ab. Ein Plattformmodell könnte z. B. ein UML-Modell der Java-Plattform sein, das entsprechende UML-Repräsentationen der Java-Klassen (z. B. Swing-Klassen) und deren Assoziationen enthält.

Ein PSM stellt dasselbe System wie das entsprechende PIM dar, allerdings auf einem niedrigeren Abstraktionsniveau. Es beinhaltet zusätzlich plattformspezifische Details und beschreibt, wie das System die Zielplattform benutzt. Ein PSM ist eine geeignete Quelle für die Generierung von Code.

Die Abbildung eines PIMs auf ein PSM erfolgt über Transformationen. Eine Transformation kann manuell oder (semi-)automatisch erfolgen. Es wird dabei zwischen Modell-Typ-Abbildungen (*Model Type Mappings*) und Modell-Instanz-Abbildungen (*Model Instance Mappings*) unterschieden. Eine Kombination der Abbildungsarten ist möglich. Bei einem Model Type Mapping wird die Abbildung der Elemente eines PIMs auf Elemente eines PSMs abhängig von den Elementtypen vorgenommen. Dabei können Attributwerte der jeweiligen Elemente berücksichtigt werden. Ein Beispiel für ein Quellmodell einer Modell-Typ-Abbildung wäre ein PIM, das Szenengraphen auf einem hohen Abstraktionsniveau modelliert. In diesem PIM könnte es Elemente eines Typs `SceneGraphNode` mit einem Attribut `nodeKind` geben. Um ein entsprechendes PSM für die Java3D [J3D]-Plattform zu erzeugen, würde ein `SceneGraphNode`-Element im PIM auf ein Java3D-`Group`-Element im PSM abgebildet, sofern `nodeKind` den Wert `GROUP` hätte und auf ein Java3D-`TransformGroup`-Element, sofern `nodeKind` den Wert `TRANSFORM` besäße.

Bei einer Modell-Instanz-Abbildung werden konkrete Elemente eines PIMs auf entsprechende konkrete Elemente eines PSMs abgebildet. Die zu transformierenden Elemente müssen dafür explizit mit Markierungen versehen werden. Die Markierungen können in einem UML-Modell z.B. Stereotypen eines UML-Profiles sein. Die Markierungen sind nicht Teil des PIMs; sie sind abhängig von der konkreten Zielplattform. Modellmarkierungen

[4]UML-Profiles sind ein leichtgewichtiger Mechanismus zur Erweiterung des UML-Metamodells. Ein UML-Profile enthält Stereotypen (*Stereotypes*), Eigenschaftswerte (*Tagged Values*) und Zusicherungen (*Constraints*). Stereotypen erweitern Elemente des UML-Metamodells. Für sie können bestimmte Eigenschaften als *Tagged Values* definiert und durch *Constraints* eingeschränkt werden.

lassen sich als Markierungen vorstellen, die sich auf einer transparenten Ebene befinden, die über das Modell gelegt wird [Obj03].

Eine (semi-)automatische Transformation kann unter Verwendung von Templates erfolgen. Templates sind parametrisierbare Modelle, die auch in Transformationsregeln verwendet werden können (vgl. auch Abschnitt 3.6). Einem Template kann eine Menge von Markierungen zugeordnet werden, um Elemente im Quellmodell zu identifizieren, die mit dem Template zu transformieren sind. Ebenso können Markierungen benutzt werden, um die Parameter des Templates mit Werten zu belegen. Damit können Werte aus dem Quellmodell in das Zielmodell (ggf. auch modifziert) übertragen werden.

Ein *Record of Transformation* speichert die bei einer Modellabbildung angewandten Transformationsregeln. Er kann damit als Grundlage dienen, um PIM und PSM bei späteren Änderungen zu synchronisieren.

Modell-Abbildungen können in einer Transformationssprache geschrieben werden. Im Rahmen der MDA wurde dafür die Sprache QVT (Query/View/Transformation) spezifiziert. Eine bekannte realisierte QVT-ähnliche Transformationssprache ist die Atlas Transformation Language (ATL).

Der Prozess der Transformation eines PIM in ein PSM kann in mehreren Stufen, also über mehrere Zwischenmodelle mit fallendem Abstraktionsniveau, erfolgen. Neben der naheliegenden Möglichkeit, ein PSM am Ende der Transformationskette in Code zu übersetzen, besteht auch die Option, ein PIM direkt in Code zu überführen [Obj03]. Letztgenannte Variante wurde im Rahmen der vorliegenden Arbeit angewandt (vgl. Abschnitt 4.2).

Als Modellierungssprache wird im Rahmen der MDA – wie bereits erwähnt – vor allem UML in Verbindung mit UML-Profiles eingesetzt. Zur Erstellung der Modelle (PSM, PIM, Plattform-Modelle) bieten sich daher insbesondere UML-Werkzeuge an, z. B. im Zusammenspiel mit leistungsfähigen Generatoren und Generatorframeworks wie *AndroMDA* [Bre04]. Prinzipiell können auch eigene Metamodelle mittels der MOF definiert werden, allerdings ist die existente Werkzeugunterstützung, z. B. um selbst konkrete Syntaxen für entwickelte Modellierungssprachen zu definieren, hier eher gering. Zumindest eine Zusammenarbeit mit MDA-Standards, z. B. durch den Import von (Meta-)Modellen im XML Metadata Interchange (XMI)-Format, erlaubt das Eclipse *Graphical Modeling Framework* (*GMF*) [GMF]. Das GMF ermöglicht in Verbindung mit dem *Eclipse Modeling Framework* (*EMF*) [EMF], basierend auf Ecore, einem MOF-ähnlichen Metametamodell, die Erstellung eigener graphischer Modellierungssprachen und Editoren. Auch Modell-Transformationen werden vom GMF unterstützt.

### 3.8.2 Software Factories

Bei der Software Factories-Initiative [GS03, GSCK04] von Microsoft liegt der Fokus im Unterschied zur MDA weniger auf Plattformunabhängigkeit, sondern eher auf einer hohen Produktivität. Ebenso geht es weniger um die Erstellung einzelner Software-Systeme, sondern um die Erschaffung ganzer Software-Produktfamilien. Dies soll erreicht werden durch die Komposition, Konfiguration und Anpassung von Framework-Komponenten, um Familien ähnlicher, im Detail aber doch unterschiedlicher Software-Produkte, wie sie inner-

halb einer Domäne häufig vorkommen, schnell entwickeln zu können. Beispiele für solche Produkt-Familien im Kontext der vorliegenden Arbeit wären 3D-Instruktoren (s. Kapitel 6) oder AR-Anwendungen im Bereich Montage und Wartung (vgl. Kapitel 9). Benötigte Komponenten und Frameworks werden zunächst auf Basis von gesammelten Erfahrungen entwickelt; sie enthalten Expertenwissen z. B. in Form von angewandten domänenspezifischen (Architektur-)Mustern. DSLs und DSL-Werkzeuge wie Editoren, Transformatoren und Compiler helfen, so genannten Konfigurationscode (vgl. Abschnitt 3.7) zu erzeugen, der auf dem Framework aufsetzt und dieses zu einem lauffähigen System (einem Mitglied der Software-Familie) komplettiert. Ein solches Vorgehen wurde teilweise auch im Rahmen dieser Arbeit angewandt (vgl. z. B. Abschnitt 5.6.2). (Visuelle) DSL-basierte Modelle spielen eine wichtige Rolle im Software-Entwicklungsprozess, z. B. als Quellmodelle für die Generierung von Framework-Konfigurationscode. Durch Änderung der Modelle und die erneute Generierung kann der Konfigurationscode schnell angepasst werden.

Eine *Software-Fabrik* (*Software Factory* – SF) verbindet drei Kernkomponenten: Das *SF-Schema*, das *SF-Template* und die erweiterbare Software-Entwicklungsumgebung (*Integrated Software Development Environment – IDE*). Ein SF-Schema spezifiziert Elemente wie Projekte, Konfigurationsdateien, zu nutzende DSLs, Modell-Transformationen und Frameworks, die zur Erstellung einer Produktlinie nötig sind, und beschreibt, wie diese zu kombinieren und zu verwenden sind. Ein SF-Template ist die Sammlung der Elemente, die im SF-Schema spezifiziert wurden. Eine erweiterbare IDE kann mit dem SF-Template konfiguriert werden und wird dadurch zu einer Software-Fabrik, die der Erstellung von Mitgliedern der anvisierten Produktfamilie dient. Nach der Erstellung einer SF gesammeltes Wissen kann auch nachträglich noch in die SF integriert werden. Ebenso kann eine SF an Drittfirmen weitergegeben werden, die ihre eigenen Erweiterungen hinzufügen können.

Benutzer der SF müssen nur die Anwendungsteile entwickeln, in denen sich die Mitglieder der Produktfamilie unterscheiden. Das gesamte Software-Produkt wird von der SF durch Kombination aller Anwendungsbestandteile automatisch generiert.

Einen wichtigen Teil der SF-Initiative verkörpern die DSL-Tools [CJKW07], die eine zum GMF (siehe oben) vergleichbare Funktionalität aufweisen, aber nicht kompatibel mit den OMG-Standards sind.

## 3.9 Modellgetriebene Entwicklung im Kontext der vorliegenden Arbeit

In dieser Arbeit wird ein domänenspezifischer Ansatz vorgestellt. Bei den in den Kapiteln 5 bis 9 beschriebenen DSLs handelt es sich um visuelle Modellierungssprachen, die Hilfestellung bei der Lösung von Problemen aus dem Bereich der 3D-Anwendungsentwicklung geben sollen. Die abstrakten Syntaxen der Sprachen wurden als MOF-konforme Metamodelle definiert. Die in den Sprachen geschriebenen Modelle können in einer XML-Repräsentation gespeichert werden. Die vorgestellten Sprachen besitzen zudem eine Ausführungssemantik, die durch XSLT-Stylesheets definiert ist. Mittels der Stylesheets wird aus den XML-

codierten Modellen kompilierbarer oder auch – wie in Kapitel 6 beschrieben – interpretierbarer Quellcode generiert.

Der hier vorgestellte Modellierungsansatz orientiert sich z. T. an der MDA. Bspw. spielen MDA-relevante Aspekte, wie die Unabhängigkeit erstellter Modelle von einer konkreten Zielplattform und die Verwendung eines MOF-konformen Metamodells, das (manuell) auf ein UML-Profile abgebildet wurde, wichtige Rollen. Berücksichtigt wurden allerdings auch Aspekte der SFs, wie die schnelle Entwicklung von domänenspezifischen Anwendungen und die automatische Erzeugung von – im Vergleich zum Umfang des Gesamtsystems – kleineren Mengen an Code, welcher Komponenten eines zuvor erstellten domänenspezifischen Frameworks (vgl. z. B. Abschnitte 9.5.2, 11.3) nutzt.

Festzustellen ist, dass die aus dem MOF-konformen Metamodell manuell generierten UML-Profiles die Defintion der Sprachen nur unvollständig umsetzen, da nicht alle Zusicherungen (*Constraints*) der Metamodelle abgedeckt werden können und zudem die Notation der Elemente eines UML-Profiles nur in Grenzen anpassbar ist. Die Verwendung von Software-Fabriken bzw. DSL-Toolkits (wie dem GMF oder den DSL Tools) zur Definition einer DSL (etwa der in dieser Arbeit vorgestellten neuen Sprache SSIML) und zur Realisierung der entsprechenden DSL-Werkzeuge können diesbezüglich deutliche Verbesserungen bewirken.

# Kapitel 4

# Der Ansatz SSIML im Überblick

Wie bereits in Kapitel 1 erwähnt, stellt die im Rahmen dieser Arbeit entwickelte SSIML-Sprachfamilie einen Ansatz dar, um die bei der Entwicklung interaktiver 3D-Anwendungen auftretenden Probleme zu überwinden. Die SSIML-Sprachen sind also domänenspezifische Sprachen (DSLs), die für den Einsatz in der Domäne der 3D-Anwendungsentwicklung konzipiert sind. Dieses Kapitel gibt zunächst einen Überblick über die Mitglieder der SSIML-Sprachfamilie und ihre Zusammenhänge, bevor in den folgenden Kapiteln die einzelnen Sprachkomponenten genauer betrachtet werden.

## 4.1 Komponenten der SSIML-Sprachfamilie

Abbildung 4.1 zeigt die Komponenten der SSIML-Sprachfamilie. Jede Komponente – d.h. jede Teilsprache der Sprachfamilie – ist in einem eigenen Paket definiert, das ein entsprechendes MOF [Obj06a]-konformes Metamodell enthält. Pakete, die SSIML-Sprachen definieren, sind `core`, `tasks`, `behaviour`, `components` und `ar`. Die jeweils in diesen Paketen definierten Sprachen sind *SSIML* (im Paket `core`), *SSIML/Tasks*, *SSIML/Behaviour*, *SSIML/Components* und *SSIML/AR*.

Die *Scene Structure and Integration Modelling Language* (SSIML), die als Basissprache gleichzeitig der SSIML-Sprachfamilie ihren Namen gibt, stellt grundlegende Elemente zur Beschreibung der Strukturen von 3D-Inhalten und von Verknüpfungen der 3D-Szene mit Bestandteilen der Rahmenanwendung zur Verfügung.

Alle anderen Sprachen bauen auf SSIML auf und können daher als SSIML-Erweiterungen betrachtet werden. *SSIML/Tasks* dient zur Realisierung aufgaben-zentrierter 3D-Anwendungen, wie z. B. interaktiven 3D-Anleitungen im Bereich Montage und Wartung. Die Komponente *SSIML/Behaviour* stellt Elemente zur Beschreibung von Animationen und Verhalten bereit. Die Sprache *SSIML/Components* enthält Elemente zur Beschreibung von 3D-Komponenten. Sie erlaubt damit eine verbesserte Strukturierung der 3D-Inhalte. Nicht zuletzt soll die *SSIML/AR*-Komponente angeführt werden, die auf die Entwicklung von Augmented Reality - Anwendungen ausgerichtet ist.

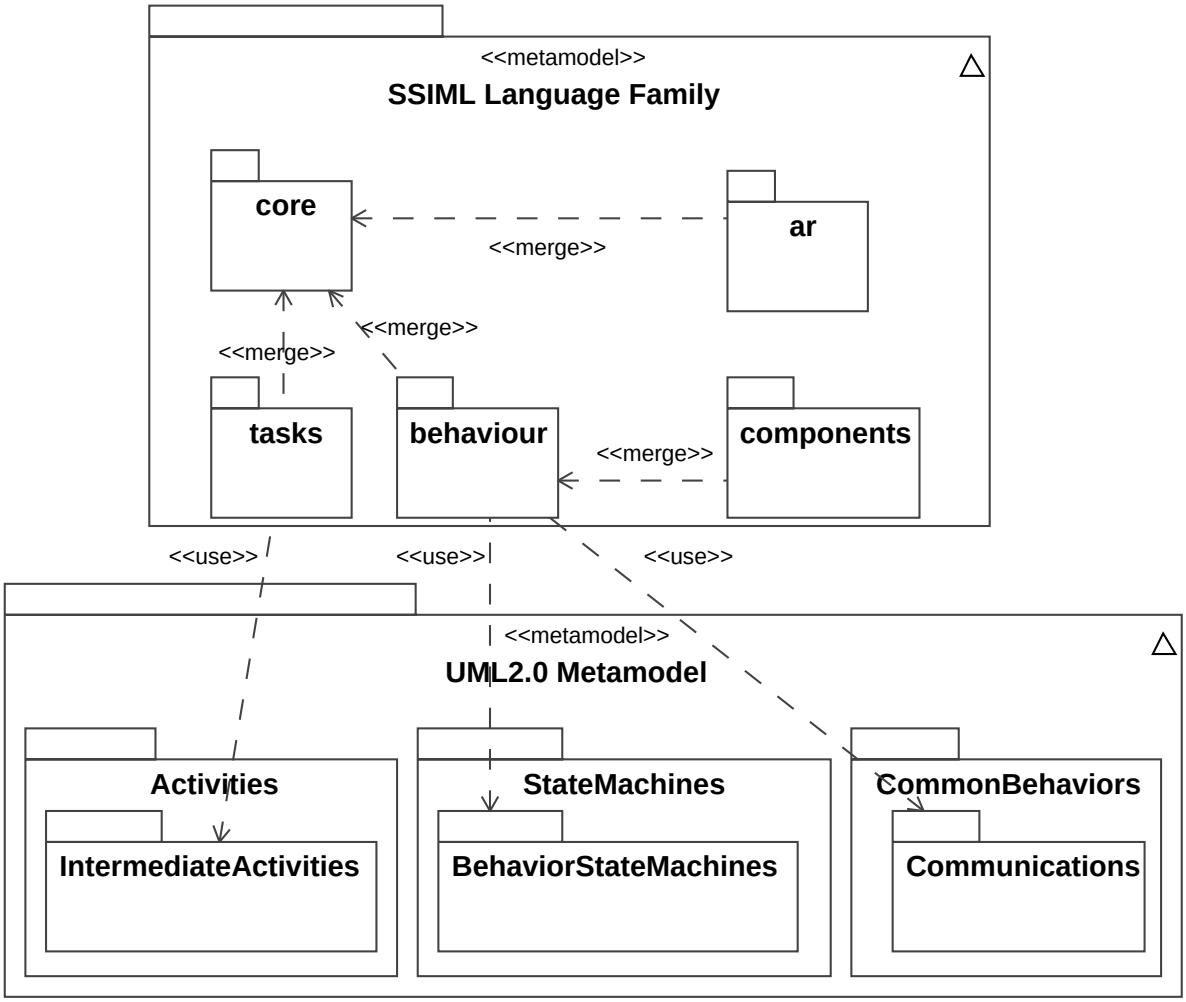



Abbildung 4.1: Komponenten der SSIML-Sprachfamilie

Wie Abbildung 4.1 zeigt, verwendet SSIML/Components neben Elementen des `core`-Paketes auch Elemente aus SSIML/Behaviour. Um Spracherweiterungen der SSIML-Kernkomponente zu definieren, wurden die Inhalte der entsprechenden Pakete mittels *Package-Merge* kombiniert bzw. verschmolzen. Bspw. beschreibt das mit dem `core`-Paket verschmolzene `ar`-Paket das Metamodell der Sprache SSIML/AR. Durch die Paketverschmelzung existieren im Paket `ar` damit alle Elemente der Pakete `core` und `ar`. Für eine detaillierte Beschreibung der Package-Merge-Semantik sei auf [Obj07c] verwiesen.

Alle Teilsprachen sind miteinander kombinierbar. Um etwa ein Metamodell mit allen SSIML/AR und SSIML/Tasks-Elementen zu erhalten, müssen beide Sprachfamilienkomponenten in einem (weiteren) Paket per Package-Merge kombiniert werden. Diese Möglichkeit wurde u. a. im Beispielszenario in Kapitel 9 genutzt.

Ein SSIML-Modellierungswerkzeug kann also entweder genau eine Komponente – nämlich die `core`-Komponente – oder mehrere kombinierte Komponenten der SSIML-

Sprachfamilie unterstützen. In dem in Abschnitt 11.3 beschriebenen Szenario etwa wurde ein Werkzeug verwendet, mit dem sich Modelle erstellen lassen, die konform zu dem durch die Verschmelzung der Pakete `ar` und `tasks` (die ihrerseits durch Verschmelzung mit dem `core`-Paket hervorgegangen sind) entstandenen Metamodell sind.

Jedes Modell, welches mit einer SSIML-Sprache oder einer Kombination von SSIML-Sprachen beschrieben ist, besitzt eine *Containment-Hierarchie*. Wurzel dieser Hierarchie ist immer ein Element vom Typ `SSIMLModel`. Dafür definiert jede SSIML-Teilsprache die Metaklasse `SSIMLModel`, die zumindest mit allen anderen nicht-abstrakten Metaklassen (direkt oder indirekt) über Kompositionsbeziehungen verknüpft ist. Bei einem Package-Merge werden auch die `SSIMLModel`-Elemente der zu kombinierenden Quell-Metamodelle verschmolzen. Das resultierende Metamodell enthält somit wiederum eine Meta-Klasse `SSIMLModel`, deren Instanz auf Modellebene alle anderen Elemente enthält.

Einige SSIML-Komponenten verwenden Elemente des UML 2-Meta-Modells (s. Abbildung 4.1). Bspw. benutzt die SSIML/Behaviour-Komponente UML 2-Zustandsautomaten.

In den folgenden Kapiteln werden die einzelnen Komponenten der SSIML-Sprachfamilie genauer beschrieben: in Kapitel 5 wird die Basissprache SSIML betrachtet, Kapitel 6 präsentiert die Sprache SSIML/Tasks, Kapitel 7 stellt SSIML/Behaviour vor, Kapitel 8 führt in SSIML/Components ein und Kapitel 9 beschreibt die Sprache SSIML/AR.

## 4.2 Abbildung von SSIML-Modellen auf Code

Da SSIML-Modelle plattformunabhängig sind, besteht eine Vielzahl von Möglichkeiten, sie in plattformspezifischen Code zu übersetzen (verschiedene Programmiersprachen und 3D-Formate, verschiedene Software-Architekturen).

In dieser Arbeit dient X3D (s. Abschnitt 2.5) als Beispielformat, um die Machbarkeit und die Möglichkeiten der Codegenerierung aus visuellen Software-Modellen im Bereich 3D zu demonstrieren. Die Vorteile von X3D sind hierbei, dass es durch seine Spezifikation hervorragend dokumentiert ist, dass kostenfrei verfügbare, erweiterbare X3D-Browser wie Xj3D [Xj3] existieren, mit denen sich 3D-Szenen testen lassen, und dass es sich durch seine Textrepräsentation sehr gut als Zielformat für eine templatebasierte Codegenerierung eignet (vgl. Abschnitt 3.6). Der in den nachfolgenden Kapiteln präsentierte Beispielcode verwendet X3D sowohl in der XML- als auch der klassischen VRML-Syntax, die als zwei konkrete Syntaxen ein- und derselben Sprache völlig gleichwertig eingesetzt werden können (s. Abschnitt 2.5).

Weiterhin ist es leicht möglich, X3D durch seine Nähe zu VRML97 – sofern eine VRML97-kompatible Untermenge von X3D-Elementen genutzt wird – ins VRML97-Format zu konvertieren und umgekehrt. Damit können auch (freie) 3D-Autorenwerkzeuge und 3D-APIs in Verbindung mit X3D verwendet werden, die nur den länger etablierten VRML-Standard unterstützen. Trotzdem soll darauf hingewiesen werden, dass X3D im Kontext der vorliegenden Arbeit stellvertretend für andere 3D-Formate wie U3D steht, deren Verwendung prinzipiell ebenfalls möglich wäre.

Neben X3D-Code zur deklarativen Beschreibung von 3D-Szenen kann auch Programmcode aus SSIML-Modellen erzeugt werden, z. B. zur Realisierung von spezifizierten Szenenzugriffen (vgl. Abschnitt 5.6) oder zur Implementierung des Verhaltens von 3D-Objekten in Skriptknoten (vgl. Abschnitt 7.3). Als Zielsprache für die Erzeugung von Programmcode wurde beispielhaft Java gewählt, da es sich besonders zur Einbindung von X3D-Szenen und -Modellen in Applikationen eignet. Das Toolkit Xj3D etwa bietet eine reine Java-Implementierung eines X3D-Browsers an und erlaubt die Integration von X3D-Szenen in eigene Anwendungen. Die Implementierung von Skriptknoten und der Szenenzugriff über das SAI werden von Xj3D entsprechend der Java-Sprachbindungsspezifikation [Int05d] unterstützt. Eine weitere Möglichkeit, die sich durch die Benutzung von Java ergibt, ist, X3D- und auch VRML-Dateien über so genannte `Loader` in Java3D-Szenengraphen umzuwandeln, deren Zustände zur Laufzeit manipuliert und abgefragt werden können.

Besonders nutzbringend ist die Abbildung von SSIML-Modellen auf Code dann, wenn sie automatisch erfolgt. So konnten Werkzeuge zur automatischen Generierung von Code aus SSIML/AR und SSIML/Tasks-Modellen bereits erfolgreich in einem Teamprojekt eingesetzt werden (s. Abschnitt 11.3). Codegeneratoren für andere SSIML-Komponenten wurden nur bis zu einer experimentellen Stufe entwickelt, um die – in den folgenden Kapiteln anhand von Beispielen vorgestellten – Regeln und Konzepte für die Modell-zu-Code-Übersetzung zu validieren.

## 4.3 Designkriterien für Sprachen der SSIML-Familie

Den Rahmen für die Designentscheidungen bei der Entwicklung von SSIML gaben die in Abschnitt 1.1 aufgelisteten Zielstellungen vor. Um eine effizientere Kommunikation zwischen 3D-Designern und Programmierern zu erreichen, sollte die Sprachnotation leicht verständlich bzw. erlernbar sein. SSIML orientiert sich daher an dem für 3D-Entwickler bekannten Konzept der Szenengraphen. Das vergleichsweise hohe Abstraktionsniveau der SSIML-Sprachen sorgt für ihre Eignung zum Anwendungsentwurf. Es lassen sich somit (semi-)formale Spezifikationen von Anwendungen vor der Implementierung erstellen, wodurch ein strukturierter Entwicklungsprozess gefördert wird. Zur Strukturierung der Entwicklung trägt auch die automatische Generierung von Code aus SSIML-Modellen bei, die einen nahtlosen Übergang vom Entwurf zur Implementierung ermöglicht. Dadurch, dass Codebestandteile aus einem Modell gleichzeitig für die verschiedenen Entwicklergruppen erzeugt werden können, wird zum einen die Konsistenz der durch die verschiedenen Entwickler zu bearbeitenden Codebestandteile gewährleistet; zum anderen wird auch eine Parallelisierung der Implementierung unterstützt. Näheres zum SSIML-Entwicklungsprozess ist in Kapitel 10 zu finden.

Für die Spezifikation einer 3D-(Teil-)Szene spielt das Konzept des SSIML-Objekts eine wichtige Rolle. Ein solches Objekt wird als abgeschlossene „atomare“ Einheit betrachtet; ihr interner Aufbau wird nicht berücksichtigt. Ein Objekt wird durch ein entsprechendes Modellelement in SSIML repräsentiert. Eine SSIML-Modell sollte dabei genau die Objekte enthalten, die im Kontext der Anwendung von Bedeutung sind und separat adressierbar

sein müssen, etwa um einzeln bewegt werden zu können. Als Beispiel lässt sich ein Roboter mit einen Greifarm betrachten, der als SSIML-Modell spezifiziert werden soll: Soll der Geifarm des Roboters sich bewegen, werden sowohl der Roboter-Rumpf als auch der mit dem Rumpf verbundene Greifarm als separate Objekte benötigt. Soll sich der Roboter nur als Gesamteinheit bewegen können, reicht es aus, wenn nur ein Objekt für den Roboter einschließlich seines Greifarms definiert wird. Um ein angemessenes Abstraktionsniveau zu erreichen, werden Details der Realisierung, wie etwa die Geometriebeschreibungen der Objekte auf Polygonebene, in einem SSIML-Modell nicht spezifiziert. Ebenso fließen keine Informationen, die sich speziell auf die Zielplattform beziehen, in das Modell ein. Am Beispiel des Roboters wird das – im Vergleich zu einer Beschreibung der 3D-Inhalte in einem konkreten Szenengraphformat wie X3D – hohe Abstraktionsniveau von SSIML deutlich; es müssen lediglich maximal zwei miteinander verbundene Objekt-Knoten (z. B. `robotBody` und `robotArm`) modelliert werden. Die Festlegung der Einzelheiten des konkreten Erscheinungsbildes des Roboters kann nach der Generierung von Code aus dem Modell auf der Implementierungsebene erfolgen.

Die Modellierung von Bestandteilen der Rahmenanwendung orientiert sich an der UML-Notation für Klassen. Die Nutzung einer verbreiteten und bekannten Notation ist wiederum vorteilhaft für den Entwickler, da der Aufwand zum Erlernen neuer Notationselemente so z. T. entfallen kann. In Abschnitt 5.3.1 werden Konzepte zur Übertragung von Klassen aus vorhandenen UML-Modellen der Rahmenanwendung in ein SSIML-Modell vorgestellt. Beziehungen zwischen Bestandteilen der Rahmenanwendung und 3D-Elementen werden als Kanten zwischen den Beziehungspartnern notiert. Dieses so genannte Interrelationenmodell stellt ein entscheidendes Konzept zur Sicherung der Konsistenz zwischen den aus dem Modell erzeugten Komponenten der 3D-Szene und der Rahmenanwendung dar.

Zur Verbesserung der Modularisierung und der Wiederverwendbarkeit von 3D-Modellen sowie zur Verbesserung der Verwaltbarkeit umfangreicherer SSIML-Modelle wurden entsprechende Konzepte eingeführt, mit denen sich Subgraphen eines SSIML-Modells in Elementen kapseln lassen, die selbst als eigenständige Einheiten in SSIML-Modellen wieder verwendet werden können (s. dazu auch Abschnitt 5.2.2 – *Composed Nodes* – und Kapitel 8).

Mit den SSIML-Sprachen lassen sich durchaus Anwendungen der „realen Welt“ modellieren, wie in den Abschnitten 11.2 und 11.3 beschrieben. SSIML wurde so konzipiert, dass es sich – abgesehen von den in Abschnitt 2.4 aufgeführten 3D-Anwendungsarten – für den Entwurf eines breiten Spektrums möglicher 3D-Anwendungen eignet. Konkrete, wichtige Designentscheidungen, etwa bei der Auswahl der Modellelemente, werden in den folgenden Kapiteln an entsprechenden Stellen beschrieben. SSIML ist *nicht* als vollständiger Ersatz für ein 3D-API zu sehen. Eher wird eine sinnvolle Aufteilung der Anteile, die aus dem Modell generierbar sind, und der Anteile, die manuell zu schreibenden Code darstellen, angestrebt. Diese sinnvolle Aufteilung wurde für die im Rahmen der vorliegenden Arbeit mit SSIML modellierten Projekte durchaus erreicht (vgl. dazu z. B. Abschnitt 11.3). Können große Teile der Anwendung aus den Modellen generiert werden, gewinnt der Entwickler Zeit, sich spezielleren, kreativeren Aufgaben zu widmen, wie der Implementierung von Spezialeffekten, z. B. Partikel- oder Morphing-Animationen.

Die Architektur der SSIML-Metamodelle erlaubt zudem eine einfache Erweiterung des Sprachumfangs durch zahlreiche abstrakte Meta-Klassen, von denen – falls notwendig – neue Elemente abgeleitet werden können.

## 4.4 Werkzeugunterstützung

Um Modelle zu erstellen und in Code zu übersetzen, wurden die SSIML-Sprachen (mit Ausnahme von SSIML/Components, s. unten) in ein UML-Werkzeug (NoMagic MagicDraw [Mag]) als UML-Profile, die weitestgehend den SSIML-Metamodellen entsprechen, integriert. Erstellte Modelle konnten so im XML-Metadata-Interchange-Format (XMI) gespeichert und mittels XSLT [Wor07b]-Stylesheets in den entsprechenden Code übertragen werden.

Es soll nicht argumentiert werden, dass diese Art und Weise der Erstellung und Konvertierung der Modelle die beste Möglichkeit darstellt. Der Ansatz wurde rein nach Gesichtspunkten der Praktikabilität gewählt, da sich damit vergleichsweise schnell ein *Proof-of-Concept* erbringen ließ. So musste zunächst kein spezielles Modellierungstool für SSIML von Grund auf neu entwickelt werden. Ebenso ist auf dieser Basis eine einfache Nutzung der von SSIML benötigten UML-Bestandteile (z. B. UML-Zustandsautomaten) möglich.

Eine Ausnahme bilden die in Kapitel 8 beschriebenen SSIML-Komponentenmodelle: Ihre Integration in UML-Werkzeuge in Form eines Profiles ist nicht sinnvoll, da die Anpassung der grafischen Notation eines Modellelements in einem UML-Profile beschränkt ist. Somit lassen sich SSIML-Komponenten aufgrund Ihrer besonderen Notation in UML-Werkzeugen nicht darstellen. Die Komponentenmodelle werden deshalb derzeit mittels eines Graphikeditors wie Microsoft Office Visio [Vis] erstellt. Der XSLT-basierte Codegenerator benötigt die Modellbeschreibung allerdings in einem XML-Format. Diese XML-Modellrepräsentation muss daher manuell (etwa mit Hilfe eines aus dem SSIML/Components-Metamodell automatisch generierten EMF [EMF]-Baumeditors) oder (semi-)automatisch (z. B. mittels der Transformation der in SVG [Wor03] codierten Modellvisualisierung) erzeugt werden. Eine weitere Variante stellt die manuelle Direktübersetzung der Modelle in Code dar. Allerdings sind alle genannten Vorgehensweisen sehr zweitaufwendig und fehleranfällig. Die Entwicklung eines spezialisierten Komponenteneditors mit angepassten Codegenerierungsmechanismen unter Benutzung von DSL-Werkzeugen wie dem Eclipse GMF [GMF] oder den Microsoft DSL Tools [CJKW07] würde hier Abhilfe schaffen.

Ein auf den Microsoft DSL Tools basierendes SSIML-Modellierungswerkzeug, das bislang jedoch nur die Elemente des SSIML-`core`-Paketes unterstützt, wurde in der Arbeit von Alzetta [Alz07] bereits prototypisch realisiert. Damit sollen neben der besseren Anpassbarkeit der Notation der Modellelemente auch einige weitere Nachteile kompensiert werden, die sich aus der Nutzung des UML-Profile-Mechanismus ergeben hatten (vgl. Abschnitt 3.9).

# Kapitel 5

# SSIML

## 5.1 Einführung

Wie bereits in Kapitel 1 herausgestellt wurde, stellt die Verknüpfung von Programmcode und 3D-Inhalten bei der Entwicklung von 3D-Anwendungen oft eine komplexe Aufgabe dar. Um den Entwickler bei der erfolgreichen Bearbeitung dieser Aufgabe zu unterstützen, wurde im Rahmen der vorliegenden Arbeit die grafische Modellierungssprache *SSIML* (*Scene Structure and Integration Modelling Language*) entwickelt, die in diesem Kapitel vorgestellt wird. Bei der Modellierung mit SSIML werden nachfolgende Hauptbestandteile einer 3D-Anwendung unterschieden:

- Die *Rahmenanwendung*, die die Anwendungslogik enthält,
- die *3D-Szene*, welche die 3D-Objekte beinhaltet und in eine Struktur einordnet, sowie
- *Beziehungen* zwischen der Rahmenanwendung und der 3D-Szene (Abbildung 5.1 oben).

Eine ähnliche Aufteilung einer 3D-Anwendung in *Subsysteme* wird von Foley et al. [FvFH95, S. 292 ff] beschrieben. Der Begriff *Subsystem* meint allerdings nicht, dass jedes Subsystem zwangsläufig in einem eigenen Programmmodul vorliegt:

> *The term subsystem does not imply major modules of code – a few calls or a procedure may be sufficient to implement a given subsystem.* [FvFH95, S. 292]

Im Gegensatz zu der oben getroffenen Unterscheidung von 3D-Anwendungskomponenten unterteilen Foley et al. die *Rahmenanwendung* in das *Front End*, dass die Darstellung der 3D-Szene und anderer Benutzungsschnittstellen-Elemente zur Aufgabe hat, den *Anwendungscode*, der keine direkten Beziehungen zur 3D-Szene aufweist, und so genannte *Reader* und *Writer*, die Daten der 3D-Szene lesen bzw. verändern können [FvFH95, S. 292 ff]. Die *Reader* und *Writer* entsprechen den oben angeführten *Beziehungen* zwischen Szene und Rahmenanwendung.

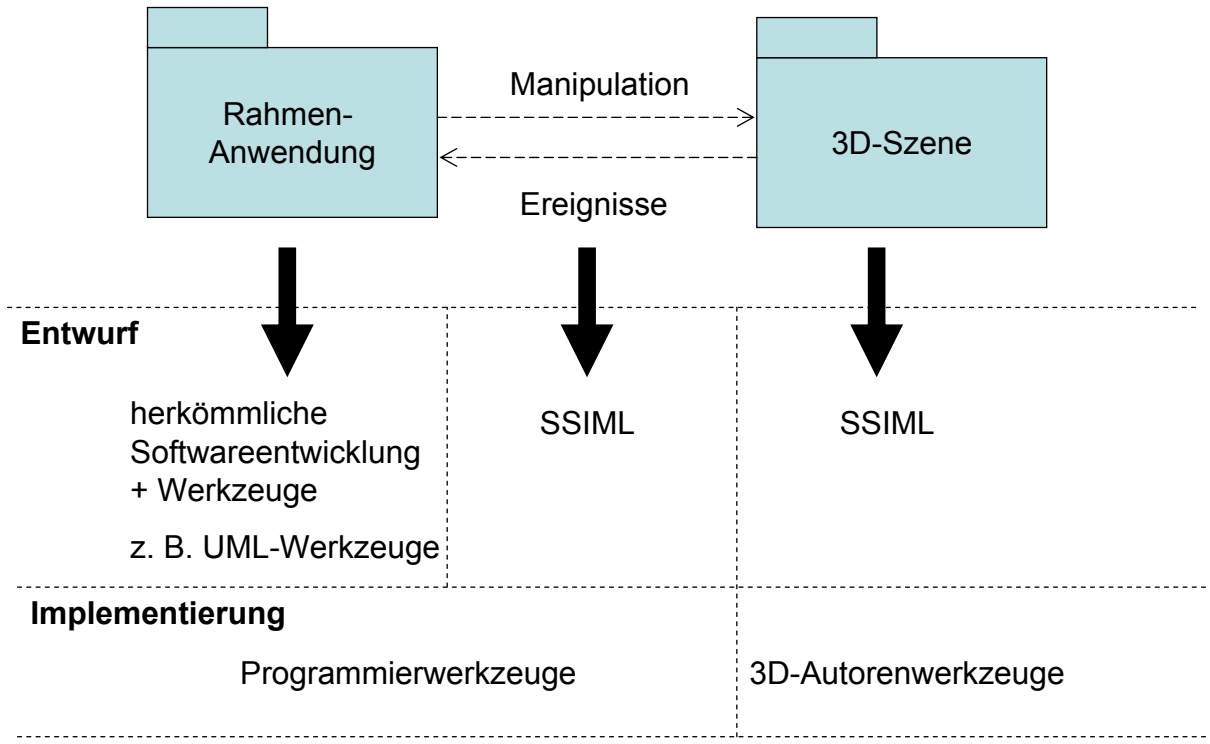



Abbildung 5.1: Entwicklungsunterstützung für 3D-Anwendungen

Während sich die Rahmenanwendung mit existierenden Modellierungssprachen wie der UML gut beschreiben lässt, erlaubt SSIML die (semi-)formale Spezifikation der 3D-Szene und ihrer Beziehungen zur Rahmenanwendung. SSIML ist für den Einsatz als plattformunabhängige Software-Entwurfssprache vorgesehen und abstrahiert daher von den spezifischen Details konkreter 3D-APIs und -Formate zur Szenenbeschreibung. Ein wichtiger Bestandteil des SSIML-Konzeptes ist die automatische Generierung von Codegerüsten aus Modellen. Diese hat das Ziel, einen nahtlosen Übergang von der Entwurfs- zur Implementierungsphase zu gewährleisten und sowohl den Programmieraufwand als auch das Auftreten von Fehlern während der Implementierung deutlich zu reduzieren.

Wie bereits im vorherigen Kapitel dargestellt, ist SSIML durch ein MOF [Obj06a]-konformes Metamodell definiert. Dieses Metamodell wird in den Abschnitten 5.2.2 und 5.3.1 genauer erläutert. Um eine Basiswerkzeugunterstützung für SSIML zu schaffen, wurde SSIML – in Form eines dem Metamodell weitestgehend entsprechenden *UML-Profiles* – u. a. in das verbreitete UML-Werkzeug NoMagic MagicDraw [Mag] integriert (vgl. Abschnitt 4.4).

### 5.1.1 Entwurfszeit und Laufzeit

Zur *Entwurfszeit* werden mit SSIML Modelle erstellt, die Gegebenheiten zur *Laufzeit* eines Systems spezifizieren. Nach dem Entwurf können die Modelle in eine Textform (z.B. Java-Code, X3D-Code) übertragen werden, die in einer Laufzeitumgebung (auch als *Laufzeitplattform* bezeichnet, z.B. einer Java Virtual Machine) als Bestandteil eines Systems ausführbar ist. In einer Laufzeitumgebung existiert dann eine *Laufzeit-Instanz* des Modells. Die Erzeugung einer Laufzeit-Instanz eines Modells bedeutet gleichzeitig, dass alle Teilelemente des Modells entsprechend den im Modell getroffenen Vorgaben instanziiert werden.

### 5.1.2 Komponenten eines SSIML-Modells

Für die Modellierung von 3D-Szenen mit SSIML in so genannten *Szenenmodellen* wurde eine szenengraphähnliche Notation gewählt. Da keine allgemein gebräuchliche Notation für Elemente von 3D-Szenengraphen existiert, wurde eine eigene kompakte Notation eingeführt. SSIML unterstützt neben der hierarchischen Zerlegung komplexer Szenenstrukturen auch die Kapselung und Wiederverwendung einzelner Szenenbestandteile und einen einfachen Mechanismus zur Beschreibung einer Menge gleichartiger 3D-Objekte.

Die Beziehungen zwischen der Rahmenanwendung und einer 3D-Szene werden im *Interrelationenmodell* definiert, welches das *Szenenmodell* enthält (Abbildung 5.2). Das Interrelationenmodell stellt entscheidende Mechanismen zur Sicherung der Konsistenz von 3D-Inhalten und Programmcode zur Verfügung. Nachfolgend werden das Szenenmodell und das Interrelationmodell ausführlicher dargestellt.


Abbildung 5.2: Komponenten eines SSIML-Modells – Interrelationen- und Szenenmodell

## 5.2 Szenenmodell

Ein *Szenenmodell* besteht aus Knoten und gerichteten Kanten, welche einen gerichteten azyklischen Graphen bilden (*Directed Acycylic Graph - DAG*). Das Modell besitzt einen Wurzelknoten, von dem aus alle anderen Knoten über gerichtete Kanten erreichbar sind. Knoten im Szenenmodell können *Attribute* besitzen.

Ein Szenenmodell in SSIML beschreibt nicht eine einzelne konkrete Szene, sondern eine *Klasse von Szenen gleicher Struktur*, also einen bestimmten *Szenentyp. Laufzeit-Instanzen des Szenenmodells* (kurz: *Szenenmodell-Instanzen*) sind *konkrete 3D-Szenen* zur Laufzeit des Systems. Der *Szenengraph* ist dabei die Datenstruktur, welche einer konkreten 3D-Szene zu Grunde liegt. Eine Szene kann während der Laufzeit verschiedene Zustände annehmen, die sich z. B. durch unterschiedliche Positionen der 3D-Objekte unterscheiden.

*Szenenspezifikationen* können wie SSIML-Szenenmodelle als Modelle der Ebene M1 (s. Abschnitt 3.3.1) aufgefasst werden (vgl. die Definition eines *Modells* in Abschnitt 3.1). Als konkrete Syntax für die Szenenspezifikation können z.B. Java (etwa für einen Java3D-Szenengraphen) oder X3D dienen. Eine Szenenspezifikation beschreibt eine konkrete Szene und definiert ihren Initialzustand. Die Szenenspezifikation kann von einer Laufzeitplattform geladen und in eine konkrete Szene, also eine speicherinterne Repräsentation, transformiert werden.

Im Gegensatz zum Szenenmodell enthält die Szenenspezifikation im Normalfall alle notwendigen Informationen, die das Erscheinungsbild der entsprechenden konkreten Szene bestimmen, wie Geometrieinformationen für alle 3D-Objekte. Einige Parameter, wie die initialen Einstellungen der virtuellen Kamera, durch welche der Betrachter die virtuelle Welt sieht, können auch durch die Laufzeitplattform (z. B. durch den X3D-Browser im Fall einer X3D-Szenenspezifikation) implizit vorgegeben sein, wodurch die explizite Festlegung dieser Parameter in der Szenenspezifikation entfallen kann. Die Szene wird zur Laufzeit von einem *Renderer* durch Traversierung des Szenengraphen in eine (Stereo-)Rastergrafik umgewandelt und auf geeigneten Geräten (z.B. Monitor, Head Mounted Display) ausgegeben. Ggf. in der Szene enthaltene Audioinformationen werden über das Soundsystem ausgegeben.

Die Unterscheidung zwischen einer konkreten Szene und einem Szenenmodell soll folgendes Beispiel verdeutlichen. Ein Szenenmodell beschreibt ganz allgemein die Struktur eines Automobils. Zwei Ausprägungen oder Instanzen dieses Szenenmodells, d.h. zwei konkrete Szenen, würden dann zwei konkrete Fahrzeuge unterschiedlicher Marken oder auch derselben Marke beschreiben – einschließlich aller Geometrien und Materialdefinitionen, welche das Erscheinungsbild des jeweiligen Fahrzeuges bestimmen. Beide Szenen können zur Laufzeit verschiedene Zustände annehmen. Bspw. kann die Kofferraumklappe eines Fahrzeuges geöffnet oder geschlossen sein. In den folgenden Abschnitten kann sich der Begriff *Szene* kontextabhängig entweder auf eine konkrete Szene zur Laufzeit auf der Meta-Ebene M0, die Spezifikation der Szene auf der Meta-Ebene M1, oder den SSIML-Elementtyp `Scene` beziehen.

Ein Szenenmodell kann auf ein oder mehrere *3D-Templates* abgebildet werden. Ein 3D-Template besitzt ein konkretes Szenengraphformat. So kann ein Szenenmodell z. B. auf ein Java3D-Template und ein X3D-Template abgebildet werden. Ein 3D-Template dient als Vorlage für eine oder mehrere Szenenspezifikationen. Eine Szenenspezifikation entsteht aus einem 3D-Template, indem das Template mit einem geeigneten Werkzeug (z. B. einem 3D-Autorenwerkzeug) angepasst und mit 3D-Inhalten (wie Objektgeometrien) angereichert wird.

Knoten in einem SSIML-Szenenmodell können in einem Szenengraphen typabhängig durch ein Szenenelement (i. d. R. durch einen Szenengraphknoten) oder eine Kombination aus mehreren Szenenelementen (die i. d. R. einen Subgraphen des Szenengraphen bilden) repräsentiert werden. Knotenattribute im Szenenmodell können, abhängig von der Zielplattform, entweder Szenengraphknoten oder Attributen von Szenengraphknoten entsprechen. Ein SSIML-`Material`-Attribut wird bspw. in einem X3D-Szenengraphen durch einen eigenen Knoten repräsentiert, in einem Java3D-Szenengraphen aber nicht.

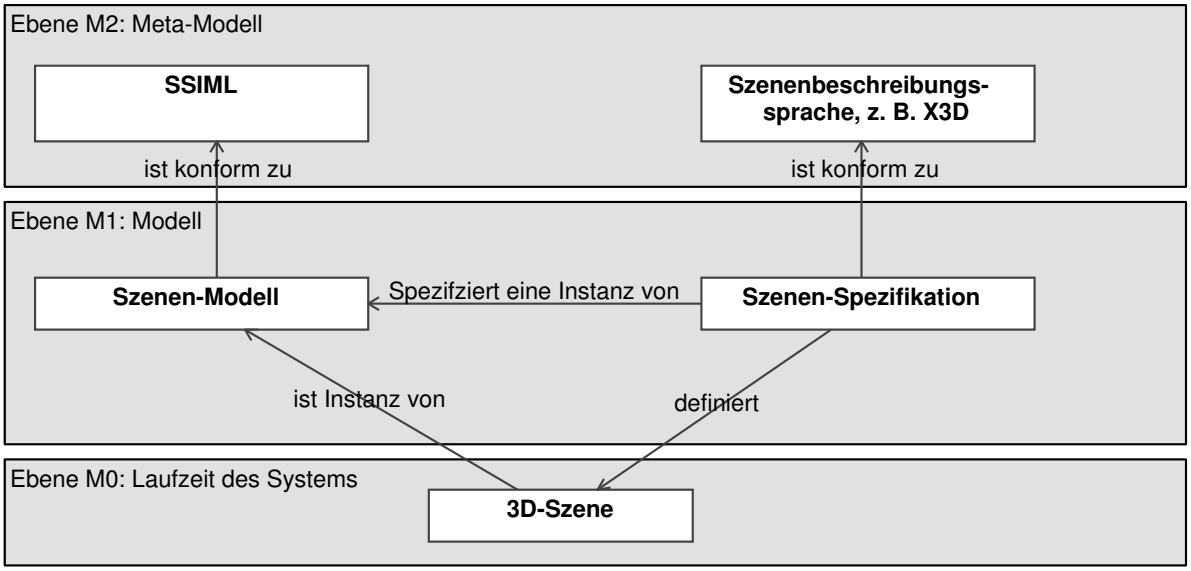



Abbildung 5.3: Zusammenhang zwischen SSIML-Modellen und Modell-Instanzen

Wie aus dem oben stehenden Text hervorgeht, kann sich der Begriff *Knoten* kontextabhängig entweder auf die Knoten eines Szenengraphen oder die Knoten eines SSIML-Szenenmodells beziehen. Ein Knoten im Szenenmodell kann in einer konkreten Szene mehrere Instanzen besitzen. Im Normalfall werden mit der Instanziierung eines Knotens auch dessen Attribute instanziiert. Auf die Regeln der Instanziierung von Knoten (und der damit verbundenen Instanziierung von Knotenattributen) wird im Abschnitt 5.2.2.7 genauer eingegangen.

Eine *Instanz* eines SSIML-Knotens wird nachfolgend auch *(3D-)Objekt, (Knoten-)* oder *(Laufzeit-)Instanz* (des Knotens) genannt. Kontextabhängig kann der Begriff der *Instanz eines SSIML-Knotens* die Instanz des Elements zur Laufzeit oder dessen Repräsentation in der Szenenspezifikation bezeichnen. Abbildung 5.3 illustriert die in diesem Abschnitt dargestellten Zusammenhänge.

### 5.2.1 Einfaches Beispiel

Zur Demonstration der verschiedenen Aspekte der Modellierung mit SSIML dient ein einfaches Beispiel. Abbildung 5.4 zeigt das Szenenmodell eines Fahrrades (*Bicycle*). Dieses setzt sich aus den Objekten `frame` (Rahmen), `handlebars` (Lenker), `frontWheel` (Vorderrad) und `rearWheel` (Hinterrad) zusammen. Der Lenker und das Hinterrad sind mit dem Rahmen des Fahrrads, das Vorderrad ist mit dem Lenker verbunden.

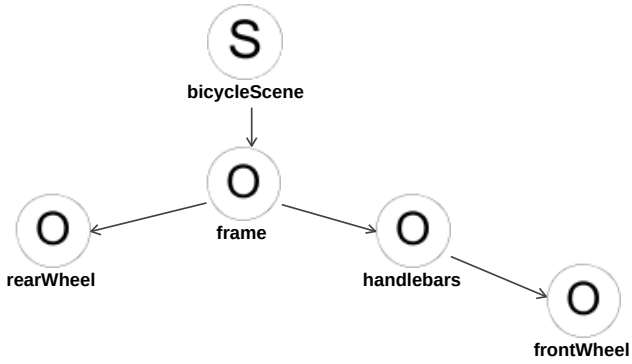



Abbildung 5.4: Einfaches Szenenmodell eines Fahrrades

### 5.2.2 Metamodell

In diesem Abschnitt wird anhand von Ausschnitten aus dem SSIML-Metamodell der Aufbau von SSIML-Szenenmodellen illustriert. Einige Zusicherungen *(Constraints)* sind dabei exemplarisch in OCL-Syntax [Obj06b] als Annotationen der entsprechenden Metamodellklassen dargestellt. Umfangreichere Constraints werden in natürlicher Sprache im Text beschrieben.

Meta-Klassen von SSIML- und UML-Modellelementen sind in entsprechenden Abbildungen in diesem Kapitel und den Kapiteln 7 bis 9 farblich gekennzeichnet. Die Bedeutung der verwendeten Farben ist aus Abbildung 5.5 ersichtlich.

#### 5.2.2.1 Elemente des Szenenmodells

Ein SSIML-Szenenmodell (`SceneModel`) besteht aus einzelnen Szenenmodellelementen (`SceneModelElement`s, Abbildung 5.6). `SceneModelElement` ist das Basiselement aller Szenenmodellelemente. Nicht alle Szenenmodellelemente müssen unmittelbar dem Szenenmodell untergeordnet sein. Einige können in Subgraphen des Szenenmodells enthalten sein. Eine besondere Gruppe der Szenenmodellelemente bilden diejenigen Elemente, die einen Namen besitzen (`NamedSceneModelElement`s). So ist ein Knoten des Szenenmodells z. B. ein `NamedSceneModelElement`, eine Beziehung zwischen einem Eltern- und einem Kindknoten aber nicht. Abbildung 5.6 zeigt, welche Modellelementtypen in einem Szenenmo-

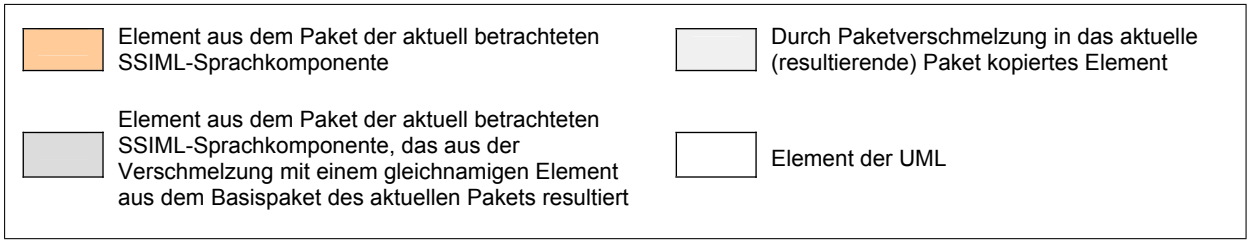



Abbildung 5.5: Farbliche Kennzeichnungen von SSIML- und UML-Meta-Klassen

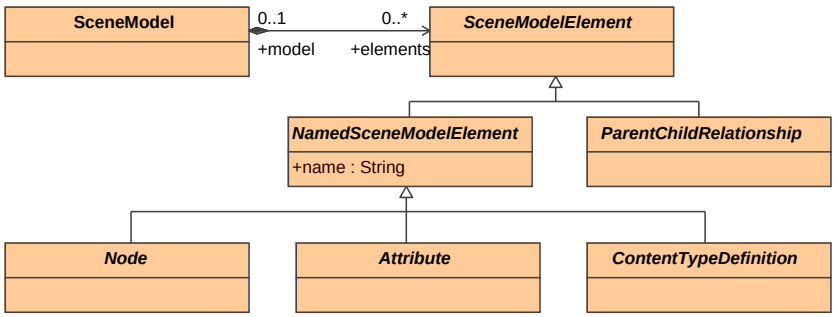



Abbildung 5.6: Typen von Elementen des Szenenmodells

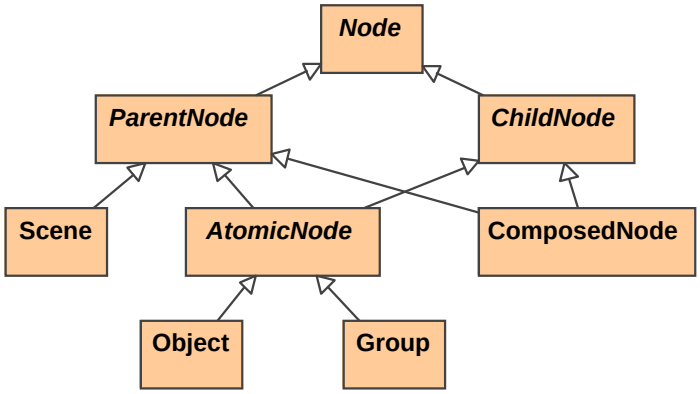



Abbildung 5.7: Vererbungshierarchie der Knoten-Meta-Klassen

dell prinzipiell vorkommen können. Die einzelnen Meta-Klassen werden in den folgenden Abschnitten genauer beschrieben.

#### 5.2.2.2 Knoten und Knotenbeziehungen

Abbildung 5.7 gibt einen Überblick über Meta-Klassen, deren Instanzen Knoten in einem SSIML-Szenenmodell sind. Ein Modellelement vom Typ `Node` ist ein Knoten innerhalb des SSIML-Szenenmodells. Außer Knoten vom Typ `Scene`, die nur als Wurzelknoten des Szenenmodells vorkommen können und daher ausschließlich Elternknoten (`ParentNode`s) sind, können alle Knoten sowohl Elternknoten als auch Kindknoten (`ChildNode`) sein. Neben *Attributen* (siehe Abschnitt 5.2.2.8) kann ein Knoten einen Inhalt (*Content*) besitzen. Die Kindknoten eines Elternknotens zählen zu seinem Inhalt (*Child Content*).

Eine Beziehung zwischen einem Eltern- und einem Kindknoten (`ParentChildRelationship`, notiert als Kante) drückt aus, dass das durch den Kindknoten repräsentierte 3D-Objekt zum Elternobjekt gehört bzw. ein Teil des Elternobjektes ist. Die räumliche Position und Orientierung des Kindobjektes wird relativ zur Position und Orientierung des Elternobjekts bestimmt.

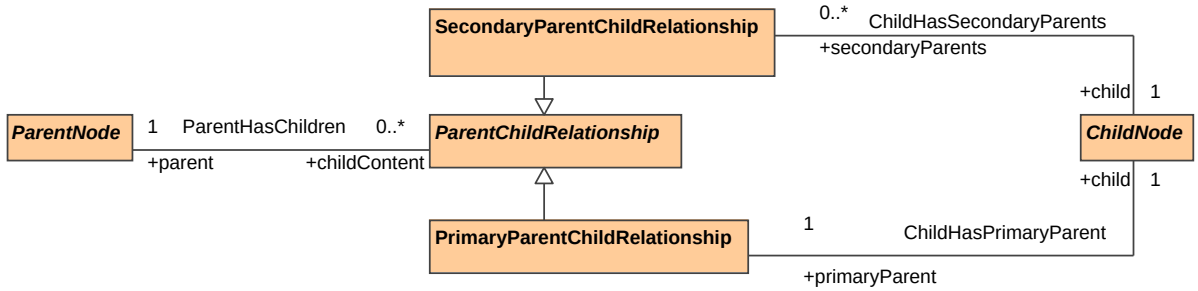



Abbildung 5.8: Beziehungen zwischen Knoten

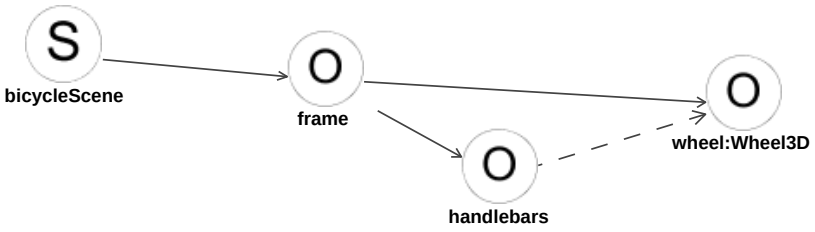



Abbildung 5.9: Variante des Fahrradbeispiels mit Primär- und Sekundärkanten

Jeder Kindknoten kann mehrere, muss aber mindestens einen Elternknoten besitzen. Mit *genau einem* seiner Elternknoten, dem *Primärelternknoten*, muss ein Kindknoten über eine primäre Eltern-Kind-Beziehung (`PrimaryParentChildRelationship`, nachfolgend kurz als *Primärkante* bezeichnet) verbunden sein, mit den restlichen Elternknoten, den *Sekundärelternknoten*, über sekundäre Eltern-Kind-Beziehungen (`SecondaryParentChildRelationships`, nachfolgend auch *Sekundärkanten* genannt). Die *Primärkindknoten* eines Elternknotens sind mit demselben über Primärkanten, seine *Sekundärkindknoten* über Sekundärkanten verbunden. Ein Elternknoten kann i. d. R. nur die Rolle entweder des Primär- oder des Sekundärelternknotens ein und desselben Kindknotens übernehmen, nicht aber beide Rollen zugleich. Eine Ausnahme tritt ein, wenn der Elternknoten ein Kompositionsknoten ist (siehe Abschnitt 5.2.2.6). Abbildung 5.8 gibt einen Überblick über die Beziehungen zwischen Knoten.

Abbildung 5.9 zeigt eine Variante des Fahrradbeispiels, in dem der Knoten `wheel` (Rad) sowohl über eine Primärkante als auch über eine Sekundärkante in das Modell eingebunden ist. Dies bewirkt beim Durchlauf eines entsprechenden Szenengraphen zur Laufzeit, dass das Objekt `wheel` zweimal dargestellt wird; zum einen als Hinterrad am Fahrradrahmen und zum anderen als Vorderrad in Verbindung mit dem Lenker.

#### 5.2.2.3 Wurzelknoten des Szenenmodells

Ein Szenenmodell muss genau einen Wurzelknoten, also einen Knoten ohne eingehende Kanten, besitzen. Soll eine Szene mit explizit einstellbaren Kamera- und Beleuchtungs-

parametern definiert werden, *muss* der Wurzelknoten des Szenenmodells vom Typ `Scene` sein. Soll etwa nur ein Teil einer umfangreicheren Gesamtszene spezifiziert oder sollen ggf. durch die Zielplattform vorgegebene Standardkamera- und -beleuchtungseinstellungen genutzt werden, kann der Wurzelknoten der Szene ein Knoten eines beliebigen Typs sein. Der Name des Wurzelknotens repräsentiert den Szenentyp, z. B. *bicycleScene.*

#### 5.2.2.4 Teilbäume

Werden nur die primären Kanten im Szenenmodell betrachtet, ergibt sich eine Baumstruktur, da jeder Kindknoten genau einen primären Elternknoten besitzt. Ein *Teilbaum* im SSIML-Szenenmodell wird beschrieben durch:

- einen Knoten im Szenenmodell (den *Teilbaumwurzelknoten*) sowie
- alle über Primärkanten erreichbaren Nachfahren des Teilbaumwurzelknotens (sofern vorhanden),
- die Menge aller zu den Knoten des Teilbaums gehörigen Attribute und
- die Menge der Primärkanten, über die die Knoten des Teilbaums miteinander verbunden sind.

#### 5.2.2.5 Atomare Knoten

Ein *atomarer Knoten* oder *Atomknoten* (`AtomicNode`) repräsentiert eine definierte Position (und Orientierung) innerhalb eines dreidimensionalen Raumes. Knoten der Typen *Objekt* (`Object`) und *Gruppe* (`Group`) sind atomare Knoten. Abbildung 5.7 zeigt die Einordnung von atomaren Knotentypen ins SSIML-Metamodell. Atomare Knoten vom Typ `Object` können Inhalte kapseln[1], wobei diese Inhalte im SSIML-Modell nicht genauer spezifiziert werden. Dadurch unterscheiden sich atomare Knoten auch von den weiter unten erläuterten Kompositionsknoten, die Inhalte einschließen, welche im Modell spezifiziert werden.

**Group** Ein Gruppenknoten (`Group`) dient der Zusammenfassung seiner Kindknoten zu einer Gruppe. Deshalb ist der Inhalt eines Gruppenknotens durch seine Kindknoten definiert (*Child Content* des Gruppenknotens). So könnte z. B. ein Szenenmodell mit einer Gruppierung von zwei Subgraphen, die jeweils ein Fahrradobjekt repräsentieren, definiert werden.

**Object** Eines der wichtigsten Konzepte in SSIML stellen Objektknoten dar (vgl. dazu auch Abschnitt 4.3). I. d. R. repräsentiert ein Objektknoten (Typ `Object`), wie der in Abbildung 5.9 dargestellte Knoten `handlebars`, ein geometrisches Szenenobjekt einschließlich seines farblichen Erscheinungsbildes (Material, Textur). Ferner kann ein Objektknoten

[1] Der gekapselte Inhalt (*Encapsulated Content*) eines atomaren Knotens ist nicht mit seinen Kindknoten (*Child Content*) zu verwechseln.

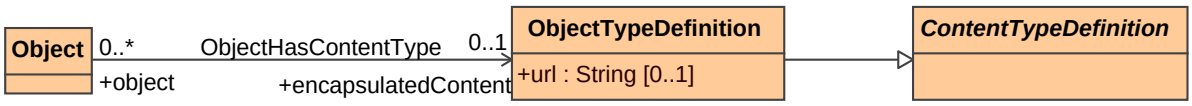



Abbildung 5.10: Die Metamodellklasse `ContentTypeDefinition`

aber auch Audioinformationen oder lokale Lichtquellen enthalten. Die detaillierte Geometrie des Objektes wird im Szenenmodell nicht spezifiziert. Der zum *Child Content* hinzu kommende eingeschlossene Inhalt eines Objektknotens (*Encapsulated Content*) wird im SSIML-Modell durch eine Referenz auf ein `ObjectTypeDefinition`-Modellelement, der *Inhaltstypdefinition* des Objektknotens, beschrieben (Abbildung 5.10). Ein `ObjectTypeDefinition`-Element besitzt einen Namen und optional eine URL, die auf die Ressource, welche die konkreten 3D-Objekt-Informationen enthält, verweist. Der Name eines `ObjectTypeDefinition`-Elements muss innerhalb des SSIML-Modells eindeutig sein. Es können in einem Szenenmodell mehrere Objektknoten auftreten, die denselben Inhaltstyp haben (Name der Inhaltstypdefinition stimmt überein). Dies bedeutet i. W., dass mehrere Kopien derselben Objektgeometrie innerhalb eines Szenengraphen existieren können. Die Angabe einer `ObjectTypeDefintion` für ein `Object` ist optional.

Im Beispiel in Abbildung 5.9 besitzt das Objekt `wheel` den Inhaltstyp `Wheel3D`. Dieser wird mit vorangestelltem Doppelpunkt hinter dem Objektnamen notiert.

#### 5.2.2.6 Kompositionsknoten und Subgraphendefinitionen

Ein spezieller Knoten ist der *Kompositionsknoten* (`ComposedNode`). Ein Kompositionsknoten kapselt einen Subgraphen des Szenenmodells (*Encapsulated Content* des Kompositionsknotens). Der entsprechende Subgraph ist in einer – innerhalb des Szenenmodells eindeutig benannten – Subgraphendefinition (`SubgraphDefinition`) spezifiziert (Abbildung 5.11). Eine Subgraphendefinition enthält die Spezifikation eines SSIML-Subgraphen (ein Subgraph in einer Subgraphendefinition ist ein zusammenhängender Graph mit genau einem Knoten mit dem Eingangsgrad[2] Null, dem *Subgraphenwurzelknoten*). Da ein solcher Subgraph selbst auch Kompositionsknoten enthalten kann, die auf Subgraphendefinitionen verweisen, können zyklische Abhängigkeiten zwischen den Subgraphendefinitionen entstehen. Dies ist in einem gültigen SSIML-Modell jedoch nicht erlaubt. Zudem enthält eine Subgraphendefinition alle Attributelemente, die mit den Knoten des Subgraphen assoziiert sind.

Abbildung 5.12(a) zeigt eine weitere Variante des Fahrradbeispiels. Die Knoten für Rahmen und Lenker sind in der Subgraphdefinition `BicycleBody` enthalten (Abbildung 5.12(b)). Der Kompositionsknoten `bodyComp` verweist auf diese Subgraphendefinition. Die Radobjekte sind mit dem Kompositionsknoten `bodyComp` verbunden. Das Vorderrad ist mit dem in `bodyComp` gekapselten Knoten `handleBars` assoziiert, das Hinterrad mit dem Knoten `frame`. Die UML-ähnliche Darstellung des `BicycleBody`-Subgraphen als Paket in

[2] Anzahl einlaufender Kanten

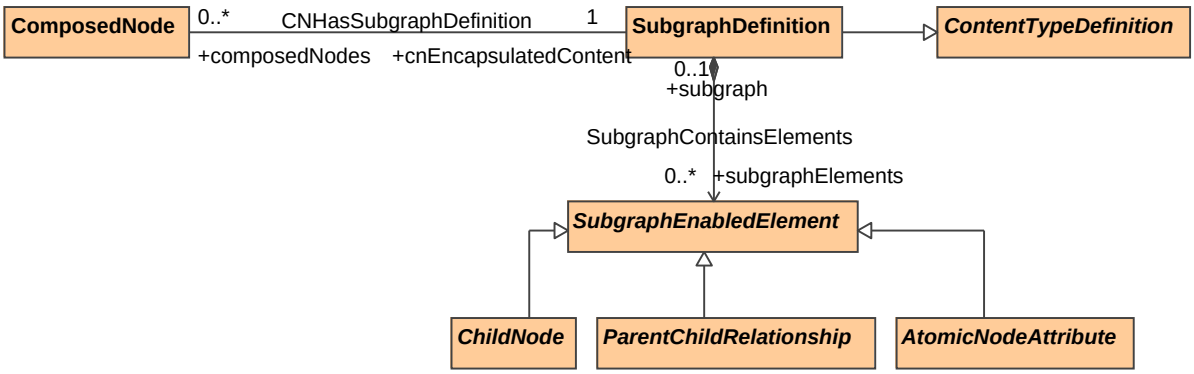



Abbildung 5.11: Zusammenhang zwischen `ComposedNode` und `SubgraphDefinition`

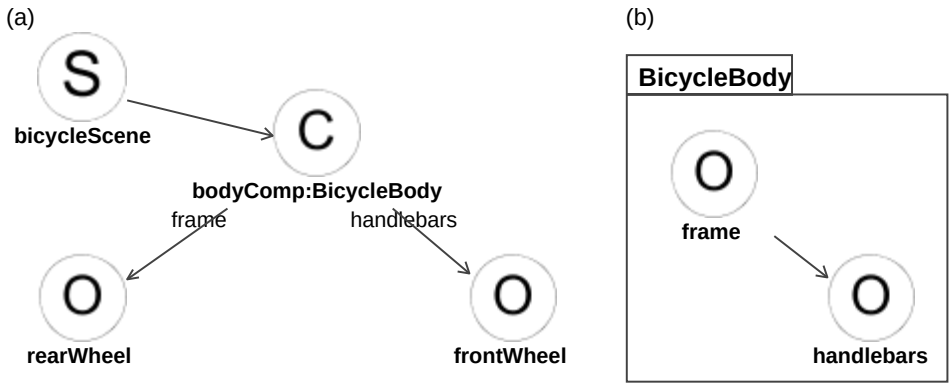



Abbildung 5.12: Variante des Fahrradbeispiels mit einem Kompositionsknoten

Abbildung 5.12(b) ist nicht Bestandteil der SSIML-Notation und wurde nur zu Illustrationszwecken verwendet.

Im SSIML-Szenenmodell können mehrere Kompositionsknoten Referenzen auf ein und dieselbe Subgraphendefinition haben. Da Kompositionsknoten mit derselben Subgraphendefinition unterschiedliche Kindknoten besitzen können, muss bei der Abbildung eines Kompositionsknotens in ein konkretes Szenengraphformat immer eine Kopie seiner Subgraphendefinition erzeugt werden, woraus Mehrfachvorkommen der gleichen Graphstruktur auf der Laufzeitplattform resultieren können.

**Kompositionsbereiche und Namensräume** Der gekapselte Inhalt eines Kompositionsknotens stellt einen *Kompositionsbereich* dar. Kompositionsknoten selbst können wieder Kompositionsknoten enthalten, so dass ein Szenenmodell aus mehreren verschachtelten *Kompositionsbereichen* bestehen kann. Die Verschachtelung von Kompositionsknoten wurde über eine Variante des *Composite*-Entwurfsmusters [GHJV94] realisiert. Durch diese Verschachtelung kommt es zu einer *Kompositionshierarchie* (in Form eines *Kompositions-*

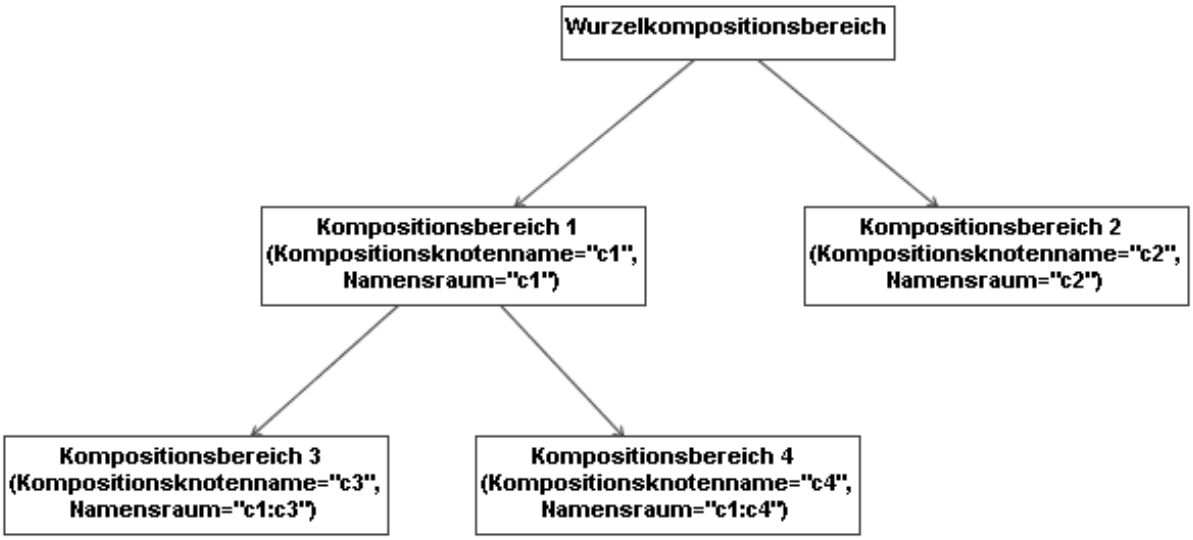



Abbildung 5.13: Beispiel einer Kompositionshierarchie

*baumes* als Sicht auf das Szenenmodell, siehe Beispiel in Abbildung 5.13) mit mehreren *Kompositionsebenen*. Die oberste Kompositionsebene enthält nur den *Wurzelkompositionsbereich* des Szenenmodells, der auch den Szenenwurzelknoten beinhaltet. Der Wurzelkompositionsbereich ist somit selbst nicht in einem Kompositionsknoten enthalten, er beinhaltet aber direkt oder indirekt die Kompositionsbereiche aller niedrigeren Kompositionsebenen. Die Blätter der Kompositionshierarchie bilden Kompositionsbereiche, in denen nur atomare Knoten vorkommen, also keine Kompositionsknoten.

Die Vorfahren eines Kompositionsbereiches im Kompositionsbaum werden nachfolgend *übergeordnete Kompositionsbereiche* oder *Überkompositionsbereiche* genannt, Nachfahren werden als *untergeordnete Kompositionsbereiche* oder *Unterkompositionsbereiche* bezeichnet.

Ein Kompositionsbereich stellt einen eigenen *Namensraum* dar. Die Bezeichnung des Namensraumes ergibt sich durch einen Pfad in der Kompositionshierarchie, also einer Abfolge von Kompositionsknoten. Die Namensraumbezeichnung wird aus den *Namen der im Pfad enthaltenen Kompositionsknoten* gebildet, wobei die einzelnen Knotennamen durch Doppelpunkte separiert werden. Der Namensraum des Wurzelkompositionsbereiches ist die leere Zeichenkette. Innerhalb eines Kompositionsbereiches müssen alle Knoten einen eindeutigen Namen besitzen. Daraus resultiert, dass auch innerhalb einer Subgraphendefintion alle Knoten einen eindeutigen Namen besitzen müssen.

**Kompositionsknoten als Eltern- und Kindknoten** Zwischen Kompositionsknoten und ihren Kindknoten gibt es besondere Beziehungen, die Instanzen der Metamodellklasse `CNChildRelationship` sind. Anders gesagt: Ein `CNChildRelationship`-Element ist eine Eltern-Kind-Beziehung, bei welcher der Elternknoten ein Kompositionsknoten ist (Abbildung 5.14). Sie besitzt ein zusätzliches Attribut `innerParents`, welches auf atomare Kno-

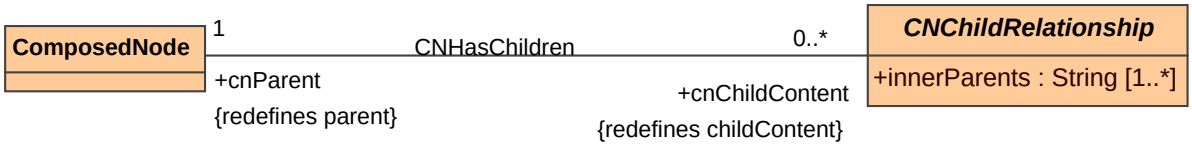



Abbildung 5.14: Zusammenhang zwischen `ComposedNode` und `CNChildRelationship`

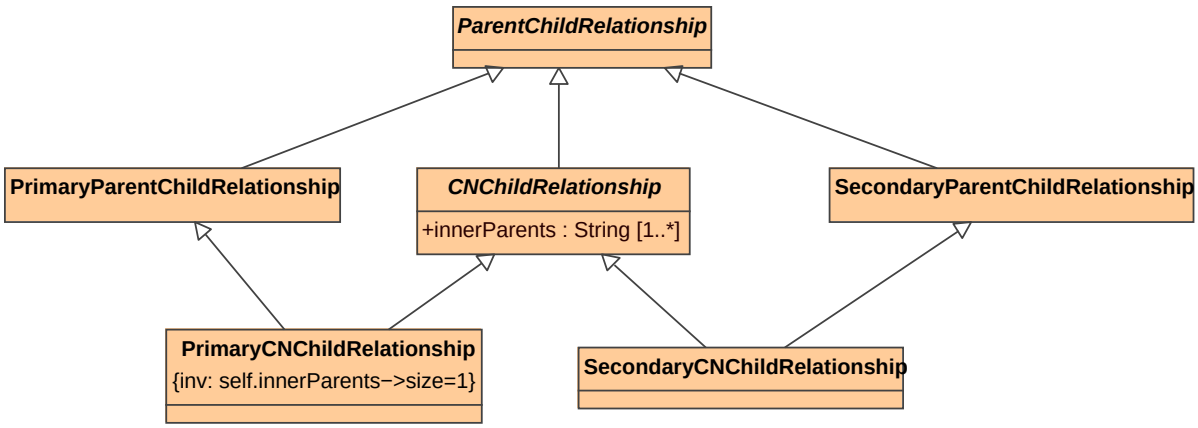



Abbildung 5.15: `CNChildRelationship` und die abgeleiteten Meta-Klassen `PrimaryCNChildRelationship` und `SecondaryCNChildRelationship`

ten innerhalb des Kompositionsknotens – die *inneren Elternknoten* – verweist. Bei einem `PrimaryCNChildRelationship`-Element darf `innerParents` genau einen Knoten innerhalb des Kompositionsknotens referenzieren, bei einem `SecondaryCNChildRelationship`-Element auch mehrere (Abbildung 5.15). Der Verweis auf innere Elternknoten wird in der SSIML-Kompositionspfadsyntax als Zeichenkette codiert (s. a. Abschnitt 5.2.2.7). Das Verweisziel muss immer ein atomarer Knoten sein. Ein Verweis auf einen Kompositionsknoten als inneren Elternknoten ist also nicht erlaubt. Enthielte etwa der Knoten `bodyComp` in Abbildung 5.12(a) statt den Knoten `handlebars` und `frame` einen weiteren Kompositionsknoten `innerBodyComp`, der seinerseits die Knoten `handlebars` und `frame` kapselte, und würde als innerer Elternknoten der Radobjekte nur `innerBodyComp` referenziert, wäre unklar, mit welchen atomaren Objekten die entsprechenden Räder (`frontWheel`, `rearWheel`) zu verbinden wären.

Eine Besonderheit ist, dass ein Kindknoten mit einem Elternknoten, der ein Kompositionsknoten ist, gleichzeitig über eine Primär- und eine Sekundärkante verbunden sein kann.

Die Angabe von `innerParents` ist notwendig, da ein Kompositionsknoten kein *atomarer* Knoten ist, sondern ein *auflösbarer* Knoten, d.h. ein Platzhalter für den von ihm gekapselten Subgraphen und somit durch den Subgraphen ersetzbar ist. Zur Laufzeit existiert ein Kompositionsknoten nicht mehr als Kapsel, sondern nur noch in aufgelöster Form, al-

so als Bestandteil (Subgraph) des Gesamtszenengraphen. Somit muss eindeutig festgelegt sein, mit welchen atomaren Knoten innerhalb des Kompositionsknotens ein Kindknoten verbunden ist.

Analog muss ein atomarer Knoten innerhalb eines Kompositionsknotens *composedNode* als *innerer Kindknoten* (*innerChild*) der Elternknoten des Kompositionsknotens dienen. Prinzip ist, dass der *atomare Wurzelknoten* des Kompositionsknotens gesucht wird, welcher die Wurzel des durch die Auflösung aller in *composedNode* direkt oder indirekt enthaltenen Kompositionsknoten entstehenden Subgraphen ist. Dieser Knoten ist dann der Knoten *innerChild*. Im Beispiel in Abbildung 5.12 wäre der innere Kindknoten des Kompositionsknotens `bodyComp` bspw. der Knoten `frame`. Der nachfolgende (weitestgehend selbsterklärende) Pseudocode skizziert einen Algorithmus zum Finden des inneren Kindknotens.

```
subgraphRoot = subgraphRootNodeOf(composedNode);

while (subgraphRoot instanceOf ComposedNode) {
    subgraphRoot_old = subgraphRoot;
    subgraphRoot_new = subgraphRootNodeOf(subgraphRoot_old);
    subgraphRoot=subgraphRoot_new;
}

innerChild = subgraphRoot;
```

#### 5.2.2.7 Erzeugung von Knoteninstanzen

Die nachfolgenden Betrachtungen beziehen sich darauf, dass alle Kompositionsknoten im Szenenmodell *aufgelöst*, d.h. durch die entsprechenden Subgraphen ersetzt werden, wodurch ein Graph entsteht, der ausschließlich *atomare* Knoten enthält.

Ein atomarer Knoten – und damit auch der gesamte Teilbaum, dessen Wurzelknoten der Knoten ist – kann mehrere (Laufzeit-)Instanzen besitzen; ein atomarer Knoten ist ein `MultiInstanceEnabledElement` (s. Abbildung 5.16). Die Zahl der Instanzen wird über die Attribute `numberOfElements` und `createCopies` der Primärkante festgelegt, deren Ziel der Knoten ist (Abbildung 5.16). Besitzt `createCopies` den Wert `true`, gibt `numberOfElements` an, mit welcher Anzahl (Laufzeit-)Instanzen des Knotens eine Instanz seines Elterknotens in Beziehung steht (Standardwert von `numberOfElements` ist `1`). Da bei der Traversierung eines gerichteten Pfades aus Primärkanten vom Szenenwurzelknoten aus zu einem Teilbaumwurzelknoten ein mehrfaches Auftreten von `createCopies` möglich ist, kann sich die Zahl der insgesamt erzeugten Teilbauminstanzen vervielfachen. Die *Zahl der Instanzen eines Kompositionsknotens* entspricht implizit der Zahl der Instanzen seines atomaren Wurzelknotens.

Besitzt `createCopies` den Wert `false` (Standard), gibt `numberOfElements` an, wie viele *parallele* Transformationen *eine* Knoteninstanz relativ zur Transformation ihrer Elternknoteninstanz zur Laufzeit besitzt. In der Praxis bedeutet dies, dass ein Kindobjekt zur Laufzeit nur einmal im Arbeitsspeicher existiert, es aber im dreidimensionalen Raum an mehreren Positionen (und in verschiedenen Lagen) relativ zum Elternobjekt dargestellt wird, nämlich so vielen, wie durch `numberOfElements` spezifiziert. So können z.B. auf einem 3D-Objekt *Tisch* (Elternobjekt) mehrere durch ein und dasselbe Kindobjekt repräsentier-

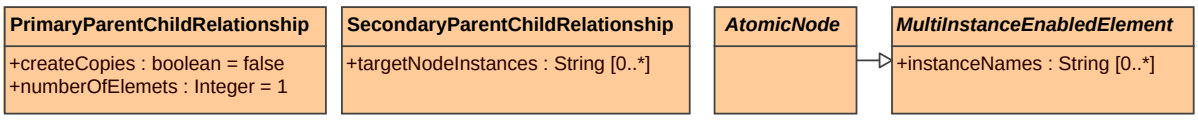



Abbildung 5.16: Instanzbezogene Attribute von `PrimaryParentChildRelationship`, `SecondaryParentChildRelationships` und `AtomicNode`

te Vasen stehen. Der Wert von `numberOfElements` muss immer größer als die Zahl *Null* sein. Ist `numberOfElements` gleich *Eins*, spielt der Wert von `createCopies` praktisch keine Rolle.

Eine Sekundärkante besitzt im Gegensatz zu einer Primärkante ein Attribut `targetNodeInstances` (Abbildung 5.16), mit dem spezifiziert werden kann, welche (Laufzeit-)Instanzen des Zielknotens der Kante adressiert werden. Die Adressierung von (Laufzeit-)Instanzen von Szenenmodellelementen wie Knoten und Knotenattributen wird im Abschnitt 5.2.2.13 näher beschrieben.

Jeder Instanz eines atomaren Knotens kann ein Name zugeordnet werden. Dies ist allerdings nur sinnvoll, wenn die Zahl der Instanzen des Knotens nicht übermäßig groß ist. Die Instanznamen werden durch das Attribut `instanceNames` des atomaren Knotens (Abbildung 5.16) in Form einer Namensliste angegeben, z.B. *{instance1, instance2, instance3}*. Der ersten Instanz des Knotens wird das erste Listenelement zugeordnet, der zweiten Instanz das zweite, usw. Enthält die Liste $n$ Elemente, obwohl es *mehr als* $n$ Instanzen gibt, wird nur an die ersten $n$ Instanzen ein Name entsprechend der Liste vergeben. Die Liste kann maximal so viele Namen beinhalten, wie Instanzen existieren. Enthält die Liste mindestens ein Element, darf sie in der Beschriftung eines Knotens hinter dem Knotennamen stehen. An dieser Stelle ist zu erwähnen, dass die Anzahl der Instanzen eines Knotens (spezifiziert durch die Attribute `createCopies` und `numberOfElements` der zum Knoten führenden Primärkante) in der Notation einer Primärkante nicht unbedingt angegeben werden muss, wenn sie gleich der Anzahl der Elemente der Instanznamensliste des Knotens ist (vgl. Abschnitt 5.2.3.1 zur Notation von Knoten und Knotenbeziehungen). Innerhalb eines Kompositionsbereiches müssen alle vergebenen Instanznamen eindeutig sein.

Abbildung 5.17 demonstriert eine praktische Anwendung der Attribute `numberOfElements` und `createCopies` in einem SSIML-Szenenmodell. Rahmen und Lenker wurden zu einem `Object`-Knoten `frameAndHandlebars` zusammengefasst, mit dem zwei Radobjekte (`rearWheel` und `frontWheel`) verbunden sind. Im Beispiel hat `createCopies` den Wert `true` und `numberOfElements` den Wert `2`.

#### 5.2.2.8 Knotenattribute

Ein Attributelement (`Attribute`) im Szenenmodell repräsentiert eine Eigenschaft der 3D-Szene selbst (z.B. einen Blickpunkt) oder die Eigenschaft eines Objektes der Szene (z. B. Position und Orientierung, Material). Die eingeführten SSIML-Attributtypen entsprechen

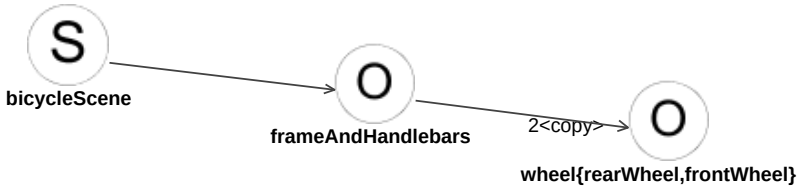



Abbildung 5.17: Variante des Fahrradbeispiels mit der Angabe von Multiplizitäten und Instanznamen

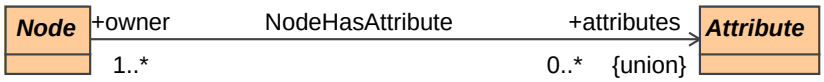



Abbildung 5.19: Knotenattribute

grundlegenden Elementen in Szenengraphformaten wie Java3D, VRML oder X3D, die zum Erscheinungsbild oder zur Interaktivität der Szene beitragen.

Ein Attribut wird einem (in besonderen Fällen auch mehreren) Knoten zugeordnet (Abbildung 5.19) und besitzt einen im Kontext des Knotens eindeutigen Namen. Manche Knotentypen erlauben eine Verbindung mit mehreren Attributen gleichen Typs, bspw. kann ein Objekt mehrere Materialien besitzen. Konkrete Attributwerte werden auf der Abstraktionsebene des Szenenmodells nicht angegeben. Die Modellierung von Attributen eines Knotens ist prinzipiell optional. Dies ist sinnvoll, um schon bei der Erstellung des Szenenmodells zu verdeutlichen, welche Objekteigenschaften später dynamisch verändert werden sollen und welche nicht. Abbildung 5.20 zeigt die Vererbungshierarchie von Attributtypen. In Abbildung 5.18 ist ein Beispielmodell für die Zuordnung eines Attributs zu einem Knoten zu sehen. Der Knoten `rearWheel` (Rad) wurde mit einem Transformationsattribut `wheelTrans` assoziiert. Durch Zugriff auf dieses Attribut kann das Radobjekt zur Laufzeit rotiert werden. Auf die einzelnen Attributtypen wird im Folgenden eingegangen.


Abbildung 5.18: Ausschnitt aus dem Fahrradbeispiel; dem Objektknoten `rearWheel` wurde ein Transformationsattribut `wheelTrans` zugeordnet

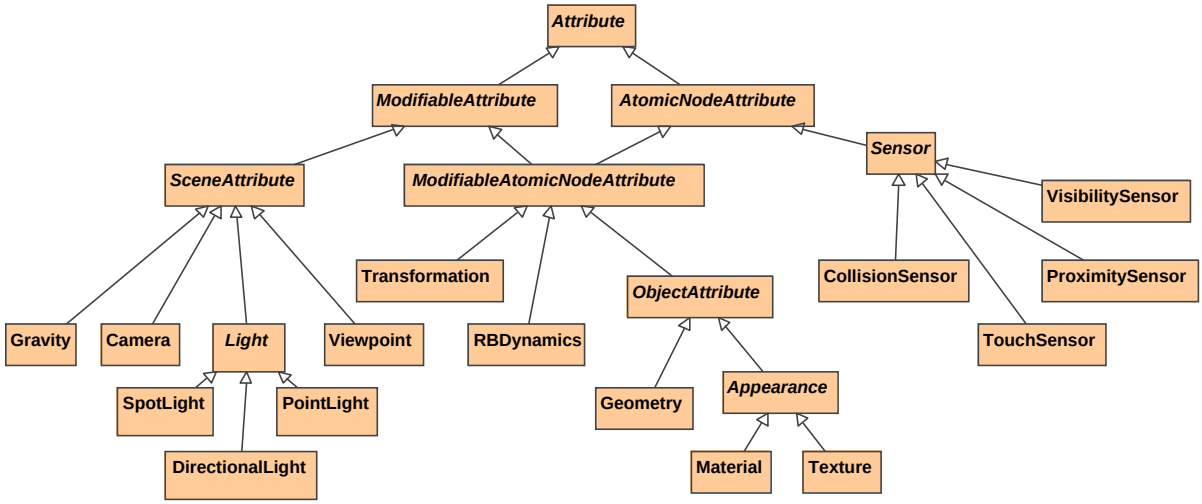



Abbildung 5.20: Vererbungshierarchie von Attributtypen

#### 5.2.2.9 Attribute der Gesamtszene

Attribute der gesamten Szene (d.h. Attribute eines Knotens des Typs `Scene`, Abbildung 5.21) sind Kamera (`Camera`), Betrachterstandpunkte (`Viewpoints`) und Attribute, die physikalische Eigenschaften der Szene beschreiben (`Gravity`). Auch globale Lichtquellen (`Lights`) zählen zu den Szenenattributen.

**Camera** Das Kameraattribut (`Camera`) repräsentiert die virtuelle Kamera, durch welche der Betrachter die 3D-Szene wahrnimmt. Eine Instanz einer virtuellen Kamera auf einer Laufzeitplattform kann Informationen über ihre Position und Orientierung, den vertikalen und horizontalen Öffnungswinkel sowie den Abstand zur vorderen und hinteren Clipping-Ebene des Viewing-Volumens, also den sichtbaren Ausschnitt der virtuellen 3D-Welt, enthalten.

**Viewpoint** Ein Blickpunktattribut verkörpert einen bestimmten Blickpunkt (`Viewpoint`) des Betrachters in der 3D-Szene. Ein Blickpunkt kapselt eine festgelegte Position und Orientierung, so dass der Benutzer zu einem bestimmten Blickpunkt „springen“ kann. Beim Übergang zu einem Blickpunkt werden Position und Orientierung der virtuellen Kamera auf die Position und Orientierung des Blickpunktes gesetzt. Ein Knoten vom Typ `Scene` kann mehrere Blickpunktattribute besitzen.

**Light, DirectionalLight, PointLight und SpotLight** Ein Lichtquellenattribut (`Light`) repräsentiert – wie der Name bereits nahelegt – eine Lichtquelle in der 3D-Szene. Ein Szenenwurzelknoten kann über mehrere Lichtquellenattribute verfügen.

Gerichtete Lichtquellen, Punktlichtquellen und Scheinwerferlichtquellen werden durch die Typen `DirectionalLight`, `PointLight` und `SpotLight` beschrieben.

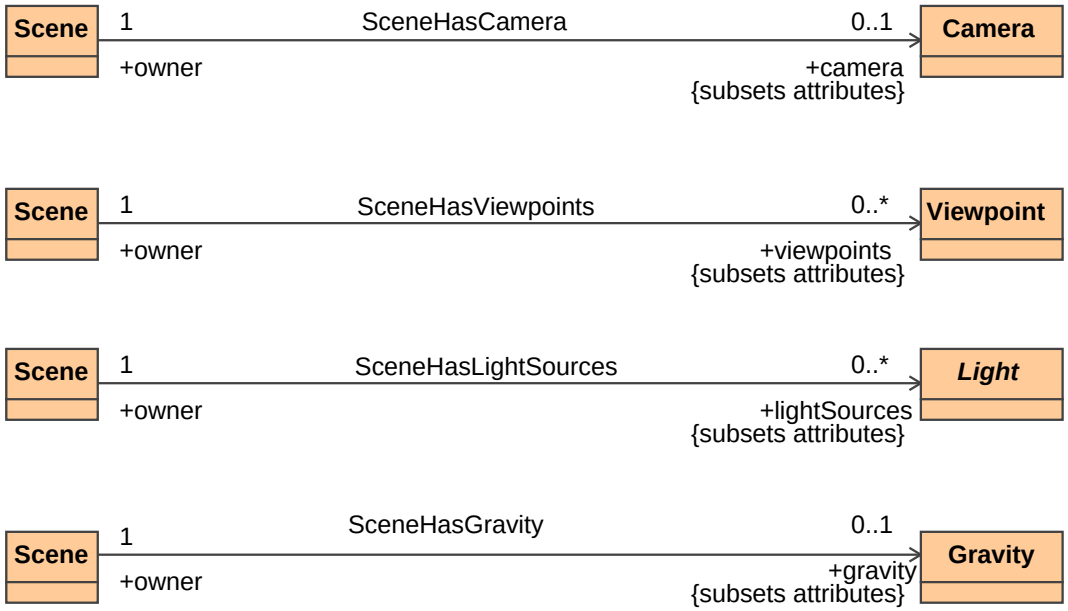



Abbildung 5.21: Attribute von `Scene`

**Gravity** Das `Gravity`-Attribut stellt die innerhalb der Szene wirkende Gravitationskraft dar. Der Standardwert für das Attribut entspricht der Erdanziehungskraft. Die Angabe eines `Gravity`-Attributes kann in Verbindung mit der Angabe von den weiter unten beschriebenen `RBDynamics`-Attributen für Objekte sinnvoll sein.

#### 5.2.2.10 Attribute von atomaren Knoten

Abbildung 5.22 illustriert die möglichen Beziehungen zwischen atomaren Knoten und ihren Attributen.

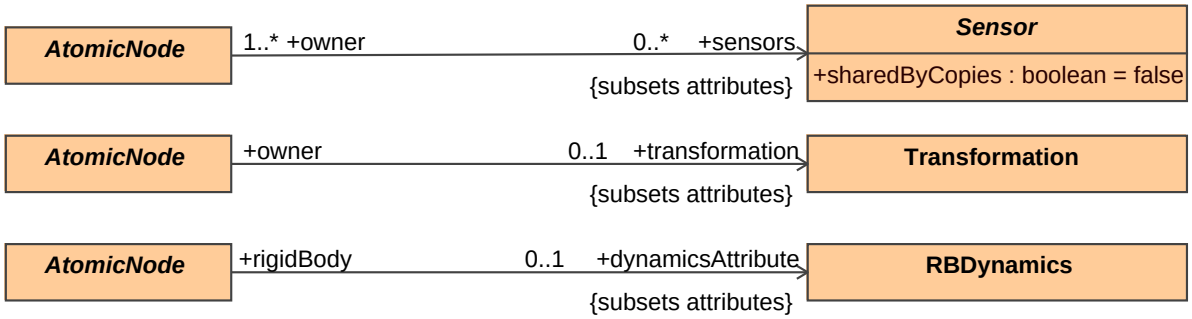



Abbildung 5.22: Attribute von atomaren Knoten

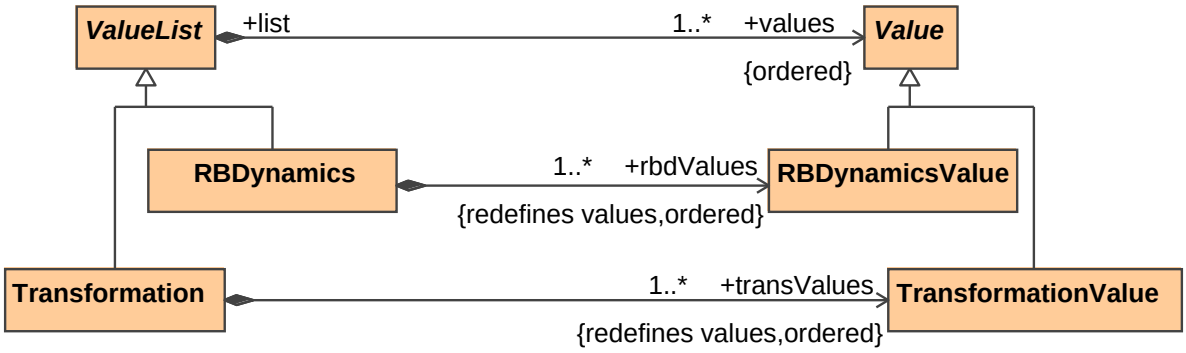



Abbildung 5.23: Listenattribute

**Transformation** Jeder atomare Knoten (Typ `AtomicNode`) besitzt konzeptuell *mindestens* eine räumliche Transformation relativ zu seinem Elternknoten. Soll diese Transformation z. B. für den externen Zugriff durch Anwendungskomponenten zur Verfügung stehen, muss sie im Szenenmodell dem atomaren Knoten als Attribut vom Typ `Transformation` zugeordnet werden. Ein solches Attribut repräsentiert dann die Position und Orientierung des Objekts, das dem atomaren Knoten in der 3D-Szene entspricht. Die Position und Orientierung eines Objektes relativ zum Elternobjekt wird in SSIML also als eine Eigenschaft des Objektes selbst betrachtet. Transformationsattribute werden als Listen von Transformationswerten aufgefasst. I. d. R. kapselt ein Transformationsattribut allerdings nur einen Transformationswert. Repräsentiert ein `Transform`-Attribut aber mehrere *parallele Transformationen* einer Knoteninstanz (vgl. Abschnitt 5.2.2.7), werden diese als Elemente in die Liste aufgenommen (Abbildung 5.23). Somit können ausgewählte Transformationen durch Angabe von Indizes gemäß den in Abschnitt 5.2.2.13 aufgeführten Regeln adressiert werden.

**Sensor** Attribute des Typs `Sensor` können `AtomicNode`-Knoten zugeordnet werden. Ein `Sensor` in einer 3D-Szene kann zur Laufzeit auf Ereignisse reagieren und entsprechende Nachrichten erzeugen, die von Anwendungsobjekten empfangen und ausgewertet werden können. Ein Sensor kann von mehreren Objekten gleichzeitig benutzt werden. Somit ist es auf der Laufzeitplattform möglich, bei der Interaktion mit verschiedenen Objekten nur einen Sensor zu beobachten und die von ihm generierten Nachrichten auszuwerten, anstatt für jedes Objekt einen separaten Sensor beobachten zu müssen. Die gemeinsame Sensornutzung für *mehrere Instanzen desselben Knotens* ist über die Sensoreigenschaft `sharedByCopies` möglich. Gilt *sharedByCopies*, kann eine Instanz eines Knotensensors von allen Instanzen des Knotens gemeinsam benutzt werden und es wird nicht für jede Knoteninstanz eine separate Sensorinstanz erzeugt. In der Notation ist ein solcher gemeinsam genutzter Sensor durch ein Pluszeichen vor dem Sensornamen gekennzeichnet (z.B. *+mySensor*). Die in SSIML definierten Sensortypen orientieren sich an den grundlegenden Sensortypen aus VRML/X3D.

**CollisionSensor, TouchSensor, ProximitySensor und VisibilitySensor** Ein Element, welches einem SSIML-Kollisionssensor (`CollisionSensor`) in einer konkreten 3D-Szene entspricht, reagiert auf eine Kollision des Avatars des Betrachters mit einem der 3D-Objekte, dem es zugeordnet ist. Ein Szenenobjekt vom Typ Berührungssensor (`TouchSensor`) generiert ein Signal, sobald der Benutzer eines der 3D-Objekte, mit denen das Sensorelement verbunden ist, bspw. durch Anklicken mit der Maus selektiert. Eine Laufzeitausprägung eines Entfernungssensors (`ProximitySensor`) generiert hingegen ein Signal, wenn die Distanz zwischen Betrachterstandpunkt und 3D-Objekt einen bestimmten Wert unterschreitet. Eine (Laufzeit-)Instanz eines SSIML-Sichtbarkeitssensors (`VisibilitySensor`) reagiert, wenn ein bestimmtes Objekt für den Benutzer sichtbar wird. Ein Sichtbarkeitssensor dient u. a. dem Zweck, performanceverbrauchende Objektanimationen nur dann durchzuführen, wenn der Benutzer diese Animationen auch wahrnimmt.

**RBDynamics** Ein `RBDynamics`-Attribut repräsentiert physikalische Eigenschaften eines geometrischen Objektes, das als Festkörper betrachtet wird (*RB* steht für *Rigid Body*), z.B. die Masse. Dies kann dann sinnvoll sein, wenn eine physikalisch basierte Animation des Objektes geplant ist. Von den in einem Knotenpfad[3] vom Szenenmodellwurzelknoten bis zu einem Blattknoten enthaltenen Knoten darf nur ein Knoten ein `RBDynamics`-Attribut besitzen, der, samt seiner Nachfahren, den Festkörper repräsentiert. Wie ein `Transformation`-Attribut ist ein `RBDynamics`-Attribut eine Liste, die für jede parallele Transformation des dem Attribut zugeordneten Knotens einen Wert enthält (vgl. Abbildung 5.23). Genaueres zum `RBDynamics`-Attribut ist in Kapitel 7 zu finden.

#### 5.2.2.11 Attribute von Object: Geometry, Appearance, Material und Texture

Einem oder mehreren `Object`-Knoten können ein oder mehrere `ObjectAttribute`s, wie `Geometry-` und `Appearance`-Attribute, zugeordnet werden (Abbildung 5.24).

Ein `Geometry`-Attribut kann mit einem oder mehreren `Object`-Knoten assoziiert werden, um die direkte Manipulation der Mikrogeometrie – also der Eckpunkte einzelner Polygone – der entsprechenden 3D-Objekte zu erlauben. Somit lassen sich z. B. Deformationen darstellen. Da die interne Zusammensetzung eines 3D-Objekts aus mehreren Polygon-Sets möglich ist, kann ein Objektknoten auch mehrere `Geometry`-Attribute besitzen.

`Appearance`-Attribute können entweder Material- oder Texturattribute sein (`Material` bzw. `Texture`). Ein `Material`-Attribut kapselt Objekteigenschaften wie Farbe und Transparenz. Ein Texturattribut bezieht sich hingegen auf die Textureigenschaften eines Objektes, bspw. in Form von Texturbitmaps. Auf der Zielplattform müssen `Appearance`-Attribute

[3] Ein *Knotenpfad* besteht zwischen einem Startknoten *k1* und einem Zielknoten *k2* in einem Szenenmodell mit aufgelösten Kompositionsknoten, wenn der Knoten *k2* ein Nachfahre des Knotens *k1* ist; also dann, wenn der Knoten *k2* von *k1* durch Traversierung von gerichteten Kanten zwischen Eltern- und Kindknoten zu erreichen ist. Ein Knotenpfad kann durch die Sequenz der bei der Traversierung überstrichenen Knoten repräsentiert werden, einschließlich des Start- und des Zielknotens. Werden bspw. bei der Traversierung der Kanten zwischen *k1* und *k2* die Knoten *k1a* und *k1b* überstrichen (außer *k1* und *k2*), wird der Knotenpfad vom Knoten *k1* zum Knoten *k2* durch die Sequenz *(k1,k1a,k1b,k2)* repräsentiert. Knotenpfade sind nicht zu verwechseln mit den weiter unten beschriebenen *Kompositionspfaden*.

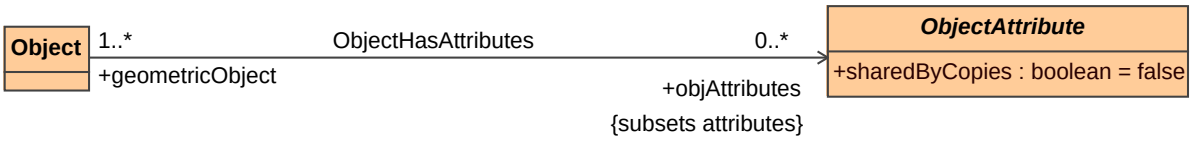



Abbildung 5.24: Attribute von `Object`

auf Materialien und Texturen abgebildet werden, die sich im gekapselten Inhalt des Objekts befinden. Wie bereits erwähnt, können `Appearance`-Attribute von mehreren Objekten gemeinsam benutzt werden. Analog zur Verwendung von Sensoren ist ein `Appearance`-Element durch Setzen der Eigenschaft `sharedByCopies` auf *true* von Instanzen ein und desselben Knotens gemeinsam nutzbar. In der Notation trägt ein gemeinsam genutztes `Appearance`-Attribut ebenfalls ein Pluszeichen vor dem Attributnamen.

#### 5.2.2.12 Attributfelder

Ein Attribut besitzt verschiedene Felder. Bei einem `Transformation`- oder `RBDynamics`-Attribut, das eine Liste von Werten darstellt, besitzt jedes Element der Liste die gleichen Felder. Attributfelder spielen eine besondere Rolle bei der Animations- und Verhaltensspezifikation mit SSIML. Sie werden daher in Kapitel 7 noch im Detail beschrieben.

#### 5.2.2.13 Adressierung von (Laufzeit-)Instanzen von Knoten und Attributen im SSIML-Modell: Kompositionspfade

*SSIML-Kompositionspfade* dienen dem Zweck der Adressierung von Instanzen von atomaren Knoten und deren Attributen. Die Syntax der Pfade orientiert sich teilweise an XPath [Wor07a] und OCL [Obj06b]. Sie wurde allerdings auf die Anforderungen des genannten Zwecks zugeschnitten, um eine möglichst kompakte Pfadnotation zu erhalten. Eine Adressierung liefert immer eine geordnete Liste (d. h. Sequenz, Folge) von Instanzen oder Instanzattributen zurück und kann somit als *Abfrage auf dem Szenengraphen zur Laufzeit* verstanden werden. In diesem Abschnitt soll der Aufbau von Kompositionspfaden eingehender betrachtet werden.

**Identifizierung und Auswahl von einzelnen Knoteninstanzen** Um eine oder mehrere Instanzen eines atomaren Knotens auszuwählen, bietet SSIML die Möglichkeit der Angabe so genannter *Instanzselektoren,* die in Klammern hinter dem Knotennamen auftreten können. Aufbau und Funktionsprinzip der Instanzselektoren werden nachfolgend dargestellt.

Besitzt ein atomarer Knoten mehrere (Laufzeit-)Instanzen, wird jeder (Laufzeit-)Instanz ein eindeutiger Code in Form eines Tupels positiver ganzer Zahlen (z.B. *(1,1,22)*) zugeordnet, um jede Instanz eindeutig identifizieren zu können. Dieser *Instanzcode* wird wie folgt gebildet.

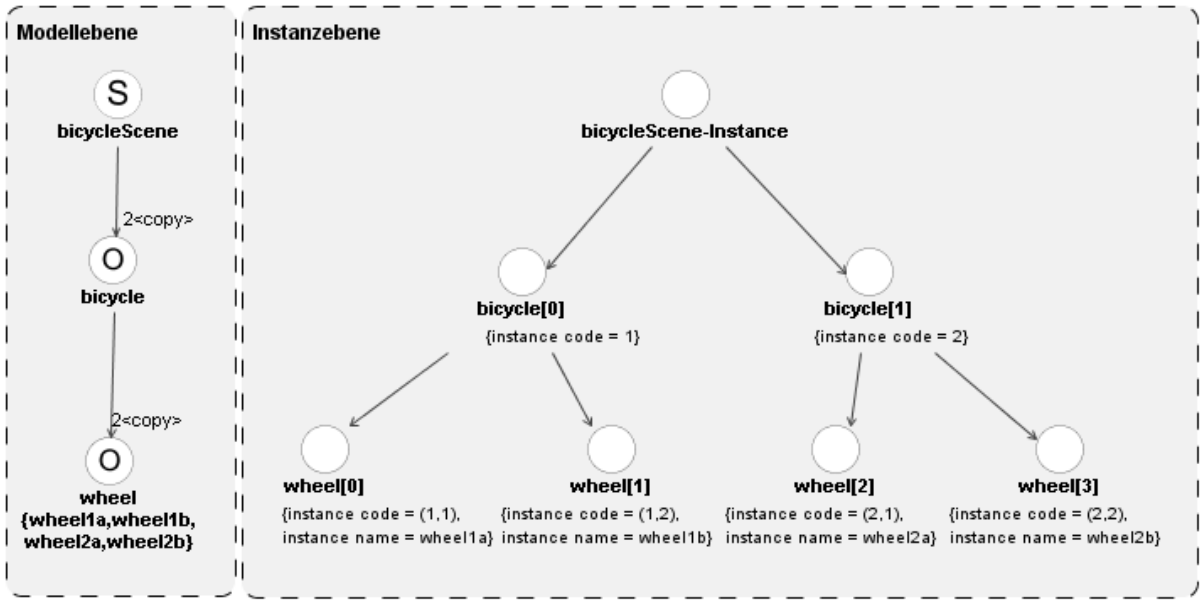



Abbildung 5.25: Beispiel eines Szenenmodells und einer Instanz des Modells

Betrachtet werden ein atomarer Elternknoten $p$ und ein atomarer Kindknoten $c$, zwischen denen eine primäre Kante $r$ besteht. Betrachtet werden weiterhin eine Instanz $p'$ von $p$ und die Instanzen $c'_i$ von $c$, wobei $p'$ auf Instanzebene (im Szenengraph) Elternelement der Elemente $c'_i$ ist. Für die Indexmenge $I$ gilt:

$I = \{i \in \mathbb{N} | (r.createCopies \rightarrow 1 <= i <= r.numberOfElements)$
$\wedge (\neg r.createCopies \rightarrow i = 1)\}$

Die Instanzen $c'_i$ bekommen zunächst den gleichen Code wie $p'$. Gilt zusätzlich $r.createCopies \wedge r.numberOfElements>1$, wird dem Code eines Elements $c'_i$ der Index $i$ angehängt. Somit kann jede Knoteninstanz durch die Verbindung des Namensraumes, des Knotennamens und des Instanzcodes eindeutig identifiziert werden. Dies soll durch eine weitere Variante des Fahrradbeispiels verdeutlicht werden (Abbildung 5.25). In einer Szene gibt es zwei Fahrräder (*Bicycles*), von denen jedes zwei Räder (*Wheels*) besitzt. Die Radobjekte des ersten Fahrrades haben dann die Instanzcodes *(1,1)* und *(1,2)*, während die Radobjekte des zweiten Fahrrades die Instanzcodes *(2,1)* und *(2,2)* besitzen. Beschrieben wird eine Knoteninstanz über ihren Code in SSIML in der Syntax *<Knotenname><Instanzcode>*, also z. B. *wheel(1,2)*.

Eine zweite Möglichkeit, eine bestimmte Instanz eines Knotens zu identifizieren, bietet ihr *Instanzindex*. Einem atomaren Knoten kann eine Liste seiner Instanzen zugeordnet werden. Somit kann jede Instanz über ihren Index in der Liste adressiert werden (die Indizes beginnen bei Null). Die Elemente der Liste sind nach ihren Instanzcodes geordnet, wobei die Elemente zunächst nach der ersten Komponente des Codes sortiert sind, innerhalb der dadurch entstehenden Elementgruppen nach der zweiten Codekomponente usw. Bspw. hat eine Instanz mit dem Code *(2,1)* einen niedrigeren Listenindex als eine Instanz mit

dem Code *(2,3)* und eine Instanz mit dem Code *(1,3)* hat einen niedrigeren Index als eine Instanz mit dem Code *(2,1)*. So würde eine Liste der Instanzen von *wheel* im vorangegangen Beispiel vier Elemente enthalten, nämlich die Instanzen mit den Codes *(1,1), (1,2), (2,1)* und *(2,2)* (in dieser Reihenfolge). *wheel(1,1)* hätte den Index Null, *wheel(1,2)* den Index Eins usw. Die Notation für die Auswahl einer Knoteninstanz über ihren Index folgt der Syntax *<Knotenname>'['<Instanzindex>']'*, also z. B. *wheel[0]*.

Die letzte Möglichkeit, eine Instanz eines Knotens zu wählen, beruht auf der Angabe des Instanznamens, sofern ein solcher Name für die interessierende Instanz vergeben wurde. Bspw. selektiert *wheel<wheel1a>* die Instanz von *wheel* mit dem Namen *wheel1a*. Zu beachten ist, dass der Instanzname in Winkelklammern eingeschlossen wird.

**Auswahl einer Teilmenge von Knoteninstanzen** Um aus der Gesamtmenge der Instanzen eines atomaren Knotens die Auswahl einer Teilmenge zu treffen, gibt es verschiedene Möglichkeiten. Die Auswahl ist über Instanzcodes, über Instanzindizes und über Instanznamen möglich. Eine vierte Option bieten die weiter unten aufgeführten *Selektionskriterien*. Eine Liste aller Instanzen eines Knotens wird durch den Knotennamen repräsentiert. Z. B. repräsentiert *wheel* die Liste aller Instanzen des Knotens *wheel* - geordnet nach den im vorangehenden Abschnitt beschriebenen Kriterien (Instanzcodes). Anstatt eine einzelne Instanz aus der Liste über die Angabe eines Indexes zu wählen, können durch Angabe mehrerer Indizes auch mehrere Instanzen selektiert werden. Es kann eine Indexmenge angegeben werden, deren Elemente entweder einzelne Indizes oder Indexintervalle sein können. Bspw. selektieren *wheel[0,1,2]* (Indexmenge mit Indizes), *wheel[0..2]* (Indexmenge mit einem Intervall) und *wheel[0..1, 2]* (Indexmenge mit Intervall und Einzelindex) dieselben Instanzen von *wheel*, nämlich *wheel[0]*, *wheel[1]* und *wheel[2]* (siehe Abbildung 5.25).

Die Möglichkeit der Angabe von Intervallen und Listen besteht auch für die Einzelkomponenten von Instanzcodes. Codekomponenten, die die Auswahl einer Teilmenge von Instanzen zur Folge haben, müssen zusätzlich mit einer geschwungenen Klammer umschlossen werden, um sie von den restlichen Codekomponenten klar zu separieren. *wheel(1,{1,2})* und *wheel(1,{1..2})* sind äquivalent und selektieren bspw. die Instanzen *wheel(1,1)* und *wheel(1,2)*. *wheel({1,2},{1..2})* hingegen wäre äquivalent zur Angabe von *wheel*, da alle Instanzen von *wheel* selektiert würden.

Weiterhin kann ein Asterisk (*) in einem Intervall anstelle des (zur Instanzselektion führenden) maximalen Indexes bzw. des Maximalwertes einer Instanzcodekomponente verwendet werden. Bspw. werden durch die Angabe *wheel(2,{1..*})* ebenso wie durch die Angabe *wheel[2..*]* die Knoteninstanzen *wheel[2]* und *wheel[3]* selektiert.

Auch über Instanznamen können mehrere Instanzen eines Knotens selektiert werden, indem eine Liste von Instanznamen angegeben wird, wie z.B. *wheel<wheel1a, wheel1b>*.

Die erwähnte vierte Möglichkeit besteht darin, Instanzen *dynamisch zur Laufzeit in Abhängigkeit von bestimmten Attributbelegungen* und *Resultaten von Methodenaufrufen* zu wählen. Diese Art der Instanzwahl ist allerdings *nur bei Interrelationen* (vgl. Abschnitt 5.3) und nicht innerhalb des Szenenmodells möglich. Die zur Auswahl herangezogenen Attribute

können entweder Attribute des instanziierten Knotens oder lesbare Attribute von *repräsentierten Klassen* des Knotens sein, d.h. Klassen, die über eine *representation*-Relation mit dem Knoten verbunden sind (vgl. Abschnitt 5.3.1.3). Ein Attribut einer Klasse (damit sind an dieser Stelle Klassen- und Instanzattribute gemeint) ist dadurch erkennbar, dass in der Notation vor dem Attribut der Name der Klasse steht, wobei Klassenname und Attributname durch einen Doppelpunkt separiert sind (z.B. *Product:category*). Ebenso kann an die Stelle des Attributes ein Methodenname treten (z.B. *Product:getPrice()*). Durch die Verknüpfung von Knotenattributfeldnamen (vgl. Abschnitt 7.2.4), Klassenattribut- und Methodennamen, konkreten Werten und definierten Operatoren, entsteht ein boolescher Ausdruck, das *Selektionskriterium*. Ausgewählt werden alle Instanzen, für die zum Zeitpunkt der Adressierung das *Selektionskriterium* zutrifft, d.h. den Wert *wahr* (*true*) annimmt. Das Selektionskriterium steht in der Notation nach dem Knotennamen, umschlossen von doppelten Winkelklammern. Erlaubte Operatoren sind für alle Operandentypen die relationalen Operatoren *Gleich* (=) und *Ungleich* (*!=*). Für boolesche Operanden sind weiterhin der unäre Operator *Nicht* (*!*) und die logischen Operatoren *Und* (*&*) und *Oder* (*|*) anwendbar. In Verbindung mit numerischen Operanden können zusätzlich die relationalen Operatoren *Größer* (*>*), *Größer-gleich* (*>=*), *Kleiner* (*<*) und *Kleiner-gleich* (*<=*) eingesetzt werden. Die Rangordnung der Operatoren entspricht derjenigen in OCL: Der *Nicht*-Operator hat eine höhere Priorität als die relationalen Operatoren, welche wiederum eine höhere Priorität als die logischen Operatoren besitzen. Durch explizite Klammerung kann die Ausführungsreihenfolge der Operationen in einem Selektionskriterium angepasst werden. Atomare (nicht zusammengesetzte) Operanden sind *Attribut(feld)namen, Methoden* von repräsentierten Klassen und *konkrete Werte*. *Konkrete Werte* sind z. B. konkrete Zahlenwerte und Zeichenketten, boolesche Werte (repräsentiert durch die Schlüsselwörter *TRUE* und *FALSE*) sowie Tupel konkreter Werte, notiert in geschweiften Klammern (z. B. repräsentiert das Wertetripel *{1,0,0}* die Farbe *Rot* im RGB-Format). Repräsentiert ein Attribut, ein Knotenattributfeld oder ein Methodenresultat eine Liste von Werten, kann ein einzelner Wert durch Angabe eines Indexes identifiziert werden (etwa *objTrans[2].translation={1,0,0}* oder *objTrans[2].translation[0]=1*). Besitzt eine Methode Parameter, können für diese alle auch an anderer Stelle im Selektionskriterium verwendbaren typkonformen Operanden eingesetzt werden.

Im nachfolgenden Beispiel werden die Instanzen eines Knotens *laptopObject* gewählt, die eine Masse kleiner als zwei Kilogramm (Knotenattributfeld *mass*) aufweisen, vom Hersteller *A* oder *B* stammen, deren Preis kleiner *2000* ist und die keine Subnotebooks sind (Attribute *manufacturer*, *price*, *screenSize* und Methode *isSubnotebook* der repräsentierten Klasse *Product*):

```
laptopObject<<rigidBodyAttr.mass<2
    & (Product:manufacturer='A'|Product:manufacturer='B')
    & Product:price<2000
    & !Product:isSubnotebook(Product:screenSize)>>
```

Ganz gleich, welche Art von Instanzselektor verwendet wird - eine Selektion von Knoteninstanzen führt immer zu einer *nach ihren Instanzcodes geordneten Instanzliste*, die dann nur noch die selektierten Elemente enthält. Aus einer Ergebnisliste können erneut Instanzen

ausgewählt werden, indem ein weiterer Instanzselektor angehängt wird, wie nachfolgendes Beispiel demonstriert:

```
laptopObject[1..5]<<Product:price<2000>>
```

**Aufbau von Kompositionspfaden** Ein Kompositionspfad besteht aus mehreren Komponenten. Prinzipiell ist ein solcher Pfad folgendermaßen aufgebaut (Beschreibungsform angelehnt an die EBNF für XML [Wor06, Abschnitt 6]):

```
<Kompositionspfad>:=
    <Absoluter Pfad>|<Relativer Pfad>
<Absoluter Pfad>:=
    '/'<Pfadkomponente>
<Relativer Pfad>:=
    <Pfadkomponente>
<Pfadkomponente>:=
    <Kompositionsknotenkomponente><Atomknotenkomponente>
    <Attributkomponente>
<Kompositionsknotenkomponente>:=
    ((<Kompositionsknotenname>|<Kompositionsknotenmenge>)':')*
<Atomknotenkomponente>:=
    (<Atomknoteninstanzen>|
    (<Atomknoteninstanzenmenge>(<Instanzselektor>)*))
<Attributkomponente>:=
    ('.'<Attributname>(<Attributindizes>)?)?
<Kompositionsknotenmenge>:=
    '{'<Kompositionsknotenname>(','<Kompositionsknotenname>)*'}'
<Atomknoteninstanzenmenge>:=
    '{'<Atomknoteninstanzen>(','<Atomknoteninstanzen>)*'}'
<Atomknoteninstanzen>:=
    <Atomknotenname>(<Instanzselektor>)*
```

Instanzen können entweder *relativ* oder *absolut* adressiert werden. Zunächst soll die absolute Adressierung betrachtet werden. Ein absoluter Pfad beginnt immer mit einem Schrägstrich (/). Die nachfolgende *Kompositionskomponente* des Pfades entspricht hier dem Namensraum des Knotens, dessen Instanz(en) adressiert werden soll(en). Anschließend erfolgt die Auswahl der *Atomknoteninstanzen*, wie oben beschrieben wurde. Sollen Attribute der Knoteninstanzen (also Attributinstanzen) adressiert werden, muss zusätzlich eine *Attributkomponente* angegeben werden. Die Attributkomponente besteht aus einem vorangestellten *Punkt,* dem Namen des Attributes (z.B. *object1Attribute*) und, falls das Attribut eine Liste von Werten beinhaltet, der Angabe der entsprechenden Listenindizes zur Adressierung bestimmter Werte. Für die Angabe der Indizes gelten dieselben Regeln wie die in Abschnitt 5.2.2.13 aufgeführten Regeln zur Auswahl von Knoteninstanzen mittels Instanzindizes bzw. Indexmengen (z.B. *object1Attribute[0,1..2]*).

Bei relativer Adressierung erfolgt die Adressierung relativ zum Kompositionsbereich des adressierenden Elements, dem *Kontext* des adressierenden Elements. Es können nur Elemente adressiert werden, die im Kontext des adressierenden Elementes zugänglich sind. Dies sind Elemente, die sich entweder im Kontext selbst oder in Unterkompositionsbereichen des Kontexts befinden. Der einführende Schrägstrich wird bei relativer Adressierung nicht angegeben. Beispiele für syntaktisch korrekt aufgebaute Kompositionspfade sind:

```
/composedNode1:innerComposedNode:object1
composedNode2:object2[0..2,3,5..*]
composedNode2:object3(2,3).obj3Attribute
```

Die Angabe von Mengen von Kompositionsknoten erlaubt es zusätzlich, Instanzen verschiedener atomarer Knoten mittels eines einzigen Kompositionspfades zu adressieren. Somit können mehrere Kompositionspfade zusammengefasst werden. Bspw. können die Pfade */car:wheel1:rim* und */car:wheel2:rim* zusammengefasst werden zu */car:{wheel1, wheel2}:rim* und die Kompositionspfade */house:chair[0..3].materialAttribute* und */house:table[1,2].materialAttribute* zu */house:{chair[0..3],table [1,2]}.materialAttribute.*

Aus einer Ergebnismenge von Instanzlisten verschiedener atomarer Knoten kann mittels der Benutzung von Instanzselektoren, die auf alle Instanzlisten gleichermaßen angewendet werden, eine Unterauswahl getroffen werden. Die Selektoren werden in der oben beschriebenen Syntax notiert. Z. B. ist *house:{chair, table}<<Product:material='Wood'>>* äquivalent zu *house:{chair<<Product:material='Wood'>>, table<<Product:material='Wood'>>}*.

**Besonderheiten bei der Darstellung von Kompositionspfaden in SSIML-Diagrammen** In der grafischen SSIML-Notation werden bei einem relativen Kompositionspfad die Pfadbestandteile im Kompositionspfad unterdrückt, die Elementen wie Knoten und Attributen entsprechen, die sich direkt im Kontext des adressierenden Elementes befinden. Dies ist möglich, da die so versteckten Informationen aus dem Modell anderweitig eindeutig hervorgehen (i. d. R. durch die Kante, welche die Beziehung zwischen dem adressierenden und dem adressierten Element repräsentiert). Werden ausgewählte Instanzen (oder deren Attribute) eines atomaren Knotens adressiert, der sich im selben Kompositionsbereich wie das adressierende Element befindet, treten in einem SSIML-Diagramm daher *Instanzselektoren* ohne voranstehende Knoten- bzw. nachfolgende Attributnamen, z.B. *[4..7].[3,4]* statt *object[4..7].objAttribute[3,4]*, oder sogar leere Pfade auf.

**Anwendung von Kompositionspfaden** Kompositionspfade können an verschiedenen Stellen in einem SSIML-Modell auftreten, wie Tabelle 5.1 illustriert. Jeder Pfad kann grundsätzlich als relativer oder absoluter Pfad dargestellt werden. Eine Umschaltung zwischen diesen Darstellungsvarianten sollte durch ein SSIML-Modellierungswerkzeug unterstützt werden. Allerdings ist eine Darstellung als absoluter Pfad in einigen Fällen unabdingbar, wie in Abschnitt 5.3.2 verdeutlicht wird.

Einige Elemente erlauben die Angabe einer Menge von Pfaden, andere nicht (vgl. Spalte *Anzahl erlaubter Pfade* in Tabelle 5.1). Wiederum ist bei einigen adressbeinhaltenden Elementen die Verwendung von Instanzselektoren nicht erlaubt, bspw. beim Attribut `innerParents`, da sonst die Regeln zur Bildung von Knoteninstanzen, wie in Abschnitt 5.2.2.7 beschrieben, verletzt würden.

### 5.2.3 Notation von Szenenmodellelementen

In diesem Abschnitt wird auf die Notation von Elementen des Szenenmodells eingegangen. Konventionen für die Beschriftung der Elemente sind in EBNF-Form [Wor06, Abschnitt 6] integriert.

Tabelle 5.1: Auftreten von Kompositionspfaden

| Metaklasse | Attribut | Anzahl erlaubter Pfade | Instanz-selektion | Adressierendes Element | Adressiertes Element |
|---|---|---|---|---|---|
| PrimaryCNChildRelation | innerParents | 1 | nein | Kindknoten | innerer Elternknoten |
| SecondaryCNChildRelation | innerParents | 1..* | nein | Kindknoten | innere Elternknoten |
| SecondaryParentChildRelation | targetNodeInstances | 0..* | ja | Elternknoten | Kindknoteninstanzen |
| InterrelationElement | je nach Unterklasse | 0..* | je nach Unterklasse | Klasse | Knoten Knoteninstanzen/ Knotenattributinstanzen |

Tabelle 5.2: Notation von SSIML-Knoten

| Knotentyp | Notation | Beispiel |
|---|---|---|
| Scene | S<br><*Szenenname*> | S<br>carScene |
| Object | O<br><*Objektname*>(<*Instanznamensliste*>)?<br>(':'<*Inhaltstypdefinition*>)? | O<br>oldCar{wartburg,trabant}:Car |
| Group | G<br><*Gruppenname*>(<*Instanznamensliste*>)? | G<br>cars |
| ComposedNode | C<br><*Knotenname*>':'<*Subgraphendefinition*> | C<br>carBody:BodyComposite |

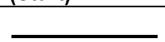
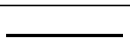
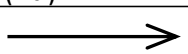
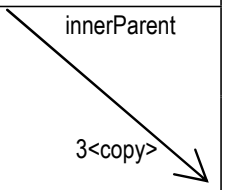
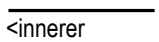
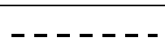
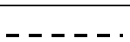
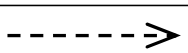
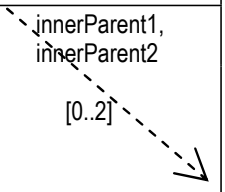
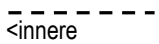
#### 5.2.3.1 Notation von Knoten und Knotenbeziehungen

Tabelle 5.2 fasst die Notation für die aus Meta-Klassen instanziierbaren SSIML-Knoten zusammen. Ein Knoten wird in SSIML allgemein – wie auch aus den bereits betrachteten Beispielen hervorgeht – als Kreisform dargestellt, die ein Symbol enthält, das den Knotentyp spezifiziert. Der Name des Knotens steht i. d. R. direkt beim Knoten.

Neben der hier vorgestellten, graphemorientierten Notation für SSIML-Elemente, die zunächst erlernt werden muss, wurde – motiviert durch eine Expertenbefragung (vgl. Abschnitt 11.1) – in der Arbeit von Jahn [Jah07] eine alternative, bildorientierte Notation erarbeitet, welche die Ansprüche kreativ ausgerichteter Entwickler in stärkerem Maß berücksichtigt. Diese Notation wird in Abschnitt 11.2 kurz präsentiert. Sie dient nicht als Ersatz, sondern als Ergänzung zu der in diesem Abschnitt vorgestellten und in der vorliegenden Arbeit durchgehend verwendeten Notation.

Tabelle 5.3 illustriert die Notation von Knotenbeziehungen. Eine Primärkante wird als durchgezogene Linie mit Pfeil vom Eltern- zum Kindknoten dargestellt. Eine Sekun-

Tabelle 5.3: Notation von Knotenbeziehungen

| Beziehungstyp | Elternknoten (Startknoten) | Ende 1 (Start) | Mittelteil | Ende 2 (Ziel) | Beispiel |
|---|---|---|---|---|---|
| Primary | AtomicNode |  |  |  (<Elementzahl> ('<ref>'\| '<copy>'))? |  innerParent 3<copy> |
| | ComposedNode |  <innerer Elternknoten> | | | |
| Secondary | AtomicNode |  |  <Zielknoten-instanzen> |  |  innerParent1, innerParent2 [0..2] |
| | ComposedNode |  <innere Elternknoten> | | | |

därkante wird hingegen als unterbrochene Linie mit Pfeil vom Eltern- zum Kindknoten dargestellt. Ebenfalls wird gezeigt, wie sich die Attribute `createCopies` und `numberOfInstances` in der Notation widerspiegeln. Der Ausdruck `<ref>` steht dabei für die Erzeugung einer Kindknoteninstanz mit parallelen Transformationen im Szenengraphen des Zielformats, bei `<copy>` werden mehrere Instanzen erzeugt.

#### 5.2.3.2 Notation von Attributen

Die Notation von Attributen und die Notation der Beziehungen zu den ihnen zugeordneten Knoten sind in Tabelle 5.4 dargestellt.

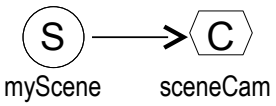
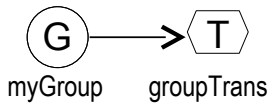
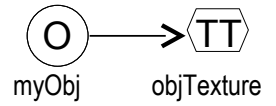
## 5.3 Interrelationenmodell

Nach der Modellierung der Szene folgt die Festlegung von Beziehungen (Interrelationen) zwischen Elementen der Szene (Knoten, Attribute) und Komponenten der Rahmenanwendung (Klassen) im so genannten Interrelationenmodell. Zunächst sollen das Metamodell und die Notation von Elementen des Interrelationenmodells beschrieben werden, bevor in Abschnitt 5.5 einzelne Aspekte der Modellierung mit SSIML noch einmal anhand eines Beispiels verdeutlicht werden.

### 5.3.1 Metamodell

Die Meta-Klasse `ApplicationClass` in SSIML entspricht der Meta-Klasse `Class` der UML. Zwischen dem SSIML-Metamodell und dem UML-Metamodell besteht demzufolge eine Abhängigkeit. Um Klassen aus einem vorhandenen UML-Modell in einem SSIML-Modell verwenden zu können, müssen diese einschließlich ihrer Paketinformation im Sinne einer Modelltransformation zunächst aus dem UML-Modell extrahiert und anschließend zu SSIML-

Tabelle 5.4: Die Notation von Attributen

| Attributtyp | Notation (ohne Beschriftung) | Attribut von | Beispiel |
|---|---|---|---|
| Camera | C | Scene |   |
| PointLight | PL | | |
| DirectionalLight | DL | | |
| SpotLight | SL | | |
| Viewpoint | V | | |
| Gravity | GR | | |
| CollisionSensor | CS | Atomic-Node |   |
| TouchSensor | TS | | |
| ProximitySensor | PS | | |
| VisibilitySensor | VS | | |
| Transformation | T | | |
| RBDynamics | RB | | |
| Material | M | Object |   |
| Texture | TT | | |
| Geometry | G | | |

`ApplicationClass`-Elementen konvertiert werden. Der Klassenname wird im Attribut `name` von `ApplicationClass` abgelegt, das Paketpräfix im Attribut `package`. Der Wert des Attributs `qualifiedName` wird aus dem Paket- und dem Klassennamen abgeleitet. Abbildung 5.26 gibt einen Überblick über die wichtigsten Bestandteile des Interrelationenmodells: Klassen, das Szenenmodell und Beziehungen zwischen Klassen und Elementen des Szenenmodells.

Durch die Etablierung von Beziehungen zwischen Komponenten der Rahmenanwendung (d. h. Klassen) und Elementen des Szenenmodells werden die in Beziehung gesetzten Szenenmodellelemente für die Rahmenanwendung sichtbar und die Zugriffsmethoden auf die Elemente werden festgelegt. Die Gesamtmenge der Beziehungen zwischen Anwendungskomponenten und dem Szenenmodell definiert gewissermaßen eine für die Anwendung maßgeschneiderte Schnittstelle des Szenenmodells.

Die Modellierung von Interrelationen findet auf der Wurzelkompositionsebene des Szenenmodells statt. Damit ist gewährleistet, dass der Entwickler einen Überblick über die Anwendungskomponenten erhält, die mit entsprechenden Szenenelementen verdrahtet sind. Ein SSIML-Modellierungswerkzeug sollte allerdings das Einblenden von Klassen und Inter-

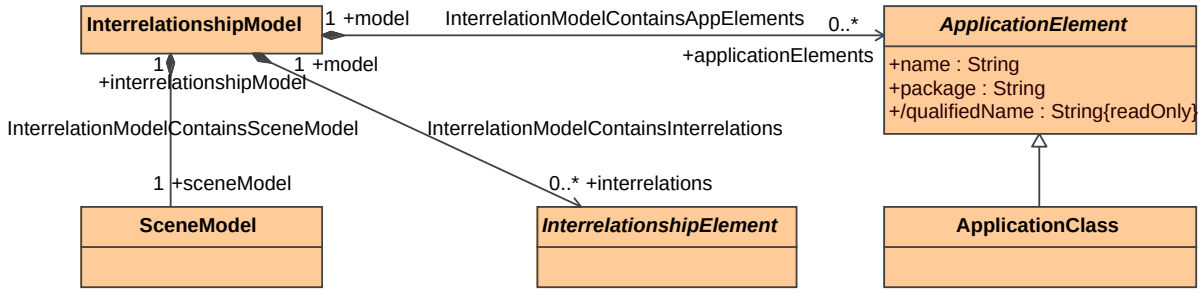



Abbildung 5.26: Bestandteile des Interrelationenmodells

relationen auf unteren Kompositionsebenen unterstützen, damit deutlich wird, auf welche Elemente innerhalb eines Kompositionsknotens eine Klasse zugreift.

Abbildung 5.27 zeigt die Vererbungshierarchie der Typen von Beziehungen (`InterrelationshipElements`) zwischen Klassen und Szenenmodellelementen in SSIML. Es werden drei Hauptarten von Beziehungen unterschieden: Aktions-, Ereignis- und Repräsentationsbeziehungen (`ActionRelationship`, `EventRelationship` und `RepresentationRelationship`).

Grundlage für die Aufstellung der Beziehungsarten war eine Analyse verschiedener Szenengraph-APIs, wie dem External Authoring Interface [Int04a], dem Scene Access Interface [Int08a], Java3D [J3D] und Open Inventor [SC92]. Dabei wurden grundlegende Funktionalitäten zur Beeinflussung der 3D-Szene und zur Überwachung von in der Szene auftretenden Zustandsänderungen oder Ereignisse zur Laufzeit untersucht, die von den APIs bereitgestellt werden. Die Aktionsbeziehungen *Adds-Child*, *Removes-Child*, *Replaces-Content* und *Modifies* repräsentieren grundlegende Aktionen zur Manipulation von Objekten der Szene. *Removes-Child*- und *Adds-Child*-Beziehungen erlauben in Kombination die Definition komplexerer Aktionen zum Austausch von Kindknoten. *Replaces-Content* ermöglicht ein einfaches Austauschen umfangreicherer Inhalte wie kompletter 3D-Modelle, die u. U. als separate Ressourcen vorliegen. Der Einsatz der *Modifies*-Relation ist dann sinnvoll, wenn nur einzelne Eigenschaften von Objekten verändert werden sollen, nicht aber das Gesamtobjekt. Die Mächtigkeit der eingeführten Beziehungstypen wurde in verschiedenen Anwendungsbeispielen geprüft. Dabei wurde eine weitere Beziehungsart, die *Reads*-Beziehung, identifiziert, die dann nutzbringend einsetzbar ist, wenn Werte aus der 3D-Szene gelesen, aber nicht verändert werden sollen, z. B. um Kollisionen von Objekten abzufragen.

In Foley et al. [FvFH95, S. 292] werden ebenfalls das *Hinzufügen*, *Entfernen* und *Austauschen* von Informationen als grundlegende Aktionen der Veränderung der Szene aufgeführt. Hinzu kommt das Extrahieren von Informationen aus der Szene, z. B. zum Zweck der Anzeige [FvFH95, S. 292].

Der Fluss von Ereignis-Nachrichten von der Szene zur Rahmenanwendung kann über SSIML-Ereignisbeziehungen modelliert werden. In APIs wie dem EAI und dem SAI, die zum Zugriff auf VRML- bzw. X3D-Szenen dienen, wie auch dem API Open Inventor werden

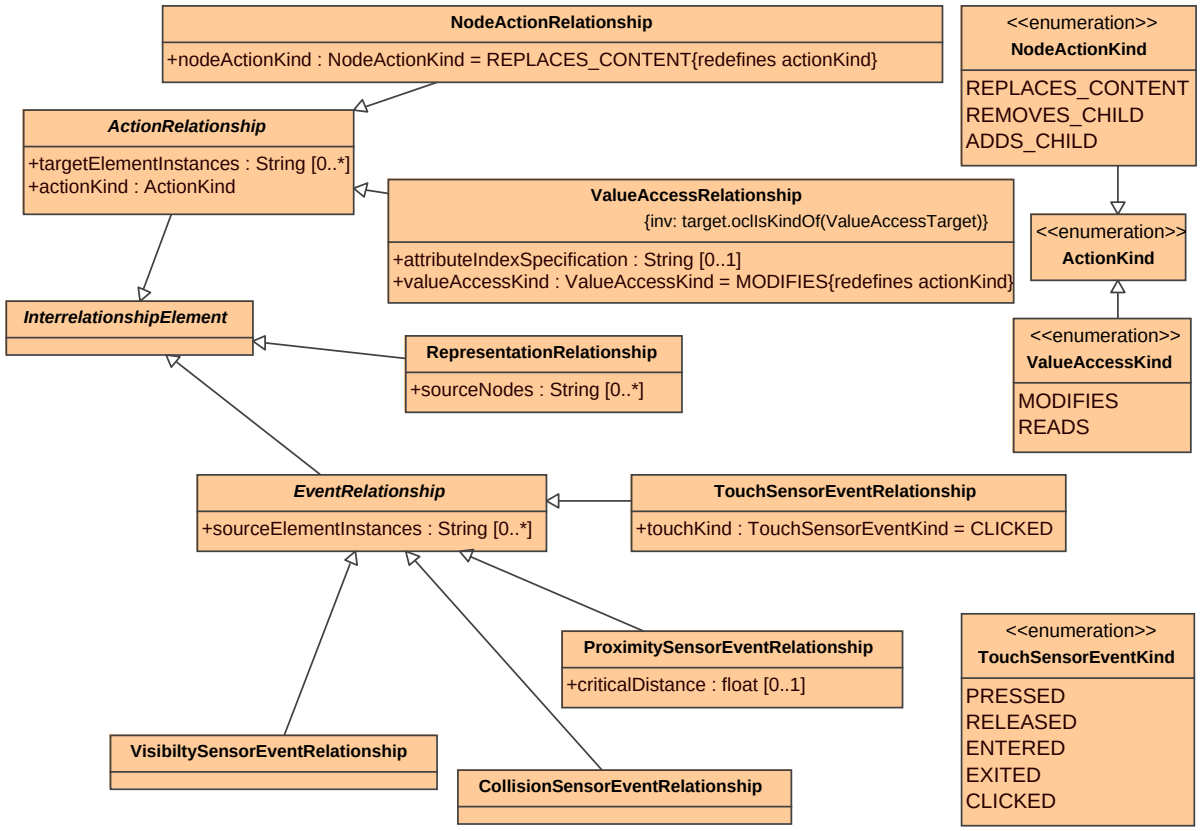



Abbildung 5.27: Die Meta-Klasse `InterRelationshipElement` und ihre Unterklassen

Ereignis-Nachrichten von Sensoren generiert, die assoziierte 3D-Objekte überwachen. In Java3D können derartige Sensoren durch so genannte `Behaviour`-Objekte abgebildet werden. Subtypen des SSIML-Typs `EventRelationship` entsprechen grundlegenden Sensor-Typen der APIs.

Die *Repräsentations*-Beziehung wurde eingeführt, um Metadaten mit einem graphischen zu Objekt assoziieren. Somit können 3D-Objekte beliebig mit Zusatzinformationen verknüpft werden, die sich nicht direkt auf die graphische Darstellung beziehen. Eine solche Information wäre etwa der Preis eines Produktes, das durch ein entsprechendes 3D-Objekt repräsentiert wird. Die genannten Beziehungstypen werden nachfolgend ausführlicher behandelt.

#### 5.3.1.1 Aktionsbeziehungen

Eine Aktionsbeziehung zwischen einer Aktionsquelle wie einer Klasse und einem Aktionsziel wie einem Knoten oder Knotenattribut sagt aus, dass Instanzen der Aktionsquelle alle oder die über Kompositionspfade ggf. angegebenen Instanzen von Aktionszielen (Attribut `targetElementInstances` von `ActionRelationship`) zur Laufzeit modifizieren können. Es existieren verschiedene Arten von Aktionsbeziehungen, die über die Attribute `node-`

`AccessKind` oder `valueAccessKind` eines `NodeActionRelationship`-Elements bzw. eines `ValueAccessRelationship`-Elements festgelegt werden (Abbildung 5.27). Die Abbildungen 5.28 und 5.29 zeigen die Zusammenhänge zwischen Aktionsquellen (`ActionSources`), Aktionsbeziehungen (`ActionRelationships`) und Aktionszielen (`ActionTargets`) wie im Metamodell:

1. Eine *Adds-Child-* oder *Removes-Child*-Beziehung (`nodeActionKind` ist `ADDS_CHILD` bzw. `REMOVES_CHILD`) besteht zwischen einer Aktionsquelle (hier: `ApplicationClass`-Element) und einem Elternknoten. Diese Beziehung bedeutet, dass die Aktionsquelle Kindknoten zu dem Elternknoten hinzufügen oder von ihm entfernen kann. Ist ein Kompositionsknoten das Zielelement, muss über das Attribut `targetElementInstances` angegeben werden, welche Instanzen atomarer Elternknoten innerhalb des Kompositionsknotens adressiert werden.

2. Eine *Replaces-Content*-Beziehung (`nodeActionKind` ist `REPLACES_CONTENT`), welche zwischen einer Aktionsquelle und einem Objektknoten oder einem Kompositionsknoten, der den adressierten Objektknoten enthält, bestehen kann, drückt aus, dass die Aktionsquelle den gekapselten Inhalt des Knotens vollständig austauschen kann. Ein Austausch gegen den leeren Inhalt ist ebenfalls möglich. Dies kann ggf. das Ersetzen kompletter Geometrien (bspw. ein Auswechseln der Felgengeometrie im Fahrzeugkonfiguratorbeispiel) erleichtern.

3. Die *Modifies*-Relation (`valueAccessKind` ist `MODIFIES`) besteht zwischen einer Aktionsquelle und einem Attribut eines Knotens der Szene oder einem Kompositionsknoten. Besteht die Beziehung zu einem Kompositionsknoten, spezifiziert das von `ActionRelationship` geerbte Attribut `targetElementInstances` die adressierten Attributinstanzen *innerhalb des Kompositionsknotens* als Menge von Kompositionspfaden. Repräsentiert das Ziel-Knotenattribut eine Werteliste, was bei einem Transformationsattribut oder bei einem `RBDynamics`-Attribut der Fall ist, kann über das Element `attributeIndexSpecification` der Beziehung zusätzlich zur Angabe von `targetElementInstances` festgelegt werden, welche Werte in der Liste adressiert werden. Eine *Modifies*-Relation drückt generell aus, dass die an der Beziehung teilnehmende Klasse die Werte der adressierten Attributinstanzen ändern kann. Es wird nicht definiert, *wie* diese Änderung durchgeführt wird. Dies kann z. B. später im generierten Programmcode festgelegt werden. Dadurch lassen sich komplexe Animationen und Verhaltensbeschreibungen realisieren. Erweiterte Möglichkeiten zur Verhaltens- und Animationsspezifikation mit SSIML werden in Kapitel 7 beschrieben. Die *Modifies*-Beziehung schließt einen lesenden Zugriff nicht aus, da es z. B. notwendig sein kann, aktuelle Attributwerte auf der Basis zuvor gelesener Werte zu berechnen.

4. Die *Reads*-Beziehung (`valueAccessKind` ist `READS`) entspricht i. W. der zuvor beschriebenen *Modifies*-Beziehung. Eine *Reads*-Beziehung zwischen einer Aktionsquelle und einem Knotenattribut spezifiziert allerdings, dass Instanzen der Aktionsquelle auf Instanzen des Knotenattributes ausschließlich lesend zugreifen.

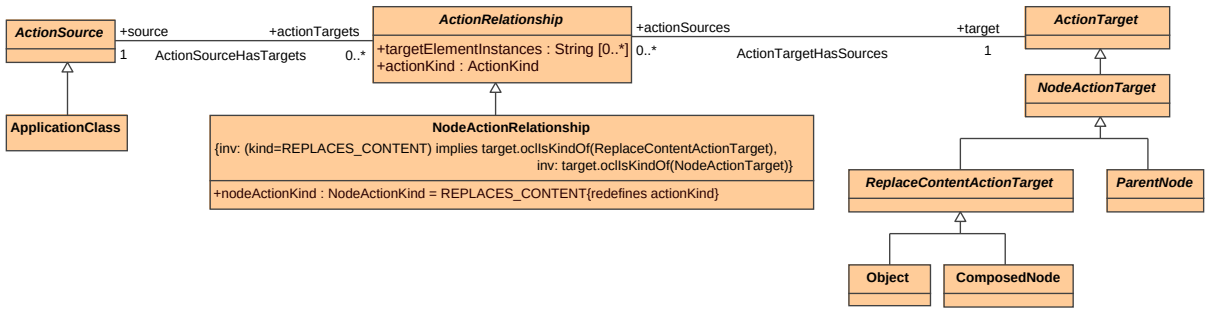



Abbildung 5.28: Aktionsbeziehungen zwischen Klassen und Knoten

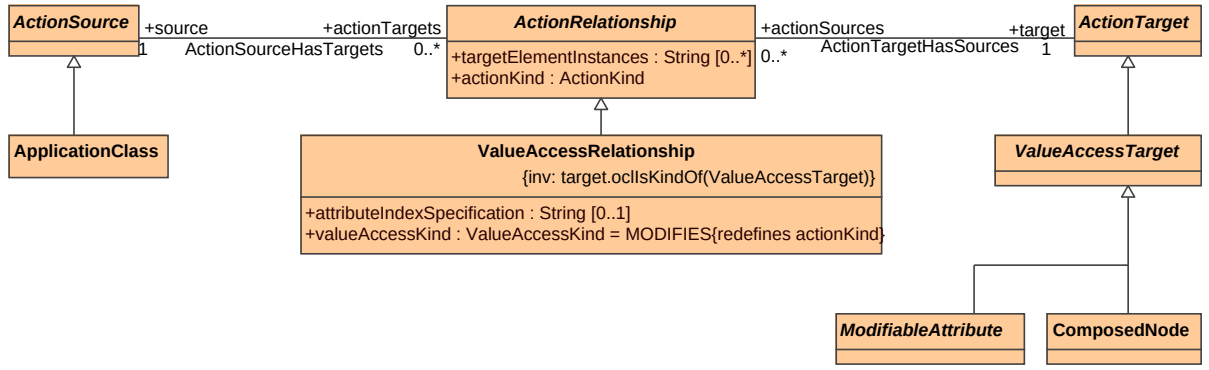



Abbildung 5.29: Aktionsbeziehungen zwischen Klassen und Attributen

Zu beachten ist, das Aktionen voneinander abhängig sein können. Das Ausführen einer Aktion zur Laufzeit kann bedingen, dass sich eine andere Aktion nicht mehr ausführen lässt; z. B. kann eine *Remove-Child*-Aktion bewirken, dass ein Kindknoten entfernt wird, auf dem ebenfalls Aktionen definiert waren.

Ein Ausschnitt aus einem Interrelationenmodell mit einer Aktionsbeziehung ist in Abbildung 5.30 zu sehen. In dieser Abbildung besitzt das SSIML-Objekt `wheel` ein Transformationsattribut `wheelTrans`. Zur Laufzeit können Instanzen der Klasse `Wheel-Updater` das Attribut `wheelTrans` eines `wheel`-3D-Objektes manipulieren, etwa um Objektrotationen zu visualisieren.

### 5.3.1.2 Ereignisbeziehungen

Eine Ereignisbeziehung (`EventRelationship`) zwischen einer Ereignisquelle, wie einem (möglicherweise in einem Kompositionsknoten gekapselten) Sensor, und einem Ereignisziel, wie einer Klasse, drückt aus, dass das Ereignisziel Nachrichten von den im Attribut `sourceElementInstances` des `EventRelationship`-Elements vermerkten Instanzen der Ereignisquelle empfängt. Die Instanzen der Ereignisquelle werden als Kompositionspfade spezifiziert. Wurden keine Instanzen explizit ausgewählt, empfängt das Ereignisziel Nachrichten von allen Instanzen der Ereignisquelle. Abbildung 5.31 stellt den beschriebenen

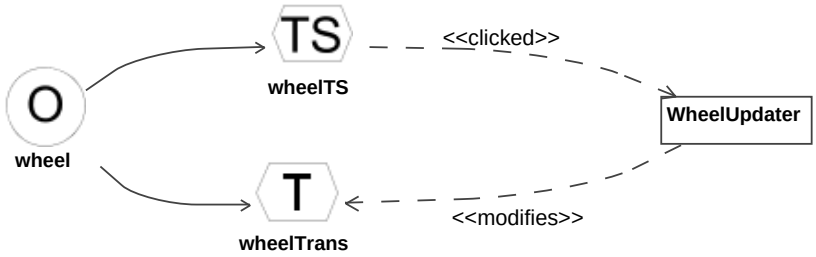



Abbildung 5.30: Ein einfaches Beispiel für Interrelationen

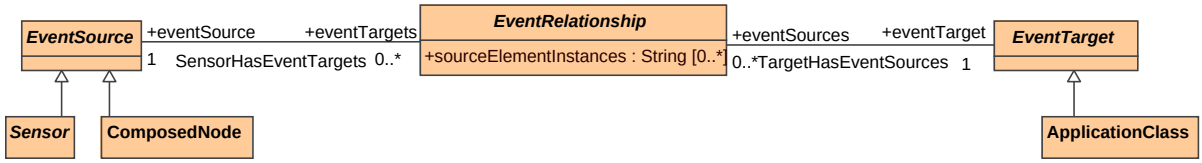



Abbildung 5.31: Ereignisbeziehungen zwischen Sensoren und Klassen

Sachverhalt im Metamodell dar. Das mit einem Sensor assoziierte 3D-Objekt wird auch als *Sensoreigentümer* bezeichnet. Hat eine Benutzeraktion ein den Sensoreigentümer betreffendes Ereignis zur Folge, generiert – wie bereits erwähnt – der entsprechende Sensor eine Ereignisnachricht (*Event*). Ein bestimmter Sensortyp kann spezielle Ereignisarten melden, wie eine Selektion oder das Sichtbarwerden des Sensoreigentümers. Z. B. versendet ein Berührungssensor (`TouchSensor`) Nachrichten, wenn der Zeiger eines Zeigegerätes (i. d. R. der Mauszeiger) in den Oberflächenbereich des *Sensoreigentümers* gezogen wurde (das `TouchSensor`-Attribut `touchKind` trägt den Wert `ENTERED`), aus diesem Bereich herausgezogen wurde (`touchKind` ist `EXITED`) oder der Zeigegerätknopf innerhalb des Bereichs gedrückt (`touchKind` ist `PRESSED`), losgelassen (`touchKind` ist `RELEASED`) oder gedrückt und anschließend wieder losgelassen wurde (`touchKind` ist `CLICKED`). Weitere Sensortypen wurden bereits in Abschnitt 5.2.2.10 beschrieben. An dieser Stelle soll nur erwähnt werden, dass die Meta-Klasse `CollisionSensorEventRelationship` über ein optionales Attribut `criticalDistance` verfügt, mit dem sich diejenige Distanz zwischen Betrachterstandpunkt und Sensoreigentümer in Metern festlegen lässt, bei deren Unterschreitung der Sensor ein Event generiert.

Im Beispiel in Abbildung 5.30 ist der `TouchSensor wheelTS` dargestellt, der über ein `TouchSensorEventRelationship`-Element (`touchKind` ist im Beispiel `CLICKED`) mit der Anwendungskomponente `WheelUpdater` verbunden ist. Damit können `WheelUpdater`-Instanzen zur Laufzeit auf entsprechende Nachrichten der `wheelTS`-Instanzen reagieren, z. B. durch Rotation der `wheel`-3D-Objekte.

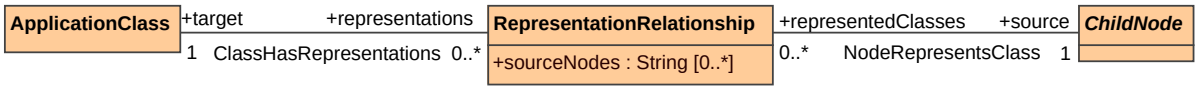



Abbildung 5.32: Repräsentationsbeziehungen zwischen Klassen und Knoten

#### 5.3.1.3 Repräsentationsbeziehungen

Eine Repräsentationsbeziehung (`RepresentationRelationship`) zwischen einem – ggf. in einem Kompositionsknoten gekapselten, über das Attribut `sourceNodes` der Beziehung adressierten – Knoten und einer Klasse deutet an, dass der Knoten die als Fachklasse fungierende Klasse *repräsentiert*. Abbildung 5.32 verdeutlicht diesen Zusammenhang im Metamodell. Repräsentierte Klassen erlauben es somit, SSIML-Modelle semantisch zu erweitern. Eine Instanz einer repräsentierten Klasse stellt zur Laufzeit Metadaten über das assoziierte 3D-Objekt bereit. Bspw. kann ein 3D-Objekt `laptopObject` in einer Verkaufsanwendung ein Produkt (`Product`) repräsentieren. Um diesen Sachverhalt auszudrücken, muss im Interrelationenmodell ein Knoten `laptopObject` mit einer Fachklasse `Product` über eine Repräsentationsbeziehung verbunden werden. Auf Instanzebene wird jeder Instanz eines repräsentierenden Knotens eine Instanz der repräsentierten Klasse zugeordnet. Eine Instanz der Klasse `Product` kann Informationen wie den Preis oder eine Beschreibung des Laptops speichern. Der Laptop wird durch eine Instanz des Knotens `laptopObject` repräsentiert.

### 5.3.2 Notation von Interrelationen

Modellelemente, die Anwendungsklassen repräsentieren, werden in SSIML genau wie in UML als Rechtecke dargestellt, welche die Klassennamen enthalten. Beziehungen zwischen Klassen und Szenenelementen werden jeweils als gerichtete Kante in Form einer Strichlinie mit Pfeilspitze vom Quell-Element zum Zielelement dargestellt. Quell-Elemente sind je nach Beziehungsart Elemente vom Typ `ActionSource`, `EventSource` oder `ApplicationClass` (bei Repräsentationsbeziehungen). Zielelemente sind Elemente vom Typ `ActionTarget`, `EventTarget` und `ChildNode` (bei Repräsentationsbeziehungen). Die Beschriftung der Kanten erfolgt entsprechend der Syntax *'<<'<Beziehungstyp>'>>' (<Szenenelemente>)? (<Distanzspezifikation>)?*. Die Bezeichnungen für die Beziehungstypen sind in Tabelle 5.5 aufgeführt.

Der jeweilige Beziehungstyp und – wenn nicht eindeutig aus der Beziehung hervorgehend – die adressierten Szenenelemente werden an die Kante angetragen. Die adressierten Szenenelemente (bei `RepresentationRelationship` Knoten, bei `ActionRelationship` Aktionszielinstanzen wie Knoteninstanzen oder Attributinstanzen, bei `EventRelationship` Ereignisquelleninstanzen wie `Sensor`-Instanzen) werden als Menge von Kompositionspfaden dargestellt. Abbildung 5.33 zeigt zwei weitere Beispiele: Bei Beispiel (a) muss das innerhalb des Kompositionsknotens adressierte Element durch einen Kompositionspfad spezifiziert werden, bei Beispiel (b) geht das adressierte Element ohne weiteres aus der

Tabelle 5.5: Notation der Beziehungsarten des Interrelationenmodells

| Übergeordneter Beziehungstyp | Beziehungsunterart | Beziehungstyp-Bezeichner |
|---|---|---|
| NodeActionRelationship | ADDS_CHILD | addsChild |
| | REMOVES_CHILD | removesChild |
| | REPLACES_CONTENT | replaces |
| ValueAccessRelationship | MODIFIES | modifies |
| | READS | reads |
| TouchSensorEventRelationship | CLICKED | clicked |
| | PRESSED | pressed |
| | RELEASED | released |
| | ENTERED | entered |
| | EXITED | exited |
| ProximitySensorEventRelationship | - | proximity |
| VisibiltySensorEventRelationship | - | visibility |
| CollisionSensorEventRelationship | - | collision |
| RepresentationRelationship | - | represents |


Abbildung 5.33: Notation von Interrelationen

Beziehung hervor. *Instanzselektoren* dürfen in den Kompositionspfaden *außer bei Repräsentationsbeziehungen* verwendet werden. Die *Distanzspezifikation* in der Pfeilbeschriftung ist optional bei Beziehungen zu `ProximitySensor`-Elementen in der Syntax *'$criticalDistance='<Distanzwert>* anzugeben. Der Distanzwert entspricht dabei dem Wert des Attributes `criticalDistance` eines Modellelementes vom Typ `ProximitySensor`.

Anwendungsklassen befinden sich im Wurzelkompositionsbereich des Szenenmodells, da sie von „außerhalb“ auf die Szene zugreifen. Entsprechend sind Kompositionspfade bei Interrelationenmodellelementen zu bilden. Ein Modellierungswerkzeug kann aber die Anzeige von Klassen auch innerhalb einer Subgraphendefinition zulassen, da es so in einigen Fällen möglich ist, die graphische Darstellung des Modells übersichtlicher zu gestalten. In diesem Fall müssen die Pfade an einem `InterrelationshipElement` absolut dargestellt werden, da im Falle der Verwendung ein und derselben Subgraphendefinition durch mehrere Kompositionsknoten nicht eindeutig hervorgeht, über welchen der Kompositionsknoten die Klasse auf das Zielelement zugreift.

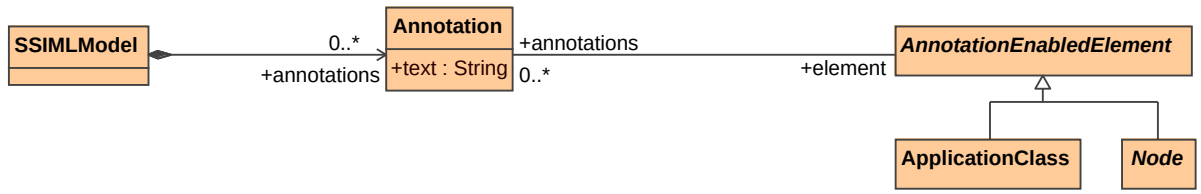



Abbildung 5.34: Kommentierung von Modellelementen

## 5.4 Kommentare

Ein SSIML-Modell kann Kommentare (`Annotations`) enthalten, die mit Elementen des Typs `AnnotationEnabledElement` verbunden werden können (Abbildung 5.34). Die Notation von Kommentaren entspricht der aus der UML bekannten Notation.

## 5.5 Beispiel

Ein Beispiel, um verschiedene Aspekte der Modellierung mit SSIML im Zusammenhang zu demonstrieren, ist ein Fahrzeugkonfigurator, wie er auf den Webseiten vieler namhafter Automobilhersteller (z. B. Ford [For]) zu finden ist. 3D-Systeme zur Produktkonfiguration sind ebenfalls Gegenstand aktueller Forschung und Entwicklung (vgl. [SAAT05]).

Der Konfigurator erlaubt die Auswahl diverser Accessoires für ein Fahrzeug. Für eine so zusammengestellte Konfiguration kann ein Gesamtpreis errechnet werden. Das Beispiel wurde hier etwas vereinfacht, um die Charakteristika von SSIML in angemessenem Umfang zu demonstrieren. Ähnliche, aber deutlich umfangreichere Beispiele waren Gegenstand der Evaluierung von SSIML (vgl. Abschnitt 11.2). Abbildung 5.35 vermittelt einen Eindruck von der graphischen Benutzungsschnittstelle des Konfigurators.

Es existieren fünf unterschiedliche Ansichten (*Views*), die verschiedene Aspekte der Konfiguratoranwendungsdaten aufzeigen: Die 3D-Ansicht (*3D-View*), die Kategorien-Liste (*Categories View*), die Zubehörliste (*Accessories View*), die Beschreibungsansicht (*Description View*) und die Ansicht des Gesamtpreises (*Full Price View*). Der Benutzer erhält die Beschreibung einer bestimmten Fahrzeugkomponente, indem er auf den entsprechenden Fahrzeugteil in der 3D-Ansicht klickt. Die Kategorienliste enthält unterschiedliche Zubehörkategorien, wie Felgen (*rims*), Spoiler usw. Wurde eine Kategorie ausgewählt, werden alle Produkte innerhalb dieser Kategorie in der Zubehörliste angezeigt. Wählt der Benutzer ein Produkt aus der Zubehörliste, wird eine Produktbeschreibung in der Beschreibungsansicht ausgegeben. Durch Selektion der Checkbox neben einem Produkt in der Zubehörliste wird das 3D-Modell des Produktes dem Fahrzeugmodell in der 3D-Ansicht hinzugefügt (bspw. wird ein Spoiler am hinteren Ende des Fahrzeuges angefügt). Wird die Produkt-Checkbox wieder deselektiert, wird das 3D-Modell des Produktes aus der 3D-Ansicht entfernt. Die Gesamtpreisansicht visualisiert den Gesamtpreis der gewählten Fahrzeugkonfiguration und

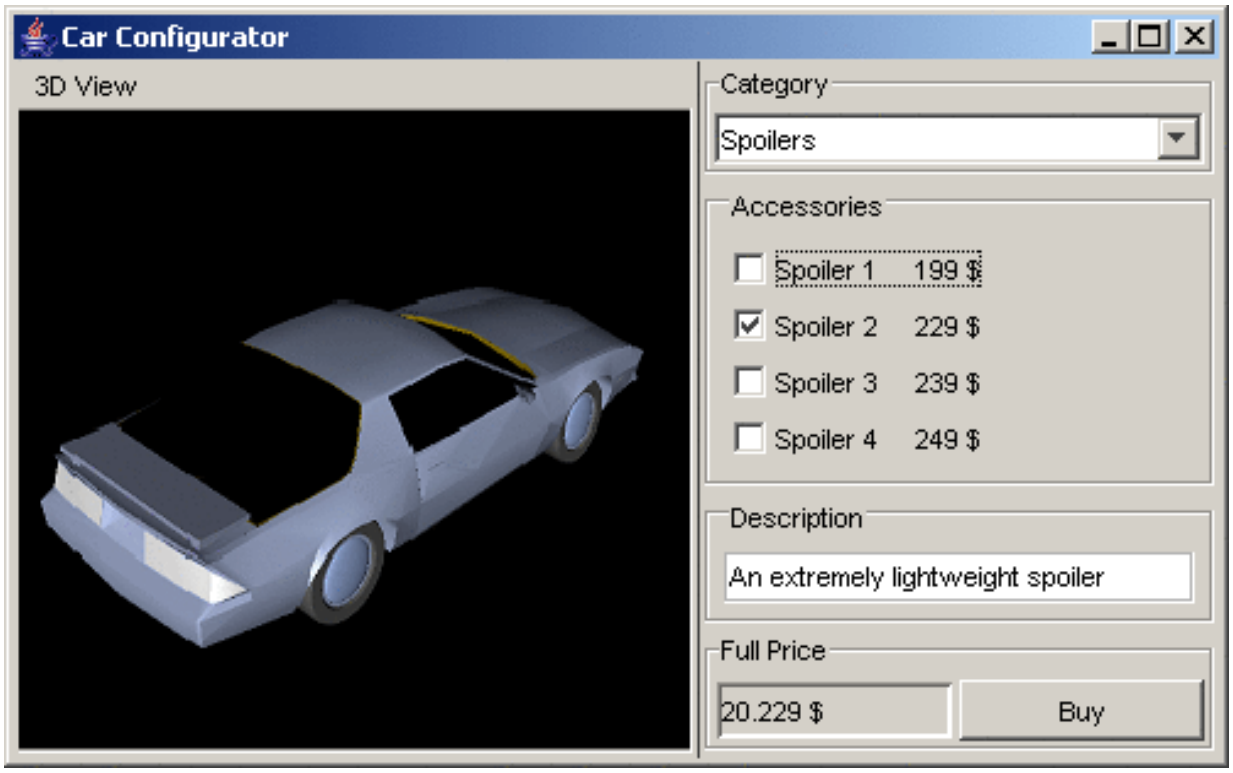



Abbildung 5.35: Benutzungsschnittstelle des Fahrzeugkonfigurators

erlaubt das Einleiten eines Kaufvorganges des konfigurierten Fahrzeuges über einen Button (*Buy*-Button).

Abbildung 5.36 zeigt einen vereinfachten Ausschnitt aus dem Klassendiagramm des Fahrzeugkonfigurators. Konkrete Zubehörteile, wie Spoiler (*spoiler*) oder Felgen (*rims*), sind Instanzen der Produktklasse (`Product`). Die oben erwähnten unterschiedlichen Ansichten erben von der Klasse `DefaultView`. Jeder `DefaultView` hat Zugriff auf die Daten des Produktkatalogs (`Catalog`) und visualisiert bestimmte Produktinformationen. Da die Daten des Katalogs von der Anwendung nicht verändert werden dürfen, besteht lediglich ein lesender Zugriff auf den Katalog. Auf der Ebene der Ansichten müssen über den Anwendungsrahmen Nachrichten ausgetauscht werden, sobald die Kategorien- oder Produktauswahl in einer Ansicht geändert wurde. Eine Nachricht muss Informationen über die neu gewählten Kategorien- und Produktdaten enthalten, damit alle Ansichten korrekt aktualisiert werden können. Da der konkrete Mechanismus zum Austausch von Nachrichten zwischen den einzelnen Ansichten für die Beziehungen zwischen Programmkomponenten der Anwendung und der 3D-Szene keine große Relevanz besitzt, wurde auf die Darstellung entsprechender Anwendungsbestandteile in Abbildung 5.36 verzichtet.

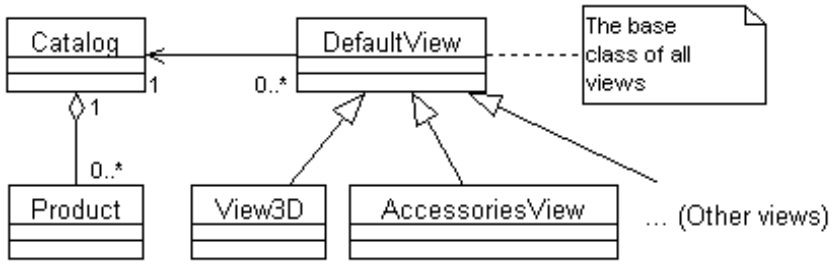



Abbildung 5.36: Vereinfachter Ausschnitt aus dem Klassendiagramm des Fahrzeugkonfigurators

### 5.5.1 Szenenmodell

In Abbildung 5.37 ist das Diagramm der *Subgraphendefinition* `BodyComposite` zu sehen. `BodyComposite` definiert einen wiederverwendbaren Subgraphen für das Szenenmodell des Fahrzeugkonfiguratorbeispiels, der den Fahrzeugrumpf beschreibt. Der Objektknoten `carBody` ist die Wurzel des `BodyComposite`-Subgraphen. `carBody` hat die Kindknoten `backDoor`, `frontHood`, `leftDoor` und `rightDoor`. Der Inhaltstyp eines Objektknotens steht jeweils hinter dem Knotennamen und ist von diesem durch einen Doppelpunkt getrennt. Der Knoten `door` hat zwei Instanzen `leftDoor` und `rightDoor,` denen die gleiche Geometrie zugrunde liegt[4]. Die modellierten Attribute deuten an, dass externe Anwendungsbestandteile später auf sie zugreifen sollen. Bspw. soll die Transformation `bodyTrans` erreichbar sein, um dem Benutzer das Rotieren des 3D-Fahrzeugmodells zur Inspektion zu erlauben. `doorTrans` sollte adressierbar sein, um etwa ein Öffnen und Schließen der Fahrzeugtüren zu visualisieren, z. B. als Reaktion eines Mausklicks auf eine der Türen. Dafür hat jede `door`-Instanz einen Berührungssensor `doorTS`, der – etwa nach einem Mausklick – eine Nachricht zur Rahmenanwendung senden kann.

Die Gesamtansicht des Fahrzeug-Szenenmodells zeigt Abbildung 5.38. Man sieht, dass im gewählten Beispiel alle Knoten über Primärkanten verbunden sind. Wie bereits erwähnt, ist der Wurzelknoten des Szenenmodells ein Knoten vom Typ `Scene`. Der Name des durch das Modell repräsentierten Szenentyps (`carScene`) ist am Wurzelknoten angetragen. Der Kompositionsknoten `bodyComp` mit dem Inhaltstyp `BodyComposite` ist das Kind des Szenenwurzelknotens. Für den Szenengraphen einer Zielplattform bedeutet das, dass anstelle des Knotens `bodyComp` der durch `BodyComposite` definierte Subgraph in die Gesamtszenenstruktur eingefügt wird. Der Kompositionsknoten `bodyComp` definiert wiederum einen Namensraum (nämlich `bodyComp`) für alle Knoten, die er enthält. Gleichfalls repräsentiert der Knoten `bodyComp` eine bestimmtes Auftreten der `BodyComposite`-Subgraphenstruktur innerhalb des Szenenmodells.

[4]Sofern die anvisierte Zielplattform es untersützt, muss die Transformation einer der `Door`-Instanzen zur Laufzeit einen negativen Skalierungsfaktor besitzen, um zwei spiegelsymmetrische Fahrzeugtüren zu erhalten. In der VRML-Spezifikation ist die Möglichkeit der Spiegelung bspw. nicht vorgesehen, beim VRML-Nachfolger X3D hingegen schon.

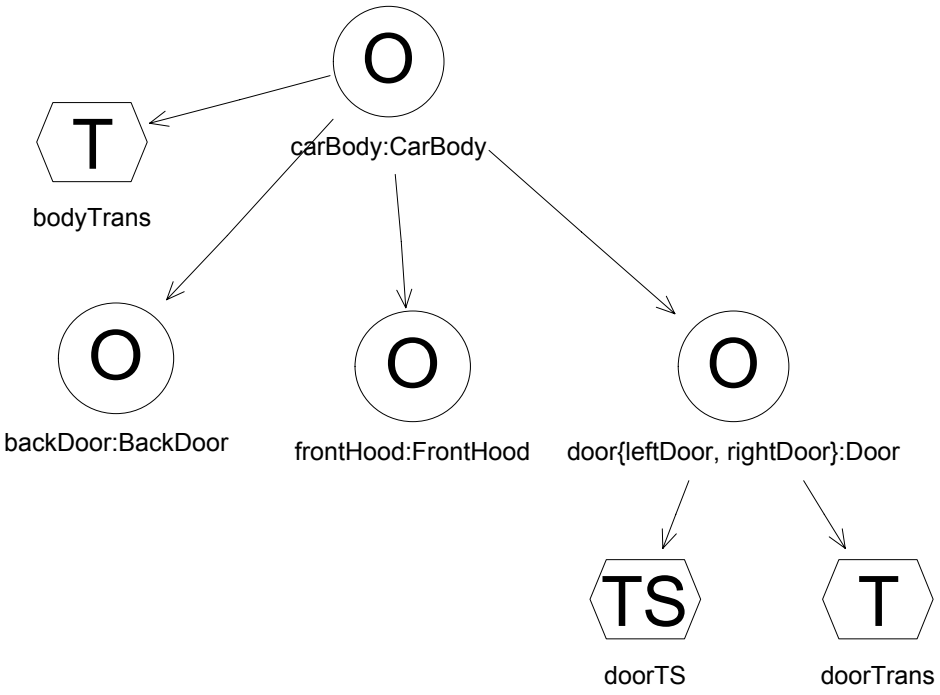



Abbildung 5.37: Die Subgraphendefinition `BodyComposite`

Die Knoten `wheel` und `spoiler` sind Kindknoten von `bodyComp`. `spoiler` ist mit dem Knoten `backDoor` verbunden, der im Kompositionsknoten `bodyComp` enthalten ist (vgl. Abbildung 5.38) und die Rolle des *inneren Elternknotens* von `backDoor` übernimmt. `wheel` ist mit `carBody` verbunden, da an der entsprechenden Kante die Bezeichnung `carBody` notiert ist. `wheel` besteht wiederum aus den Knoten `tire` und `rim`. Die Multiplizität am Endpunkt der Kante zum Knoten `wheel` gibt an, dass in einem konkreten Szenengraphen dem Element `carBody` vier `wheel`-Elemente in der Transformationshierarchie untergeordnet sind. Der Ausdruck `<ref>` hinter der Multiplizität weist darauf hin, dass zur Laufzeit genau eine Instanz von `wheel` mit vier parallelen räumlichen Transformationen existiert. Dies kann von Vorteil sein, wenn man den gekapselten Inhalt des `rim`-Knotens (d.h. die Felgengeometrie) austauschen möchte. Anstatt den Austausch viermal vorzunehmen, reicht es, die Felgengeometrie an einer Stelle im Arbeitsspeicher zu ersetzen.

Die Knoten `rim` und `spoiler` besitzen Berührungssensorattribute (`TouchSensor`-Attribute). Die Rahmenanwendung kann zur Laufzeit Nachrichten von Instanzen dieser Sensorattribute abfangen, etwa um Beschreibungen der in der 3D-Ansicht angeklickten Felgen- und Spoilerobjekte in der Beschreibungsansicht anzuzeigen (s. Abbildung 5.35).

Kommentare können ebenfalls in ein SSIML-Diagramm eingebunden werden (Abbildung 5.38 zeigt, dass `tire` einen Kommentar besitzt). Kommentare können später mit im Zielformat generiert und mit geeigneten Werkzeugen inspiziert werden.

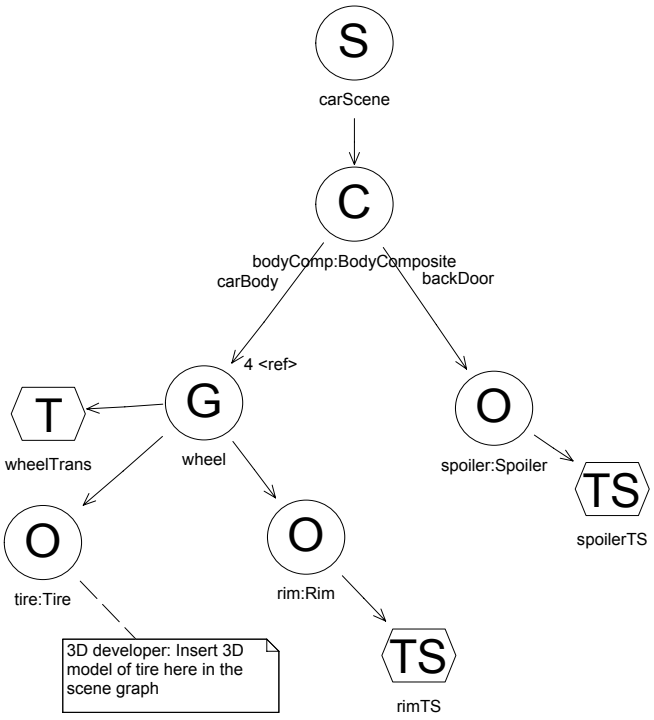



Abbildung 5.38: Das SSIML-Szenenmodell des Fahrzeugkonfigurators

### 5.5.2 Interrelationenmodell

In Abbildung 5.39 ist das Diagramm des Interrelationenmodells für das Fahrzeugkonfiguratorbeispiel zu sehen. Die Klasse `View3D` kann die Transformationen des `carBody`-3D-Objektes sowie die Transformationen der Instanzen der Knoten `door` und `wheel` modifizieren, um bspw. das Rotieren des gesamten Fahrzeuges, das Öffnen- und Schließen der Türen und das Drehen der Räder zu visualisieren. Die Adressierung von Knoteninstanzen innerhalb des Kompositionsknotens `bodyComp` ist über die Angabe von Kompositionspfaden möglich. Weiterhin kann die Klasse `View3D` die Geometrien der Instanzen der Knoten `rim` und `spoiler` austauschen (mit `<<replaces>>` annotierte Beziehungen).

Die Klasse `View3D` erhält Nachrichten über aufgetretene Klicks durch die Berührungssensoren `rimTS`, `spoilerTS` und die Sensoren, welche mit den Türobjekten verbunden sind. Anzumerken ist, dass der Kompositionspfad `door<leftDoor, rightDoor>.doorTS` auch durch den Pfad `door.doorTS` ersetzt werden könnte, da beide Pfade dieselbe Menge von `Sensor`-Instanzen (nämlich alle Instanzen des `Sensor`-Attributs `doorTS`) adressieren. Durch die Anbindung an die Sensoren wird es Instanzen der Klasse `View3D` (im Beispiel gibt es nur eine Instanz der Klasse `View3D`) möglich, auf die empfangenen Nachrichten angemessen zu reagieren, z. B. durch Anzeige von Informationen zu den angeklickten 3D-Objekten in der Beschreibungsansicht (*Description View*) oder durch Starten einer Animation des angeklickten Objektes wie dem Öffnen und Schließen einer Fahrzeugtür.

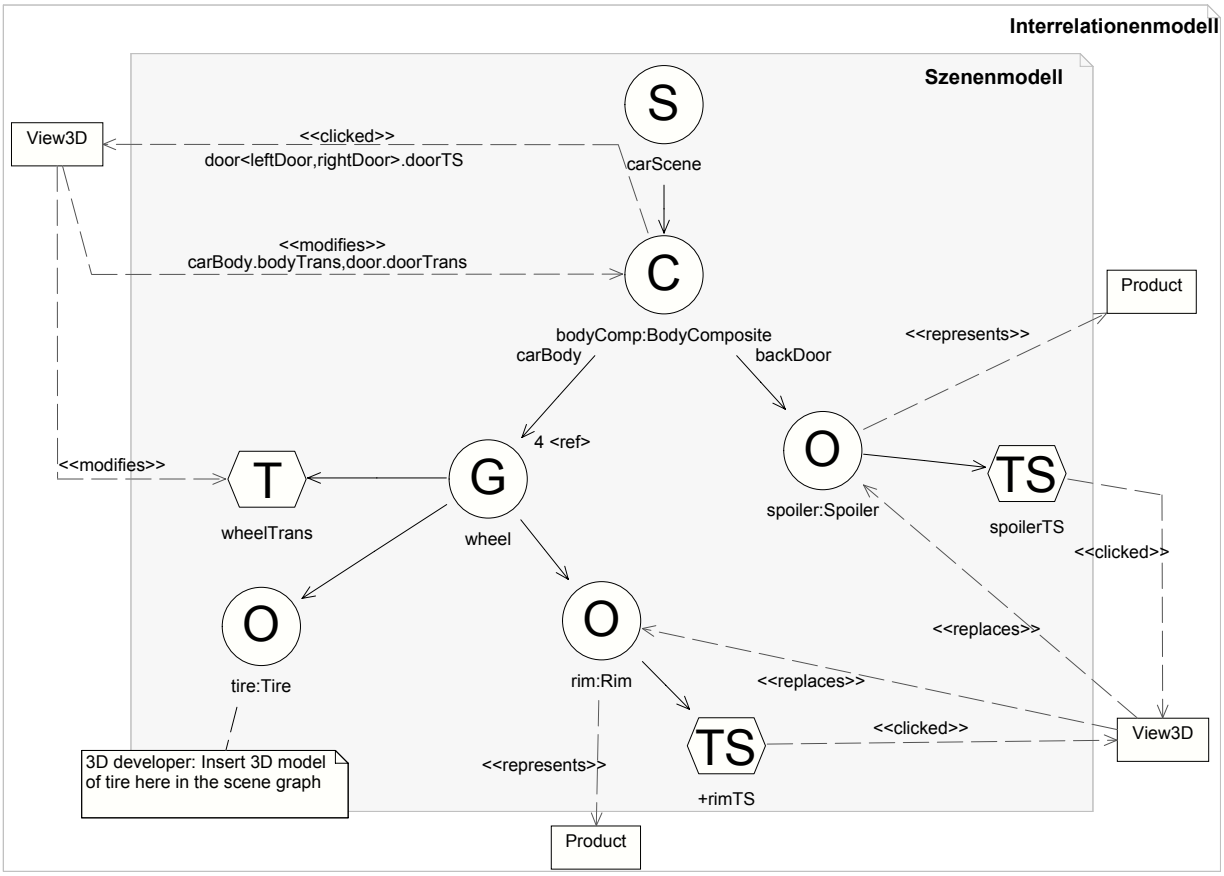



Abbildung 5.39: SSIML-Interelationenmodell des Fahrzeugkonfigurators

Ein Beispiel für eine *Repräsentationsbeziehung* im Interrelationenmodell des Fahrzeugkonfigurators ist die Beziehung zwischen dem Felgenobjekt (`rim`) und der Klasse `Product`. Das Felgenobjekt repräsentiert ein Produkt. Das Produkt enthält Metainformationen über das 3D-Felgenobjekt, z.B. einen Namen, eine Beschreibung, eine Preisangabe, eine Referenz auf eine Bilddatei und eine Referenz auf ein 3D-Modell (in Abbildung 5.39 nicht dargestellt). Falls der Benutzer ein bestimmtes Produkt aus einer Zubehörliste auswählt, z. B. einen bestimmten Felgentyp, kann das entsprechende 3D-Felgenmodell über die Produktinformationen identifiziert und geladen, im Szenengraph ausgetauscht (*replaceContent*-Beziehung zwischen `View3D` und `rim`) und schließlich in der 3D-Ansicht dargestellt werden.

Ein und dieselbe Modellkomponente kann zum Zweck einer besseren Übersichtlichkeit auch mehrmals in einem SSIML-Diagramm dargestellt werden, wenn dadurch das Modell gültig bleibt. Als Beispiele sind die Klassen `View3D` und `Product` im Diagramm in Abbildung 5.40 mehrfach vorhanden.

## 5.6 Modell-Code-Abbildung

An dieser Stelle soll beispielhaft für eine bestimmte Kombination von Sprachen und Programmbibliotheken gezeigt werden, wie Elemente aus dem SSIML-Szenen- und Interrelationenmodell auf Code abgebildet werden können, um Anwendungsgerüste zu erhalten, welche die Modellsemantik möglichst vollständig umsetzen. Dabei wird der Fokus auf den generierten Code, also das Ergebnis der Codegenerierung, und weniger auf den konkreten Weg vom Modell zum Code gelegt. Als Demonstrationsobjekt dient der Fahrzeugkonfigurator aus dem vorherigen Abschnitt.Die hier getroffenen Aussagen beziehen sich auf die Abbildung von SSIML-Modellen, die ein Szenenmodell beinhalten, das eine vollständige Szene repräsentiert. Die Generierung von Code aus Modellen mit Teilszenen (vgl. Abschnitt 5.2.2.3) kann allerdings in ähnlicher Art und Weise erfolgen.

### 5.6.1 Abbildung von Szenenmodellen

Wie bereits in Kapitel 4 erwähnt, wurde als Zielformat für die Beschreibung der 3D-Szene das Standardformat X3D gewählt. Codebeispiele werden im X3D-Format unter Verwendung der klassischen VRML-Syntax angegeben.

#### 5.6.1.1 Aufteilung der Szene auf Dateiebene

Zunächst besteht die Frage, in welche Datenressourcen eine Szene bei der Generierung untergliedert wird. Der prototypisch im Rahmen dieser Arbeit implementierte, auf XSLT basierende Generator erzeugt zunächst eine – nach dem Wurzelknoten des Szenenmodells benannte – X3D-Hauptdatei, in der die Gesamtstruktur der Szene beschrieben ist. Kompositionsknoten werden aufgelöst und der aus ihnen generierte Code wird mit in die X3D-Gesamtszene eingegliedert. Somit enthält die Hauptdatei nach Abschluss des Generierungsprozesses nahezu den gesamten, aus Modelldaten erzeugten Code. Die benötigten 3D-Modelle (Geometriebeschreibungen) werden in separaten Dateien abgelegt. Sie werden aus der Hauptdatei mittels so genannter X3D-*Inline*-Knoten über URLs referenziert. Eine 3D-Modelldatei entspricht einer Inhaltstypdefinition eines SSIML-Objektknotens (`ObjectTypeDefinition`). Wurde in der Inhaltstypdefinition keine URL spezifiziert, kann automatisch eine leere oder mit einem Dummy-Objekt (z.B. einem Würfel) versehene 3D-Modelldatei mit dem Namen der `ObjectTypeDefinition` generiert werden. Wurde im Modell auch keine Inhaltstypdefinition für einen Knoten spezifiziert, erhält der generierte X3D-Inline-Knoten ein leeres URL-Feld und es wird keine Datei für das 3D-Modell generiert. Die Inhalte der ggf. generierten 3D-Modelldateien werden erst von 3D-Entwicklern durch detaillierte Geometriebeschreibungen ersetzt. Alle generierten Ressourcen werden in einem Ordner zusammengefasst. Für das Modell des Fahrzeugkonfigurators würden z. B. folgende Ressourcen generiert: `carScene` (Hauptdatei), `Tire`, `Rim`, `Spoiler`, `CarBody`, `BackDoor`, `FrontHood`, `Door`.

#### 5.6.1.2 Benennung generierter Elemente

Natürlich muss gewährleistet sein, dass sich jeder generierte X3D-Knoten durch einen eindeutigen, aber auch aussagekräftigen Namen ansprechen lässt. Namen von X3D-Knoten, die einer SSIML-Knoteninstanz entsprechen, werden nach dem Prinzip *<Namensraum des SSIML-Knotens>'/'<SSIML-Instanzkennung>* gebildet, wobei die *Instanzkennung* entweder der *Instanzname* ist (falls ein solcher vergeben wurde) oder sich aus dem *Knotennamen* und – wenn der Knoten mehrere Instanzen besitzt – dem *Instanzcode* zusammensetzt. Komponenten des Instanzcodes werden durch Unterstriche getrennt.

Namen von X3D-Knoten, die SSIML-Attributinstanzen entsprechen (Attributabbilder), werden entsprechend der Syntax *<X3D-Knotenname der Instanz des Attributbesitzers>'_'<Attributname>* erzeugt. Gehört eine Attributinstanz zu mehreren SSIML-Knoteninstanzen, (z.B. im Falle eines *Shared Sensors*), wird der Name des Attributabbildes nur für die erste Instanz generiert; für die weiteren Knoteninstanzen wird nur eine Referenz auf das bereits generierte Attributabbild erzeugt. Die Zuweisung eines Namens zu einem Knoten in X3D erfolgt mittels des Ausdrucks *'DEF' <X3D-Knotenname>*, eine Referenz auf einen so markierten Knoten wird durch *'USE' <X3D-Knotenname>* beschrieben. Eine Besonderheit kann bei Transformationsattributen von Knoten auftreten: Besitzt eine Knoteninstanz mehrere parallele Transformationen, wird jedem der X3D-Transformationsknotennamen eine laufende Nummer angehängt.

#### 5.6.1.3 Abbildung von SSIML-Knoten auf X3D-Code

Der `Scene`-Knoten des SSIML-Modells wird auf einen X3D-Gruppenknoten abgebildet, der alle weiteren Szenenelemente enthält. Wie oben bereits angedeutet, werden bei der Generierung des X3D-Codes alle Kompositionsknoten aufgelöst. Das bedeutet, dass die entsprechenden Subgraphenstrukturen, die von Kompositionsknoten referenziert werden, mit in die Gesamtszenenstruktur eingeordnet werden. Im Fahrzeugkonfiguratorbeispiel würde bspw. der Kompositionsknoten `bodyComp` (vgl. Abbildung 5.38) aufgelöst und in die Gesamtszene integriert.

Jeder *atomare Knoten* im SSIML-Modell wird auf die im Modell spezifzierte Anzahl an Instanzen abgebildet. Für jede Instanz eines SSIML-Knotens werden im Normalfall mehrere X3D-Knoten, die einen X3D-Teilszenengraphen bilden, erzeugt. Für jede Instanz eines atomaren Knotens wird eine X3D-Transformationsgruppe generiert. Wenn im Modell entsprechend spezifiziert, werden X3D-`Sensor`-Knoten für die Instanz erzeugt. Der Inhalt einer Instanz wird auf einen X3D-Gruppenknoten, nachfolgend als *X3D-Inhaltsgruppe* bezeichnet, abgebildet. Für eine Instanz eines Objektknotens (z. B. `rim`, s. Abbildung 5.38) beinhaltet die X3D-Inhaltsgruppe im `children`-Feld sowohl den X3D-Code für den gekapselten Inhalt der Instanz als auch deren Kindknoten. Der gekapselte Inhalt einer Objektknoteninstanz wird – wie bereits beschrieben – auf einen X3D-`Inline`-Knoten abgebildet, welcher ggf. die durch den Inhaltstyp im SSIML-Modell festgelegte externe 3D-Modell-Ressource referenziert (z.B. *Rim.x3dv*). Für einen SSIML-Gruppenknoten entfällt der gekapselte In-

halt. Der folgende Quelltext-Ausschnitt zeigt den aus dem SSIML-Objektknoten `rim` und dem Attribut `rimTS` generierten X3D-Code.

```
#X3D Code for rim
  Transform {
    children [
      DEF rimTS TouchSensor {}
      DEF rim Group {children [Inline{url "Rim.x3dv"}]}
]}
```

Ein weiteres Codebeispiel ist unten dargestellt. Da die Instanzen `leftDoor` und `rightDoor` beide denselben Inhaltstyp (`Door`) haben, referenzieren ihre Abbilder auch im X3D-Code ein und dieselbe 3D-Modelldatei (*Door.x3dv*).

```
#X3D code for leftDoor
DEF bodyComp/leftDoor_doorTrans Transform ...
  DEF bodyComp/leftDoor Group {children [
    Inline {url "Door.x3dv"}]} ...
#X3D code for rightDoor
DEF bodyComp/rightDoor_doorTrans Transform ...
  DEF bodyComp/rightDoor Group {children [
    Inline {url "Door.x3dv"}]} ...
```

#### 5.6.1.4 Abbildung von SSIML-Attributen

Die Abbildung von SSIML-Knotenattributen einschließlich der Stelle, an welcher der Attributcode in den X3D-Szenengraphen eingefügt wird, hängt vom jeweiligen Attribut selbst ab. Bspw. wird ein Transformationsattribut ganz anders behandelt als etwa ein `Appearance`-Attribut oder ein `Geometry`-Attribut. Letztgenannte Attribute erfordern Zugriff auf den gekapselten Inhalt eines Objektknotens, da in diesem die Geometrie eines 3D-Objekts einschließlich seiner Textur- und Materialeigenschaften definiert ist. Ein Zugriff auf derart gekapselte Objekteigenschaften kann in X3D mittels `Import-/Export`-Statements erfolgen. Diese Anweisungen ermöglichen den Zugriff auf X3D-Knoten, die in einer durch ein `Inline`-Statement referenzierten Datei enthalten sind. Da Materialien, Texturen und Geometrien in X3D durch eigene Knoten repräsentiert werden, ist somit ein Zugriff auf diese Elemente gegeben. Transformationsattribute, die in X3D auf Transformationsgruppen abgebildet werden, müssen außerhalb der X3D-Inhaltsgruppe im Code stehen, da es mehrere parallele Transformationen derselben Inhaltsgruppe geben kann. Entsprechendes wird in nachfolgend illustriert; dargestellt ist der für das Fahrzeugkonfiguratorbeispiel generierte Code für zwei Radobjekte (`wheel_1` und `wheel_2`).

```
#X3D code for the first wheel
DEF wheel_1_wheelTrans Transform {
  children [
    DEF wheel Group {
      children [
        #X3D Code for rim
        ...
        #X3D Code for tire
        ...]}]}
DEF wheel_2_wheelTrans Transform {
  children [
    USE wheel
  ]
}
```

Besondere Regeln gelten für die Codeabbildung von `RBDynamics` und `Gravity`-Attributen. Da diese Attribute eine spezielle Bedeutung für physikbasierte Animationen haben, wird auf ihre Abbildung in Kapitel 7, dessen Schwerpunkt das Verhalten und die Animation von 3D-Objekten ist, eingegangen.

#### 5.6.1.5 Alternative Möglichkeiten der Szenengenerierung

Es kann sinnvoll sein, Subgraphendefinitionen auf Implementierungsebene in eigenen Komponenten zu kapseln, um ihre Wiederverwendbarkeit zu gewährleisten. Eine Komponente `Door` könnte dann z.B. in verschiedenen X3D-Szenen eingesetzt werden. X3D bietet dafür das Konzept der so genannten X3D-Prototypen (*X3D-Prototypes*). Ein Prototyp besitzt eine definierte Schnittstelle und kapselt einen Rumpf, der die Implementierung des Prototyps als X3D-codierten Subgraphen enthält. Elemente im Rumpf können mit der Schnittstelle des Prototyps verbunden und über diese dann adressiert werden. Instanzen des Prototyps (`ProtoInstance`s), die Instanzen von SSIML-Kompositionsknoten entsprechen, lassen sich wie vordefinierte X3D-Knoten in einen X3D-Szenengraphen einfügen.

Eine Einschränkung bei Prototypen besteht darin, dass über die Schnittstelle keine X3D-Knoten, sondern nur Felder von X3D-Knoten im Prototyprumpf angesprochen werden können. Somit lassen sich etwa bei einem X3D-Gruppenknoten entweder alle Kindknoten (`children`-Feld des Knotens) oder gar keine Kindknoten adressieren. Dies stellt dann ein Problem dar, wenn (analog zum Hinzufügen von Kindknoten zu Kompositionsknoten) ein X3D-Knoten über die Prototypschnittstelle als zusätzlicher Kindknoten mit einem X3D-Knoten im Rumpf einer Prototyp-Instanz verbunden werden soll. Dies ist zwar prinzipiell möglich, erfordert aber zusätzliche Maßnahmen, z. B. das Einbinden von Skriptcode über X3D-`Script`-Nodes. Um derartige Work-Arounds zu vermeiden, müssen für hinzuzufügende X3D-Kindknoten, die sich außerhalb des Prototyprumpfes befinden, im Rumpf zusätzliche Gruppenknoten eingefügt werden, deren `children`-Felder sich dann separat ansprechen lassen.

Weiterhin müssen nahezu alle Felder von X3D-Knoten im Prototyprumpf über die Prototypschnittstelle adressierbar sein, da sich die Schnittstelle mit den Basiselementen von SSIML nicht spezifizieren lässt, aber trotzdem eine flexible Anwendung eines Prototyps in unterschiedlichem Kontext (unterschiedliche Szenen) gewährleistet sein soll. Eine Erweiterung von SSIML, die das Konzept von 3D-Komponenten aufgreift und die Möglichkeiten der Kompositionsknoten ausbaut, um auch höheren Anforderungen an Wiederverwendbarkeit zu genügen, wird in Kapitel 8 vorgestellt.

#### 5.6.1.6 Validierungswerkzeug

Um sicherzustellen, dass eine mit einem 3D-Autorenwerkzeug vervollständigte VRML/X3D-Szene noch konform zum entsprechenden SSIML-Szenenmodell ist, wurde ein einfaches Validierungswerkzeug entwickelt. Damit ist es möglich, eine unmittelbar aus einem SSIML-Modell generierte Szene mit einer angepassten Szene bzw. einer vollständig manuell erstellten Szene hinsichtlich der Struktur und der Benennung der Szenenobjekte

zu vergleichen. Das Werkzeug gibt entsprechende Meldungen aus, wenn Unterschiede zwischen den zwei zu vergleichenden Szenen festgestellt werden.

Zur Demonstration der Anwendung des Validierungswerkzeuges soll folgendes Szenario betrachtet werden: Bei der Generierung des Codes für das SSIML-Szenenmodell des Fahrzeugkonfigurators wird der `TouchSensor` `rimTS` erzeugt (siehe oben). Der 3D-Designer lädt nun das erzeugte Gerüst für die 3D-Szene zur Komplettierung in ein Autorenwerkzeug. Bei der Vervollständigung benennt der 3D-Designer den Sensorknoten `rimTS` in `rimTouchSensor` um. Die im Autorenwerkzeug komplettierte Szene wird nun in einer neuen Datei abgelegt. Der Programmier lädt diese Datei über das 3D-API. Aufgrund der Umbenennung des `TouchSensor`s ist es allerdings nicht mehr möglich, den Sensor korrekt zu adressieren, da der aus dem Interrelationenmodell generierte Programmcode (s. dazu auch nächster Abschnitt) nach wie vor den ursprünglich im Modell definierten Knotennamen, nämlich `rimTS`, enthält. An dieser Stelle kann das Validierungswerkzeug eingesetzt werden, um die ursprünglich generierte Szene mit der nachbearbeiteten 3D-Szene zu vergleichen. Das Validierungswerkzeug deckt auf, dass sich die Namen der jeweils an der gleichen Szenengraphposition vorkommenden `TouchSensor`-Knoten in der Original- und der modifizierten Szene unterscheiden. Während der Sensor im Original den Namen `rimTS` trägt, ist er in der angepassten Szene mit der Bezeichnung `rimTouchSensor` versehen. Durch die Aufdeckung des Fehlers mit Hilfe des Validierungswerkzeuges kann dieser schnell behoben werden. Ein ähnliches Szenario, in dem das Validierungswerkzeug gewinnbringend einsetzbar wäre, bestünde bspw. darin, dass der Designer einen Knoten des Szenengraphs versehentlich löscht oder an eine andere Position innerhalb der Knotenhierarchie verschiebt.

### 5.6.2 Abbildung von Interrelationenmodellen

Für die Abbildung von SSIML-Interrelationen auf Code dienen beispielhaft Java als Zielsprache und das Scene Access Interface [Int08a] (SAI) als generische Schnittstelle für den Zugriff auf X3D-Szenen.

#### 5.6.2.1 Abbildung von Aktions- und Repräsentationsbeziehungen

Für jede `ActionRelationship`-Beziehung, die eine zugreifende Klasse zu einem Element des Szenenmodells hat, wird eine entsprechende Zugriffsmethode generiert. Dabei werden auch Repräsentationsbeziehungen berücksichtigt, die ein SSIML-Knoten, der Ziel einer Aktionsbeziehung ist, zu Fachklassen besitzt. Instanzen repräsentierter Klassen (hier der Klasse `Product`) werden als Parameter an die Methode übergeben, da sie für die auszuführende Aktion wichtige Informationen kapseln, wie die URL auf das 3D-Modell, welches die Geometrien des adressierten Szenengraphknotens ersetzen soll. Der Codeausschnitt unten zeigt die generierte Methode für die Beziehung zum Austausch der Felgengeometrie im Fahrzeugkonfigurator.

```
public void replaceRim(Product rim) {
  String url = null;
  //To do: get String url from Product rim
  ...
```

```
  X3DNode[] newValue = {};
  if ((url != null)&&!url.equals("")) {
    String[] urls = {url};
    newValue = browser.createX3DFromURL(urls).getRootNodes();
  }
  X3DNode node = scene.getNamedNode("rim");
  MFNode contentField = (MFNode)node.getField("children");
  contentField.setValue(newValue.length, newValue);
}
```

#### 5.6.2.2 Abbildung von Ereignisbeziehungen

Während im vorherigen Abschnitt der Programmcode für den Zugriff auf Szenenelemente vorgestellt wurde, liegt der Fokus in diesem Abschnitt auf der Behandlung von Ereignissen, die innerhalb der Szene auftreten. Für jede – nachfolgend als *Besitzerklasse* bezeichnete – Klasse, die über eine Ereignisbeziehung (`EventRelationship`) mit einem SSIML-Sensor verbunden ist, wird eine Implementierung von `X3DFieldEventListener` (Schnittstelle im SAI) generiert. Der Name des generierten `Listener`s setzt sich aus dem Namen seiner Besitzerklasse, dem Namen der entsprechenden Szene, einem Hinweis auf das 3D-Format und dem Wort *Listener* zusammen (z.B. `View3DCarSceneX3DListener`). Nach der Instanziierung bekommt ein spezialisierter `X3DFieldEventListener` über die Methode `setOwner` eine Referenz auf seine Besitzerklasse zugewiesen. Ein `X3DFieldEventListener` verfügt weiterhin über eine Methode `readableFieldChanged`, die ein `X3DFieldEvent` als Argument akzeptiert. Innerhalb der `readableFieldChanged`-Methode können als Reaktion auf Ereignisse Operationen der Besitzerklasse aufgerufen werden. Eine passende Implementierung der `readableFieldChanged`-Methode eines generierten `X3DFieldEventListeners` kann teilweise automatisch erzeugt werden. Lücken im Programmcode müssen durch den Programmierer gefüllt werden. Der folgende Quellcode zeigt einen Teil der generierten Implementierung der Methode `readableFieldChanged` aus dem Fahrzeugkonfiguratorbeispiel.

```
public void readableFieldChanged(X3DFieldEvent ev) {
  if (ev.getData().equals("rimTS_clicked")) {
    //To do: Implement event handling code here
    <Display description in view>
  } else if ...
}
```

#### 5.6.2.3 Integration des generierten Codes in die Gesamtanwendung

Eine Kernidee des SSIML-Ansatzes ist es, aus ein und demselben Modell sowohl Codeskelette für die 3D-Szene als auch für den Programmcode in der Art zu generieren, dass die generierten Bestandteile untereinander konsistent sind. So lässt sich z. B. sicherstellen, dass die Benennungen von 3D-Objekten, auf welche die Rahmenanwendung zugreift, im Programmcode und in der 3D-Szene übereinstimmen. Welche 3D-Objekte mit Teilen der Rahmenanwendung verknüpft sind, geht aus den Interrelationen des SSIML-Modells hervor. Um die in den vorangehenden Abschnitten vorgestellten Codekomponenten sinnvoll in eine Gesamtanwendung integrieren zu können, müssen zunächst die Anforderungen an eine solche Integration formuliert werden:

- Eine Szene kann für unterschiedliche Klassen der Rahmenanwendung unterschiedliche maßgeschneiderte Schnittstellen bereitstellen, da der Code für den Zugriff auf die Szene sich auf verschiedene Anwendungskomponenten verteilen kann. Jeder Klasse sollen genau die Zugriffsmethoden bereitgestellt werden, die sie benötigt. Z. B. kann es in einer virtuellen Welt Klassen mit verschiedenen Verantwortlichkeiten geben: eine Klasse zur Navigation durch die Szene, eine Klasse zur Einstellung von Kameraparametern und weitere Klassen zur Kapselung von Animationen und objektspezifischem Verhalten.

- Eine Szene soll möglichst flexibel austauschbar sein, ohne die Implementierung der zugreifenden Klassen komplett ändern zu müssen (z.B. Nachladen einer neuen Level bei einem 3D-Spiel, Austausch des zu konfigurierenden Fahrzeugmodells). Für Szenen, die Instanzen desselben SSIML-Modells sind, ist dies kein Problem, da die Schnittstellen zur Szene gleich bleiben. Komplizierter wird es, wenn zwar die gleichen Klassen auf die Szene zugreifen, sich aber die Szenenschnittstelle verändert, z. B. wenn bei einem 3D-Fahrzeugkonfigurator ein Fahrzeugmodell durch ein Modell mit anderen Konfigurationsmöglichkeiten ausgetauscht wird.

Um den genannten Anforderungen gerecht zu werden, wurde ein Satz von vorgefertigten Klassen und Schnittstellen in Form eines einfachen Frameworks (genannt *Integrator3D*) entworfen, das die benötigten Grundfunktionalitäten bereitstellt. Das UML-Diagramm in Abbildung 5.40 gibt einen Überblick über die Basiselemente des Frameworks, generierte Klassen und ihre Beziehungen untereinander. Basierend auf dem Framework lässt sich automatisch eine Schnittstelle erzeugen, die dem Programmierer einerseits einen maßgeschneiderten Zugriff auf die 3D-Szene erlaubt, andererseits aber die geforderte Flexibilität bietet. Ein Großteil der Schnittstellenimplementierung wird ebenfalls generiert. Das Framework ist i. W. in zwei Schichten organisiert: einer szenenformatabhängigen und einer szenenformatunabhängigen Schicht. Eine Anwendungskomponente kann mit der Szene über die formatunabhängige Schnittstelle kommunizieren und wird somit von Besonderheiten eines zu Grunde liegenden Szenengraphformates nicht tangiert.

Zum Zweck des einfachen Austauschs kompletter 3D-Szenen wird die Schnittstelle zum Zugriff auf die Szene nicht direkt in die zugreifende Klasse generiert. Anstatt dessen wird für jede Klasse, die über eine im Interrelationenmodell definierte Beziehung Zugriff auf das Szenenmodell hat, eine separate `Connector`-Klasse (*Connector*: Verbindungsstück) erzeugt, welche speziell die Szenenzugriffsmethoden für die jeweilige Klasse bereitstellt. Ein `Connector`, der eine Klasse `AClass` mit einer X3D-Szene `AScene` verbindet, trägt den Namen `AClassASceneX3DConnector`. Bspw. trägt ein `Connector`, der eine Klasse `View3D` mit einer Szene `CarScene` verbindet, den Namen `View3DCarSceneX3DConnector`.

Über die `init`-Methode kann dem `Connector` eine Referenz auf die 3D-Szene übergeben werden, auf welche er den Zugriff ermöglichen soll. Die Szene muss eine Instanz des Szenenmodells sein, für welches der `Connector` generiert wurde. Wird eine Szene geladen, die Instanz eines anderen Szenenmodells ist (also eine andere Struktur besitzt), wird auch der `Connector` ausgetauscht.

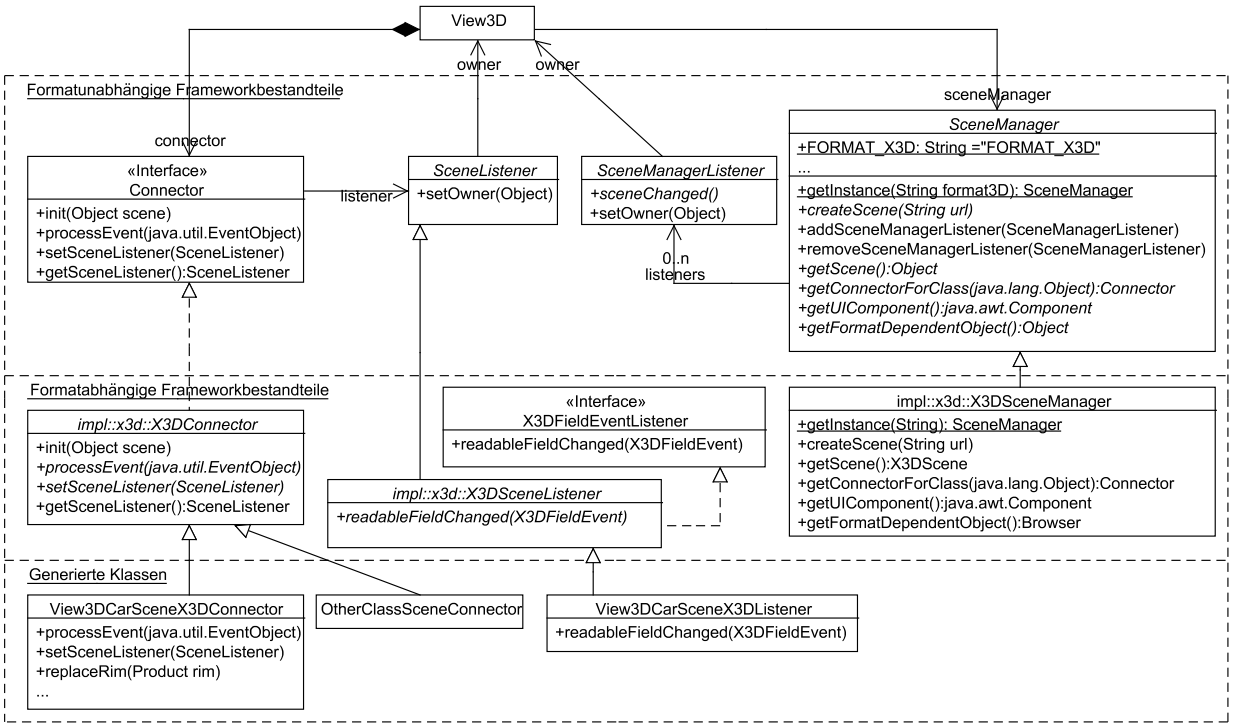



Abbildung 5.40: Zusammenhänge zwischen generierten Klassen und Klassen des Integrator3D-Frameworks

Weiterhin werden die für eine Klasse generierten Methoden für Aktions- und Repräsentationsbeziehungen (vgl. Abschnitt 5.3.1) in den `Connector` der Klasse aufgenommen. Nach dem Laden der Szene werden nahezu alle Änderungen in der Szene durch Ereignisse hervorgerufen (ein Beispiel für ein Ereignis wäre die Selektion eines bestimmten Felgentyps im Fahrzeugkonfigurator durch den Benutzer). Um einer Klasse über einen `Connector` flexiblen Zugriff auf eine austauschbare Szene zu erlauben, wird das *Zustand*-Entwurfsmuster (*State Pattern*) [GHJV94] angewendet. Das State Pattern nutzt die Eigenschaft des Polymorphismus objektorientierter Sprachen aus (gleiche Erscheinung mehrerer Objekte nach außen, aber unterschiedliches Verhalten). Dafür erben alle konkreten `Connector`-Klassen von einer abstrakten `Connector`-Klasse, der `Connector`-Basisklasse. Die Basisklasse besitzt eine Methode `processEvent`, der ein `java.util.EventObject` (`EventObject` ist die Basisklasse aller Java-Ereignisse) übergeben werden kann. Die zugreifende Klasse hat eine Referenz auf eine (austauschbare) Instanz einer konkreten `Connector`-Klasse, die sie über die Schnittstelle der abstrakten `Connector`-Basisklasse anspricht (Abbildung 5.40). Die zugreifende Klasse kann über die `processEvent`-Methode Ereignisnachrichten als spezialisierte `EventObject`s an ihr `Connector`-Objekt senden. Die `processEvent`-Methode muss in einer konkreten (abgeleiteten) `Connector`-Klasse vom Programmierer implementiert werden. Der Implementierungsaufwand für die Methode ist relativ gering, da abhängig vom eingetretenen Ereignis lediglich eine bereits generierte Methode des `Connector`s aufgerufen werden muss. Durch eine szenenspezifische Implementierung von `processEvent` kann

auf gleiche Ereignisnachrichten mit einer der aktuell geladenen Szene angepassten Aktion reagiert werden. Die übergebenen `EventObject`s müssen alle Informationen enthalten, die notwendig sind, um die anvisierte Aktion auszuführen.

An dieser Stelle soll erwähnt werden, dass neben dem Aufruf über die `processEvent`-Methode die in den `Connector` generierten Szenen-Zugriffsmethoden auch direkt aufgerufen werden können, da alle Methoden öffentlich zugänglich sind. Dies kann dann sinnvoll sein, wenn nur auf einen bestimmte Szenentyp (Szenen, die Instanzen desselben Szenenmodells sind) zugegriffen wird.

Damit eine Klasse auf von Sensoren in der 3D-Szene ausgelöste Ereignisse reagieren kann, muss der ihr zugeordnete `X3DFieldEventListener` instanziiert und bei ihrem aktuellen `Connector` mittels der Methode `setSceneListener` registriert werden. Bei der Generierung eines konkreten `Connector`s wird automatisch eine Implementierung seiner `setSceneListener`-Methode erzeugt, die aus dem SSIML-Interrelationenmodell abgeleitet wird. Innerhalb der `setSceneListener`-Methode wird der übergebene `Listener` bei den abzuhörenden Sensorenfeldern registriert. Folgender Quellcode zeigt die Definition der Methode `setSceneListener` des `View3DCarSceneX3DConnector`s aus dem Fahrzeugkonfiguratorbeispiel.

```
public void setSceneListener(X3DFieldEventListener l) {
  X3DField f1 =
    scene.getNamedNode("rimTS").getField("touchTime");
  //specifies the clicked-event
  f1.setUserData("rimTS_clicked");
  f1.addX3DEventListener(l);
  X3DField f2 = ...
}
```

Das Framework stellt zusätzlich die Klasse `SceneManager` zur Verfügung. Ein `SceneManager` beinhaltet einige Hilfsfunktionen, über die eine Klasse auf komfortable Art und Weise Zugriff auf den ihr zugeordneten `Connector` erhält. Die Funktionen des `SceneManager`s sind im Einzelnen:

- `getInstance` ist eine statische Methode, die eine formatspezifische Implementierung des `SceneManager`s zurückgibt. Das zurückgelieferte Objekt ist ein *Singleton*. Nachfolgende Aufrufe von `getInstance` ergeben als Resultat immer dieselbe `SceneManager`-Instanz, wobei der Formatparameter keine Wirkung mehr besitzt.

- `createScene`: Diese Methode lädt eine 3D-Szene. Als Parameter muss die URL auf die zu ladende 3D-Datei übergegeben werden.

- Die Methoden `addSceneManagerListener` und `removeSceneManagerListener` erlauben es, `SceneManagerListener`-Objekte zu registrieren, die bei Änderungen der Szene (normalerweise dem Austausch der Szene) informiert werden.

- Die Methode `getUIComponent` liefert eine GUI-Komponente zurück, auf der die 3D-Szene gerendert wird.

- `getConnectorForClass`: Diese Methode lädt, instanziiert und initialisiert den `Connector`, der das als Parameter übergebene Objekt mit der aktuellen 3D-Szene verbindet. Der Klassenname des zu erzeugenden `Connector`s kann aus dem Klassennamen des übergebenen Objektes und dem Namen des Szenenwurzelknotens abgeleitet werden. Ebenso wird der `SceneListener` für die aktuelle Klassen-Szenen-Kombination geladen und erzeugt. Das Objekt wird als Besitzer (*Owner*) des `SceneListener`s gesetzt und der `SceneListener` wird beim `Connector` registriert. Die erzeugte `Connector`-Instanz wird schließlich als Resultat der Methode zurückgegeben.

- Die Methoden `getScene` und `getFormatDependentObjekt` werden im Regelfall von der auf die Szene zugreifenden Klasse nicht benötigt. In Ausnahmefällen stellen diese beiden Methoden aber zusätzlich geforderte Funktionen zur Verfügung. `getScene` liefert als Resultat eine Referenz auf die 3D-Szene, `getFormatDependentObject` stellt weitere formatspezifische Daten bereit, bspw. bei der X3D-Implementierung des `SceneManager`s eine Referenz auf das X3D-Browser-Objekt.

## 5.7 Verwandte Arbeiten

Die meisten bekannten Ansätze, die sich mit dem Softwareentwurf für 3D-Anwendungen beschäftigen, konzentrieren sich auf das Design von Kernaspekten (bspw. Interaktionstechniken) reiner Virtual Reality (VR)- oder Augmented Reality (AR)-Anwendungen. Konzepte und Werkzeuge zur Integration von 3D-Inhalten und Anwendungslogik stehen dabei kaum im Mittelpunkt. Auch die Unterstützung verschiedener Entwicklergruppen wird eher selten berücksichtigt.

Die Interaction Techniques Markup Language (InTML) [FGH02] und das MariGold Toolset [WH01] etwa erlauben die Spezifikation von Verhaltenskomponenten, wie Interaktionstechniken, auf einer abstrakten Ebene. Weiterhin ist es mit diesen Ansätzen möglich, vollständige Anwendungen zu entwerfen, indem Gerätekomponenten, Interaktionstechniken und virtuelle Objekte miteinander verbunden werden. InTML ist eine XML-basierte Sprache. Sie kann aber als Grundlage für grafische Editoren benutzt werden. Das MariGold-Toolset ermöglicht eine Software-Spezifikation auf der Basis visueller Graphen und unterstützt die Generierung von Code für die Maverik-VR-Plattform [HCK+99].

Smith et. al [SW06] schlagen eine aufgabengetriebene Vorgehensweise zur Dekomposition von 3D-Objekten vor, um einen geeigneten Detailgrad der Objekte zu erreichen. Aus den zum Lösen einer Benutzeraufgabe erforderlichen Interaktionen wird schrittweise der Detailgrad abgeleitet. Zur Modellierung der 3D-Objekte wird eine an Szenengraphen angelehnte Baumnotation verwendet. Die vorgeschlagene Methodik könnte ergänzend zum SSIML-Ansatz angewendet werden, um die zu modellierenden 3D-Objekte zu identifizieren.

Eines der wenigen Projekte, die UML zur Unterstützung der Entwicklung von 3D-Anwendungen einbeziehen, ist die Augmented Presentation and Interaction Language (APRIL) [LS05], eine XML-basierte Sprache zur Beschreibung interaktiver AR-

Präsentationen. APRIL verwendet UML-Zustandsautomaten zur Spezifikation von AR-Präsentationsabläufen (so genannten *Stories*). Eine Ablaufbeschreibung kann im XMI [Obj07b]-Format gespeichert werden, anschließend in das APRIL-Format übersetzt und in eine mit APRIL spezifizierte Gesamtpräsentation integriert werden. Die fertige Präsentation kann auf einem Player, der auf der Studierstube-AR-Plattform [SFH+02] basiert, ausgeführt werden. Obwohl APRIL auf andere Aspekte als Szenenstruktur und Verknüpfung von Anwendung und 3D-Inhalten fokussiert, so gibt es doch Gemeinsamkeiten zwischen APRIL und SSIML, insbesondere die Benutzung von graphischen Modellierungswerkzeugen und die Generierung von Code aus visuellen Modellen.

Im Gegensatz zu den oben beschriebenen entwurfsorientierten Ansätzen ist das Ziel komponentenbasierter Ansätze, die Entwicklung von 3D-Anwendungen durch das Zusammensetzen komplexer Szenen aus vordefinierten Bausteinen zu erleichtern, welche bereits Geometrien, Audioinformationen und Verhalten kapseln können. Zur Erstellung interaktiver 3D-Szenen werden dabei meist proprietäre Werkzeuge verwendet. Der von Dörner und Grimm [DG00] vorgestellte Ansatz erlaubt z. B. das Erstellen von 3D-Szenen mittels Komponenten, die auf dem Java-Beans-Komponentenmodell und Java3D basieren. Im Gegensatz zu SSIML sind diese komponentenbasierten Ansätze aufgrund des niedrigeren Abstraktionsniveaus i. d. R. nicht für einen Entwurf vor der Implementierung vorgesehen.

Jedoch liefern einige Forschungsprojekte im Bereich komponentenbasierter 3D-Entwicklung interessante Ideen für den Entwurf von 3D-Software. Bspw. sieht CONTIGRA [DHM02], eine XML-basierte Sprache zur plattformunabhängigen Beschreibung von 3D-Komponenten und -Anwendungen, die Generierung von Code für verschiedene Formate und Sprachen vor, so dass eine mit CONTIGRA beschriebene Anwendung auf verschiedene 3D-Plattformen abgebildet werden kann. Weiterhin unterstützt CONTIGRA verschiedene Autorengruppen (3D-Designer, Programmierer und Sound-Ingenieure), indem die Sichtbarkeit eines Parameters einer 3D-Komponente individuell für eine bestimmte Autorengruppe festgelegt werden kann. Damit können im eigens entwickelten CONTIGRA-Autorenwerkzeug Eigenschaften einer Komponente abhängig von einer Autorengruppe ein- und ausgeblendet werden. Eine plattformunabhängige Szenenbeschreibung, automatische Codegenierung für verschiedene Zielplattformen und die Unterstützung eines interdisziplinären Entwicklungsprozesses sind ebenfalls Ziele des SSIML-Ansatzes.

Ein Forschungsprojekt, welches einen iterativen Entwurfsprozess mit einer 3D-Komponentenbibliothek und dazu passenden Werkzeugen kombiniert, ist das i4D Framework [GPRR00]. Die Autoren des Frameworks wurden, ähnlich wie der Autor der vorliegenden Arbeit, von der Problemstellung motiviert, 3D-Inhalte auf einfache Art und Weise in (Multimedia-)Anwendungen integrieren zu können. i4D zielt besonders auf die Unterstützung von Autorengruppen wie Designern ab, die nicht über tiefgründigere Programmiererfahrung verfügen. Daher werden Techniken im i4D-Entwicklungsprozess verwendet, die aus dem Bereich des Multimedia-designs stammen, z.B. die Erstellung eines Storyboards. Im Gegensatz zu i4D fußt SSIML auf traditionellen Entwicklungsansätzen aus dem Software-Engineering, um die Integration von 3D-Inhalten in ein breites Spektrum von Anwendungen zu ermöglichen.

Ein weiteres Software-System, dem eine ähnliche Problemstellung wie SSIML zugrunde

liegt, ist das Thekla-Software-System [PSB07]. Das Thekla-System erlaubt es, die Vorteile von 2D- und 3D-Benutzungsschnittstellen durch die Schaffung von hybriden Benutzungsschnittstellen zu vereinen. Allerdings handelt es sich bei Thekla nicht um einen modellgetriebenen Ansatz. Das System stellt Implementierungswerkzeuge zur einfachen Verknüpfung und Synchronisation von 2D- und 3D-Benutzungsschnittstellen zur Verfügung. Thekla basiert auf dem Studierstube-Framework [SFH+02] und ist somit auch für die Realisierung von AR- und Mixed Reality (MR)-Anwendungen einsetzbar. Mit Thekla lassen sich 3D-Eingabegeräte nicht nur zur Selektion und Manipulation von 3D-Objekten, sondern auch zur Kontrolle von 2D-Benutzungsschnittstellen einsetzen.

## 5.8 Zusammenfassung

In diesem Kapitel wurde die visuelle Modellierungssprache SSIML (Scene Structure and Integration Modelling Language) vorgestellt, welche u. a. für den abstrakten Entwurf von komplex strukturierten 3D-Szenen, die letztendlich ein Modell einer 3D-Benutzungsschnittstelle repräsentieren, konzipiert wurde. Eine besondere Innovation des Ansatzes ist die Möglichkeit, Zusammenhänge zwischen 3D-Inhalten und der Rahmenanwendung, in welche die Inhalte zu integrieren sind, in so genannten Interrelationenmodellen zu spezifizieren. Mittels des SSIML-Ansatzes soll die Zusammenarbeit von Entwicklern, wie 3D-Designern und Programmierern, die mit völlig unterschiedlichen Konzepten und Werkzeugen arbeiten, erleichtert werden, vor allem wenn zwischen der Rahmenanwendung und den eingebetteten 3D-Inhalten zahlreiche Interaktionsbeziehungen bestehen.

Die Sprache SSIML basiert auf einem Metamodell, welches auch auf ein UML-Profile abgebildet wurde, um die Integration von SSIML in bestehende UML-Werkzeuge zu ermöglichen. Beispielhaft wurde SSIML in das verbreitete UML-Werkzeug *MagicDraw* eingebunden. Ein Vorteil der angesprochenen Werkzeugintegration ist, dass sich SSIML-Modelle ohne zusätzlichen Entwicklungsaufwand im XML-Metadata-Interchange (XMI) - Format abspeichern lassen. Das XMI-Format ist eine geeignete Basis zur Übersetzung der Modelle in Code mittels XSLT. Aus den Modellen lässt sich Code für verschiedene Plattformen und Entwicklergruppen generieren, der mittels spezialisierter Werkzeuge weiter bearbeitet werden kann. Bspw. kann aus ein und demselben Modell für den 3D-Entwickler Code im X3D-Format und für den Programmierer Java-Code generiert werden, womit die Konsistenz der einzelnen generierten Codekomponenten gesichert wird. Der erzeugte Programmcode bietet eine Schnittstelle zum Zugriff auf die 3D-Szene, welche entsprechend der Spezifikation im Modell genau auf die Anwendung zugeschnitten ist. Im Gegensatz zu einer generischen Schnittstelle zur Szene, wie sie bspw. das *Scene Access Interface* darstellt, wird damit der Zugriff gezielt auf diejenigen Szenenelemente eingeschränkt, die tatsächlich für die jeweilige Anwendung von Interesse sind. Ein weiterer Vorteil der angepassten Schnittstelle ist, dass Programmierfehler bei der Adressierung von Objekten der Szene nicht erst zur Laufzeit aufgedeckt, sondern bereits in der Programmierumgebung vermieden werden können, wenn diese Mechanismen wie automatische Syntaxprüfung und kontextsensitive Codevervollständigung *(Code Completion)* zur Verfügung stellt.

Durch die SSIML-Modelle als “Vertrag” zwischen den Entwicklern und durch die automatische Codegenerierung reduziert sich die Menge des zu schreibenden Codes, die Zahl der Implementierungsfehler und letztendlich auch die Entwicklungszeit. Die SSIML-Modelle können außerdem sowohl als Kommunikationshilfe dienen, um Designideen zwischen einzelnen Entwicklern zu diskutieren, als auch zu Dokumentationszwecken eingesetzt werden.

# Kapitel 6

# SSIML/Tasks

## 6.1 Einführung

Ein wichtige Gruppe von 3D-Anwendungen im Bereich Montage, Wartung und Reparatur stellen interaktive 3D-Handbücher (*Interactive 3D Manuals*) oder 3D-Instruktoren dar. Interaktive 3D-Handbücher dienen i. d. R. dazu, Service-Mitarbeiter, aber auch Endkunden bei der Durchführung von Montage- und Demontagearbeiten von Produkten durch eine Schritt-für-Schritt-Anleitung zu unterstützen [Pol07, ZHBH03]. Sie können zudem bereits im Vorfeld eines geplanten Arbeitsganges zu Trainingszwecken eingesetzt werden, um Kosten und Zeit zu sparen.

Im Vergleich zu papierbasierten Dokumenten bieten 3D-Handbücher die Möglichkeit des Einsatzes fortgeschrittener Visualisierungs- und Interaktionstechniken. Bspw. können innerhalb eines Manuals komplizierte Vorgänge als Animationen visualisiert werden. Ebenso könnte ein Objekt aus verschiedenen Perspektiven betrachtet werden, etwa um Objektdetails hervorzuheben oder eine räumliche Vorstellung vom gesamten Objekt zu vermitteln.

3D-Handbücher sind Gegenstand aktueller Forschung (z. B. [ZHBH03]) und Technik (wie etwa das Werkzeug *Rapid Manual* von Parallelgraphics [Rap] zeigt) sowohl im Bereich der virtuellen als auch im Bereich der erweiterten Realität .

Ein wichtiger Aspekt von 3D-Handbüchern ist die sequenzielle Abarbeitung von Aufgaben (so genannten *Tasks*) zum Erreichen eines Ziels, welches im vorliegenden Fall die Montage oder Demontage eines Produktes oder Bauteils ist.

Um die Erstellung interaktiver 3D-Handbücher besser zu unterstützen, als dies mit den Grundelementen von SSIML möglich ist, wird in diesem Kapitel die SSIML-Erweiterung SSIML/Tasks eingeführt, wobei die Grundmotivation für SSIML auch für SSIML/Tasks zutrifft. SSIML/Tasks erlaubt die Modellierung einer Abfolge von Tasks, die der Benutzer zum Erreichen eines Ziels abarbeiten muss, in so genannten Taskflow-Modellen. SSIML/Tasks ist die kleinste Komponente der SSIML-Sprachfamilie. Für die im Rahmen dieser Arbeit an die Taskmodellierung gestellten Anforderungen hat sich der Umfang von SSIML-Tasks allerdings als ausreichend erwiesen.

Tabelle 6.1: Beispiel einer Liste von Tasks des Benutzers

| Task-Nummer | | Task-Name | | Benutzer | System |
|---|---|---|---|---|---|
| 1. | | RemovePCCover | | Entferne PC-Abdeckung, um Zugang zum Mainboard zu erhalten | Rendere halbtransparente Überlagerung der PC-Abdeckung |
| 2. | | RemoveMetalCover | | Lokalisiere Graphikkarte, entferne Metallabdeckung um Zugang zum Graphikkarten-Slot zu erhalten | Rendere halbtransparente Überlagerung der Slotabdeckung |
| 3. | 3.1. | InstallGraphics-Card | FitInGraphicsCard | Richte Grafikkarte aus, drücke Grafikkarte in den Slot | Rendere halbtransparente Überlagerung für Karte und Slot |
| | 3.2. | | FastenGraphicsCard | Befestige Graphikkarte mit einer Schraube | Zeige eine virtuelle Schraube an der Stelle, an der die reale Schraube einzusetzen ist |
| 4. | | ConnectPowerSupply | | Wenn die Karte eine Stromversorgung benötigt, verbinde die Karte mit der internen Stromversorgung des PCs | Erweitere den Stromversorgungsanschluss auf der Grafikkarte mit einer halbtransparenten Überlagerung |
| 5. | | ReplacePCCover | | Bringe die PC-Abdeckung wieder an | Rendere eine halbtransparente Überlagerung für die PC-Abdeckung |

Das SSIML/Tasks-Metamodell verwendet Elemente des UML-Metamodells. Damit kann SSIML/Tasks vergleichsweise einfach in bestehende UML-Werkzeuge integriert werden. Ein dem SSIML/Tasks-Metamodell entsprechendes *Profile* wurde exemplarisch in das UML-Werkzeug MagicDraw integriert.

## 6.2 Beispiel

Als einfaches Beispiel, an dem die Möglichkeiten der Modellierung mit SSIML/Tasks demonstriert werden sollen, dient eine Augmented Reality - Anwendung aus dem Bereich Montage und Wartung, die den Benutzer beim Einbau einer Grafikkarte in einen PC unterstützen soll. Auf dieses Beispiel wird in Kapitel 9 zurückgegriffen, um zu zeigen, wie ein Taskflow-Modell mit einem SSIML-Szenenmodell kombiniert werden kann, um eine taskabhängige Visualisierung von 3D-Objekten zu ermöglichen. Außerdem wurde eine Variante des Beispielszenarios zur informellen Evaluierung von SSIML(/AR) verwendet, worauf in Abschnitt 11.3 genauer eingegangen wird.

Tabelle 6.1 beschreibt die Teilaufgaben und die möglichen Reihenfolgen ihrer Abarbeitung durch den Nutzer für das gewählte Beispiel. Dabei wird auch aufgeführt, welche Informationen das System für einen bestimmten Nutzertask darstellen muss.

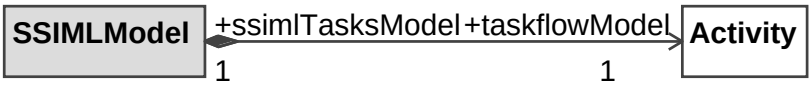



Abbildung 6.1: Ausschnitt aus dem SSIML/Tasks-Metamodell

## 6.3 Metamodell

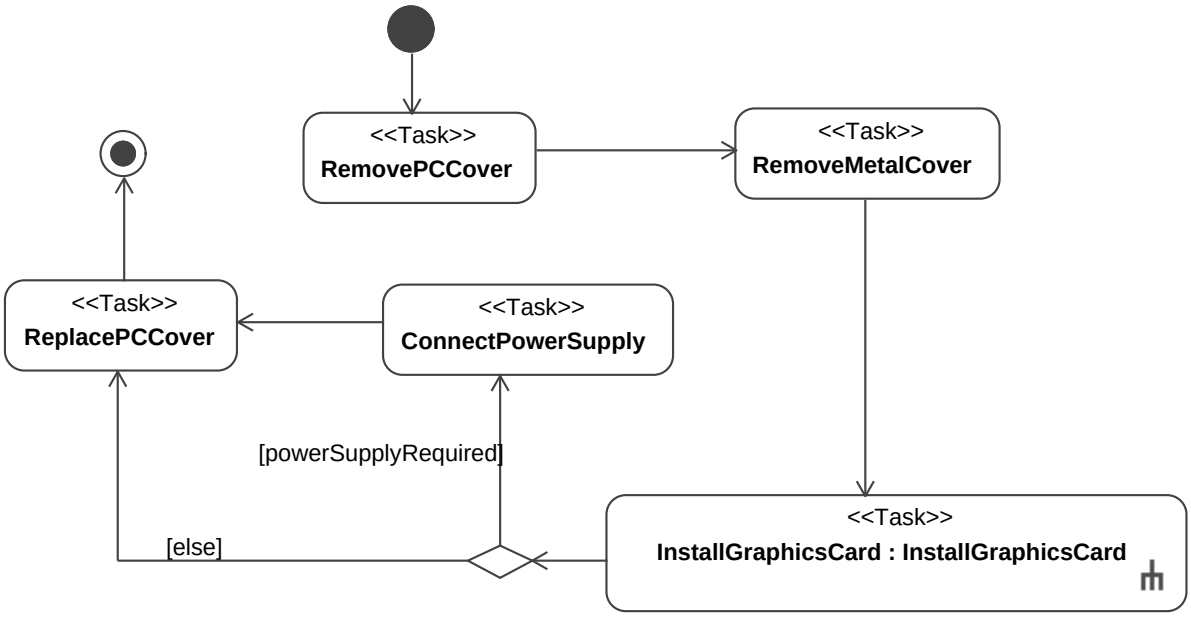
### 6.3.1 Taskflow-Modell

Die wichtigste Komponente eines SSIML/Tasks-Modells ist das Taskflow-Modell, welches den Typ `Activity` (Aktivität) aus dem UML-Metamodell besitzt (Abbildung 6.1). Jeder Task wird durch eine UML-Aktion (genauer: ein Element des Typs `CallBehaviorAction`) modelliert, die im Taskflow-Modell enthalten ist.

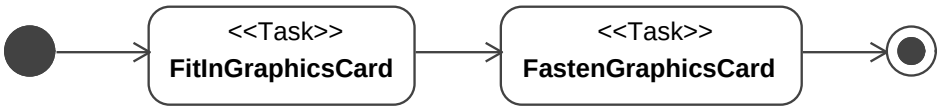
Abbildung 6.2(a) zeigt das Taskflow-Modell für das Grafikkarteninstallationsszenario. Wie zuvor erwähnt, wird jeder Task wird durch eine separate UML-Aktion modelliert. Ein Task muss einen eindeutigen Namen innerhalb des Taskflow-Modells besitzen. Nach Abarbeitung eines Tasks kann mit der Bearbeitung des nachfolgenden Tasks begonnen werden, wobei aufeinanderfolgende Tasks über Transitionen des Typs `ControlFlow` (Kontrollfluss) verbunden sind. Tasks können ggf. Subtasks enthalten, um die Verwaltung komplexerer Taskstrukturen zu erleichtern. Dafür referenziert ein Task eine weitere Aktivität, die einen Subtaskflow mit den entsprechenden Subtasks kapselt. Beispielsweise wurde der Task `InstallGraphicsCard` in die Subtasks `FastenGraphicsCard` und `FitInGraphicsCard` unterteilt (Abbildung 6.2(b)).

Alternative Taskübergänge sind über Entscheidungsknoten ebenfalls spezifizierbar. Benötigt die zu installierende Grafikkarte bspw. eine Verbindung zur internen Stromversorgung des PCs, erfolgt ein Übergang zum Task `ConnectPowerSupply`. Erst nach Abarbeitung dieses Tasks kann mit dem Task `ReplacePCCover` begonnen werden. Ist keine zusätzliche Stromversorgung notwendig, kann vom Task `FitIngraphicsCard` (Untertask von `InstallGraphicsCard`) direkt zum Task `ReplacePCCover` übergegangen werden. Im Taskflow-Modell wird durch die Verwendung des `else`-Guards an der Transition von einem Entscheidungsknoten zu einem Folgeelement sichergestellt, dass es für jeden Task einen Nachfolgetask gibt bzw. die Tasksequenz beendet werden kann. Die `else`-Transition wird dann durchlaufen, wenn keine der anderen, vom Entscheidungsknoten ausgehenden Transitionen durchlaufen werden kann.

Um den Ablauf des Taskflows besser kontrollieren zu können, ist es sinnvoll, nur binäre (Ja-/Nein-)Entscheidungen zu modellieren. Eine Entscheidung mit mehr als zwei Alternativen kann durch mehrere aufeinanderfolgende binäre Entscheidungen abgebildet werden. Andernfalls besteht die Möglichkeit, dass mehrere Guards, die an alternativen Transitionen angetragen wurden, gleichzeitig den Wert *true* annehmen. In einem solchen Fall könnte vor dem Durchlaufen einer Transition nicht mehr eindeutig bestimmt werden, welcher Task der Nachfolger des gerade abgeschlossenen Tasks ist.




(a) Haupt-Taskflow-Modell

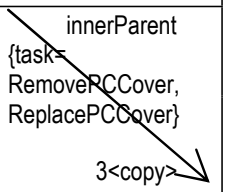
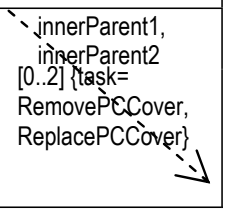



(b) Subtaskflow-Modell des Tasks `InstallGraphicsCard`

Abbildung 6.2: Beispiel eines Taskflow-Modells

Das Taskflow-Modell beschreibt nicht, wie ein Softwaresystem die vollständige Bearbeitung eines Tasks erkennt. Dies muss nach einer späteren Übersetzung des Taskflow-Modells in ein plattformspezifisches Format im Code festgelegt werden. Im einfachsten Fall könnte der erfolgreiche Abschluss eines Tasks explizit durch den Benutzer per Tastendruck bestätigt werden. Eine ähnliche Methode wurde von Boeing in einem Verkabelungsinstruktionssystem umgesetzt [CMGJ99]. Aber auch andere Mechanismen zur Steuerung des Taskflows sind vorstellbar, wie ein Taskübergang durch Spracheingabe, Gestenerkennung oder das automatische Tracking des Fortschrittes des Benutzers bei der Bearbeitung eines Tasks. Der gewählte Mechanismus hängt nicht zuletzt stark von der Einsatzsituation der betrachteten Anwendung ab. Benötigt der Benutzer zur Bearbeitung der Tasks bspw. beide Hände, kann eine Sprachsteuerung sinnvoller sein als eine Bestätigung der Fertigstellung eines Tasks durch ein tastenbasiertes Eingabegerät. Kommt es hingegen darauf an, dass ein Kommando korrekt erkannt wird und Fehleingaben weitestgehend ausgeschlossen werden, kann eine tastenbasierte Eingabe die bessere Wahl darstellen.

Tabelle 6.2: Notation von `TaskConstrainedEdge`-Elementen

| Beziehungstyp | Elternknoten (Startknoten) | Ende 1 (Start) | Mittelteil | Ende 2 (Ziel) | Beispiel |
|---|---|---|---|---|---|
| Primary | LocatedNode |  |  '{task=' <TaskList>'}' |  (<Elementzahl> ('<ref>'\| '<copy>'))? |   |
| | ComposedNode |  <innerer Elternknoten> | | | |
| Secondary | LocatedNode |  |  <Zielknoten-instanzen> '{task=' <TaskList>'}' |  |   |
| | ComposedNode |  <innere Elternknoten> | | | |

### 6.3.2 Task-Constrained Edges

Ein weiterer wichtiger Bestandteil des SSIML/Tasks-Metamodells ist ein Inkrement des SSIML-`core`-Elements `ParentChildRelationship`, dessen Instanzen mit einer Menge von Tasks in Beziehung gesetzt werden können. Daher wurde im SSIML/Tasks-Metamodell ein Element `ParentChildRelationship` mit dem Attribut `task` definiert, dass auf Modellebene eine Menge von Referenzen auf UML-`CallBehaviorAction`-Elemente enthalten kann. Um das Element `ParentChildRelationship` des `core`-Paketes mit dem `task`-Attribut zu erweitern, wurde das Paket der SSIML/Tasks-Metaklassen mit dem SSIML-`core`-Paket per UML-*Package-Merge* verschmolzen. Eine Instanz der um das `task`-Attribut erweiterten Metaklasse `ParentChildRelationship` wird auch als *Task-Constrained Edge* bezeichnet.

Eine Task-Constrained Edge ist mit einem oder mehreren Tasks aus dem Taskflow-Modell (s. voriger Abschnitt) assoziiert. Eine Laufzeit-Entsprechung einer solchen Kante kann nur traversiert werden, wenn einer der ihr zugeordenten Tasks der aktuelle Benutzertask ist. D. h. die 3D-Inhalte von Instanzen des Kantenzielknotens und deren Nachkommen im Szenengraph werden nur dann angezeigt, wenn die Kante zwischen dem Zielknoten und seinem Elternknoten mit dem gerade aktiven Task assoziiert ist. Durch die Verwendung von Task-Constrained Edges im Zusammenhang mit dem Taskflow-Modell ist es möglich, dem Benutzer nur die Informationen zu präsentieren, die für den gerade zu bearbeitenden Task von Bedeutung sind. Tabelle 6.2 zeigt die Notation von Task-Constrained Edges einschließlich einfacher Beispiele. In den Beispielen werden die Laufzeitentsprechungen der Kantenzielknoten nur dann dargestellt, wenn der aktuelle Benutzertask entweder der Task `RemovePCCover` oder `ReplacePCCover` ist. Auf die konkrete Anwendung der Task-Constrained Edges wird in Kapitel 9 im Detail eingegangen.

## 6.4 Modell-Code-Abbildung

Da die Abbildung von Task-Constrained Edges auf Code im engen Zusammenhang mit der Codeabbildung des Szenenmodells steht, wird darauf ebenfalls in Kapitel 9 anhand eines konkreten Beispiels ausführlicher eingegangen. Im vorliegenden Abschnitt steht hingegen die Generierung der Taskflow-Beschreibung im Vordergrund. Zum besseren Verständnis der nachfolgenden Erläuterungen soll hier nur erwähnt werden, dass Task-Constrained Edges in einem 3D-Format auf so genannte `Switch`-Knoten abgebildet werden können, welche Gruppenknoten sind, bei denen sich das Rendering von enthaltenen Kindknoten ein- und ausschalten lässt. Durch externen Zugriff auf einen Switchknoten der 3D-Szene zur Laufzeit ist damit eine taskabhängige Visualisierung von 3D-Inhalten möglich.

Für die Übersetzung des Taskflow-Modells wurde im Rahmen der vorliegenden Arbeit beispielhaft ein Ansatz realisiert, der die Beschreibung des Taskflows nicht direkt in Programmiersprache wie Java übersetzt (obgleich dies möglich wäre), sondern in ein einfaches, kompaktes und von Menschen lesbares XML-Format. Ein Dokument in diesem Format enthält die Abfolge der auszuführenden Einzeltasks sowie Entscheidungsknoten (`Options`-Knoten), falls eine Auswahl zwischen alternativen Tasks möglich ist. Unten stehender Code zeigt ein Beispiel der XML-codierten Taskflow-Beschreibung für das Grafikkarteninstallationsszenario. Wie zu sehen ist, werden zusammengesetzte Tasks bei der Codegenerierung in ihre Einzeltasks aufgelöst.

```
<?xml version="1.0" encoding="UTF-8"?>
<TaskSequence name="tasks">
   <Task name="RemovePCCover"/>
   <Task name="RemoveMetalCover"/>
   <Task name="FitInGraphicsCard"/>
   <Task name="FastenGraphicsCard"/>
   <Options>
      <Option name="powerSupplyRequired">
         <Task name="ConnectPowerSupply"/>
         <Task name="ReplacePCCover"/>
      </Option>
      <Otherwise>
         <TaskRef name="ReplacePCCover"/>
      </Otherwise>
   </Options>
</TaskSequence>
```

Eine solche Taskflow-Beschreibung wird von einem einfachen Java-basierten Player (dem `TaskflowManager`) geladen und kann durchlaufen werden, wobei die Schnittstelle des Players Methoden bietet, um den Übergang zum nächsten bzw. die Rückkehr zum letzten Task von außen zu steuern. Gibt es beim Übergang von einem Task zum nächsten mehrere Alternativen, werden Informationen aus der so genannten `contextMap` gelesen, um zu bestimmen, welches der nächste Task ist. Die `contextMap`, welche eine Instanz der `java.util.Properties`-Klasse ist, enthält eine Menge von Schlüssel-Wert-Paaren, wobei die Werte den primitiven Typ `Boolean` haben. Die Anfangsinhalte der `contextMap` werden aus einer Java-Properties-Textdatei geladen, welche ebenfalls automatisch aus dem Taskflow-Modell generiert wird. Diese Datei enthält für alle Entscheidungsmöglichkeiten einen Eintrag mit dem Namen des entsprechenden *Guards* aus dem Taskflow-Modell und dem Standardwert *true*. Die Werte in der `contextMap` können zur Laufzeit über entspre-

chende, in der `Properties`-Klasse definierte Methoden modifiziert werden. Für den dafür notwendigen externen Zugriff auf das `contextMap`-Objekt stellt der `TaskflowManager` die Methode `getContextMap` zur Verfügung.

Eine `else`-Transition (s. Abschnitt 6.3.1) wird in der XML-codierten Taskflow-Spezifikation durch einen `Otherwise`-Knoten repräseniert. Es kann auch vorkommen, dass aufgrund mehrerer gleichzeitig wahr gewordener Bedingungen mehrere alternative Übergänge zwischen Elementen der Taskflow-Beschreibung möglich sind. In diesem Fall wird immer die erste zutreffende Option entsprechend der XML-Beschreibung gewählt. Enthält beispielweise ein `Options`-Element zwei `Option`-Elemente *replacementRequired* und *componentDamaged* (in dieser Reihenfolge), deren Guards beide beide *true* sind, wird das erste Element (*replacementRequired*) ausgewählt.

Damit der `TaskflowManager` die für den aktuellen Task relevanten Objekte einblenden und nach dessen Abarbeitung wieder ausblenden kann, benötigt er Zugriff auf die entsprechenden `Switch`-Knoten der 3D-Szene. Daher wird eine weitere Java-Properties-Datei generiert, in die sowohl Informationen aus dem Taskflow-Modell als auch aus dem SSIML-Szenenmodell einfließen. In diese Datei wird für jeden Task (*Schlüssel*) aus dem Taskflow-Modell eine Liste der zugehörigen `Switch`-Knotennamen (d.h. eine Liste mit den Namen derjenigen `Switch`-Knoten, die durch den Task aktiviert werden sollen) als *Wert* generiert. Die Beziehung zwischen einem Task und den entsprechenden `Switch`-Knoten kann aus den an die *Task-Constrained Edges* angetragenen Zusicherungen (vgl. Abschnitt 6.3.2) extrahiert werden.

## 6.5 Verwandte Arbeiten

Visuelle Taskmodelle können die Struktur von (komplexen) Tasks durch hierarchische Zerlegung in Subtasks verdeutlichen, wodurch Taskbäume (Task Trees) entstehen, deren Knoten die einzelnen (Sub-)Tasks repräsentieren. Weiterhin können zeitliche Zusammenhänge zwischen Tasks (der Task-Flow) modelliert werden. Sowohl zur Modellierung der Taskstrukur als auch der Taskreihenfolge gibt es eine Vielzahl von Ansätzen, wobei die meisten Ansätze es erlauben, sowohl zeitliche als auch strukturelle Zusammenhänge zwischen Tasks zu spezifizieren.

Einer der ältesten Task-Beschreibungstechniken, deren Ideen in viele spätere Ansätze eingeflossen sind, ist die Hierarchische Task-Analyse (*Hierarchical Task Analysis* - HTA [AD69, ADSG71]). Einer der Hauptverdienste der HTA ist sicherlich die klare Darstellung von Taskstrukturen. Spätere bekannte Ansätze zur Taskmodellierung, die durch visuelle Werkzeuge gestützt werden, sind bspw. die *Concur Task Trees* (CTTs) [MPS02] und die *Groupware Task Analysis* (GTA) [Wel01]. Die CTTs kombinieren das Konzept der Taskbäume mit Zeitoperatoren, die auf den LOTOS-Operatoren [Int98] basieren. Diese Zeitoperatoren werden mit in den Taskbaum integriert. Die GTA zielt auf die Modellierung komplexer Umgebungen ab, in denen viele Benutzer mit interaktiven Systemen arbeiten. Die Werkzeugunterstützung für die GTA wurde durch die *Euterpe Workbench* [Wel01] realisiert. Die Euterpe Workbench bietet verschiedene Sichten auf ein Taskmodell:

Der Hierarchy Viewer visualisiert bspw. die Taskstruktur, während der Fokus des Workflow Viewers auf der Präsentation des Task-Flows liegt.

Auch UML erlaubt die Modellierung von Tasks, insbesondere durch die Verwendung von Aktivitätsdiagrammen. Besonders gut lässt sich mit Aktivitätsdiagrammen der Task-Flow darstellen. Durch die Zerlegung von Aktivitäten in Teilaktivitäten sind allerdings auch strukturelle Zusammenhänge zwischen Tasks modellierbar. Als Basis für SSIML/Tasks wurden UML-Aktivitätsdiagramme gewählt, da UML als De-facto-Standard einen höheren Verbreitungs- und Bekanntheitsgrad besitzt, als Ansätze, die ausschließlich der Taskmodellierung dienen. Damit entfällt für den Entwickler, der bereits Erfahrungen mit UML besitzt, der Aufwand, eine weitere Task-Modellierungssprache erlernen zu müssen. Außerdem kann SSIML/Tasks relativ einfach mittels des UML-Profiling-Mechanismus' in UML-Werkzeuge integriert werden. Ebenso wird SSIML/Tasks dem Prinzip des SSIML-Gesamtkonzeptes gerecht, Elemente des UML-Metamodells wiederzuverwenden, die sich zur Lösung einer gegebenen Problemstellung anbieten.

## 6.6 Zusammenfassung

In diesem Kapitel wurde die Bedeutung interaktiver 3D-Handbücher bzw. 3D-Instruktoren in aufgabenorientierten Domänen wie Montage und Wartung unterstrichen. Derartige Anwendungen setzen fortgeschrittene 3D-Visualisierungtechniken ein, um Schritt- für Schritt-Anleitungen für die Montage oder Demontage von Produkten und Bauteilen zu geben. Zur Unterstützung der Entwicklung solcher 3D-Anwendungen wurde die SSIML-Erweiterung SSIML/Tasks eingeführt. SSIML/Tasks erlaubt die Spezifizierung von Arbeitsschrittfolgen in so genannten Taskflow-Modellen. Taskflow-Modelle werden als UML-Aktivitäten modelliert. Zusätzlich wurde das Konzept der Task-Constrained Edges eingeführt. Durch die Kombination von Taskflow-Modellen und Task-Constrained Edges kann eine taskabhängige Darstellung von Objekten der 3D-Szene erreicht werden. Somit kann dem Benutzer diejenige Auswahl an Informationen präsentiert werden, die für den aktuellen Arbeitsschritt von Bedeutung ist.

Die werkzeuggestützte Abbildung der Taskflow-Modelle auf Code sorgt für einen nahtlosen Übergang zur Implementierung. Der für ein SSIML-Taskflow-Modell generierte Code benötigt keine manuelle Nachbearbeitung, wodurch Entwicklungskosten und -zeit gespart werden können.

Ein sinnvoller Einsatz einiger SSIML/Tasks-Elemente ist allerdings nur in Kombination mit anderen SSIML/Komponenten, wie der Kernkomponente SSIML oder SSIML/AR möglich. Auf die konkrete Anwendung von Task-Constrained Edges wird daher in Kapitel 9 ausführlicher eingegangen.

# Kapitel 7

# SSIML/Behaviour

## 7.1 Einführung

Wichtige Elemente moderner interaktiver 3D-Anwendungen sind detaillierte Animationen und komplexe Verhaltensweisen von 3D-Objekten. Jedoch stellt deren Integration nach wie vor eine Herausforderung für den Entwickler dar. Dies zeigt sich u. a. an der Zahl aktueller Forschungsarbeiten, welche sich mit der Verhaltensspezifikation beschäftigen [BE05, DR03].

Eine Ursache für diese Situation ist darin zu sehen, dass es an Konzepten und Werkzeugen mangelt, die die Verhaltensspezifikation auf einer abstrakteren Ebene als der Implementierungsebene ermöglichen. Derartige Konzepte und Werkzeuge könnten bereits vor der Implementierung Hilfestellung leisten, auf hohem Abstraktionsniveau Verhalten zu spezifizieren, komplexe Verhaltensstrukturen in übersichtlichere und leichter erfassbare Teilstrukturen zu zerlegen und in diesem Zuge die Stabilität, Wiederverwendbarkeit und Wartbarkeit der Verhaltenskomponenten zu verbessern. Durch automatische Codegenerierung könnte auch hier ein nahtloser Übergang zur Implementierungsphase und damit die Reduzierung von Implementierungsfehlern und des Implementierungsaufwandes erreicht werden.

An dieser Stelle soll nicht verschwiegen werden, dass einige Autorenumgebungen, wie der Internet Scene Assembler von Parallelgraphics [ISA] oder Adobe Director [Dir], durchaus die Möglichkeit bieten, Verhalten mittels visueller Werkzeuge zu spezifizieren. Jedoch wird das Potenzial des Entwicklers oft durch werkzeugspezifische Metaphern eingeschränkt. Bspw. beschränken viele Werkzeuge die Animationsunterstützung auf Keyframe-Animationen unter Verwendung der Zeitleistenmetapher. Benutzer - Objekt- und Objekt - Objekt-Interaktionen sowie spezielle Animationsarten, wie physikbasierte oder bestimmte prozedurale Animationen, lassen sich häufig entweder gar nicht oder nur durch die Verwendung von Skript- oder Programmiersprachen realisieren. Letzteres bedeutet einen Übergang auf ein niedrigeres Abstraktionsniveau und damit die Erhöhung des Realisierungsaufwandes. Hinzu kommt, dass bei solchen Autorenumgebungen Skriptcode und 3D-Inhalte bei der Dateispeicherung oft nicht separiert werden, was die Wartung des Verhaltenscodes erschwert.

Im Gegensatz zu grafischen Autorenwerkzeugen haben high-level 3D-Grafikbibliotheken und Frameworks (z. B. Java 3D [J3D], Open Inventor [SC92]) Stärken bei der Unterstützung von Codemodularisierung und ermöglichen normalerweise eine klare Trennung von Verhalten und Geometrie. Gerade dies ist bei größeren interdisziplinären Projekten von Bedeutung, in die eine Vielzahl von Entwicklern mit unterschiedlichen Kompetenzen (z. B. Programmierer, Designer) involviert sind. Die Separation von Code und Inhalt ermöglicht nicht nur die parallele Entwicklung von Programmcode und 3D-Szenen, sondern verbessert auch die Wiederverwendbarkeit, die Erweiterbarkeit und die Wartbarkeit der einzelnen Softwarebestandteile. Neben den bereits erörterten Vorteilen bieten 3D-Bibliotheken und -Frameworks zusätzlich meist einen hohen Grad an Flexibilität bezüglich der Entwicklung komplizierter Verhaltensweisen und Animationen. Genau darin besteht jedoch auch ihr größter Nachteil: Die Spezifikation von Objektverhalten auf Codeebene unter Benutzung von - oft komplex strukturierten - Bibliotheken und Frameworks kann sich zu einer zeitraubendenden Tätigkeit entwickeln, bei der ein hoher Aufwand für das Beseitigen von Implementierungsfehlern betrieben werden muss.

In diesem Kapitel wird deshalb ein Ansatz präsentiert, der das Ziel hat, die Vorteile von grafischen Autorenwerkzeugen und Frameworks zu kombinieren. Der Hauptfokus liegt auf der *plattformunabhängigen Spezifikation von Verhalten und Animation auf der Entwurfsebene*. Weitere Aspekte beinhalten

- die hierarchische Dekomposition komplexer Verhaltensbeschreibungen,
- die Kapselung des 3D-Objektverhaltens in Komponenten und – damit zusammenhängend – die Trennung von Verhalten und Geometrie
- die Unterstützung parallel ablaufender Animationen,
- die Unterstützung von verschiedenen Animationsarten, z.B. von keyframe- und physikbasierten Animationen, sowie
- die Übersetzung der Verhaltensbeschreibungen in plattformspezifischen Code.

Für die Verhaltenspezifikation wird eine visuelle Modellierungssprache eingeführt, welche auf UML 2-Zustandsautomaten [Obj07c] basiert. Diese erlaubt eine verständliche und klare Verhaltensbeschreibung auf einer moderaten Abstraktionsebene. Die zweite Säule des Ansatzes ist die Scene Structure and Integration Modelling Language (SSIML), welche die Beschreibung von 3D-Strukturen und ihre Anbindung an Anwendungslogik sowie Verhaltenskomponenten ermöglicht. Die Kombination von UML 2 -Zustandsautomaten und SSIML führt zur SSIML-Erweiterung *SSIML/Behaviour*.

### 7.1.1 Verhalten graphischer Objekte

Burrows und England [BE05] unterstreichen, dass viele Autoren den Begriff *Verhalten* (*Behaviour*) im Kontext graphischer Anwendungen benutzen, ohne ihn klar zu definieren. In dieser Arbeit wird der Begriff *Objektverhalten* durch die Aktionen definiert, die ein

graphisches Objekt ausführt. Werden diese Aktionen durch auftretende Ereignisse hervorgerufen, spricht man (im Gegensatz zu autonomen Verhalten) von einem *ereignisbasierten Verhalten* [HMB03]. Ereignisse können bspw. durch Zeitgeber (etwa bei einem Time-Out), den Benutzer (z. B. durch einen Mausklick) oder andere Objekte (z. B. bei Kollisionen) ausgelöst werden. Der Begriff *Interaktion* beschreibt den gesamten Prozess, ausgehend von der ereignisauslösenden *Aktion* eines Interaktionspartners bis hin zur *Reaktion* des anderen Interaktionspartners. Abhängig von den Partnern der Interaktion wird zwischen *System-Objekt-Interaktion*, *Benutzer-Objekt-Interaktion* und *Objekt-Objekt-Interaktion* unterschieden [BE05]. In dieser Arbeit wird die Spezifikation ereignisbasierten Verhaltens beschrieben. Die Spezifikation von Interaktionen wird hingegen nicht genauer betrachtet.

### 7.1.2 UML 2-Zustandsautomaten

UML 2-Zustandsautomaten spiegeln den aktuellen Stand der Forschung und Technik hinsichtlich der Theorie und Notation von Zustandsautomaten wider. Sie sind Varianten der Harel-State Charts [Har87], die selbst eine Erweiterung hierarchischer Zustandsautomaten (*Hierarchical State Machines - HSMs*) darstellen.

#### 7.1.2.1 Eignung von UML 2-Zustandsautomaten zur Beschreibung des Verhaltens graphischer Objekte

UML 2-Zustandsautomaten bieten einen visuellen Formalismus zur Spezifikation des Verhaltens von Objekten. Wichtige Gründe für die Eignung von UML 2-Zustandsautomaten als Fundament der Verhaltensbeschreibung graphischer Objekte sind:

- Komplexe Verhaltensweisen können mit Hilfe von zusammengesetzten Zuständen (*Composite States*) und Unterzustandsautomaten (*Substate Machines*) hierarchisch strukturiert werden.
- Unterzustandsautomaten können innerhalb eines Zustandsautomaten wiederverwendet werden.
- UML 2 stellt spezielle Erweiterungsmechanismen für Zustandsautomaten bereit.
- Über orthogonale Regionen innerhalb eines Zustandes können parallele Abläufe spezifiziert werden.
- Zeitgesteuerte Zustandsübergänge sind definierbar.

#### 7.1.2.2 Überblick über UML 2- Zustandsautomaten

Der an dieser Stelle präsentierte Überblick über UML 2-Zustandsautomaten dient dem besseren Verständnis der Darlegungen in den nachfolgenden Abschnitten. Basiselemente von UML 2-Zustandsautomaten sind *Zustände* und *Transitionen*. Ein Zustand repräsentiert eine Zuordnung von Werten zu Eigenschaften eines Objekts zu einem bestimmten

Zeitpunkt. Ein Zustand kann nur über eine Transition betreten oder verlassen werden. Ein einfacher Zustand wird in der UML-Notation durch ein Rechteck mit abgerundeten Ecken dargestellt. Das Rechteck enthält den Namen des Zustandes. Neben „echten“ Zuständen gibt es auch als *Pseudo-Zustände* bezeichnete Elemente. Ein wichtiger Pseudo-Zustand ist der *Startzustand*. Das Durchlaufen eines Zustandsautomaten beginnt mit dem Startzustand und endet mit dem so genannten *Endzustand*. Eine Transition verbindet zwei Zustände. Sie wird als gerichtete Kante zwischen den zwei verbundenen Zuständen dargestellt. Die Ausführung einer Transition, d.h. der Übergang von einem Zustand in den über die Transition angebundenen Zustand, hat keine zeitliche Dauer, ein Übergang erfolgt also unmittelbar. Eine Transition wird durchlaufen, nachdem sie ausgelöst wurde. Es existieren verschiedene *Auslöser* (*Trigger*) in UML 2. Die folgenden Auslöser finden in SSIML/Behaviour Verwendung.

- *SignalTrigger*: Die Transition wird durch ein Ereignis (das *Signal Event*) ausgelöst, das vom System übermittelt wurde. Solch ein Auslöser könnte bspw. eine Benutzerinteraktion, wie ein Mausklick, sein.
- *ChangeTrigger*: Die Transition wird ausgelöst, sobald ein boolescher Ausdruck wahr wird, z. B. wenn einer Objekteigenschaft ein bestimmter Wert zugeordnet wird.
- *TimeTrigger*: Die Transition wird nach einer gewissen Zeitdauer oder zu einem definierten Zeitpunkt ausgelöst.
- *CompletionTrigger*: Die Transition wird ausgelöst, wenn die so genannte *do-Aktivität* (*do activity*) des Quellzustandes der Transition beendet wurde.

Eine Transition wird mit einem Ausdruck mit der Syntax *<Auslöser> ('['<Guard>']')? ('/'<Aktivität>)?* versehen. *<Auslöser>* bezeichnet den oder die Auslöser der Transition (es sind mehrere Auslöser spezifizierbar). Bei einem *SignalTrigger* wird der Signalname notiert, optional gefolgt von einer Liste von Signalparametern. Die Parameterwerte können bspw. im nachfolgenden *Guard* oder in der Aktivität der Transition abgefragt werden. Ein *ChangeTrigger* wird in der Syntax *'when ('<boolescher Ausdruck>')'* notiert. Ein *TimeTrigger* hat die Syntax *'at('<Zeitpunkt>')'* oder *'after('<Zeitdauer>')'*, wobei Zeitdauer der Zeitraum nach dem Eintreten in den Zustand ist. Ein *CompletionTrigger* wird nicht extra notiert. Zu erwähnen ist außerdem, dass eine Transition, die vom Startzustand ausgeht, keinen Auslöser besitzt. Als Zusatz zum Auslöser kann noch ein *Guard* spezifiziert werden. Ein *Guard* ist ebenfalls ein boolescher Ausdruck. Ein *Guard* überprüft bspw., ob einer Objektvariablen ein bestimmter Wert zugeordnet ist. Eine Transition kann nur durchlaufen werden, wenn der *Guard* wahr ist. Schließlich kann eine Aktivität definiert werden, die bei einem Zustandsübergang ausgeführt wird. Diese darf aber definitionsgemäß keine Zeit in Anspruch nehmen.

Zusätzlich haben Zustände *eigene Aktivitäten*. Diese sind die *entry-, do-* und *exit*-Aktivität. Die *entry*-Aktivität wird beim Betreten, die *exit*-Aktivität beim Verlassen des Zustandes ausgeführt. Die *do*-Aktivität wird ausgeführt, während der Zustand *aktiv* ist. Im Gegensatz zur *entry*- und *exit*-Aktivität kann die *do*-Aktivität unterbrochen werden.

In den nächsten Abschnitten werden die Sprache SSIML/Behaviour sowie ihre Abbildung auf Code genauer betrachtet. In Abschnitt 7.4 folgt eine Diskussion verwandter Arbeiten.

## 7.2 Metamodell

Da SSIML/Behaviour z. T. auf UML 2-Zustandsautomaten basiert, werden entsprechende Bestandteile des UML 2-Metamodells wiederverwendet. Zum SSIML/Behaviour-Metamodell wurde ein entsprechendes UML-Profile definiert, dass die Integration der Modellierungssprache in UML-Werkzeuge ermöglicht. Konkret wurde SSIML/Behaviour auf diese Weise in das UML-Werkzeug MagicDraw [Mag] integriert.

Die Kernelemente von SSIML wurden mittels Paketzusammenführung (*Package-Merge*) [Obj07c] in SSIML/Behaviour übernommen. Eine Paketzusammenführung ist deshalb notwendig, da in SSIML/Behaviour Elemente des SSIML-`core`-Metamodells teilweise verändert werden.

### 7.2.1 Verhalten und Verhaltensdefinitionen

Ein *Verhalten* (Metaklasse `Behaviour`) in SSIML ist in einer Verhaltensdefintion (Metaklasse `BehaviourDefinition`) spezifiziert. Ein *Verhalten* besitzt eine Referenz auf die ihr zugehörige Verhaltensdefinition und kann mit Attributen von Knoten eines SSIML-Szenenmodells assoziiert werden. Die Instanzen der Attribute können zur Laufzeit durch das Verhaltensobjekt verändert werden. In Abbildung 7.1 wird dieser Sachverhalt dargestellt (für eine Erklärung der farblichen Kennzeichnung der Metaklassen s. Kapitel 5, Abbildung 5.5). Welche Instanzen eines Zielattributes adressiert werden, kann über das Attribut `targetInstanceValues` der Beziehung zwischen dem Verhalten und den Zielattribut (Metaklasse `BehaviourRelationship`) in Form einer Menge von relativen Kompositionspfaden (s. Abschnitt 5.2.2.13) festgelegt werden. Über die Kompositionspfade können auch einzelne Werte selektiert werden, wenn ein Zielelement wie ein Transformationsattribut oder ein `RBDynamics`-Attribut eine Liste von Werten beinhaltet. Das Attribut `sourceStates` der Metaklasse `BehaviourRelationship` gibt an, in welchen Zuständen (vgl. Abschnitt 7.2.4.3) des Verhaltens die Zielelementinstanzen einer `BehaviourRelationship`-Beziehung verändert werden.

Das Verhalten eines (zusammengesetzten) 3D-Objektes wird durch alle `Behaviour`-Elemente bestimmt, die den Attributen der Knoten des Objekts zugeordnet sind. Ein SSIML-Verhalten besitzt zur Laufzeit *genau eine* Instanz (nachfolgend als *Verhaltensobjekt* bezeichnet). Ist ein Verhalten mit einem Knotenattribut im SSIML-Graphen über eine `BehaviourRelationship`-Beziehung assoziiert, wird zur Laufzeit die Instanz des Verhaltens mit allen Instanzen des Knotenattributes verbunden, sofern nicht eine Einschränkung der Menge der Attributinstanzen über das Attribut `targetInstanceValues` der `BehaviourRelationship`-Beziehung getroffen wurde. Ist etwa ein Verhalten `wheelRotationBehaviour` mit einem Transformationsattribut `wheelTrans` verbunden, welches vier Instanzen besitzt,

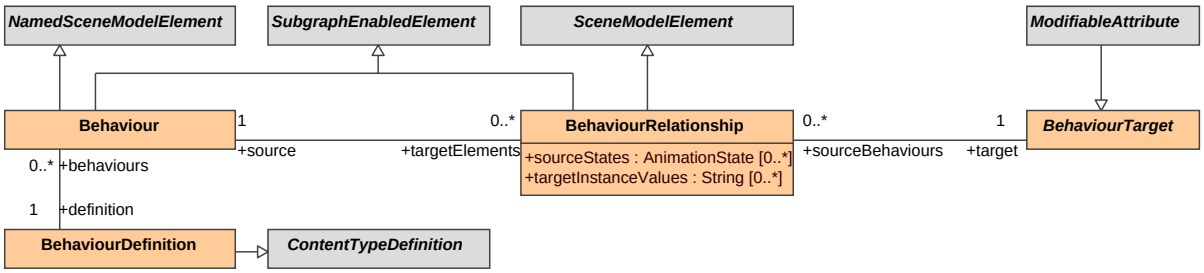



Abbildung 7.1: Die Metaklassen `Behaviour`, `BehaviourDefinition` und `BehaviourRelationship`

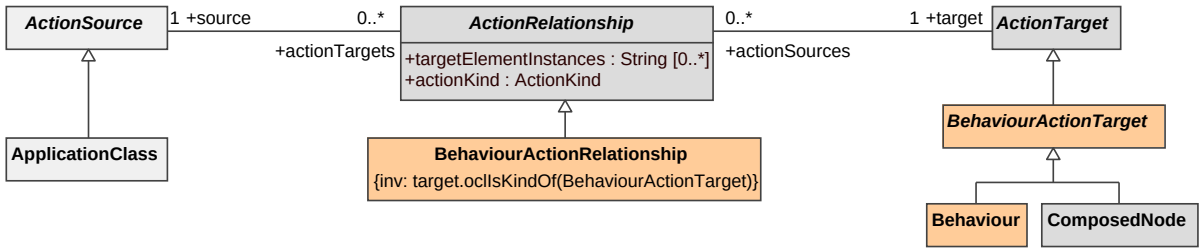



Abbildung 7.2: Beziehung zwischen `ApplicationClass` und `Behaviour`

wird die Instanz von `wheelRotationBehaviour` mit allen vier Instanzen von `wheelTrans` verknüpft.

Ein Verhaltensobjekt kann zur Laufzeit Nachrichten empfangen, die durch die in Abschnitt 7.2.3 beschriebenen `MessageEvent`s repräsentiert werden. Diese Events können von Anwendungskomponenten (Instanzen von `ApplicationClass`-Elementen) oder Sensoren der 3D-Szene stammen, deren Entsprechungen im SSIML-Modell mit dem Verhalten verbunden sind. Die Abbildungen 7.2 und 7.3 illustrieren diesen Sachverhalt anhand von Ausschnitten aus dem SSIML/Behaviour-Metamodell. Der Wert des Attributes `actionKind`, das `BehaviourActionRelationship` von der Klasse `ActionRelationship` erbt, hat für eine `BehaviourActionRelationship`-Instanz keine Auswirkungen, da für ein `BehaviourActionRelationship`-Element keine Aktionsunterarten unterschieden werden.

Abbildung 7.4 zeigt eine Variante des Interrelationenmodells aus dem Fahrzeugkonfiguratorbeispiel (Abbildung 5.38) aus Kapitel 5. Die Transformation (`hoodTransform`) der

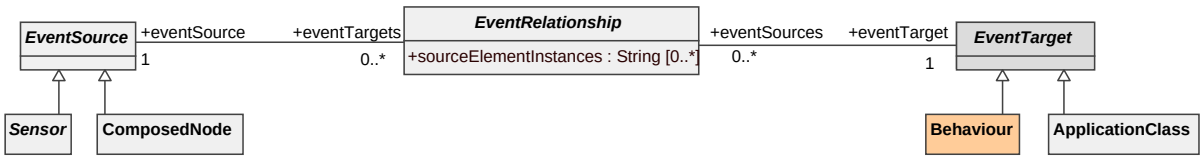



Abbildung 7.3: Beziehung zwischen `Sensor` und `Behaviour`

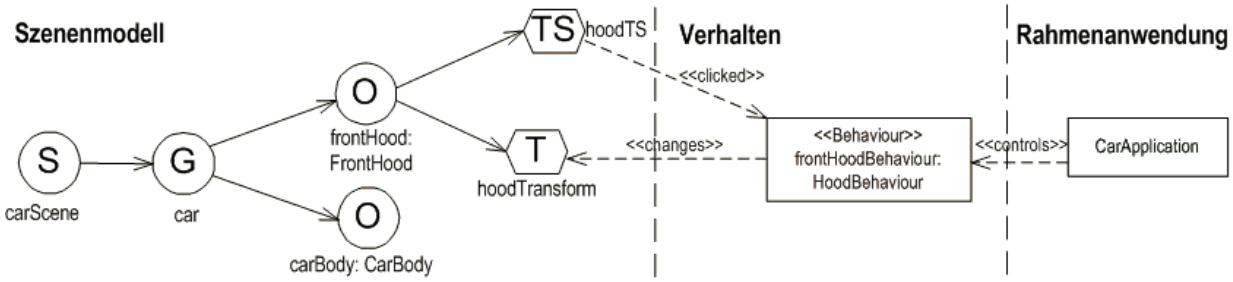



Abbildung 7.4: Interrelationenmodell mit Verhaltensobjekt

Motorhaube (`Object frontHood`) wird durch das `frontHoodBehaviour`-Verhalten beeinflusst, welches in einer Verhaltensdefinition `HoodBehaviour` spezifiziert wurde. `CarApplication`-Objekte können das `frontHoodBehaviour`-Verhaltensobjekt durch das Senden von Nachrichten kontrollieren (eine Anwendungskomponente kann Nachrichten zu allen Verhaltensobjekten senden, zu denen entsprechend des SSIML-Modells Beziehungen bestehen). Zu Zwecken der Übersichtlichkeit darf ein und dasselbe Verhalten auch mehrfach in einem SSIML-Diagramm dargestellt werden.

Nachrichten von Sensoren setzen sich aus der Bezeichnung der Sensorinstanz und der Spezifikation des aufgetretenen Ereignistyps zusammen (z. B. „hoodTS clicked“). Die Sensorinstanzbezeichnung setzt sich zusammen aus dem Sensornamen (einschließlich des Namensraumpräfixes, aber ohne vorangehenden Schrägstrich) sowie – falls mehrere Sensorinstanzen existieren – einem Instanzselektor. Der Ereignistyp wird durch die Zeichenketten *clicked, pressed, released, entered, exited* (bei `TouchSensor`), *proximity* (bei `ProximitySensor`), *visibility* (bei `VisibilitySensor`) und *collision* (bei `CollisionSensor`) bestimmt. Im Gegensatz dazu können Nachrichten von Anwendungskomponenten beliebig sein. Ebenso können Anwendungskomponenten, die eine Beziehung zu einem Behaviour-Objekt aufweisen, das Behaviour-Objekt aktivieren oder deaktivieren. Nach einer Deaktivierung kann das Verhalten erneut aktiviert werden. Die Aktivierung bzw. Deaktivierung eines Behaviour-Objektes bedeutet den Beginn der Ausführung eines dem Verhaltensobjekt zugeordneten Zustandsautomaten (s. nächster Abschnitt) bzw. den Abbruch der Ausführung des Automaten.

### 7.2.2 Behaviour-Engines

Eine Verhaltensdefinition enthält die Spezifikation eines Verhaltens in Form eines erweiterten UML 2-Zustandsautomaten, der auch *Behaviour-Engine* („Verhaltensmaschine“) genannt wird (Abbildung 7.5). Eine Behaviour-Engine kann besondere Zustände, so genannte `AnimationStates` beinhalten.

Muss die Anwendung auch direkt auf ein Attribut eines 3D-Objektes zugreifen, bspw. um Transformationswerte zu lesen und damit Kollisionen zwischen 3D-Objekten zu prüfen, können zusätzlich *Reads*-Beziehungen (Abschnitt 5.3.1.1) zwischen dem entsprechenden Knotenattribut und der Anwendungskomponente modelliert werden.

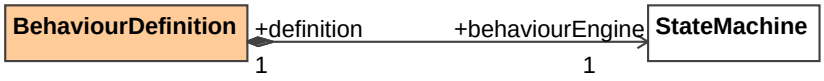



Abbildung 7.5: Behaviour-Engine im Metamodell

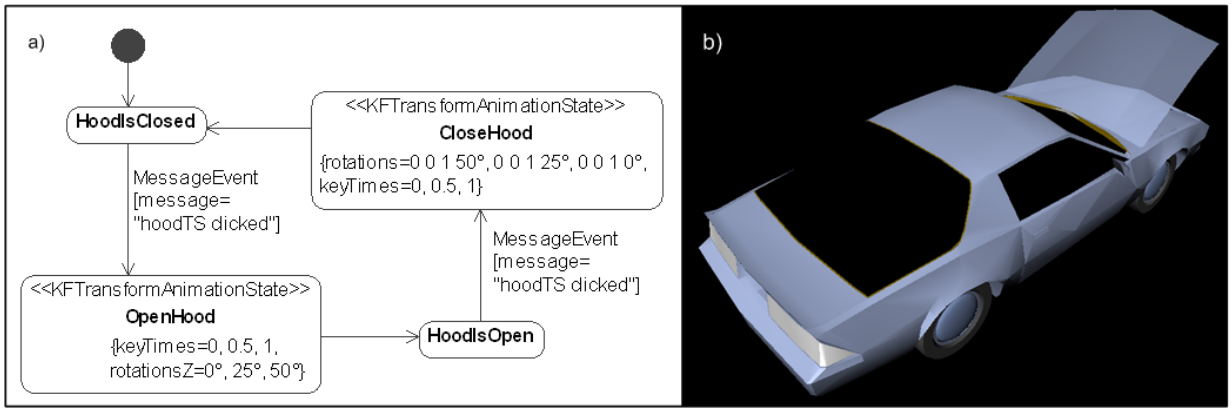



Abbildung 7.6: Behaviour-Engine-Beispiel mit (a) der Behaviour-Engine und (b) der 3D-Visualisierung des beschriebenen Verhaltens

Ein einfaches Beispiel für eine Behaviour-Engine ist in Abbildung 7.6 zu sehen. Die Engine enthält die Spezifikation des `frontHoodBehavior`-Verhaltens aus Abbildung 7.4. Bevor das Beispiel eingehender erklärt wird, sollen zunächst die grundlegenden Modellelemente von Behaviour-Engines beschrieben werden.

### 7.2.3 Funktionen, Prozeduren und Ereignisse

Für die Verhaltensbeschreibung mittels Behaviour-Engine-Zustandsautomaten wurden einige nützliche Funktionen, Prozeduren und Ereignisse vordefiniert. Die nachfolgend vorgestellte Auswahl an Funktionen, Prozeduren und Ereignissen erwies sich für die im Rahmen der vorliegenden Arbeit modellierten Beispiele als ausreichend.

- Die Funktion *absolute* (Syntax *'absolute('<Vector>')'*) errechnet den Betrag eines Vektors. Die Anwendung dieser Funktion kann u. a. im Zusammenhang mit *ChangeTriggern* sinnvoll sein. Ein Beispiel wäre die Überprüfung, ob der Betrag eines Beschleunigungsvektors (also die Beschleunigung) ein bestimmtes Limit übersteigt.

- Die Funktion *instanceOf* (Syntax *'instanceOf('<Object>','<Type>')'*) gibt einen booleschen Wert zurück (*true* oder *false*), abhängig davon, ob *<Object>* Instanz des Typs *<Type>* ist.

- Die Prozedur *sendMessage* (Syntax *'sendMessage('<String>')'*) sendet eine Nachricht von einem Verhaltensobjekt zu den assoziierten Anwendungskomponenten. Die Nachricht kann von der Anwendung genutzt werden, um Auslöser für Transitionen

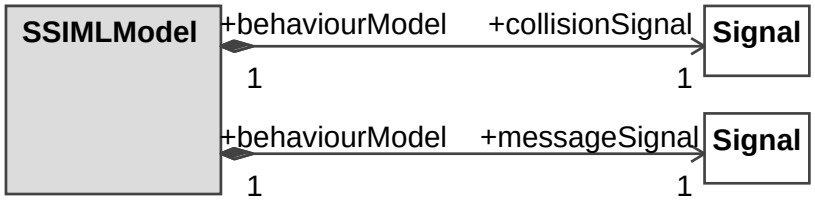



Abbildung 7.7: Signale als Bestandteile eines SSIML/Behaviour-Modells

anderer Behaviour-Engines zu generieren. Dabei kann die Anwendung die empfangene Nachricht weiterleiten oder neue Nachrichten zum Versand erzeugen. Eine plattformspezifische Implementierung des Nachrichtenübermittlungsmechanismus' muss neben der Nachricht auch eine Referenz auf den Nachrichtensender mit übermitteln, um dem Empfänger die Möglichkeit zu geben, die Identität des Senders zu prüfen. *sendMessage* kann z.B. als *entry*- oder *exit*-Aktivität eines Zustandes oder als Transitionsaktivität benutzt werden.

- Ein vordefiniertes Signal, um Zustandsübergänge auszulösen, trägt den Namen *MessageEvent* (nicht zu verwechseln mit der UML-Metaklasse `MessageEvent`). Ein *MessageEvent* kann indirekt ein Resultat einer *sendMessage*-Aktivität sein. Ein *MessageEvent* enthält als Parameter sowohl die gesendete Nachricht (*message* vom Typ `String`) als auch die Referenz auf den Sender (*sender*). Wird *MessageEvent* an einer Transition als *SignalTrigger* verwendet, können in einem *Guard* zusätzlich der Inhalt der Nachricht und der Sender überprüft werden. Die Syntax für einen solchen *Guard* wäre wie folgt: *'[message='<String>' and '<Senderüberprüfung>']'*. Eine konkretes Beispiel wäre eine Transition mit folgender Beschriftung: *MessageEvent[message = 'helicopter started' and instanceOf(sender, Helicopter)]*. Diese Transition würde also nur durchlaufen, wenn in dem *MessageEvent* eine Nachricht 'helicopter started' enthalten und der Sender der Nachricht eine Instanz des Typs *Helicopter* wäre.

- Ein weiteres vordefiniertes Signal trägt den Namen *CollisionEvent*. Eine *CollisionEvent*-Nachricht kann z. B. zu einem Verhaltensobjekt gesendet werden, wenn eines der 3D-Objekte des Verhaltensobjektes an einer Kollision beteiligt ist. Das *CollisionEvent* enthält neben der Referenz auf den Nachrichtensender eine Liste von Instanznamen der an der Kollision beteiligten 3D-Objekte. Auf welche Art die Kollision abgefragt wurde, ist nicht Gegenstand von SSIML/Behaviour.

Ein Werkzeug, das SSIML/Behaviour unterstützt, muss beim Anlegen eines neuen SSIML/Behaviour-Modells automatisch ein *MessageEvent*-Signal und ein *CollisionEvent*-Signal als vorkonfigurierte Modellelemente mit generieren. Diese Elemente können als feste Modellbestandteile in Transitionsauslösern in Behaviour-Engines verwendet werden (Abbildung 7.7).

Tabelle 7.1: Attribute und ihre Felder

| Attribut von | Feldname | Wertetyp | Werteformat-Beschreibung | Wertebereich | Standardwert |
|---|---|---|---|---|---|
| Transformation (TransformationValue) | translation | Float[3] | <x><y><z> in m | (-∞,+∞) | (0, 0, 0) |
| | rotation | Float[4] | <Achse><Winkel> (Winkel in rad) | Achse: [-1,1]; Winkel: (-∞,+∞) | (0, 0, 1, 0) |
| | uniformScale | Float[1] | <Skalierungsfaktor> | (-∞,+∞) | 1 |
| | nonUniformScale | Float[3] | <x><y><z> | (-∞,+∞) | (1, 1, 1) |
| Material | fillColour | Float[3] | <R><G><B> | [0,1] | (1, 1, 1) |
| | diffuseColour | Float[3] | <R><G><B> | [0,1] | (1, 1, 1) |
| | specularColour | Float[3] | <R><G><B> | [0,1] | (1, 1, 1) |
| | transparency | Float[1] | <Transparenz> | [0,1] | 0 |
| Camera | orientation | Float[4] | <Achse><Winkel> (Winkel in rad) | Achse: [-1,1]; Winkel: (-∞,+∞) | (0, 0, 1, 0) |
| | position | Float[3] | <x><y><z> in m | (-∞,+∞) | (0, 0, 0) |
| | fieldOfViewX | Float[1] | <Öffnungswinkel horizontal> in rad | (0,π) | 0.785 |
| | fieldOfViewY | Float[1] | <Öffnungswinkel vertikal> in rad | (0,π) | 0.785 |
| RBDynamics (RBDynamicsValue) | mass | Float[1] | <Masse> in kg | (0,+∞) | 1 |
| | linearDisplacement | Float[3] | <x><y><z> in m | (-∞,+∞) | (0, 0, 0) |
| | linearVelocity | Float[3] | <x><y><z> in m/s | (-∞,+∞) | (0, 0, 0) |
| | forces | Float[nx3] | <x><y><z> in N | (-∞,+∞) | - |
| | angularDisplacement | Float[4] | <Achse><Winkel> (Winkel in rad) | Achse: [-1,1]; Winkel: (-∞,+∞) | (0, 0, 1, 0) |
| | angularVelocity | Float[3] | <x><y><z> in rad/s | (-∞,+∞) | (0, 0, 0) |
| | torques | Float[nx3] | <x><y><z> in Nm | (-∞,+∞) | - |
| | translationDamping | Float[1] | <Dämpfungsfaktor> in 1/s | [0,1] | 0.001 |
| | rotationDamping | Float[1] | <Dämpfungsfaktor> in 1/s | [0,1] | 0.001 |
| | collisionRadius | Float[1] | <Radius> in m | (0,+∞) | 1 |
| | elasiticity | Float[1] | <Elastizität> | [0,1] | 1 |

## 7.2.4 Animationen in SSIML/Behaviour

### 7.2.4.1 Maßeinheiten, Koordinatensystem und Transformationsreihenfolge

Um dem Entwickler zu erlauben, eine Vorstellung davon aufzubauen, wie eine spezifizierte Animation letztlich auf dem Bildschirm präsentiert wird, muss eindeutig festgelegt sein, wie die Parameterwerte einer Animation zu interpretieren sind. In SSIML/Behaviour werden Abstände in Metern angegeben, Zeitwerte in Sekunden und Winkel in Radian (z. B. *1.57*). Zusätzlich sollte ein SSIML/Behaviour-Werkzeug die Eingabe bzw. Anzeige von Winkeln in (Alt-)Grad unterstützen, da die Gradangabe oft eine komfortable Alternative zur Winkelangabe in Radian darstellt (z. B. *45°* statt *0.785*). Ein rechtshändiges Koordinatensystem und die Rechte-Hand-Regel für Rotation werden eingesetzt. Translation wird vor Rotation und Skalierung ausgeführt. Diese Regeln müssen auch berücksichtigt werden, wenn ein SSIML/Behaviour-Modell in ein Zielformat übersetzt wird.

### 7.2.4.2 Attribute und Felder

SSIML-Knotenattribute enthalten Felder, die modifiziert werden können. Felder müssen insbesondere berücksichtigt werden, wenn es um die Spezifikation von Animation und Verhalten geht. Tabelle 7.1 zeigt entsprechende Attributtypen und ihre Felder.

Die Semantik der Felder eines `RBDynamics`-Attributes ist etwas komplexer. Besitzt ein `Object` ein `RBDynamics`-Attribut, wird es als starrer Körper (*R*igid *B*ody) betrachtet. In diesem Fall werden Position und Orientierung des Objektes zu einem bestimmten Zeitpunkt durch physikalische Gesetzte in Verbindung mit den Inhalten der Felder des `RBDynamics`-Attributes bestimmt.

Die Masse (`mass`) eines Objektes wird benötigt, um Beschleunigungen zu berechnen, wenn Kräfte (`forces`) oder Drehmomente (`torques`) auf das Objekt einwirken, oder um die resultierende Geschwindigkeit nach einer Kollision festzustellen. Das Objekt wird als Massepunkt betrachtet, der sich im Ursprung des lokalen Objektkoordinatensystems befindet.

`linearVelocity` und `angularVelocity` repräsentieren die lineare bzw. die Winkelgeschwindigkeit eines Objekts.

Die Werte der Felder `translationDamping` (Translationsdämpfung) oder `rotationDamping` (Rotationsdämpfung) erlauben es, die Bewegung eines Objektes über die Zeit zu dämpfen. Dämpfung ist z. B. nützlich, um Luftreibung zu simulieren. Der *Damping*-Wert für die lineare Bewegungsdämpfung wirkt sich wie folgt aus: Zu einem Zeitpunkt $t$ wirkt auf ein Objekt eine Kraft von *-1 *translationDamping* (lineare Geschwindigkeit von (t-1))*mass*. Bei einer Beschleunigung von Null kann mit der Dämpfung die Bewegung von Objekten bis zum Stillstand verlangsamt werden, bei einer Beschleunigung größer Null kann die Geschwindigkeit auf einen Maximalwert begrenzt werden. Die Anwendung von `rotationDamping` ist analog zur Anwendung von `translationDamping`.

Der Kollisionsradius (Feld `collisionRadius`) kann benutzt werden, um eine einfache Kollisionserkennung zwischen Objekten zu realisieren. Der `collisionRadius` definiert eine auf Kollision zu testende Kugel um das geometrische Objekt. Der Kugelmittelpunkt befindet sich im Ursprung des lokalen Koordinatensystems des Objekts.

Der Elastizitätswert (`elasticity`) wird mit der resultierenden Geschwindigkeit nach einer Kollision multipliziert, um die Geschwindigkeitsverringerung aufgrund des Verlustes kinetischer Energie zu simulieren.

`linearDisplacement` beschreibt die Position (Translation) des Objektes, während `angularDisplacement` die aktuelle Orientierung des Objektes im Achsen/Winkel-Format darstellt. Die Werte dieser Felder müssen zur Laufzeit mit den Werten der Felder `translation` und `rotation` des Transformationsattributes des Objektes synchronisiert werden. Repräsentiert ein Transformationsattribut eines Objektes eine Liste paralleler Transformationen, repräsentiert das `RBDynamics`-Attribut ebenso ein mehrwertiges Attribut, welches für jeden Transformationswert die Menge der `RBDynamics`-Attributfelder bereitstellt.

#### 7.2.4.3 Animationszustände

Um Animationen innerhalb von Behaviour-Engines zu beschreiben, wurde das Konzept des Animationszustandes (*Animation State*) eingeführt. Ein Animationszustand ist ein Zustand, in welchem ein zugeordnetes Objekt animiert wird; d. h. in einem Animationszustand werden die Feldwerte der Attribute des zu animierenden Objektes modifiziert. Im Gegensatz zu einem „normalen“ Zustand kann für einen Animationszustand nicht explizit eine *do*-Aktivität definiert werden. Die *do*-Aktivität eines Animationszustandes wird durch Werte von Eigenschaften eines solchen Zustandes bestimmt, die einen Animationsablauf beschreiben. Eine Animation beginnt also direkt nach dem Ausführen der *entry*-Aktivität. Eine weitere Einschränkung ist, dass ein Animationszustand kein zusammengesetzter Zustand sein darf, also keine Unterzustände enthalten darf.

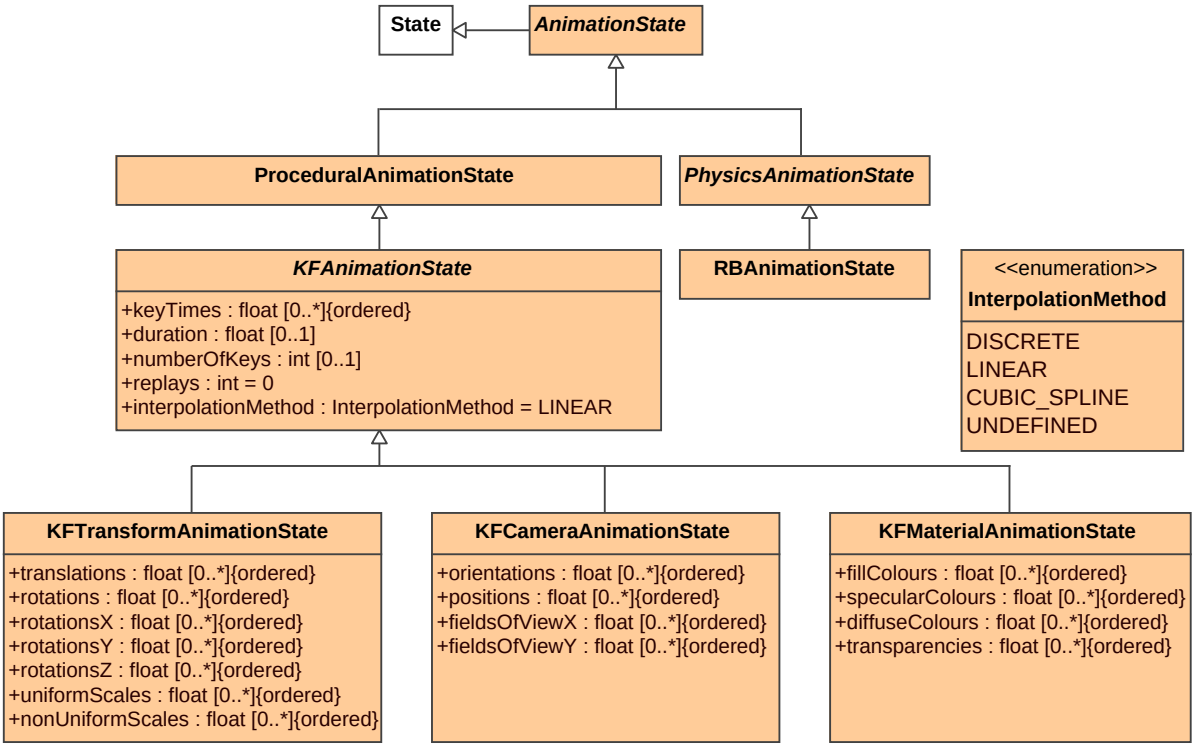



Abbildung 7.8: Hierarchie der `AnimationState`-Typen

**Kriterien für die Definition von Animationszustandstypen** Es wurde eine Hierarchie von Typen für Animationszustände definiert, um die Spezifikation verschiedener Animationsarten zu ermöglichen (Abbildung 7.8). Die Animationszustände und ihre Attribute wurden auf Basis einer Betrachtung der Animationsunterstützung in aktuellen 3D-Formaten und APIs (X3D und Java 3D), Autorenwerkzeugen (Blender3D [Ble], Internet Scene Assembler [ISA]) sowie Physikkomponenten (Bullet [Bul], X3D-Rigid-Bodies-Component [Int08b]) festgelegt. Dazu wurden grundlegende Animationsmöglichkeiten identifiziert, die von den untersuchten Plattformen unterstützt werden. Plattformspezifische Funktionalitäten, die in SSIML/Behaviour nicht abbildbar sind, können durch Erweiterung des aus einem Behaviour-Modell generierten Programmcodes realisiert werden.

**Hierarchie von Animationszustandstypen im Metamodell** Der Basistyp aller Animationszustände ist `AnimationState`. Dieser stellt eine direkte Erweiterung der UML 2-`State`-Metaklasse dar.

In einem Zustand für prozedurale Animationen (`ProceduralAnimationState`) ist der Attributwert, der in einem solchen Zustand verändert wird, eine Funktion der Zeit. Jedoch wird die konkrete Animationsfunkton nicht im Modell beschrieben sondern muss später im Code implementiert werden, der aus dem Modell generiert wurde. Dieser Animationstyp kann für sehr spezielle prozedurale Animationen verwendet werden.

Ein `KFAnimationState` ist ein Zustand, in dem ein graphisches Objekt durch eine Keyframe-Animation animiert wird. Die Eigenschaft `keyTimes` enthält eine Liste von Zeitwerten. Für jeden `keyTimes`-Wert muss ein entsprechender Schlüsselwert für die Animation definiert werden (siehe weiter unten in diesem Abschnitt). Der Animationstimer startet von Null, nachdem ein `KFAnimationState`-Zustand betreten wurde. Alle `keyTimes`-Werte müssen relativ zum Startzeitpunkt definiert werden. Anstatt der Deklaration einer Liste von `keyTimes`-Werten ist es auch möglich, eine Zeitdauer (`duration`) zu definieren. Die Zeitdauer gibt die Zeitspanne an, in der die Animation abläuft. Die `keyTimes`-Werte müssen dann aus der angegebenen Zeitdauer berechnet werden. Dafür wird die Zeitdauer in $n$ Intervalle gleicher Länge unterteilt, wobei $n$ die *Anzahl von Schlüsselwerten vermindert um Eins* ist. Die Grenzen der einzelnen Intervalle repräsentieren dann die `keyTimes`-Werte. Die Anzahl der Schlüsselwerte kann durch das `KFAnimationState`-Attribut `numberOfKeys` festgelegt werden. Der Wert der Eigenschaft `replays` definiert die Zahl der Wiederholungen der Animation. Ist `replays` der Wert *-1* zugeordnet, so wird die Animation solange wiederholt, bis sie abgebrochen wird, z. B. durch Auslösung eines Zustandsüberganges. Standardgemäß hat `replays` zunächst den Wert *0*. Die Eigenschaft `interpolationMethod` beschreibt die Methode für die Interpolation der Schlüsselwerte. Vordefinierte Werte sind hier `DISCRETE` (diskret), `LINEAR` (Standard), `CUBIC_SPLINE` (kubisch) oder `UNDEFINED` (undefiniert). Ist der Wert von `interpolationMethod` `UNDEFINED`, kann der Programmcode für die Interpolationsroutine nicht automatisch generiert, sondern muss später manuell hinzugefügt werden. Dies kann dann sinnvoll sein, wenn eine besondere Interpolationsmethode angewendet werden soll.

Der Typ der Schlüsselwerte, die interpoliert werden, hängt davon ab, welcher Subtyp von `KFAnimationState` verwendet wird. Beim Typ `KFTransformAnimationState` können Schlüsselwerte für Translationen (`translations`), Rotationen um eine bestimmte Achse (`rotationsX`, `rotationsY`, `rotationsZ`) oder beliebige Achse (`rotations`) und Skalierungen (`uniformScales`, `nonUniformScales`) definiert werden. `translations` enthält eine Liste von Verschiebungsvektoren, `rotations` eine Liste von Orientierungswerten im Achse/Winkel-Format. Bei Rotation um die x-, y- oder z-Achse mittels `rotationsX`, `rotationsY` bzw. `rotationsZ` kann auf die Angabe der Rotationsachse innerhalb der Schlüsselwerte verzichtet werden. `uniformScales` beinhaltet eine Liste mit Skalierungsfaktoren. Bei `nonUniformScales` kommt vor jedem Skalierungsfaktor noch eine Skalierungsachse (3D-Vektor) hinzu.

Ein Beispiel für eine Behaviour-Engine mit Keyframe-Animationen ist die Behaviour-Engine, die das Verhalten der Motorhaube aus dem Fahrzeugbeispiel beschreibt (Abbildung 7.6). Zu Beginn ist die Motorhaube geschlossen (Zustand `HoodIsClosed`). Empfängt die Behaviour-Engine eine Nachricht *„hoodTS clicked“* vom `TouchSensor`, der mit dem `frontHood`-Objekt verbunden ist, wird die Animation `OpenHood` gestartet. Sobald die Animation beendet ist, wird die Transition zum Zustand `HoodIsOpen` durchlaufen (*CompletionEvent*). Die Transition zu `CloseHood` wird durchlaufen, wenn ein weiteres *„hoodTS clicked“*-Ereignis eintritt. Im `CloseHood`-Zustand wird die Haube wieder durch eine Keyframe-Animation geschlossen. Nachdem das Schließen der Haube beendet ist, geht die *Behaviour-Engine* erneut in den `HoodIsClosed`-Zustand über.

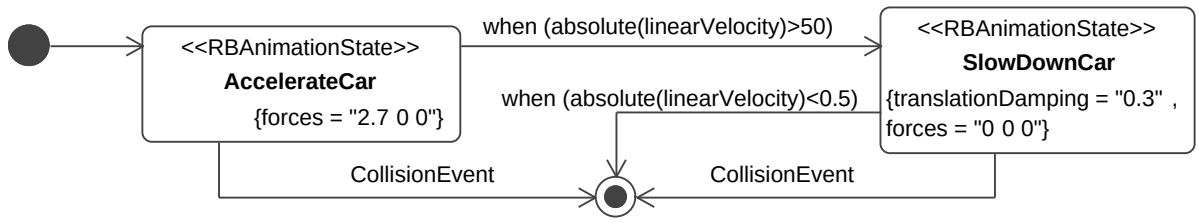



Abbildung 7.9: Beispiel einer Behaviour-Engine mit `RBAnimationState`-Zuständen

Ein `KFMaterialAnimationState` erlaubt die Animation von Materialparametern wie Farbe (RBG-Format) oder Transparenz. Die entsprechenden Attribute für die Schlüsselwerte eines `KFMaterialAnimationState` tragen die Namen `fillColours`, `diffuseColours`, `specularColours` und `transparencies`.

Ein `KFCameraAnimationState` kann hingegen benutzt werden, um über die Schlüsselwertattribute `orientations` und `positions` den Blickpunkt des Benutzers (d.h. Position und Orientierung der virtuellen Kamera, durch welche der Benutzer die Szene sieht) zu beeinflussen. Damit können z. B. Kamerafahrten simuliert werden. Ebenso ist eine Manipulation der *Field-Of-View*-Werte der `Camera` möglich.

In einem Zustand vom Typ `RBAnimationState` wird die Bewegung eines starren Körpers, repräsentiert durch ein SSIML-`Object` mit einem `RBDynamics`-Attribut, simuliert. Ein `RBAnimationState` verfügt über die gleichen Attribute wie ein `RBDynamicsValue`. Die initialen Werte der Festkörpereigenschaften, wie die Masse (`mass`) eines Objekts können über die Feldwerte des `RBDynamics`-Attributes, über entsprechende Attribute eines `RBAnimationState`-Zustandes oder später bei der Implementierung manuell im generierten Code festgelegt werden. Ein im Zustand festgelegter Wert (z.B. `mass = 10`) überschreibt dabei den ggf. im Knotenattribut festgelegten Wert. Die Bezeichnungen der Attribute eines `RBAnimationState`-Zustandes entsprechen denen der `RBDynamics`-Attribute. Wird ein `RBAnimationState`-Zustand über einen *ChangeTrigger* verlassen, kann innerhalb des *ChangeTriggers* ein Attribut des Animationszustandes abgefragt werden, z.B. *when(linearVelocity=Sequence{34,0,0})*. Die *ChangeTrigger*-Ausdrücke werden in OCL [Obj06b]-Syntax angegeben.

Ein Beispiel für eine physikbasierte Animation mit *ChangeTriggern* ist in Abbildung 7.9 zu sehen. Die Behaviour-Engine beschreibt das Verhalten eines Fahrzeuges. Im `AccelerateCar`-Zustand ist die Beschleunigung des Fahrzeuges konstant. Als Resultat steigt die Geschwindigkeit des Fahrzeuges. Sobald die Geschwindigkeit den Wert 50 übersteigt, wird die Beschleunigung auf Null gesetzt. Im Zustand `SlowDownCar` verlangsamt dann das Auto die Geschwindigkeit aufgrund des Dämpfungswertes von 0.3. Wenn die Geschwindigkeit kleiner als 0.5 ist, hält das Fahrzeug und die Abarbeitung des Zustandsautomaten wird beendet. Kollidiert das Fahrzeug mit einem anderen Objekt, wird ebenfalls der Endzustand erreicht.

Folgt auf einen physikbasierten Animationszustand ein keyframebasierter Animationszustand, ist zur Laufzeit zu beachten, dass die aktuellen physikalischen Parameter (Masse, aktuelle Geschwindigkeit etc.) in den Feldern einer `RBDynamics`-Attributinstanz gespeichert

bleiben, solange die Keyframe-Animation ausgeführt wird. Folgt auf einen `KFAnimationState` erneut ein `RBAnimationState`, wird das Objekt auf Basis der gespeicherten Werte animiert, wenn innerhalb des `RBAnimationState`s keine anderen Startparameter spezifiziert wurden. Die Anfangsposition und -orientierung eines physikalisch zu animierenden Objektes wird zur Laufzeit durch die aktuell in seiner Transformation gespeicherten Werte bestimmt. Umgekehrt muss die Transformation (Position und Orientierung) eines Objektes während der Physikanimation mit den von der Physikengine ausgegebenen Daten synchronisiert werden.

#### 7.2.4.4 Sequenzielle und synchrone Animationsausführung

Die Modellierung sequenzieller Animationen und die Einstellung der Startzeitpunkte von Animationen ist mit SSIML/Behaviour relativ einfach. Verzögerungen werden mittels einfacher Zustände ohne Aktivitäten modelliert, die über zeitgetriggerte Transitionen verlassen werden. Animationsabfolgen sind durch die Verknüpfung von mehreren Animationszuständen über ungetriggerte Transitionen umsetzbar.

Allerdings kann es manchmal sinnvoll sein, nebenläufige Animationen zu modellieren. Da definitionsgemäß ein Objekt zu jedem beliebigen Zeitpunkt einen festgelegten Zustand im Zustandsautomaten haben muss, ist es nicht möglich, dass zwei oder mehrere „einfache“ Zustände zur selben Zeit aktiv sind. Jedoch kann ein zusammengesetzter Zustand in Regionen eingeteilt werden, um parallele Abläufe zu definieren. Ein zusammengesetzter Zustand mit Regionen wird als orthogonaler zusammengesetzter Zustand bezeichnet. Ein Beispiel ist in Abbildung 7.10 zu sehen, die das Modell eines Zylinderkolbens illustriert, wie er in Dampfmaschinen oder Fahrzeugmotoren vorkommen kann. Das Objekt in Abbildung 7.10(b) besteht aus drei Teilen, die sich synchron bewegen müssen. Während *part1* und *part2* Zylinder sind, ist *part3* eine rotierende Scheibe. *part1*, *part2* und *part3* bilden eine Transformationshierarchie (Abbildung 7.10(a)). Die Winkel $\alpha$ und $\beta$ in Abbildung 7.10(b) sind Funktionen der Zeit. Um die korrekten Transformationswerte für die zu animierenden Objekte zu einem gegebenen Zeitpunkt zu erhalten, müssen die drei Animationszustände *Part1Animation*, *Part2Animation* und *Part3Animation* gleichzeitig aktiv sein und zur gleichen Zeit betreten werden. Dies wird durch die Verwendung von Regionen erreicht (Abbildung 7.10(c)).

### 7.2.5 Verknüpfung von Animationszuständen und Attributen

Welche Attribute von Objekten, oder anders ausgedrückt, welche Felder von Attributinstanzen von SSIML-Knoten in einem Animationszustand verändert werden (und ggf. zur Feststellung von Initialwerten gelesen werden), hängt von den Typen des Animationszustandes und der Zielattribute ab. Tabelle 7.2 fasst zusammen, welche Attributtypen mit welchen Animationszustandstypen kombinierbar sind.

Wird keine Einschränkung der Zielattributinstanzen getroffen, werden in einem Animationszustand die Inhalte aller Instanzen von Attributen verändert, auf welche die folgenden Aussagen zutreffen:

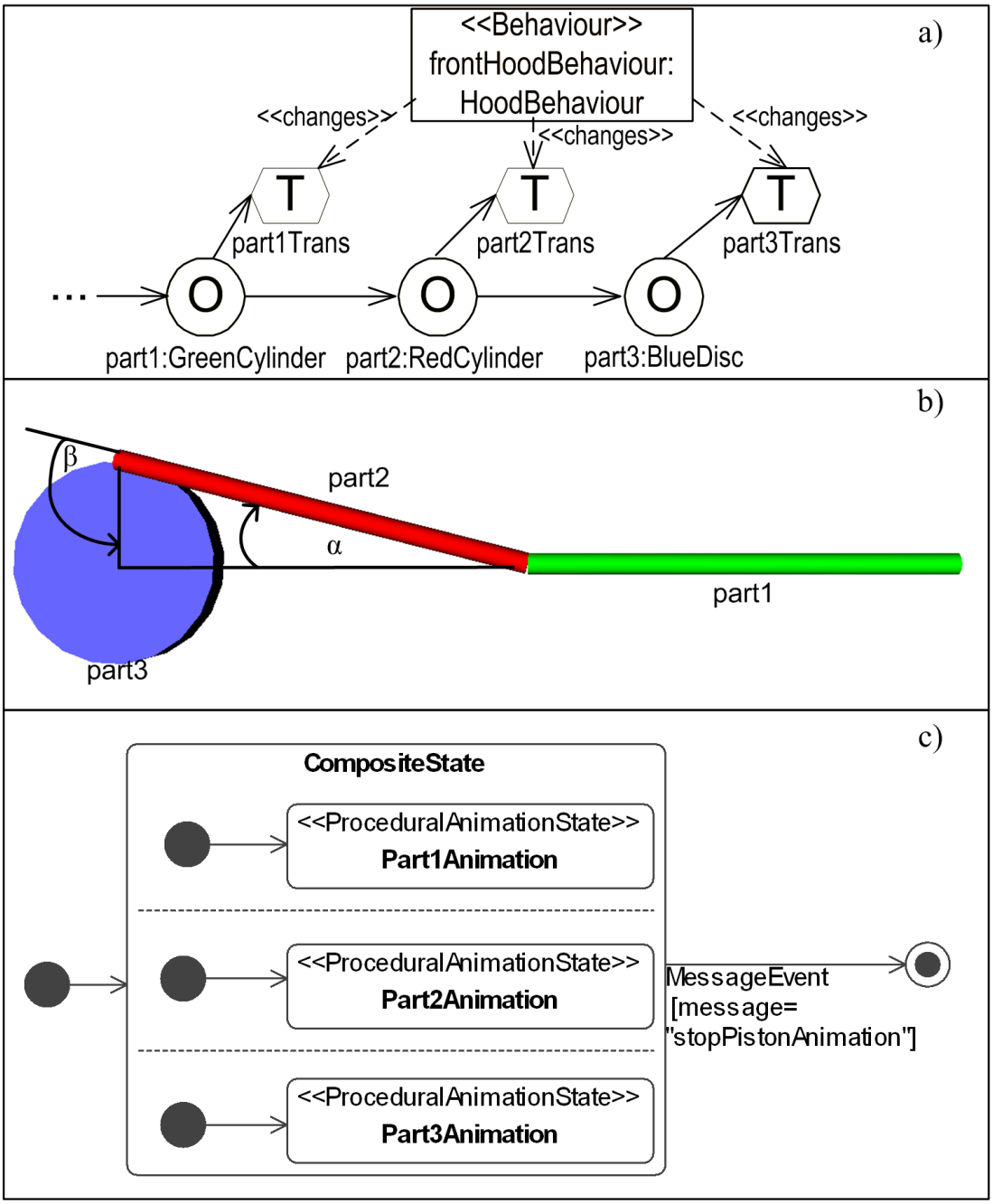



Abbildung 7.10: Beispiel für parallele Animationsabläufe; (a) Interrelationenmodell, (b) 3D-Visualisierung, (c) Behaviour-Engine

Tabelle 7.2: Mögliche Kombinationen von Animationszustandstypen und Attributtypen (+: Kombination möglich; -: Kombination nicht möglich)

| Attribute-Subtyp / AnimationState-Subtyp | Material | Camera | Transformation | RBDynamics |
|---|---|---|---|---|
| ProceduralAnimationState | + | + | + | - |
| KFTransformAnimationState | - | - | + | |
| KFCameraAnimationState | - | + | - | - |
| KFMaterialAnimationState | + | - | - | - |
| RBAnimationState | - | - | - | + |

- Der Typ des Attributes ist mit dem Typ des aktuellen Animationszustandes kombinierbar.
- Das Attribut ist über eine `BehaviourRelationship`-Beziehung mit dem `Behaviour`-Element, dessen Behaviourdefinition den aktuellen Animationszustand enthält, verbunden.

Um die Menge der Attributinstanzen einzuschränken, die in einem Animationszustand verändert werden, kann an der `BehaviourRelationship`-Beziehung zwischen dem `Behaviour`-Element und dem Zielattribut angetragen werden, von welchen Animationszuständen das Attribut verändert wird (Attribut `sourceStates` von `BehaviourRelationship`) und welche Attributinstanzen verändert werden (Attribut `targetInstanceValues` von `BehaviourRelationship`).

### 7.2.6 Notation

Die Tabellen 7.3 und 7.4 fassen die Notationen von SSIML/Behaviour-Elementen noch einmal zusammen. Tabelle 7.4 zeigt zudem ein Beispiel einer `BehaviourRelationship`-Beziehung, die nur für bestimmte Animationszustände und für bestimmte Attributinstanzen eine Änderung der Feldwertinhalte zulässt.

## 7.3 Modell-Code-Abbildung

Eine wichtige Anforderung ist, dass sich Verhaltensdefinitionen aus den SSIML-Modellen auf Codeebene in beliebigen Szenen wieder verwenden lassen. Daher ist es sinnvoll, Verhaltensdefinitionen in Komponenten zu kapseln. Bspw. kann eine Komponente ein Verhalten kapseln, das das Öffnen und Schließen einer Tür beschreibt. Diese Komponente kann dann in verschiedenen 3D-Szenen, in denen Türobjekte vorkommen, die das gleiche Verhalten aufweisen, wieder verwendet werden. Weiterhin ist es wünschenswert, dass ein 3D-Designer eine solche Verhaltenskomponente in ein 3D-Autorenwerkzeug laden und dort mit 3D-Objekten assoziieren kann.

Tabelle 7.3: Notation der SSIML/Behaviour-Elemente `Behaviour` und `AnimationState`

| Elementtyp | Notation | Beispiel |
|---|---|---|
| Behaviour | '<<Behaviour>>' <Verhaltensname>':' <Verhaltensdefintionsname> | <<Behaviour>> frontHoodBehaviour: HoodBehaviour |
| AnimationState | '<<'<Animationszustandstyp>'>>' <Animationszustandsname> ('{'<Attribut-Wert-Zuweisungen>'}')? | <<RBAnimationState>> AccelerateObject {forces = 2 0 0} |

Tabelle 7.4: Notation der SSIML/Behaviour-Elemente `BehaviourRelationship` und `BehaviourActionRelationship`

| Beziehungstyp | Startelementtyp | Zielelementtyp | Ende 1 (Start) | Mittelteil | Ende 2 (Ziel) | Beispiel |
|---|---|---|---|---|---|---|
| BehaviourRelationship | Behaviour | BehaviourTarget (ModifiableAttribute) | (<Animationszustände>)? | '<<changes>>' (<Zielelement-instanzwerte>)? | --> | ani1,ani3 <<changes>> [1,2,3].[0..2] |
| BehaviourAction-Relationship | ActionSource (Class) | ActionTarget (Behaviour, ComposedNode) | | '<<controls>>' (<Pfad zum Ziel-Verhalten>)? | --> | <<controls>> targetBehaviour |

Um derartige Verhaltenskomponenten zu realisieren, bieten sich verschiedene Wege an. Eine Möglichkeit ist, eine Verhaltensdefinition in Programmcode (z.B. Java) zu übersetzen. Für diese Variante wäre eine Plattform ähnlich der 3D-Beans-Plattform [DG00] als Zielplattform denkbar.

Mit Hinblick auf die Verwendbarkeit in Autorenwerkzeugen, wie SwirlX3D [Swi] oder Media Machines Studio [Med], wurde allerdings ein anderer Weg beschritten, indem die Verhaltensdefinitionen auf das deklarative Format X3D abgebildet wurde. Ein weiterer Vorteil von X3D ist, dass eine größere Zahl an Interpolatoren (z. B. `PositionInterpolator`, `OrientationInterpolator`) zur Realisierung von Animationen bereits zur Verfügung steht. Zudem ist in X3D über die Rigid-Body-Physics-Komponente [Int08b] eine Unterstützung für physikbasierte Animationen integriert.

Eine Kapselung von Verhalten wird durch X3D-Prototypen [Int08b] möglich. Ein X3D-Prototyp kann in einer separaten Datei definiert werden und über bestimmte X3D-Anweisungen (EXTERNPROTO-Statements) in eine X3D-Szene eingebunden werden (vgl. a. Abschnitt 5.6.1.5). Durch Instanziierung des Prototyps kann eine Verhaltensdefiniti-

on für mehrere 3D-Objekte wiederverwendet werden, wobei jedes 3D-Objekt seinen eigenen Zustand speichert. Die Prototypinstanzen können mit verschiedenen X3D-Knoten und -Feldern assoziiert werden, z.B. durch die Verbindung der Prototypinstanzfelder mit Knoten oder Knotenfeldern über `USE`- bzw. `ROUTE`-Anweisungen. Informationen darüber, welche Felder im Initialzustand der Szene verbunden sind, werden aus dem SSIML-Modell extrahiert.

Von einer externen Anwendung können die Instanzen der Prototypen über das Scene Access Interface (SAI) ähnlich wie normale X3D-Knoten adressiert werden. Dort, wo die von X3D zur Verfügung gestellten Funktionalitäten nicht ausreichen, können Skriptknoten eingebunden werden, die mittels Ecmascript [Int05c] oder Java [Int05d] implementiert werden. Aufgrund der Mächtigkeit von Java, der einfachen Realisierbarkeit einer weiteren Modularisierung der Verhaltensbeschreibung durch Klassen und der Möglichkeit, eine Vielzahl von Java-Bibliotheken zu nutzen, wurde im vorliegenden Projekt Java zur Implementierung von Skriptknoten gewählt. Als Basis diente Xj3D [Xj3], eine Umsetzung der X3D-Java-Sprachbindungsspezifikation [Int05d].

### 7.3.1 Abbildung eines `Behaviour`-Elements

Der folgende Beispielquellcode zeigt die Verknüpfung einer Prototypinstanz `frontHoodBehaviour` mit einem `TouchSensor` für die `TouchSensorEventRelationship`-Beziehung des Typs `CLICKED` aus Abbildung 7.4. Der Berührungssensor und die Prototypinstanz sind über `ROUTE`-Anweisungen verbunden. Um den Prototyp von einer konkreten Szene zu entkoppeln, bietet er eine generische Schnittstelle für das Empfangen von Nachrichten an (Feld `message`), anstatt einer Schnittstelle, die speziell Nachrichten von `hoodTS` empfängt. Eine *Click*-Ereignisnachricht eines X3D-`TouchSensor`s muss daher erst in eine Zeichenkette konvertiert werden. Für jede Beziehung zwischen einem Sensor und einem Verhalten im SSIML-Szenenmodell wird daher ein X3D-Skript generiert, das entsprechende Sensornachrichten als Zeichenkette umcodiert. Im vorliegenden Beispiel übernimmt diese Aufgabe der Skriptknoten `hoodTSClickedMessageGenerator`. Der Konvertierungscode für die Sensornachricht ist in einer externen Java-Klasse enthalten.

```
...
<ExternProtoDeclare
        name="HoodBehaviour"
        url='"HoodBehaviourPrototype.x3d#HoodBehaviour"'>
        <field name="message" accessType="inputOnly" type="SFString"/>
        ...
</ExternProtoDeclare>
...
<TouchSensor DEF="hoodTS"/>
...
<Script DEF="hoodTSClickedMessageGenerator"
        url="&quot;HoodTSClickedMessageGenerator.class&quot;">
        <field accessType="inputOnly" name="clicked" type="SFTime"/>
        <field accessType="outputOnly" name="message_changed"
         type="SFString"/>
</Script>
...
<ProtoInstance name="HoodBehaviour" DEF="frontHoodBehaviour"/>
```

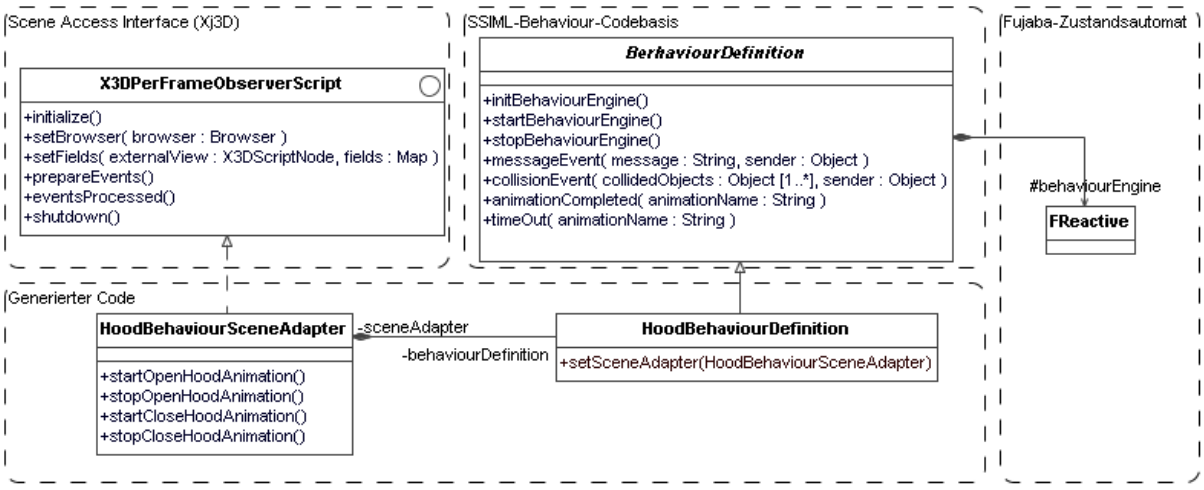



Abbildung 7.11: Abbildung der Verhaltensdefinition `HoodBehaviour` auf Code – Basisklassen, Schnittstellen und generierte Klassen (vereinfachte Darstellung)

```
...
<ROUTE fromNode="hoodTS" fromField="touchTime"
       toNode="hoodTSClickedMessageGenerator" toField="clicked"/>
<ROUTE fromNode="hoodTSClickedMessageGenerator"
       fromField="message_changed" toNode="frontHoodBehaviour"
       toField="message"/>
...
```

## 7.3.2 Abbildung einer Verhaltensdefinition

Eine auf einen X3D-Prototyp abgebildete Verhaltensdefinition enthält einen Skriptknoten mit der Java-Implementierung des Zustandsautomaten, der die Steuerung des Ablaufes der Einzelanimationen realisiert. Der Zustandsautomat wird in einer separaten Klasse gekapselt, die die Klasse `BehaviourDefinition` erweitert (Abbildung 7.11). Im Beispiel aus Abbildung 7.6 ist dies die Klasse `HoodBehaviourDefinition`.

Zudem existiert eine Hilfsklasse, die den Skriptknoten mit dem `BehaviourDefinition`-Objekt verbindet, wie im obigen Beispiel die Klasse `HoodBehaviourSceneAdapter`. Die Hilfsklasse implementiert die Schnittstelle `X3DPerFrameObserverScript`, welche Bestandteil des Java-SAIs ist.

### 7.3.2.1 Abbildung des Zustandsautomaten

Zur Abbildung von Zustandsautomaten auf Code gibt es verschiedene Möglichkeiten, wie die Realisierung durch `Switch`-Statements, die Anwendung des *State Patterns* [GHJV94] oder die Benutzung objektorientierter Zustandstabellen (vgl. [Köh00, Zün02]). Die Generierung von Code aus UML-Zustandsautomaten wurde bereits in vielen Forschungsarbeiten diskutiert (z. B. [NT05, Köh00, Zün02]) und ist Gegenstand der Executable UML-Methode (xUML [MB02]). Auch im industriellen Umfeld existieren Werkzeuge, die Zustandsautoma-

ten auf Code abbilden. Diese werden insbesondere bei der Spezifikation von eingebetteten Systemen und Realzeitanwendungen eingesetzt (z.B. STATEMATE [HLN+88, STA], Rational Rose Technical Developer [Ros]). Auf die Abbildung von UML-Zustandsautomaten auf Code wird daher nicht im Detail eingegangen; hingegen stehen animationsrelevante Aspekte im Vordergrund.

Bei der Entwicklung der Codeabbildung von SSIML-Behaviour-Engines wurde beispielhaft die Zustandsautomaten-Java-Bibliothek der Entwicklungsumgebung Fujaba verwendet, die in den Arbeiten von Köhler [Köh00] und Zündorf [Zün02] genauer dokumentiert ist. Die Laufzeitrepräsentation einer Behaviour-Engine wird von einem `BehaviourDefinition`-Objekt referenziert (Abbildung 7.11). Dieses verfügt über eine `initBehaviourEngine`-Methode, in welcher der Zustandsautomat initialisiert wird, indem die notwendigen Zustände und Transitionen erzeugt und verbunden werden.

Weiterhin besitzt eine `BehaviourDefinition` eine `startBehaviourEngine`- und eine `stopBehaviourEngine`-Methode zum Ausführen bzw. Abbruch der Ausführung des Zustandsautomaten. Ein Auslöser (*Trigger*) für Zustandsübergänge wird in der Schnittstelle einer konkreten `BehaviourDefinition` als Methode mit entsprechenden Parametern abgebildet. Für *MessageEvents* enthält die Behaviour-Engine bspw. eine Methode zum Empfangen von Nachrichten (Methode `messageEvent`), wobei der erste Parameter der Methode die entsprechende Nachricht ist und als zweiter Parameter eine Referenz auf das Objekt übergeben werden kann, das der „Verursacher“ des Ereignisses ist. Triggerlose Zustandsübergänge (*CompletionTrigger*) werden auf eine spezielle Methode `animationCompleted(animationStateName)` abgebildet, die nach Ablauf einer Animation aufgerufen wird. Zeitabhängige Zustandsübergänge werden über die Methode `timeOut(animationStateName)` realisiert. Für jeden *ChangeTrigger* der `BehaviourDefinition` wird eine separate Methode erzeugt, die beim Gültigwerden des booleschen Ausdrucks im *ChangeTrigger* aufgerufen wird. Ebenso wird für jeden *Guard* eine Methode erzeugt, die einen Wahrheitswert zurückgibt und die Variablen des im SSIML-Modell spezifizierten *Guards* als Parameter akzeptiert (*ChangeTrigger-*/*Guard*-bezogene Methoden nicht dargestellt in Abbildung 7.11). Nachfolgend ist der für die Transition vom Zustand `HoodIsClosed` zum Zustand `OpenHood` aus dem *Hood*-Behaviour-Engine-Beispiel generierte *Guard* zu sehen.

```
public boolean guardForMessageEventFromHoodIsClosedToOpenHood(String message) {
  return (message.equals("hoodTS clicked"));
}
```

Wird eine der *Trigger*-Methoden aufgerufen, wird, falls der *Guard* gilt, ein Zustandsübergang ausgelöst. Außerdem enthält eine `BehaviourDefinition` für jede im Modell definierte *entry*-Aktion, *do*-Aktivität oder *exit*-Aktion eines Zustandes Methoden, die beim bzw. nach dem Eintritt in bzw. Austritt aus dem Zustand aufgerufen werden. Bei einem Animationszustand wird bei Aufruf der *do*-Methode die entsprechende Animation gestartet. Der Aufruf zum Starten einer Animation wird an die Adapterklasse delegiert, die das `X3DPerFrameObserverScript` implementiert.

```
public void doActionForOpenHood() {
  sceneAdapter.startOpenHoodAnimation();
}
```

Wird ein Zustandsübergang ausgelöst, bevor eine Animation beendet ist, muss die Animation abgebrochen werden. Die erste Anweisung einer *exit*-Operation eines Animationszustandes muss daher die Animation anhalten, die in diesem Zustand ausgeführt wird.

#### 7.3.2.2 Einbettung der Verhaltensbeschreibung in X3D-Prototypen

Nachfolgend soll nun auf die Realisierung der X3D-Prototypen für Verhaltensdefinitionen etwas genauer eingegangen werden. Die Prototypschnittstelle enthält Felder, um die Ausführung des Verhaltens zu starten (das Boolean-Feld `engineStarted` empfängt den Wert `true` bei *nicht gestarteter* Behaviour-Engine) oder zu stoppen (das Boolean-Feld `engineStarted` empfängt den Wert `false` bei *gestarteter* Behaviour-Engine). Ein Stoppen bricht die Ausführung der Behaviour-Engine komplett ab. Nach einem Neustart wird die Engine reinitialisiert und die Abarbeitung des Zustandsautomaten beginnt von vorn.

Wie in Abschnitt 7.3.2.1 bereits erwähnt, können Nachrichten in Form von Zeichenketten (Feld `messageEvent`) an einen Prototyp, der eine Verhaltensbeschreibung kapselt, übermittelt werden. Ebenso enthält die Prototypschnittstelle ein *Boolean*-Feld für *CollisionEvents*. Ein *CollisionEvent* trat auf, wenn das `SFBool`-Feld `collisionEvent` den Wert `true` empfängt.

Zusätzliche Informationen, wie eine Referenz auf den Verursacher eines *MessageEvents* oder die Menge der an einer Kollision beteiligten Objekte können als Benutzerdaten mit einem Feld einer Prototypinstanz assoziiert werden. Die X3D-Java-Sprachanbindung sieht dafür die Methoden `setUserData(Object)` und `getUserData` vor.

Nachfolgender Code zeigt einen Ausschnitt der Prototypschnittstelle für die Verhaltensdefinition `HoodBehaviour` aus Abbildung 7.9.

```
<ProtoDeclare name="HoodBehaviour">
  <ProtoInterface>
    <field name="message" accessType="inputOnly" type="SFString"/>
    <field name="engineStarted" accessType="inputOnly" type="SFBool"/>
    <field name="collisionEvent" accessType="inputOnly" type="SFBool"/>
    ...
    </ProtoInterface>
   <ProtoBody>...</ProtoBody>
</ProtoDeclare>
```

#### 7.3.2.3 Abbildung von KFAnimationStates – Realisierung von keyframebasierten Animationen

Als Beispiel für eine Abbildung einer Verhaltensdefinition, die `KFAnimationState`s enthält, soll wiederum die Verhaltensdefinition `HoodBehaviour` dienen. Neben den zuvor erwähnten Feldern wird jede modellierte Eigenschaft eines Animationszustandes einer Behaviour-Engine auf ein Prototypfeld abgebildet, das die aktuellen Animationswerte ausgibt.

```
<ProtoDeclare name="HoodBehaviour">
  <ProtoInterface>
   ...
   <field name="openHoodRotation" accessType="outputOnly" type="SFRotation"/>
   <field name="closeHoodRotation" accessType="outputOnly" type="SFRotation"/>
  </ProtoInterface>
  <ProtoBody>...</ProtoBody>
</ProtoDeclare>
```

Im Rumpf (*Body*) des Prototyps wird zunächst ein X3D-Zeitgeber (*TimeSensor*) `TimeBase` erzeugt, der die zeitliche Basis für alle Animationen der Behaviour-Engine bereitstellt. Des Weiteren wird für jeden Animationszustand ein `TimeSensor` generiert, der nach dem Eintritt in den Animationszustand (Startzeit des `TimeSensor`s) aktiviert wird. Der `TimeSensor` enthält aus dem Modell generierte Informationen über die Dauer eines Animationszyklus' (Feld `cycleInterval`) und darüber, ob die Animation in einer Endlosschleife laufen soll (Feld `loop` ist `true`). Außerdem wird für jedes modellierte Eigenschaftsfeld eines Animationszustandes (z.B. `rotation`, `translation`) ein entsprechender Interpolator erzeugt, z. B. für den Zustand `OpenHood` aus dem `HoodBehaviour`-Beispiel ein `OrientationInterpolator OpenHoodRotationInterpolator`. `TimeSensor`-Objekte und die dazugehörigen Interpolatoren werden durch `ROUTE`-Statements verbunden. Die Ausgabefelder der Interpolatoren (`value_changed`) werden mit den jeweiligen Ausgabefeldern der Prototypschnittstelle verbunden.

```
<ProtoBody>
  <Group>
    <!-- Timebase is always running-->
    <TimeSensor DEF="TimeBase" loop="true"/>

    <!-- TimeSensor to control the OpenHood-Animation -->
    <TimeSensor DEF="OpenHoodAnimationTimeSensor" enabled="false"
      cycleInterval="1"  loop="false"/>

    <!-- OrientationInterpolator for OpenHood-rotationsZ -->
    <OrientationInterpolator DEF="OpenHoodRotationInterpolator"
      key="0.0 0.5 1.0" keyValue="0 0 1 0.0 0 0 1 0.44 0 0 1 0.87">
      <!-- connect output to output field of Prototype -->
      <IS>
        <connect nodeField="value_changed" protoField="openHoodRotation"/>
      </IS>
    </OrientationInterpolator>

    ...
    <!--connect TimeSensors and Interpolators-->
    <ROUTE fromNode="OpenHoodAnimationTimeSensor" fromField="fraction_changed"
      toNode="OpenHoodRotationInterpolator" toField="set_fraction"/>
    ...
  <Group>
</ProtoBody>
```

Die eigentliche Verhaltensbeschreibung wird in einem Skriptknoten gekapselt. Dieser besitzt direkten Zugriff auf alle Animationszeitgeber (`TimeSensor`s), um die jeweiligen Animationen zu starten oder zu stoppen, sowie direkten Zugriff auf die Zeitbasis, um die aktuelle Zeit auszulesen, etwa um die Startzeit einer Animation zu bestimmen. Basierend auf der Startzeit einer Animation kann auch die Stoppzeit derselben in Abhängigkeit von

Animationsdauer und Anzahl der Wiederholungen errechnet werden. Die erhaltenen Werte können beim entsprechenden Animationszeitgeber über die Felder `startTime` und `stopTime` gesetzt werden.

Von besonderem Interesse ist zudem das Feld `isActive` eines Animationszeitgebers. Bei Aktivierung des `TimeSensors` (Animationsbeginn zur Startzeit) wechselt das Feld den Wert von `false` nach `true` und es wird ein entsprechendes Ereignis generiert. Bei Deaktivierung des `TimeSensors` (Wechsel des Feldwertes von `isActive` von `true` nach `false`; Animationsende, z.B. wenn Stoppzeit erreicht wurde) wird ebenfalls eine Ereignisnachricht erzeugt. Die Ereignisnachrichten werden vom Skriptknoten empfangen und können dort ausgewertet werden. Wichtig ist vor allem das Animationsende, da dieses einen Zustandsübergang durch einen *CompletionTrigger* bewirken kann.

Weitere Felder des Skriptknotens entsprechen – abgesehen vom Präfix `sc_` – gleichnamigen Feldern der Prototypschnittstelle und sind mit diesen verbunden, wie untenstehendes Beispiel verdeutlicht.

```
<ProtoBody>
  <Group>
  ...
    <Script DEF="SC" url="&quot;HoodBehaviourSceneAdapter.class&quot;"
      directOutput="true" mustEvaluate="true">

      <!-- fields which reference other nodes -->
      <field name="sc_TimeBase"
        accessType="initializeOnly" type="SFNode">
        <TimeSensor USE="TimeBase"/>
      </field>

      <field name="sc_OpenHoodAnimationTimeSensor"
        accessType="initializeOnly" type="SFNode">
        <TimeSensor USE="OpenHoodAnimationTimeSensor"/>
      </field>
      ...
      <!--map Script fields to Prototype fields if necessary-->
      <field  accessType="inputOnly" name="sc_message" type="SFString"/>
      <IS>
        <connect nodeField="sc_message" protoField="message"/>
      </IS>
      ...
    </Script>
    ...
  <Group>
</ProtoBody>
```

Die Java-Adapterklasse, welche den Skriptknoten mit dem Java-Zustandsautomaten verbindet, wird über das `url`-Feld des Skriptknotens referenziert. Die Adapterklasse implementiert die Schnittstelle `X3DPerFrameObserverScript`. Ein `X3DPerFrameObserverScript` kann eine Referenz auf den X3D-Browser halten, der die Szene lädt, initialisiert und die Laufzeitumgebung für die Szene bereitstellt. Damit kann ein `X3DPerFrameObserverScript` direkt mit dem Browser kommunizieren. Zusätzlich erhält ein `X3DPerFrameObserverScript` Zugriff auf alle Felder des Skriptknotens in Form einer Java-`Map`, die jedem Feldnamen das entsprechende Feld zuordnet. Ein `X3DPerFrameObserverScript` besitzt eine Initialisierungsmethode, die während der Initialisierungsphase der X3D-Szene aufgerufen wird, also bevor der X3D-Browser die

Szene anzeigt und X3D-Ereignisse (X3D-Events) verarbeitet werden. In dieser `initialize`-Methode wird die Instanz der `BehaviourDefinition` erzeugt und ihre Methode `initBehaviourEngine` wird aufgerufen. Zudem werden alle Felder aus der dem Skript übergebenen `Map` gelesen und zum einfachen Zugriff in Instanzvariablen abgelegt, z. B. das Feld, welches die aktuelle Zeit der Zeitbasis enthält. Zusätzlich werden allen Eingabefeldern `X3DFieldEventListener` hinzugefügt, um auf entsprechende Ereignisse adäquat reagieren zu können. `X3DFieldEventListener` rufen die *Trigger*-Methoden der `BehaviourDefinition` auf und lösen somit Zustandsübergänge aus. Zeitabhängige Zustandsübergänge (*TimeTrigger*) können über Java-`Timer` (`java.util.Timer` -Klasse) realisiert werden, die beim Starten einer Animation mit dem im Modell angegebenen Zeitwert initialisiert werden und bei Animationsablauf die aktuelle Animation anhalten und die `timeOut`-Methode der `BehaviourDefinition` aufrufen.

Der nachstehende Code zeigt einen Ausschnitt der Implementierung der `initialize`-Methode der Klasse `HoodBehaviourSceneAdapter`. Der Code demonstriert anhand des Feldes `messageField`, wie auf die Felder des Skriptknotens zugegriffen wird. Das Feld `messageField` ist für den Empfang von externen Nachrichten verantwortlich.

```
public class HoodBehaviourSceneAdapter
        implements X3DPerFrameObserverScript {
...
private BehaviourDefinition behaviourDefinition;
private Map fields;
private SFString messageField;
...
public void setFields(X3DScriptNode externalView, Map fields) {
  this.fields = fields;
}
public void initialize() {
  behaviourDefintion = new BehaviourDefinition();
  behaviourDefinition.setSceneAdapter(this);
  behaviourDefintion.initBehaviourEngine();

  ...
  messageField = (SFString)fields.get("sc_message");
  messageField.addX3DEventListener(new X3DFieldEventListener() {
    public void readableFieldChanged(X3DFieldEvent evt) {
      //Send the message to the behaviour engine
      String theMessage = messageField.getValue();
      Object theSender = messageField.getUserData();
      behaviourDefinition.messageEvent(theMessage,theSender);
    }
  }
  ...
}
}
```

Die Adapterklasse definiert für jede durch einen Animationszustand beschriebene Animation eine Methode zum Starten und – falls die Notwendigkeit dafür aus dem SSIML/Behaviour-Modell hervorgeht (z. B. wenn die Animation durch ein *MessageEvent* vorzeitig beendet werden kann) – eine Methode zum Abbrechen der Animation (z. B. `startOpenHoodAnimation` und ggf. `stopOpenHoodAnimation`). Die animationsrelevanten Methoden werden durch die Behaviour-Engine nach Eintritt in einen bzw. bei Austritt aus einem Animationszustand aufgerufen. Da der Zugriff auf Felder von Knoten der X3D-Szene

zur Laufzeit nur zu definierten Zeitpunkten möglich ist, wird zum Zeitpunkt des Aufrufs einer animationsbezogenen Methode nicht direkt auf die Szene zugegriffen, sondern es wird ein sogenanntes `ActionObject`, welches den auszuführenden Szenenzugriff kapselt, erzeugt und einer Warteschlange mit synchronisiertem Zugriff hinzugefügt. Das folgende Codebeispiel zeigt ein solches `ActionObject`, das die Animation *OpenHood* startet:

```
public class HoodBehaviourSceneAdapter
        implements X3DPerFrameObserverScript {
...
private List actionQueue = Collections.synchronizedList(new ArrayList());
private X3DNode openHoodAnimationControl;
...
private interface ActionObject {
  public void execute();
}
...
public void initialize() {
  ...
  //get access to the TimeSensor which controls the animation
  SFNode n = (SFNode)fields.get("sc_OpenHoodAnimationTimeSensor");
  openHoodAnimationControl = (X3DNode)n.getValue();
  ...
}

public void startOpenHoodAnimation() {
  actionQueue.add(new ActionObject() {
    public void execute() {
      //get access to the required fields
      SFTime timeField = (SFTime)openHoodAnimationControl.getField("startTime");
      SFBool animationStarter = (SFBool)openHoodAnimationControl.getField("enabled");

      //set start-time to current time and start animation
      timeField.setValue(timeBaseTime.getValue());
      animationStarter.setValue(true);
    }
  }
}

//This method is called on a per-frame basis
public void prepareEvents() {
  //All actions are executed in the order they were added to the list
  for (int i = 0; i<actionQueue.size(); i++) {
    ActionObject ao = (ActionObject) actionQueue.get(i);
    ao.execute();
  }
  //After execution, all action objects are removed from the list
  actionQueue.clear();
}
}
```

Die `ActionObject`s werden innerhalb der `X3DPerFrameObserverScript`-Methode `prepareEvents`, welche kontinuierlich (d.h. einmal pro Frame) aufgerufen wird, in gleicher Reihenfolge wie beim Hinzufügen aus der Warteschlange entnommen, und die von einem `ActionObject` gekapselte Aktion wird durch Aufruf der `ActionObject`-Methode `execute` ausgeführt.

#### 7.3.2.4 Abbildung von RBAnimationStates – Realisierung von physikbasierten Animationen

Eine Prototypinstanz, die ein Behaviour mit `RBAnimationStates` kapselt, besitzt Direktzugriff auf die zu manipulierenden X3D-*Rigid-Body*-Objekte und auf die Transformationen der geometrischen Objekte, welche die Rigid-Body-Objekte in der virtuellen Welt visualisieren (d.h. die durch die Physikanimation beeinflussten). So können Positions- und Orientierungswerte eines X3D-Rigid-Body-Objektes mit dem Positions- und Orientierungswerten des zugehörigen Transformknotens vor dem Start einer physikbasierten Animation synchronisieren werden. Umgekehrt lässt sich das Abgleichen einer Objekttransformation mit denen durch die Physik-Engine berechneten Daten in X3D durch die Definition von `ROUTE`-Statements erreichen. Der nachfolgende Codeausschnitt demonstriert die Anwendung einer physikbezogenen Verhaltensbeschreibung für das Beispiel aus Abbildung 7.9:

```
EXTERNPROTO CarBehaviour [
    #fields generated for AccelerateCar-AnimationState:
    #fields for rigid bodies whose properties
    #will be changed in this AnimationState
    #and their corresponding transformations (same order)
    initializeOnly MFNode accelerateCarTargetRBDynamics
    initializeOnly MFNode accelerateCarTargetTransforms

    #other fields
    ...
] "CarBehaviourPrototype.x3d#CarBehaviourPrototype"

DEF carTrans Transform {
   children [
    #content definition of car
    ...
  ]
}

#definition of rigid bodies
RigidBodyCollection {
   ...
   gravity 0 -9.8 0
   collider CollisionCollection {
      collidables [
         DEF carCS CollidableShape {
            #Sphere with specified collisionRadius
            shape Sphere {radius 1.0}
         }
            #other collidable shapes
            ...
      ]
   }
   bodies [
      #rigid body definition for car
      DEF carRBDynamics RigidBody {
         geometry USE carCS
         position 0 0 0
         orientation 0 0 1 0
         forces []
         ...
      }
      #other rigid bodies
      ...]
}
```

```
#Prototype instance of CarBehaviour
DEF carBehaviour CarBehaviour {
    #get direct access to the nodes
    #which are modified in the AnimationStates
    accelerateCarTargetTransforms USE carTrans
    accelerateCarTargetRBDynamics USE carRBDynamics
    ...
}

#synchronize rigid body transformation with transform node of car
ROUTE carRBDynamics.position TO carTrans.translation
ROUTE carRBDynamics.orientation TO carTrans.rotation
```

## 7.4 Verwandte Arbeiten

Die Idee, Zustandsautomaten für die Beschreibung des Verhaltens graphischer Objekte anzuwenden, ist nicht grundsätzlich neu. Eine Domäne, in der Zustandsautomaten bereits benutzt werden, ist die Animation virtueller Kreaturen, insbesondere virtueller Menschen. Multon et al. [MFCGD99] zeigen ein Beispiel, in dem mit Hilfe von Zustandsautomaten verschiedene Phasen des menschlichen Gehens simuliert werden. Ieronutti und Chittaro [IC05] beschreiben eine Softwarearchitektur, in welcher die Verhaltensweisen virtueller Menschen durch Zustandsautomaten kontrolliert werden.

Auch in der Web3D-Domäne existieren Ansätze, Verhalten und Animation über Zustandsmaschinen zu steuern. Ein Beispiel ist hier Behaviour3D [DR03], eine XML-basierte sowie X3D- und SMIL [Wor98]-orientierte Komponentenarchitektur zur Verhaltensspezifikation. Ein anderer Vertreter des Web3D-Bereichs ist RUBE [FLPS03]. RUBE ist ein Framework, das die Simulation des Verhaltens von realen Systemen u. a. auf Basis einfacher endlicher Zustandsautomaten unterstützt. RUBE bietet eine VRML-basierte Visualisierung der Simulation.

Jacobs demonstriert, wie visuelle Zustandsautomaten zusammen mit automatischer Codegenerierung benutzt werden können, um die Programmierung des Verhaltens von grafischen Objekten in 3D-Spielen zu erleichtern [Jac05]. Aus den Zustandsautomaten generierte Programmteile werden dabei derart von anderen Programmbestandteilen separiert, dass nachträgliche Änderungen direkt im Diagrammeditor spezifiziert werden können und anschließend der komplette Verhaltenscode neu generiert wird. Eine nachträgliche Anpassung des generierten Codes ist damit nicht nötig.

Dessen ungeachtet besitzt SSIML/Behaviour einige charakteristische Merkmale, durch die es sich von den oben angeführten Ansätzen unterscheidet. Die meisten Arbeiten trennen die Spezifikation der Animationen von der Verhaltensspezifikation im Zustandsautomaten. Die Integration der Animationsbeschreibung in einen Zustand wie in SSIML/Behaviour kann es einem Entwickler erleichtern, bereits vor der programmtechnischen Umsetzung der Animation eine vergleichsweise konkrete Vorstellung des Animationsablaufs auszubilden. Bspw. würde das Modell in Abbildung 7.9 durch den Verzicht auf die Darstellung der Informationen innerhalb der Animationszustände deutlich an Aussagekraft verlieren. Außerdem können parallele Abläufe innerhalb von Zustandsautomaten mit den oben skizzierten An-

sätzen nicht realisiert werden. Ebenso fehlt es an Werkzeugen, die einen Verhaltensentwurf mittels graphischer Notationen unterstützen und die automatische Generierung von Code für verschiedene Plattformen ermöglichen, da die meisten Ansätze Zustände und Transitionen als plattformspezifische Komponenten auf Codeebene definieren.

Ein interessanter Ansatz aus dem Bereich Augmented Reality ist die Augmented Presentation and Intertaction Language (APRIL [Led04, LS05] - vgl. a. Abschnitt 5.7 und Abschnitt 9.6). APRIL erlaubt die Spezifikation des Ablaufes von *Augmented Reality* - Präsentationen, basierend auf UML-Zustandsautomaten. Animationen werden in Zuständen gekapselt und Transitionen werden von Interaktionen ausgelöst. Allerdings wird auch hier eine Trennung der Spezifikation des Interaktionsverhaltens im Zustandsautomaten und der Festlegung der eigentlichen Animationsabläufe vorgenommen. Die Modelle können in ein APRIL-spezifisches XML-Format übersetzt werden. Der Aspekt der Plattformunabhängigkeit ist bei APRIL – im Gegensatz zu SSIML/Behaviour – kein Schwerpunkt, da APRIL-Präsentationen speziell auf die Abarbeitung auf der Studierstube-Plattform [SFH+02] ausgerichtet sind.

Alice [CPGB94, Con97, PAB+97, CAB+00] ist eine Autorenumgebung, die die Spezifikation von Verhalten und Animation von 3D-Objekten – wie SSIML/Behaviour – mittels visueller Editoren erlaubt. Alice wendet sich vor allem an Neulinge im Bereich 3D-Computergrafik und 3D-Animation. Obwohl Alice keinen Ansatz auf der Basis von Zustandsautomaten verfolgt, gestaltet sich die Spezifikation von weniger komplexem Verhalten in Alice sehr intuitiv. In Alice werden parallele und/oder sequenzielle Animationen in einer Baumstruktur beschrieben. Die Blattknoten des Baumes repräsentieren einfache Animationsanweisungen. Interaktionen werden getrennt von Animationen in einer separaten Ansicht *Events* nach dem Prinzip *Stimulus – Objekt – Reaktion* festgelegt. Allerdings werden bestimmte Animationsarten, wie physikbasierte Animationen, standardmäßig nur sehr eingeschränkt oder gar nicht unterstützt.

## 7.5 Zusammenfassung

In diesem Kapitel wurde die Sprache SSIML/Behaviour als Erweiterung von SSIML vorgestellt, welche die Verhaltensbeschreibung von 3D-Objekten auf einem abstrakten Niveau erlaubt. Der Ansatz erfüllt die in Abschnitt 7.1 aufgestellten Anforderungen. Im Gegensatz zu anderen Lösungen können mit SSIML/Behaviour Animationen verschiedener Arten beschrieben werden, wie keyframe- und physikbasierte Animationen. Weiterhin können Animationsabläufe durch zusammengesetzte Zustände hierarchisch zerlegt und – im Kontrast zu verwandten Ansätzen – auch parallelisiert werden, was die Realisierung und Verwaltung komplexer Verhaltensstrukturen und Animationen ermöglicht. Das Verhalten der Objekte wird durch den Austausch von Ereignisnachrichten gesteuert. Somit kann das Verhalten eines graphischen Objektes durch Benutzeraktionen (z.B. Mausklicks), Systemnachrichten (z.B. Time-Outs) oder Nachrichten, die von anderen Objekten generiert wurden, beeinflusst werden.

SSIML/Behaviour wurde als leistungsfähige Modellierungssprache – basierend auf UML 2-Zustandsautomaten – zur Spezifikation von Animationen und Verhalten konzipiert. Neben dem SSIML/Behaviour-Metamodell existiert ein UML-Profile für SSIML/Behaviour, das beispielhaft in das UML-Werkzeug MagicDraw integriert wurde. Ebenso wurden in diesem Kapitel Konzepte zur Abbildung von SSIML/Behaviour-Modellen auf plattformspezifischen Code vorgestellt, die den Übergang zur Implementierung erleichtern.

# Kapitel 8

# SSIML/Components

## 8.1 Einführung

Eine Anzahl von Forschungsarbeiten (z.B. [DHM02, Dac04, DG00], vgl. auch Abschnitt 8.6) beschäftigt sich mit der Vereinfachung der Entwicklung von 3D-Applikationen durch die Anwendung von Komponentenkonzepten. Im Einzelnen bieten 3D-Komponenten die folgenden Vorteile:

- eine schnellere Szenenentwicklung durch den Einsatz kombinierbarer vordefinierter Softwarebausteine, deren Stabilität gewährleistet werden kann,
- die Dekomposition komplexer Szenenstrukturen durch die Kapselung von Subgraphen in Komponenten, wodurch ein einfacher und flexibler Austausch von Szenenbestandteilen möglich und die Wartbarkeit von großen 3D-Szenen verbessert wird sowie
- eine Trennung von Objektverhalten und Objektgeometrie; dadurch kann ein Verhalten bspw. mit mehr als einem Objekt assoziiert werden.

Weitere Vorteile ergeben sich aus der Tatsache, dass eine 3D-Komponente eine Menge von Eigenschaften mit definierten Zugriffsmöglichkeiten besitzt, welche die Schnittstelle der Komponente repräsentieren:

- Die Komplexität der Komponentenimplementierung wird verborgen.
- Der Zugriff auf Komponenteninhalte wird in gewünschter Weise beschränkt.
- Die Klarheit und Wartbarkeit des Anwendungscodes, der benötigt wird, um Szeneninhalte zu modifizieren, wird verbessert. 3D-Komponenten erleichtern damit die Integration von 3D-Inhalten in externe Anwendungen.
- Das visuelle Erscheinungsbild einer 3D-Komponente kann durch Parametrisierung einfach angepasst werden.

Dessen ungeachtet kann der Einsatz von 3D-Komponenten auch mit Erschwernissen verbunden sein. I. d. R. müssen die Komponenten auf Implementierungsebene in Textform spezifiziert werden. Dies verkompliziert die Beschreibung von Komponenten mit vielen Eigenschaften, komplexen inneren Strukturen und nicht-trivialem Verhalten. Ebenso kann die textliche Spezifikation einer großen Zahl von Komponentenverknüpfungen unübersichtlich werden. Einen Ansatz zur Erleichterung stellen hier Notationen und Werkzeuge dar, die eine visuelle Spezifikation von Komponenten und deren Verknüpfungen oberhalb des Implementierungsniveaus erlauben.

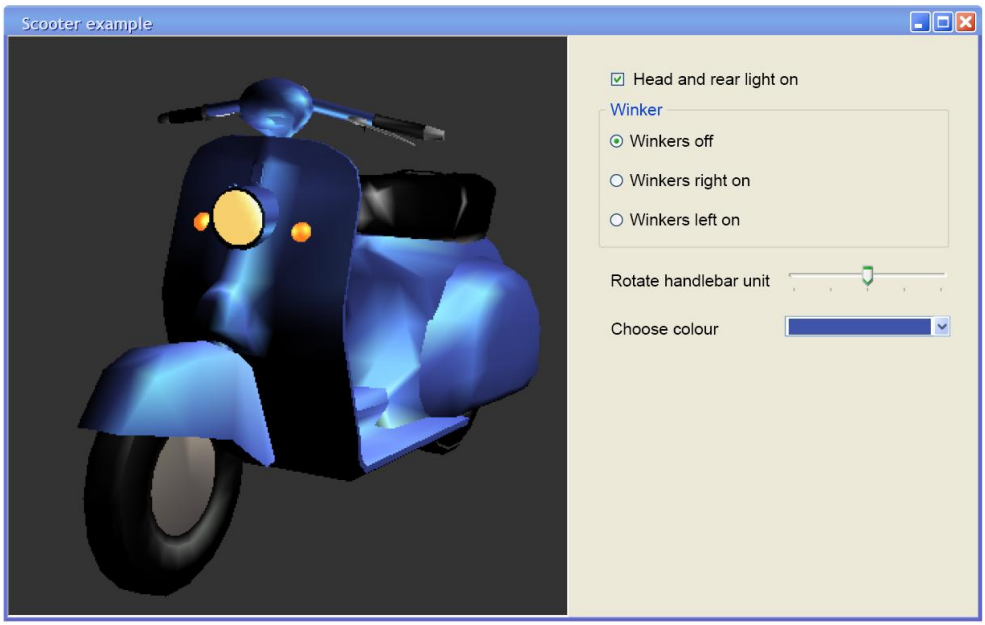
Mit SSIML/Components wird ein Ansatz vorgestellt, welcher – im Kontrast zu verwandten Arbeiten (s. Abschnitt 8.6) – die Spezifikation von 3D-Komponenten und Komponentenverknüpfungen mit Hilfe einer visuellen Notation in der Entwurfsphase des Entwicklungsprozesses erlaubt. SSIML/Components ist eine Erweiterung der graphischen Modellierungssprache SSIML (vgl. Kapitel 5). Durch die Verbindung von SSIML mit Komponentenkonzepten kann zudem die Integration interaktiver 3D-Grafik in Anwendungen weiter vereinfacht werden. Im Vergleich zu SSIML-Kompositionsknoten stellt SSIML/Components verbesserte Mechanismen zur Verwaltung und Dekomposition komplexerer Szenen zur Verfügung.

## 8.2 Beispiel

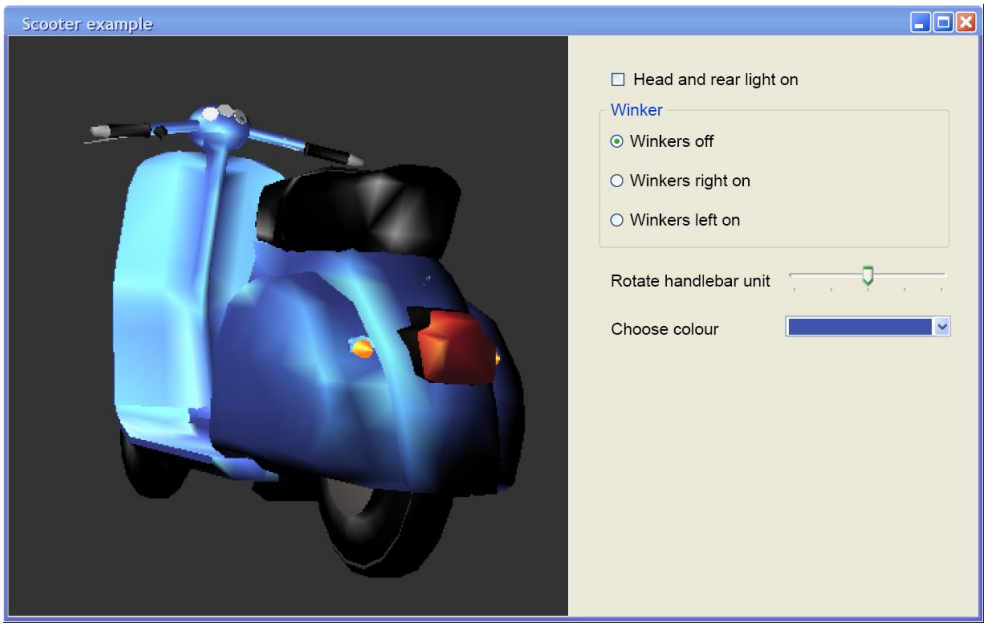
Zur Demonstration des Einsatzes der Sprache SSIML/Components in nachfolgenden Abschnitten soll ein einfaches Beispiel aus dem Gebiet der Produktvisualisierung dienen: ein Motorrollermodell. Die Abbildungen 8.1(a) und 8.1(b) zeigen die Benutzungsschnittstelle der Anwendung. Neben dem Rotieren des gesamten Rollermodells hat der Anwender verschiedene Möglichkeiten, mit der 3D-Benutzungsschnittstelle zu interagieren. So können die Lenkeinheit innerhalb eines vorgegebenen Winkels bewegt, alle Lampen (Vorder- und Rückleuchte sowie Blinkerlampen) ein- und ausgeschaltet, und auch die Farbe des Rollers geändert werden. Die 3D-Szene des Motorrollers lässt sich in 3D-Komponenten unterteilen, welche die zur Realisierung der Anwendung notwendigen Geometrie- und Verhaltensbeschreibungen kapseln.

## 8.3 Metamodell

Das hier vorgestellte Metamodell zur Beschreibung von 3D-Komponenten lehnt sich an bekannten Ansätzen aus Forschung und Technik, wie CONTIGRA [DHM02, Dac04], 3D-Beans [DG00] oder X3D-Prototypen [Int04b] an. Allerdings bietet es Mechanismen, die über andere Ansätze hinausgehen. Dazu gehört z. B. die Erweiterung von 3D-Komponenten über Mehrfachvererbung oder die Definition von Beziehungen zwischen 3D-Komponenten und einer Rahmenanwendung. Obwohl auch die UML die Möglichkeit der Komponentenspezifikation bietet, eignet sich das UML-Komponentenkonzept nur bedingt für die Übertragung auf 3D-Komponenten. Bspw. stellt eine UML-Komponente eine Menge von




(a) Vorderansicht




(b) Rückansicht

Abbildung 8.1: Benutzungsschnittstelle der Motorroller-Produktvisualisierung

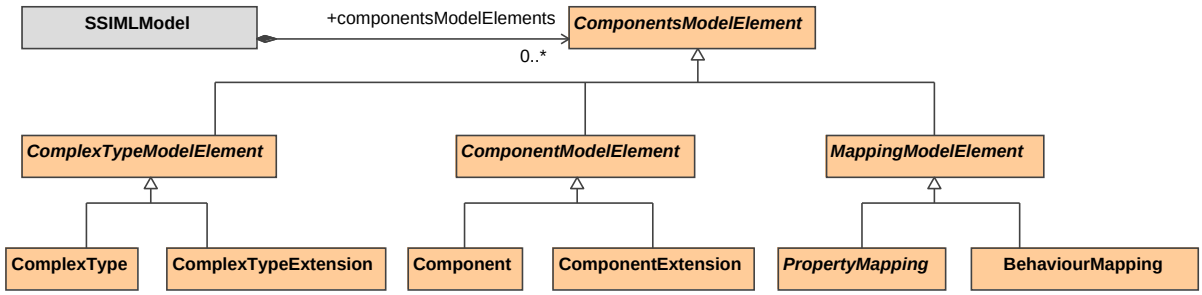



Abbildung 8.2: Containment-Hierarchie der Elemente des SSIML/Components-Metamodells

Schnittstellen bereit, die von anderen Komponenten verwendet werden können. Eine 3D-Komponente hingegen stellt eine Menge von einzelnen Eigenschaften zur Verfügung, die beeinflusst werden können, um z. B. das visuelle Erscheinungsbild der 3D-Repräsentation einer Instanz der Komponente zu ändern.

Abbildung 8.2 zeigt die Containment-Hierarchie der Elemente des SSIML/Components-Metamodells (Farb-Legende s. Kapitel 5, Abbildung 5.5). SSIML/Components baut auf SSIML/Behaviour auf. Die einzelnen SSIML/Components-Elemente werden im Weiteren dargestellt.

### 8.3.1 Einfache und komplexe Datentypen

In SSIML/Components existieren verschiedene Datentypen. Datentypen können einfache (*Simple Types*) oder komplexe Typen (*Complex Types*) sein. Einfache Datentypen sind `STRING`, `INT`, `FLOAT`, `BOOLEAN` und – für SSIML-Knoteninstanzen, codiert als Kompositionspfade – `NODE` (Abbildung 8.3). Diese Typenauswahl dient der Unterstützung grundlegender Datentypen potenzieller SSIML/Components-Zielformate wie Contigra oder X3D. Ein einfacher Typ ist ein *Listentyp*, wenn das Attribut `isList` des einfachen Typs den Wert *Wahr* (`true`) enthält. Listentypen werden Elementen zugewiesen, die eine Liste von Werten eines Datentyps speichern können. `numberOfComponents` bestimmt, falls angegeben, bei Listentypen die Zahl der Wertkomponenten in der Liste. Ein RGB-Farbwert hätte z. B. drei Wertkomponenten: Eine R-, eine G- und eine B-Komponente. Ein Wert eines einfachen Typs oder eine Wertkomponente eines Listentyps wird in einem SSIML/Components-Modell immer als Zeichenkette (*String*-Repräsentation) abgelegt, die sich in den ihr zugeordneten Typ konvertieren lässt. Ein bestimmter RGB-Wert, dessen Wertkomponenten den Typ `FLOAT` haben, kann z. B. durch die aufeinander folgenden 3 Zeichenketten *„0.5“*, *„0“*, *„0“* repräsentiert werden.

Das Konzept der *komplexen Typen* erlaubt Softwareentwicklern, ihre eigenen Datentypen zusammenzusetzen. Ein komplexer Typ (Abbildung 8.4) besteht aus einer Menge von Elementen (*Complex Type Elements*). Ein `ComplexTypeElement` besitzt einen Namen (`name`), einen einfachen Typ (`simpleType`), einen optionalen Wertebereich (`valuesRange`)

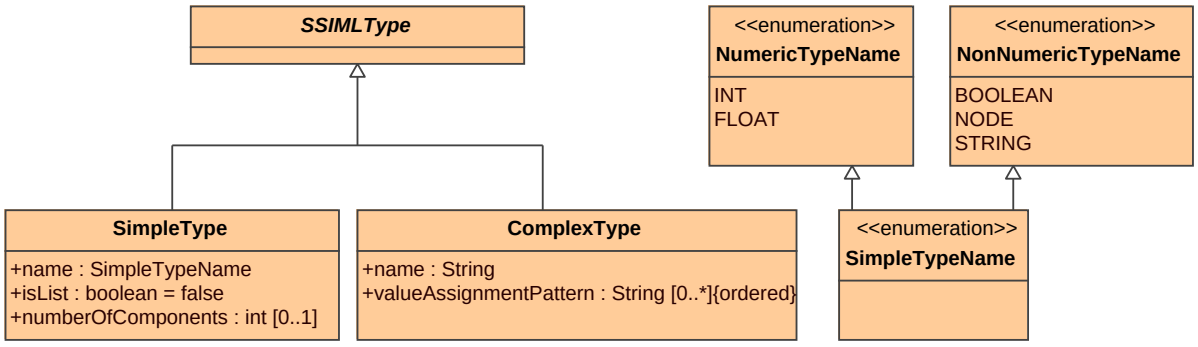



Abbildung 8.3: Einfache und komplexe Datentypen

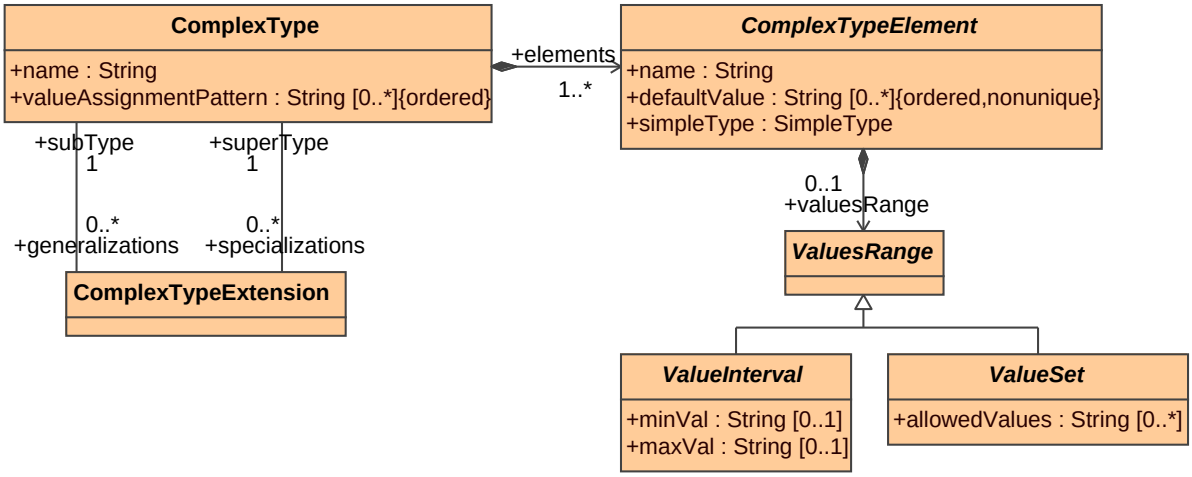



Abbildung 8.4: `ComplexType` und in Beziehung stehende Elemente des Metamodells

und einen optionalen Standardwert (`defaultValue`). Ein Wertebereich (Metaklasse `ValuesRange`) kann entweder durch einen Minimalwert und/oder einen Maximalwert oder eine Liste von erlaubten Werten definiert werden (in Abbildung 8.4 unten rechts). Die Angabe eines Minimal- bzw. eines Maximalwertes ist nur bei numerischen Typen (`INT` und `FLOAT`) möglich. Bei Listentypen gilt der spezifizierte Wertebereich für alle Wertkomponenten.

Die vorgestellten Mechanismen zur Einschränkung von Wertebereichen komplexer Typen lehnen sich an die aus XML-Schema [Wor04a, Wor04b, Wor04c] bekannten Möglichkeiten an. Die Überprüfung der Einhaltung der Einschränkungen auf Modellebene muss durch ein SSIML/Components-Modellierungswerkzeug gewährleistet werden.

Die Definition eines Standardwertes für ein `ComplexTypeElement` ist nicht zwingend notwendig. Wird kein Standardwert angegeben, kann dieser später noch in dem aus einem Modell generierten Code manuell festgelegt werden.

Ein komplexer Typ erlaubt es auch, Daten eine Semantik zuzuordnen. Dies kann von

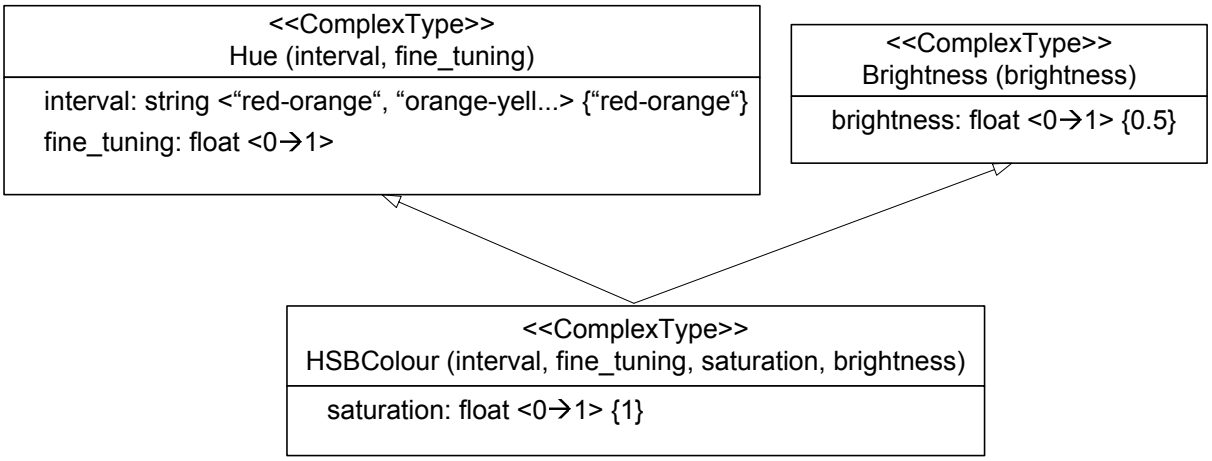



Abbildung 8.5: Die komplexen Typen `Hue`, `Brightness` und `HSBColour`

Bedeutung sein, wenn ein Modell auf plattformspezifischen Code abgebildet wird. Zum Beispiel kann ein `FLOAT`-Wert verschiedenen Feldern eines Szenenelements (etwa dem Feld *Radius* einer Kugel oder dem Feld *Transparenz* eines Materials) zugeordnet werden. Wird allerdings ein komplexer Typ *SphereRadius* definiert, welcher den Float-Wert enthält, wird damit ausgedrückt, dass der Wert dem Radius einer Kugel zugeordnet werden soll.

Das Wertzuweisungsmuster (*Value Assignment Pattern*) eines komplexen Typs (vgl. Abbildung 8.4) beschreibt die Reihenfolge, in welcher die Werte für Elemente eines komplexen Typs spezifiziert werden. Die einzelnen Komponenten des Musters entsprechen Namen von Elementen des komplexen Typs, wobei kein Name mehrfach vorkommen darf. Außerdem müssen nicht zwangsweise alle Elemente eines komplexen Typs über sein Wertzuweisungsmuster angesprochen werden können. Elemente, die nicht Bestandteil des Wertzuweisungsmusters sind, können zwar mit einem Standardwert initialisiert, jedoch später nicht mehr modifiziert werden. Wie an anderer Stelle noch ausgeführt wird, dienen Wertzuweisungsmuster dazu, einer 3D-Komponenteneigenschaft eines komplexen Typs einen Wert zuzuweisen.

Das Konzept der komplexen Typen soll an einem Beispiel verdeutlicht werden. Im Beispiel wird der komplexe Typ `HSBColour` spezifiziert, der es dem Entwickler erlaubt, eine Farbe im HSB-Farbraum zu definieren. Da sich das HSB-Modell an der menschlichen Farbwahrnehmung orientiert, gilt die Definition einer Farbe im HSB-Farbraum oft intuitiver als ihre Definition im RGB-Farbraum. Daher existieren auch einige 3D-APIs, die das HSB-Farbmodell unterstützten (z. B. [Ell99]). In Abbildung 8.5 ist der komplexe Typ `Hue` (Farbton) zu sehen. `Hue` hat zwei Elemente: `interval` und `finetuning` (Feineinstellung). `interval` ist vom Typ `STRING` und ermöglicht die Definition eines Farbton-Intervalls. Erlaubte Werte sind `red-orange`, `orange-yellow`, `yellow-green`, `green-blue`, `blue-violet`, und `violet-red`. Das Standardintervall ist `red-orange`. Das `FLOAT`-Element `finetuning` mit einem Wertebereich von Null bis Eins drückt aus, wo der Farbton innerhalb des gewählten Intervalls liegen soll. Wäre bspw. das Farbintervall `red-orange` selektiert und der Feinein-

stellungswert wäre gleich Null, würde der Farbton *Rot* beschrieben. Im gleichen Intervall definiert ein Feintuning-Wert von 1 den Farbton *Orange*, ein Feintuningwert von 0.5 ergibt eine Mischung aus gleichen Anteilen von „reinem“ Rot und „reinem“ Orange. Im Vergleich zur Spezifizierung eines Farbtons allein über eine Zahl bietet der komplexe Typ `Hue` die Möglichkeit, einen Farbton auf intuitive Weise zu beschreiben. Gleichfalls lassen sich mehr Farbtöne beschreiben als bei einer einfachen Adressierung eines Farbtons über seinen Namen (z: B. *Rot*). Das Wertzuweisungsmuster wird in geschwungenen Klammern nach dem Namen des komplexen Typs angegeben. Das Wertzuweisungsmuster legt fest, dass zunächst für `interval` und anschließend für `finetuning` ein Wert angegeben werden muss.

Abbildung 8.5 zeigt einen weiteren komplexen Typ: `Brightness.` Dieser dient zur Beschreibung von Helligkeitswerten. `Brightness` besitzt ein entsprechendes Element `brightness`, welches den Helligkeitswert speichert. `Brightness` wird durch einen Gleitkommawert zwischen einschließlich Null und Eins repräsentiert.

Komplexe Typen können im Sinne einer Spezialisierung erweitert werden. Dies wird durch die Metaklasse `ComplexTypeExtension` in Abbildung 8.4 ausgedrückt. In Abbildung 8.5 erweitert der Typ `HSBColour` die Typen `Hue` und `Brightness`. `HSBColour` wird daher auch *Subtyp* von `Hue` und `Brightness` genannt, während `Hue` und `Brightness` *Supertypen* von `HSBColour` sind. `HSBColour` erbt alle Elemente seiner Supertypen. Wie Abbildung 8.5 zeigt, ist bei der Erweiterung komplexer Typen Mehrfachvererbung erlaubt.

Weiterhin ist es möglich, zusätzlich zur Menge der geerbten Elemente neue Elemente zu definieren. In Abbildung 8.5 wurde `HSBColour` um das Element `saturation` erweitert, welches die Sättigung einer Farbe beschreibt. Ein Sättigungswert von Null entspricht dabei keiner Sättigung, während ein Wert von Eins die maximale Sättigung repräsentiert. Der Standardwert der Sättigung ist Null.

Das Wertzuweisungsmuster eines Subtyps kann Namen aller Elemente des komplexen Typs (einschließlich Namen geerbter Elemente) enthalten (siehe Typ `HSBColour` in Abbildung 8.5).

Wertebereiche oder Standardwerte von Supertypelementen können redefiniert werden. Der Wertebereich eines Elements kann als *Invariante* angesehen werden. Ein Subtyp darf daher den Wertebereich eines von einem Supertyp geerbten Elements einschränken, nicht aber erweitern. Dies folgt aus dem Liskovschen Substitutionsprinzip [LW94]. Eine Invariante, die für einen Supertyp gilt, muss auch für dessen Subtyp gelten. Wenn z. B. das Element `brightness` im Supertyp `Brightness` einen Wert zwischen einschließlich Null und Eins besitzt, dann muss `brightness` auch im Subtyp `HSBColour` einen Wert zwischen einschließlich Null und Eins besitzen. Dies wäre nicht mehr sichergestellt, wenn der Wertebereich von `HSBColour` erweitert werden würde.

#### 8.3.1.1 Auflösung von Namenskonflikten bei Mehrfachvererbung

Besitzen zwei Supertypen eines komplexen Typs gleichnamige Elemente, kommt es zu *Namenskonflikten* (auch als *Namenskollisionen* bezeichnet). Bei derartigen Konflikten sind prinzipiell zwei Fälle zu unterscheiden:

1. Entweder *Namen* der in Konflikt stehenden Elemente *stimmen überein*, aber die Element-*Typen* sind *unterschiedlich* oder

2. *Namen und Typen* der kollidierenden Elemente *stimmen überein.*

Eine Übereinstimmung der Typen bedeutet, dass die Elemente exakt den gleichen einfachen Typ[1] besitzen.

Ein SSIML/Components-Werkzeug sollte eine semi-automatische Konfliktauflösung unterstützen. Fall 1 kann gelöst werden, indem ein oder mehrere Namen der in Konflikt stehenden Elemente (bei zwei bzw. mehr als zwei in Konflikt stehenden Elementen) durch den Nutzer geändert werden. Zu beachten ist, dass die Namensänderung eines `ComplexType`-Elements auch an die Supertypen und Subtypen des komplexen Typs weitergegeben werden muss. Dies sollte automatisch erfolgen.

Im 2. Fall muss der Benutzer entscheiden, ob eine Konfliktauflösung, wie für Fall 1 vorgeschlagen, erfolgen soll. Drücken die kollidierenden Elemente z. B. semantisch dasselbe Konzept aus, kann eine automatische Auflösung ohne Namensänderungen von `ComplexType`-Elementen stattfinden. Ein entsprechendes SSIML/Components-Werkzeug muss dazu beim erbenden komplexen Typ ein namens- und typgleiches Element anlegen, welches die kollidierenden Elemente der Supertypen redefiniert. Das redefinierende Element muss der Bedingung gerecht werden, dass ein komplexer Typ den Wertebereich einer Eigenschaft eines Supertyps einschränken, aber nicht erweitern darf (vgl. Abschnitt 8.3.1). Der Wertebereich des Elements des erbenden komplexen Typs muss also aus der Schnittmenge der Wertebereiche der entsprechenden Elemente der Supertypen gebildet werden. Ein Standardwert aus Supertypen kann für das redefinierende Element nur übernommen werden, wenn die Standardwerte der entsprechenden Supertypelemente gleich sind. Andernfalls bleibt der Standardwert des redefinierenden Elements so lange undefiniert, bis der Nutzer den Wert manuell festlegt, da nicht automatisch bestimmt werden kann, von welchem Supertyp der Standardwert übernommen wird. Zudem können die Standardwerte der Supertypelemente außerhalb des Wertebereichs des redefinierenden Elements liegen. Besitzen bspw. zwei Supertypen *S1* und *S2* eines komplexen Typs *C* zwei gleichnamige `FLOAT`-Elemente *e* mit den Wertebereichen (Intervallen) *[0.0, 0.8]* und *[0.2, 1.0]* und den Standardwerten *0.1* und *0.9*, ergibt sich für das redefinierte Element *e* des komplexen Typs *C* ein Wertebereich *[0.2, 0.8]*. Beide Standardwerte für *e* in *S1* und *S2* liegen damit außerhalb des Wertebereichs von *e* in *C*.

### 8.3.2 Komponenten

Abbildung 8.6 (Mitte, links) zeigt das Element `Component` (Komponente) im SSIML/Components Metamodell. Im Gegensatz zu komplexen Typen können Komponenten einen Subgraphen der Szene (einschließlich der Elemente, die innerhalb eines Subgraphen erlaubt sind) in ihrem Rumpf (*Component Body*) kapseln. Eine Komponente besitzt außerdem

[1] Zwei einfache Typen sind gleich, wenn die Belegung ihrer Attribute (`name`, `isList`, `numberOfComponents`) übereinstimmt.

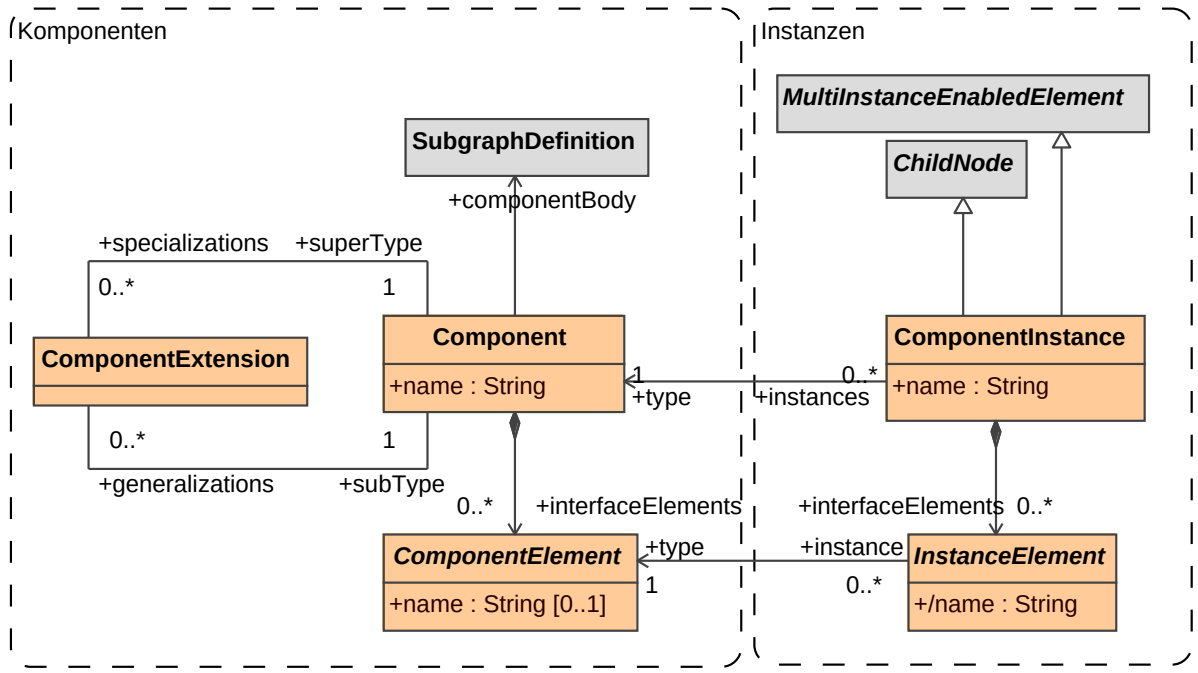



Abbildung 8.6: `Component, ComponentInstance` und in Beziehung stehende Elemente im Metamodell

einen Namen und eine Menge von Elementen vom Typ `ComponentElement`, welche die Schnittstelle der Komponente bilden.

Ein spezielles `ComponentElement` ist eine Komponenteneigenschaft (`ComponentProperty`, in Abbildung 8.7 unten links). Eine solche Eigenschaft hat einen (optionalen) Standardwert eines komplexen oder einfachen Typs (vgl. Abbildung 8.3). Weiterhin können über die Attribute `read` und `write` einer Eigenschaft verschiedene Zugriffsmethoden definiert werden, wie *Read* (nur Lesen erlaubt, d.h. nur `read` gilt), *Write* (nur Schreiben erlaubt, nur `write` gilt), *ReadWrite* (Lesen und Schreiben erlaubt, `read` und `write` gilt) und *Init* (weder `read` noch `write` gilt). *Init* bedeutet, dass nach der Initialisierung einer Eigenschaft, z. B. mit einem im Modell definierten Standardwert, kein Wertzugriff mehr möglich ist. Abbildung 8.8 illustriert die Notation von Komponenteneigenschaften und ihren Zugriffsmethoden.

Außerdem kann eine Komponente `ComponentBehaviourPort`-Elemente beinhalten (Abbildung 8.7 unten rechts), die Schnittstellen zur Steuerung des Komponentenverhaltens bereitstellen.

Wie komplexe Typen können auch Komponenten erweitert werden, wobei das Erweiterungskonzept analog zu dem für komplexe Typen ist (siehe Abschnitt 8.3.1). Eine Komponente erbt die Elemente ihrer Supertypen. Im Gegensatz zu komplexen Typen sind bei Komponenten zusätzlich die Zugriffsmethoden von Eigenschaftselementen in Subtypen redefinierbar. Zugriffsmethoden können in einem Subtyp erweitert, nicht aber wieder

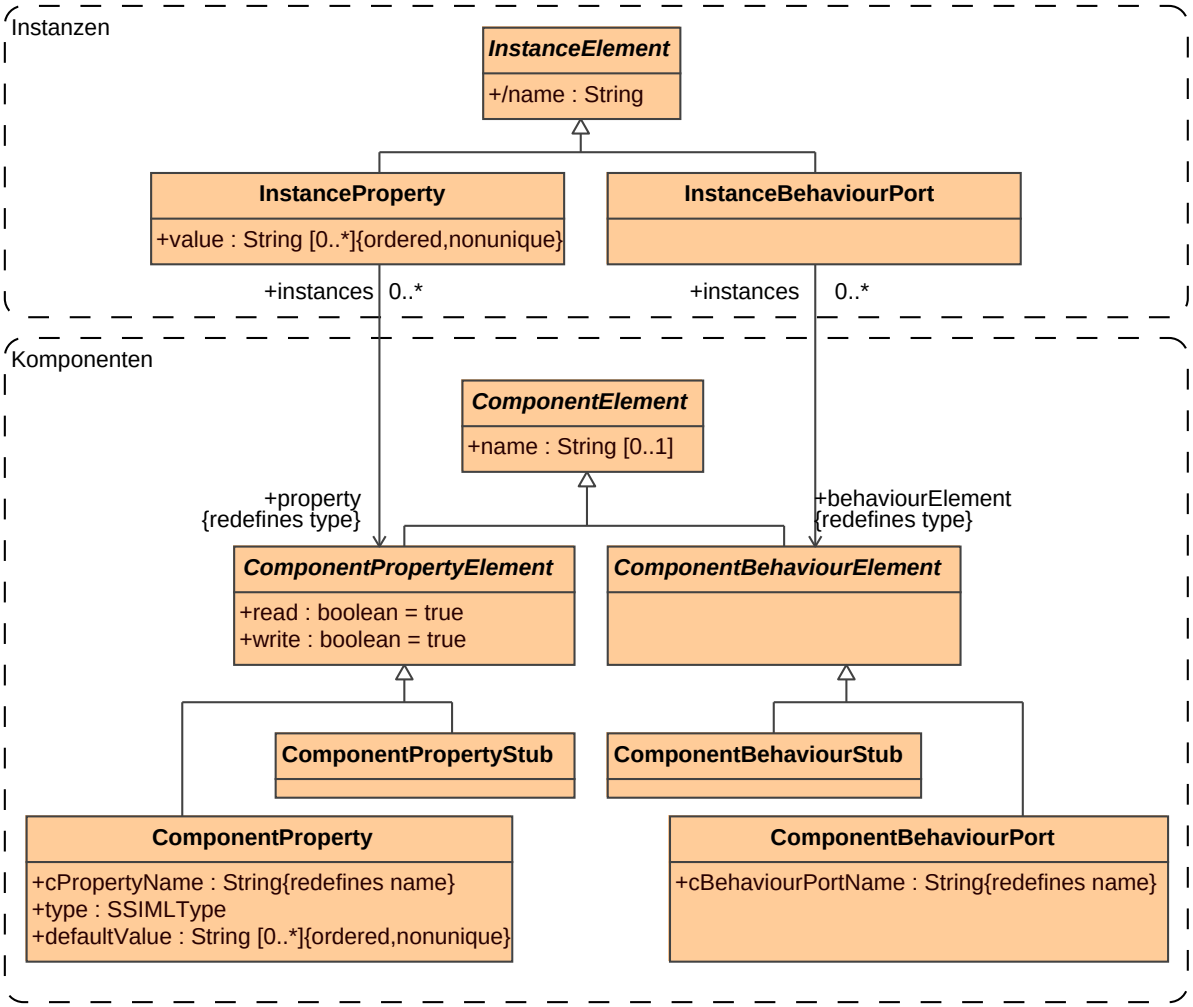



Abbildung 8.7: Elemente von Komponenten und Komponenteninstanzen

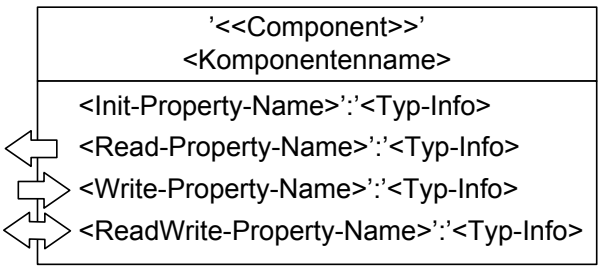



Abbildung 8.8: Notation von Komponenteneigenschaften und ihren Zugriffsmethoden

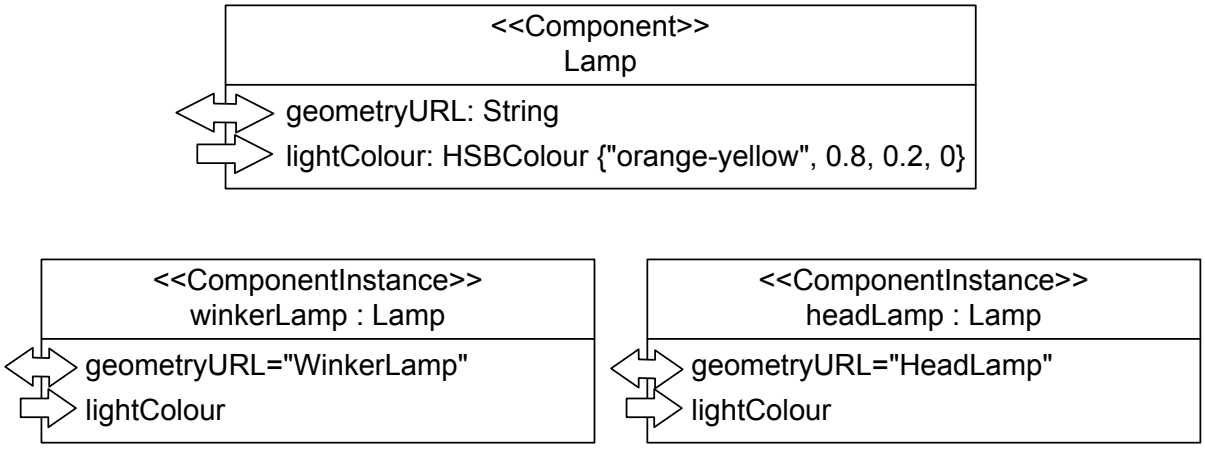



Abbildung 8.9: Die Komponente `Lamp`

eingeschränkt werden. Die für einen generischeren Komponententyp definierten Zugriffsmöglichkeiten für eine Eigenschaft muss also auch für die entsprechende Eigenschaft seiner spezielleren Subtypen gelten. *Init* kann durch *alle* anderen Zugriffsmethoden, *Read* und *Write* können durch *ReadWrite* erweitert werden. Namenskonflikte durch Mehrfachvererbung können bei Komponenten ebenfalls analog zur Auflösung von Namenskonflikten bei komplexen Typen behandelt werden (vgl. Abschnitt 8.3.1.1). Zu beachten ist, dass, falls eine bei den Supertypen einer Komponente kollidierende Eigenschaft in der Komponente redefiniert werden soll, die Zugriffsmethode der redefinierenden Eigenschaft die oben aufgestellten Regeln erfüllen muss. Das `read`- und das `write`-Attribut des redefinierenden Elementes können durch ODER-Verknüpfungen der `read`- bzw. `write`-Attribute der entsprechenden Eigenschaftselemente der Supertypen korrekt gesetzt werden.

In Abbildung 8.9 ist exemplarisch eine Komponente `Lamp` dargestellt. `Lamp` soll Lampenobjekte repräsentieren. `Lamp` besitzt zwei Eigenschaften: `geometryURL` und `lightColour`. `geometryURL` ist eine Zeichenkette, welche die URL enthält, die auf die Datei mit der Lampengeometriebeschreibung verweist. Die URL kann sowohl gelesen als auch geschrieben werden. Damit wird ein einfaches Austauschen der Lampengeometrie zur Laufzeit der 3D-Anwendung möglich. Die Eigenschaft `lightColour` vom komplexen Typ `HSBColour` kann benutzt werden, um das Ein- und Ausschalten des Lichtes zu simulieren. Für `lightColour` wurde der Standardwert `{‘‘orange-yellow‘‘,0.8,0.2,0}` angegeben. Dies entspricht einem wenig gesättigtem Gelb mit der Helligkeit Null, also der Farbe Schwarz. Durch die Manipulation des Helligkeitswertes kann das Licht ein- (maximale Helligkeit) und ausgeschaltet (minimale Helligkeit) werden.

#### 8.3.2.1 Komponenteninstanzen

Eine Komponente kann instanziiert werden. Abbildung 8.6 zeigt diesen Sachverhalt im Metamodell. Wird eine Komponenteninstanz (`ComponentInstance`) im Modell erzeugt, wird für jedes Element der zugehörigen Komponente ein Komponenteninstanzelement

(`InstanceElement`) erzeugt, mit dem Komponentenelement assoziiert und der Komponenteninstanz hinzugefügt. Dies ist notwendig, damit z. B. Eigenschaftswerte einzelner Instanzen geändert werden können, ohne dabei die für die Komponente ggf. definierten Standardwerte zu überschreiben. Eine Komponenteninstanz enthält auch für die geerbten Elemente der assoziierten Komponente entsprechende Instanzelemente. Welche Arten von Instanzelementen existieren und wie diese mit den jeweiligen `ComponentElement`-Subtypen in Beziehung stehen, ist in Abbildung 8.7 unten dargestellt. Bspw. ist eine Instanzeigenschaft (`InstanceProperty`) in einem SSIML/Components-Modell immer mit einer Komponenteneigenschaft assoziiert. Der Wert einer Instanzeigenschaft kann über das Attribut `value` angegeben werden. Wird kein Wert für eine Instanzeigenschaft festgelegt, wird – falls möglich – der für die zugehörige Komponenteneigenschaft definierte Standardwert übernommen.

Eine Komponenteninstanz stellt – wie der Wurzelkompositionsbereich des SSIML-Szenenmodells – einen *abgeschlossenen Kompositionsbereich* dar. Im Unterschied zu Kompositionsknoten ist es bei einer Komponenteninstanz nicht möglich, über einen Kompositionspfad auf die inneren Elemente der Instanz zuzugreifen, da ein Zugriff auf diese Elemente lediglich über die Komponentenschnittstelle erfolgen kann.

Komponenteninstanzen können selbst als Kindknoten in die Szene eingebunden werden und auf die gleiche Art und Weise wie atomare SSIML-Knoten über einen Kompositionspfad adressiert werden. Wie Abbildung 8.6 zeigt, ist eine Komponenteninstanz auch ein Kindknoten. Sie kann selbst allerdings keine Kindknoten besitzen, sie erbt also nicht von `ParentNode`.

In Abbildung 8.9 sind zwei Instanzen der Komponente `Lamp` dargestellt. Eine Instanz beschreibt eine Blinkerlampe (`winkerLamp`), während die andere Instanz (`headLamp`) ein Frontlicht repräsentiert. `winkerLamp` und `headLamp` besitzen unterschiedliche Geometrien. Bspw. ist die Geometrie von `winkerLamp` in einer Ressource `WinkerLamp` definiert.

#### 8.3.2.2 Zusammengesetzte Komponenten und Verhaltensintegration

Es wäre sinnvoll, den Zustand der Lampe (s. Beispiel in Abschnitt 8.3.2) nur durch das Senden entsprechender *Ein-/Aus*-Nachrichten ändern zu können, anstatt die Helligkeitswerte der Lichtquelle modifizieren zu müssen. Dies wird durch das Einbetten eines Verhaltensobjekts (`Behaviour`) in eine Komponente möglich (s. a. Kapitel 7). Wie bereits angedeutet, kann eine Komponente einen Subgraphen einschließlich all seiner Elemente, also auch ggf. vorhandene Verhaltensobjekte, enthalten.

Abbildung 8.10 zeigt die Definition der Komponente `ToggleLamp`. Neben der Eigenschaft `description`, die es erlaubt, eine Instanz der Komponente mit einer Beschreibung zu versehen, wurde die innere Struktur im Rumpf der Komponente modelliert. `ToggleLamp` enthält eine Instanz der Komponente `Lamp` und eine Instanz eines Verhaltensobjektes (`Behaviour`), d.h. eine Instanz der Verhaltensdefinition `ToggleLightBehaviour` mit dem Namen `toggleLight`. `toggleLight` ist mit der Eigenschaft `lightColour` der Komponenteninstanz `theLamp` assoziiert. Außerdem ist `toggleLight` mit einem so genannten *Verhaltensstumpf* (*Behaviour Stub*) verbunden. Dieser erlaubt den externen Zugriff auf die

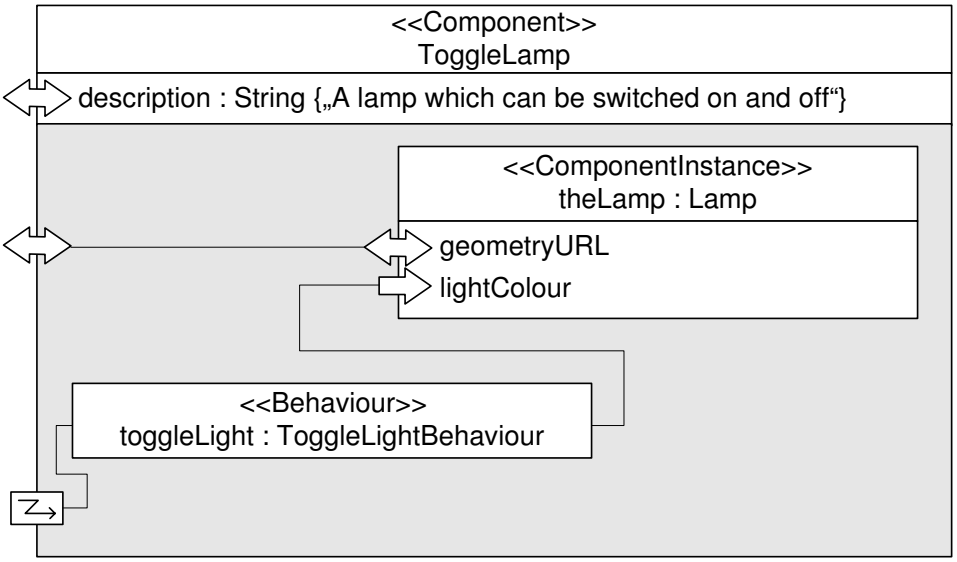



Abbildung 8.10: Die Komponente `ToggleLamp`

Schnittstelle des Verhaltensobjektes. Damit ist es möglich, über `toggleLight` die Farbe der Lichtquelle in Abhängigkeit von extern gesendeten Nachrichten (z. B. `MessageEvent`s, Start- und Stopp-Anweisungen) entsprechend der Verhaltensbeschreibung, die in der Verhaltensdefinition `ToggleLightBehaviour` spezifiziert wurde, zu modifizieren.

Abbildung 8.11 zeigt verhaltensbezogene Elemente im Metamodell. Die hellgrau dargestellten Metaklassen entstammen SSIML/Behaviour und werden in SSIML/Components wiederverwendet. Für eine genauere Beschreibung dieser Metaklassen sei auf Kapitel 7 verwiesen. Die SSIML/Components-Metaklasse `Behaviour`, deren Instanzen Verhaltensobjekte wie `toggleLight` aus dem obigen Beispiel sind, stellt ein durch Paketverschmelzung entstandenes Inkrement des gleichnamigen SSIML/Behaviour-Elements dar.

Aus dem Diagramm geht außerdem hervor, dass ein Verhaltensstumpf durch ein `BehaviourMapping`-Element nicht nur mit einem Verhaltensobjekt (wie `toggleLight` im Beispiel), sondern auch mit einem Instanzverhaltensport (`InstanceBehaviourPort`) verbunden werden kann. Ein Instanzverhaltensport ist wie eine Instanzeigenschaft ein `InstanceElement` und mit einem entsprechenden Komponentenverhaltenselement (`ComponentBehaviourElement`, vgl. Abbildung 8.7) assoziiert. Ein Instanzverhaltensport stellt die Schnittstelle zur Verfügung, die zur Steuerung eines von der Instanz gekapselten Verhaltens benötigt wird.

In der Spezifikation eines Verhaltens, das mit einer oder mehreren Eigenschaften einer Komponenteninstanz assoziiert werden soll, lässt sich aus der Menge der in Kapitel 7 definierten Animationszustandstypen nur der Typ `ProceduralAnimationState` einsetzen, da sich alle anderen nicht-abstrakten Animationszustandstypen auf konkrete Knotenattributtypen beziehen. Allerdings sind komponentenorientierte Erweiterungen der existierenden Animationszustandstypen durchaus denkbar. Weiterhin ist zu beachten, dass der Schreib-

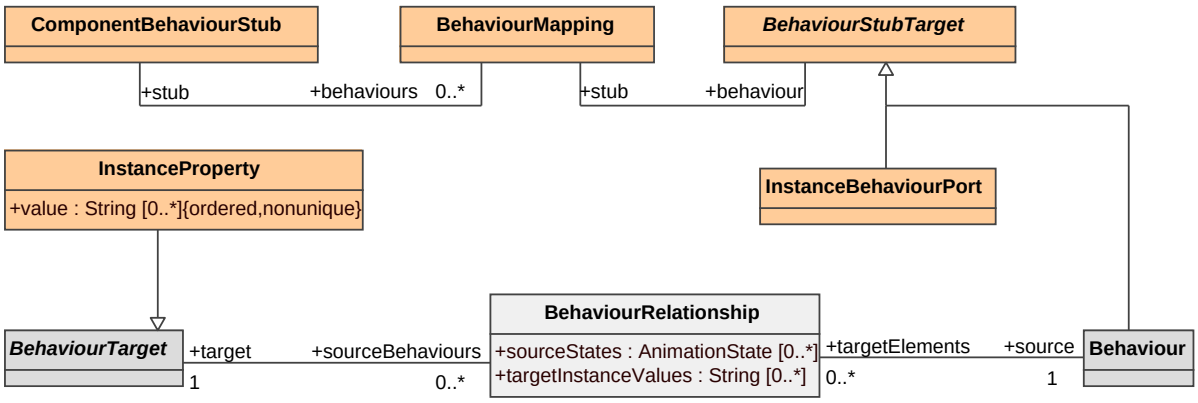



Abbildung 8.11: Verhaltensbezogene Elemente im SSIML/Components-Metamodell

Tabelle 8.1: Regeln für die Abbildung äußerer Komponenteneigenschaften auf innere Komponenteninstanzeigenschaften

| **Externe Eigenschaft** | Read | Write | ReadWrite |
|---|---|---|---|
| **Interne Eigenschaft** | Read, ReadWrite | Write, ReadWrite | ReadWrite |

zugriff für Komponenteninstanzeigenschaften, die durch ein Verhaltensobjekt verändert werden sollen, erlaubt sein muss.

Eigenschaften einer Komponente, welche auf eine oder mehrere Eigenschaften von in der Komponente enthaltenen Komponenteninstanzen (so genannte *innere Komponenteninstanzeigenschaften*) abgebildet werden können, werden als *Eigenschaftsstumpf* (*Property Stub*) am Rand des visuellen Komponentenmodells angetragen. Ein *Eigenschaftsstumpf* übernimmt in Bezug auf eine *innere Komponenteninstanzeigenschaft* die Rolle der *äußeren Komponenteneigenschaft*. Abbildung 8.12 (rechts) zeigt diesen Sachverhalt im Metamodell. In Abbildung 8.10 besitzt die Eigenschaft `geometryURL` der inneren Komponenteninstanz `theLamp` einen *Eigenschaftsstumpf*. Mit anderen Worten: die Eigenschaft `geometryURL` der Komponente `ToggleLamp` wird auf die Eigenschaft `geometryURL` der Komponenteninstanz `theLamp` abgebildet. Zudem können Eigenschaften innerer Komponenteninstanzen geblockt werden, indem sie nicht in die Schnittstelle der umgebenden Komponente aufgenommen werden. Der Typ der äußeren Eigenschaft muss mit dem Typ einer assoziierten inneren Eigenschaft übereinstimmen. In der grafischen Notation wird der Typ der äußeren Eigenschaft nicht mit abgebildet, da er sich aus dem Typ der inneren Eigenschaft ergibt. Ob eine äußere Eigenschaft auf eine innere Eigenschaft abgebildet werden kann, hängt außerdem von der Zugriffsmethode der inneren Eigenschaft ab. Tabelle 8.1 zeigt, welche Zugriffsmethoden miteinander kombinierbar sind.

Eine veränderbare und nicht lesbare äußere Eigenschaft kann auch auf mehrere innere veränderbare Eigenschaften abgebildet werden, wenn die inneren Eigenschaften den gleichen Wertetyp besitzen. Für lesbare Eigenschaften ist dies nicht möglich, da sonst nicht

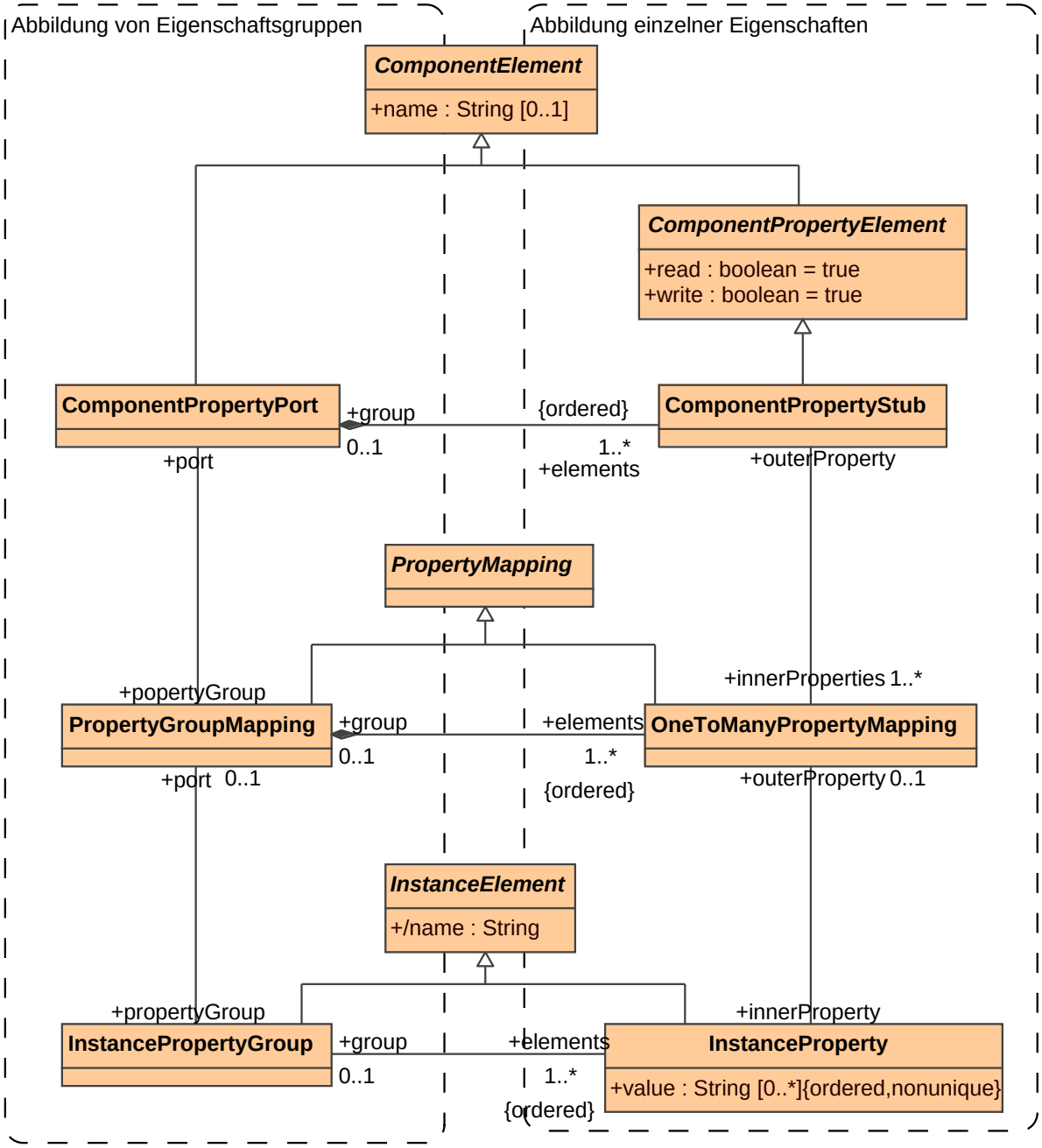



Abbildung 8.12: Abbildung von Komponenteneigenschaften auf Komponenteninstanzeigenschaften

eindeutig hervorgeht, von welcher inneren Eigenschaft der Wert gelesen werden soll.

Ein `ComponentInstance`-Modellelement kann innerhalb eines abgeschlossenen Kompositionsbereiches selbst mehrere *Laufzeitinstanzen* besitzen; es ist ein `MultiInstanceEnabledElement` (vgl. Abbildung 8.6). Daher kann für die Abbildung eines Komponentenelements auf Elemente einer inneren Komponenteninstanz über das optionale `String`-Attribut `whichRunTimeInstances` eines `MappingModelElements` (s. Abbildung 8.2, Attribute nicht dargestellt) in Form von Instanzselektoren angegeben werden, welche Laufzeitinstanzen der Komponenteninstanz adressiert werden. Ein denkbares Beispiel für die Anwendung von `whichRunTimeInstances` wäre folgendes: Eine Komponente `Scooter` beinhaltet eine Komponenteninstanz `wheel` mit einem Verhaltens-Port `rotate`, von welcher zur Laufzeit zwei Kopien existieren (`wheel[0]` und `wheel[1]`). Die Komponente `Scooter` besitzt einen Verhaltensstumpf `rotateFirstWheel`, der auf den Instanzverhaltensport `rotate` der Komponenteninstanz `wheel` abgebildet wird. Durch Angabe des Instanzselektors *[0]* für das Attribut `whichRunTimeInstances` des `BehaviourMapping`-Elements, welches `rotateFirstWheel` mit `rotate` verbindet, wird sichergestellt, dass zur Laufzeit über die Schnittstelle `rotateFirstWheel` nur auf die erste Kopie von `wheel` (also `wheel[0]`) zugegriffen wird.

Wurde ein lesender Zugriff auf Eigenschaften mehrerer Laufzeitinstanzen einer `ComponentInstance` modelliert (d.h. `whichRunTimeInstances` adressiert mehrere Laufzeitinstanzen), müssen bei der Durchführung eines Lesezugriffs zur Laufzeit über die Schnittstelle der kapselnden Komponente die Werte der Eigenschaften aller adressierten Kopien als eine geordnete Wertefolge (Wert von Kopie 1, Wert von Kopie 2, usw.) zurückgegeben werden.

#### 8.3.2.3 Einordnen von Komponenteninstanzen in die Szenengraphstruktur

Abbildung 8.13 zeigt die zusammengesetzte Struktur der Komponente `ScooterMainUnit` des Motorroller-Beispiels. Der Aufbau der Komponente ist relativ komplex, da er eine Vielzahl von Möglichkeiten der Komponentenmodellierung mit SSIML/Components demonstrieren soll. In der Praxis ist es zwecks besserer Übersichtlichkeit und Verwaltbarkeit eines visuellen Komponentenmodells allerdings sinnvoll, Komponenten mit zahlreichen inneren Elementen in weniger komplexe Unterkomponenten oder Kompositionsknoten zu zerlegen. Die Komponente `ScooterMainUnit` beinhaltet eine Gruppe `scooterMainGroup`, die weitere Komponenteninstanzen enthält. Die Instanzen repräsentieren die vorderen und hinteren Blinklichter des Motorrollers (`leftFrontWinkerLamp`, `rightFrontwinkerLamp`, `leftRearWinkerLamp`, `rightRearWinkerLamp`), sowie dessen Vorder- und Rückleuchte (`headLamp`, `rearLamp`) und den Rumpf des Motorrollers einschließlich aller weiteren Rollerkomponenten (`body`). `body` besitzt eine Eigenschaft `bodyColour`, die es erlaubt, die Farbe des Motorrollers anzupassen. Weiterhin enthält die Komponente `ScooterMainUnit` mehrere Verhaltensobjekte: ein Objekt (`toggleMainLight`) kontrolliert das simultane An- und Ausschalten des Vorder- und des Rücklichts, zwei weitere Objekte (`toggleLeftWinker` und `toggleRightWinker`) sind für das gleichzeitige Blinken der rechten oder linken Blinkleuchten (hinten und vorn) verantwortlich.

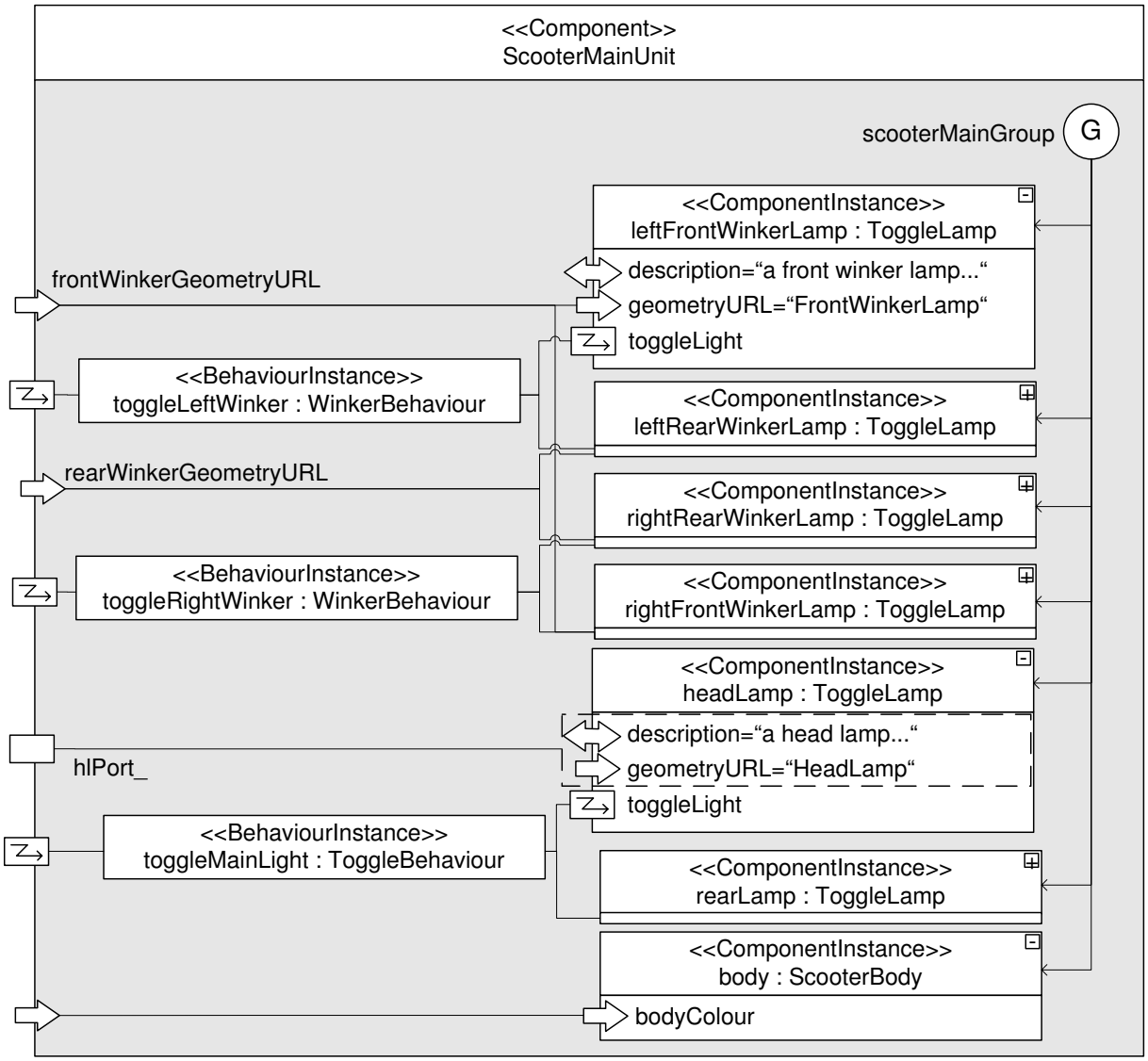



Abbildung 8.13: Die Komponente `ScooterMainUnit`

#### 8.3.2.4 Eigenschaftsgruppen und -ports

In Abbildung 8.13 ist zu sehen, dass die Eigenschaften von `headLamp` eine Eigenschaftsgruppe bilden, da beide Eigenschaften durch eine unterbrochene Linie umrahmt sind. Die Eigenschaftsgruppe ist mit einem so genannten Komponenteneigenschaftsport (`ComponentPropertyPort`, dargestellt durch ein Rechteck auf dem Rahmen der Komponente `scooterMainUnit`) verbunden. Ein Eigenschaftsport stellt eine Gruppe von Eigenschaften einer inneren (in einer Komponente gekapselten) Komponenteninstanz für den externen Zugriff – ggf. unter neuen Namen – zur Verfügung. In der *Instanzansicht* einer Komponente wird nicht der Eigenschaftsport selbst dargestellt, sondern die Eigenschaftselemente, die der Port enthält. Demnach ist die Zugehörigkeit einer Eigenschaft zu einem Port nur in der *Definitionsansicht* einer Komponente – wie in der Definitionsansicht der Komponente `ScooterMainUnit` in Abbildung 8.13 – erkennbar.

Abbildung 8.12 (linker Teil) verdeutlicht den Zusammenhang zwischen Komponenteneigenschaftsports (`ComponentPropertyPorts`) und Eigenschaftsgruppen von inneren Komponenteninstanzen (`InstancePropertyGroups`) im Metamodell. Ein `ComponentPropertyPort` enthält für jedes Element, welches aus der – dem Port (über ein `PropertyGroupMapping`) zugeordneten – `InstancePropertyGroup` stammt, einen entsprechenden `ComponentPropertyStub` mit identischen Zugriffsrechten (Abbildung 8.12 oben). Die Zuordnung eines *Elements* einer `InstancePropertyGroup` zu einem *Stub* eines Ports wird in einem `OneToManyPropertyMapping`-Element (hier in Form einer 1:1-Abbildung) gespeichert. Die Stubs eines Ports und die entsprechenden `OneToManyPropertyMapping`-Elemente werden in der konkreten Notation einer Komponente nicht dargestellt (s. z. B. die mit dem Port `hlPort_` verknüpfte Eigenschaftsgruppe in Abbildung 8.13).

Instanzen einer Komponente können auch vereinfacht dargestellt werden. Bei einer solchen vereinfachten Darstellung werden Eigenschaften der Instanz nicht explizit angezeigt. Bspw. wurden in Abbildung 8.13 die Eigenschaften von `leftRearWinkerLamp` ausgeblendet; trotzdem bleibt ersichtlich, welche Eigenschaften der kapselnden Komponente auf Eigenschaften der gekapselten Komponenteninstanz abgebildet werden.

Eine Eigenschaft kann mehr als einmal im Modell initialisiert werden. Die Regel, aus der sich letztendlich der Initialwert einer Eigenschaft ableitetet, ist folgende: Hat eine Komponenteneigenschaft einen komplexen Typ, werden die Standardwerte des komplexen Typs durch die Initialwerte der Komponenteneigenschaft ersetzt. Die Initialwerte einer Komponenteneigenschaft werden ggf. durch die Initialwerte einer Instanz der Komponente ersetzt. Bspw. würde der Initialwert der Eigenschaft `description` von `headLamp` in Abbildung 8.9 den Initialwert der Eigenschaft `description` der Komponente `Lamp`, sofern ein solcher definiert wäre, ersetzen. Enthält eine Komponenteninstanz (die äußere oder kapselnde Instanz) eine andere Komponenteninstanz (die innere oder gekapselte Instanz) und wird eine Eigenschaft der äußeren Instanz auf eine Eigenschaft der inneren Instanz abgebildet, so ersetzt der Initialwert der Eigenschaft der äußeren Instanz denjenigen der inneren Instanz.

#### 8.3.2.5 Benennung von Stumpf- und Port-Elementen

Bei der Benennung von Elementen von Komponenten muss in jedem Fall sichergestellt werden, dass ein `InstanceElement` innerhalb der Menge aller Instanzelemente ein und derselben `ComponentInstance` einen eindeutigen Namen besitzt.

Die Vergabe eines Namens für ein `ComponentProperty`-Element ist obligatorisch. Ein `ComponentPropertyStub`-Element hingegen muss dann explizit benannt werden, wenn es auf mehrere `InstanceProperty`-Elemente abgebildet wird. Andernfalls wäre nicht eindeutig, von welchem der `InstanceProperty`-Elemente der Stumpfname zu übernehmen ist. Wird ein `ComponentPropertyStub` nur auf eine `InstanceProperty` abgebildet, kann der Name dieser für den Stumpf implizit übernommen werden, wenn dadurch die eingangs in diesem Abschnitt genannte Bedingung nicht verletzt wird.

Die Angabe eines `ComponentPropertyPort`-Namens ist optional. Wird explizit ein Name festgelegt (z. B. `hlPort_` in Abbildung 8.13), dann trägt eine `InstanceProperty`, die über den Port zur Verfügung gestellt wird (also Instanz einer der Eigenschaftsstümpfe des Ports ist), den Namen des Ports, verbunden mit dem entsprechenden Stumpfnamen (z. B. `hlPort_description`). Besitzt der Port hingegen keinen Namen, werden die `InstanceProperty`-Elemente, die der Port umfasst, nur nach den entsprechenden Eigenschafts-Stümpfen benannt. Auch hier gilt, dass die oben genannte Bedingung nicht verletzt werden darf.

Analog verhält es sich bei Komponentenverhaltenselementen (Typ `ComponentBehaviourElement`). Während einem `ComponentBehaviourPort` explizit ein Name zugeordnet werden muss, kann ein `ComponentBehaviourStub`-Name von dem Element übernommen werden, auf das der Port abgebildet wird (d.h. ein Element vom Typ `Behaviour` oder `InstanceBehaviourPort`), wenn es sich um eine 1:1-Abbildung handelt und die eingangs genannte Bedingung nicht verletzt wird.

### 8.3.3 Interrelationenmodell

In Abbildung 8.14 ist das Interrelationenmodell der Motorroller-Anwendung zu sehen. Der Motorroller ist in zwei Haupteinheiten unterteilt: Die Lenkeinheit (`handleBarsUnit`) und die Haupteinheit (`mainUnit`). Die Lenkeinheit kapselt den Lenker, die Frontgabel und das Vorderrad. Die Komponenteninstanz `mainUnit` wird durch die Komponente `Scooter-MainUnit` definiert, die in Abschnitt 8.3.2.3 vorgestellt wurde. Die `SceneManager`-Klasse hat Zugriff auf die Transformation der Lenkeinheit, um Steuerungsbewegungen zu simulieren. Zusätzlich kann `SceneManager` direkt auf die Schnittstelle der Komponenteninstanz zugreifen. Dies wird durch eine gerichtete Kante, ausgehend vom `SceneManager,` verdeutlicht, die am Rand der Komponente endet.

Es ist auch möglich, eine Klasse (d. h. ein SSIML-`ApplicationClass`-Element) mit einer oder mehreren Eigenschaften einer Komponenteninstanz zu assoziieren (s. Auschnitt aus dem Metamodell in Abbildung 8.15). Dies bedeutet nicht, dass der Zugriff für die Klasse prinzipiell auf bestimmte Eigenschaften der Komponenteninstanz beschränkt ist. Allerdings wird damit ausgedrückt, dass die Klasse ausschließlich die mit ihr assoziier-

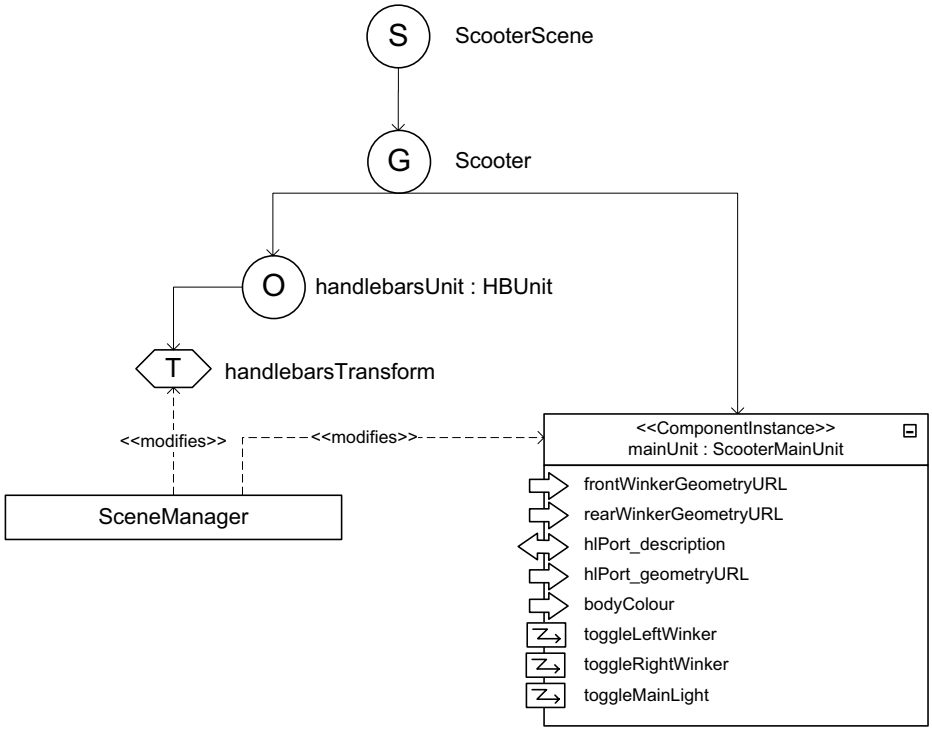



Abbildung 8.14: Das Interrelationenmodell der Motoroller-Anwendung

ten Eigenschaften der Komponenteninstanz benutzt. Dies kann bei der Übersetzung des Modells zu Code ggf. entsprechend berücksichtigt werden.

An dieser Stelle sei angemerkt, dass Beziehungen vom Typ `NodeActionRelationship` (vgl. Abschnitt 5.3.1.1) nicht in Verbindung mit Komponenteninstanzen verwendet werden dürfen, da eine Komponenteninstanz weder das Hinzufügen noch das Entfernen von Kindknoten erlaubt. Zudem ist auch ein Austausch des Inhalts der Komponenteninstanz im Gegensatz zum Austausch des Inhalts eines Atom-Knotens nicht möglich, da auf den Inhalt der Komponenteninstanz nur über ihre Schnittstelle zugegriffen werden kann.

Existieren zur Laufzeit mehrere Kopien einer `ComponentInstance`, zu denen eine Klasse Zugriff hat, bedeutet dies nicht, dass die Klasse parallel auf alle zugreift. Vielmehr besitzt sie die Möglichkeit, zu einem bestimmten Zeitpunkt genau auf eine der `ComponentInstance`-Kopien zuzugreifen. Dies muss bei der Generierung des Programmcodes für die Klasse entsprechend berücksichtigt werden.

## 8.4 Notation

Die Tabellen 8.2, 8.3 und 8.4 geben noch einmal einen Überblick über die Notation wichtiger Elemente von SSIML/Components.

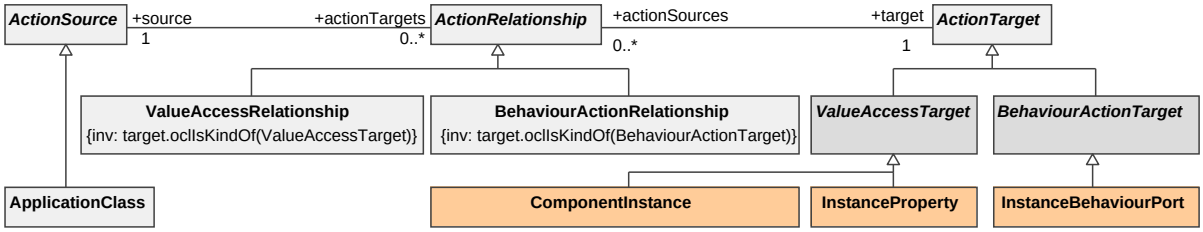



Abbildung 8.15: Zugriff auf Elemente von Komponenteninstanzen durch Anwendungsklassen

Tabelle 8.2: Notation von komplexen Typen, Komponenten und Komponenteninstanzen

| Elementtyp | | Notation | Beispiel |
|---|---|---|---|
| ComplexType | | '<<ComplexType>>' <Name>'('<Wertzuweisungsmuster>')' <br> <ComplexType-Elemente> | <<ComplexType>> Brightness(value) <br> value: float <0→1> {0.3} |
| Component | Mit Inhalt | '<<Component>>' <Name> <br> <Komponenten-Elemente> <br> <Komponenten-Inhalt> | <<Component>> MyComponent <br> initOnlyProperty : Brightness {0.6} |
| | Ohne Inhalt | '<<Component>>' <Name> <br> <Komponenten-Elemente> | |
| Component-Instance | offen | '<<ComponentInstance>>' <Name>':'<Typ> [-] <br> <Instanz-Elemente> | <<ComponentInstance>> myInstance:MyComponent [+] |
| | ge-schlossen | '<<ComponentInstance>>' <Name>':'<Typ> [+] | |

Tabelle 8.3: Notation von Elementen von komplexen Typen, Komponenten und Komponenteninstanzen

| Besitzer | Elementtyp | Notationsteil/ Zugriffsmeth. | Notation | Beispiel |
|---|---|---|---|---|
| ComplexType | ComplexType-Element (CTE) | Name und Typ | <CTE-Name>':'<CTE-Typ> | transparency:float <0→1>{0.5} |
| | | Wertebereich | ('<'(<Minimalwert>'→' <MaximalWert>)\| (<Wert>(','<Wert>)*)'>')? | |
| | | Standardwert | ('{'<Standardwert>'}')? | |
| Component | Component-Property (CP) | Read | ⇦ <CP- Name>':'<CP-Typ>('{'<Initialwert>'}')? | ⇔model3D:String {“file:///C:/model.x3d”} |
| | | Write | ⇨ | |
| | | ReadWrite | ⇔ | |
| | Component-PropertyStub (CPS) | Read | ⇦ (<CPS- Name>)? | ⇔my3DModel |
| | | Write | ⇨ | |
| | | ReadWrite | ⇔ | |
| | Component-BehaviourPort (CBP) | - | <CBP-Name> | toggleLightBehaviour |
| | Component-BehaviourStub (CBS) | - | (<CBS-Name>)? | toggleLightBehaviour |
| | Component-PropertyPort (CPP) | - | (<CPP-Name>)? | myProps |
| Component-Instance | Instance-Property (IP) | Read | ⇦ <IP-Name>('='<IP-Wert>)? | ⇔my3DModel= “file:///C:/model2.x3d” |
| | | Write | ⇨ | |
| | | ReadWrite | ⇔ | |
| | Instance-Property-Group (IPG) | - | <IPG> | ⇔my3DModel ⇨ transparency=0.7 |
| | Instance-Behaviour-Port (IBP) | - | <IBP-Name> | toggleLightBehaviour |

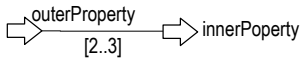
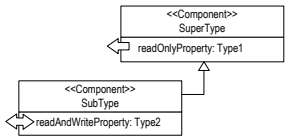
## 8.5 Modell-Code-Abbildung

Wie bereits in den vorangehenden Kapiteln soll als Zielformat der Codeabbildung von SSIML-Komponenten beispielhaft das Format X3D dienen. Als X3D-Codierung wird an dieser Stelle die klassische VRML-Codierung (*Classic VRML Encoding* [Int05b]) verwendet.

### 8.5.1 Abbildung von SSIML-Datentypen

Die Abbildung einfacher SSIML-Datentypen (*Simple Types*) auf entsprechende X3D Datentypen ist relativ einfach. Bspw. kann ein SSIML-`String` direkt in einen X3D-`SFString` übersetzt werden – oder in einen X3D-`MFString`, falls `isList` gilt.

Tabelle 8.4: Notation wichtiger Beziehungs-Typen in SSIML/Components

| **Beziehungsart** | **Start-Element-Art** | **Ziel-Element-Art** | **Notation** | **Beispiel** |
|---|---|---|---|---|
| Abbildung (Mapping) | Komponenten-Stub-Element, Komponenten-Port-Element | Komponenten-Instanz-Element, Behaviour | (<Ziel-Laufzeit-Instanzen>)? |   |
| Erweiterung (Extension) | Komplexer Typ/ Komponente (Subtyp) | Komplexer Typ/ Komponente (Supertyp) | |   |

Komplexe SSIML-Typen können auf X3D-Prototypen abgebildet werden. Es gibt sowohl komplexe Typen, die direkt zu X3D-Datentypen konvertiert werden können (ein Beispiel wäre ein komplexer Typ `RGBColour`, da für RGB-Farben ein entsprechender X3D-Typ existiert), als auch solche, bei denen dies nicht möglich ist (z. B. `HSBColour`).

Um einen Code auf der Zielplattform zu erhalten, welcher der Bedeutung eines bestimmten komplexen Typs optimal entspricht, müssen für jeden neu definierten komplexen Typ entweder die Modell-zu-Code-Transformationsbeschreibungen (z. B. XSLT-Stylesheets) oder der generierte Code selbst angepasst werden. Die Anpassung der Transformationsbeschreibung für einen komplexen Typ ist insbesondere dann lohnenswert, wenn dieser universell einsetzbar ist oder häufig benötigt wird. Dies würde bspw. für den komplexen Typ `HSBColour` zutreffen. Ohne Anpassung der Transformationsbeschreibung kann ein komplexer Typ nach generischen Regeln abgebildet werden, wobei allerdings eine nachträgliche manuelle Vervollständigung des Codes i. d. R. notwendig ist. Wie ein nach generischen Regeln für einen komplexen Typ generierter X3D-Code aussehen kann, soll nachfolgend am Beispiel des komplexen Typs `HSBColour` (vgl. Abbildung 8.5, Abschnitt 8.3.1) erläutert werden.

```
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;
            }
          }
    "]
    ...
```

Der Codeausschnitt oben zeigt die Abbildung des komplexen Typs `HSBColour` auf X3D-Code. `HSBColour` wird in X3D durch einen Prototyp gleichen Namens repräsentiert. Da Prototypen andere Prototypen nicht erweitern (spezialisieren) können, werden auch Elemente, die `HSBColour` von anderen SSIML-Komponenten wie `Brightness` erbt, mit in die Prototypschnittstelle übernommen. Alle Elemente des komplexen Typs werden in entsprechende Felder der Prototypschnittstelle übersetzt. Besteht ein komplexer Typ bspw. aus mehreren SSIML-`Float`-Elementen, werden diese Elemente auf entsprechende `SFFloat`-Felder der Prototypschnittstelle abgebildet. Ein komplexer Typ spezifiziert keinerlei Zugriffsbeschränkungen für seine Elemente; daher wird für jedes Element des komplexen Typs ein `initializeOnly`, `inputOnly` and `outputOnly`-Feld in der Schnittstelle des Prototyps generiert. Für das SSIML-`HSBColour`-Element `brightness` sind dies die X3D-Felder `brightness`, `set_brightness` und `brightness_changed`. Die Erzeugung von drei separaten Feldern anstatt eines `inputOutput`-Feldes ist notwendig, um die einzelnen Felder mit Hilfe von `IS`-Anweisungen mit Skript-Funktionen verbinden zu können, etwa um Eingabewerte zu validieren. Ein `initializeOnly`-Feld wird mit dem Wert instanziiert, der ggf. im SSIML-Modell für das dem Feld entsprechende Element definiert wurde. Da im Beispiel für die Elemente von `HSBColour` zusätzlich Wertebereiche festgelegt wurden, muss zur Laufzeit geprüft werden, ob die den Prototypfeldern zugeordneten Werte innerhalb der vereinbarten Wertebereiche liegen. Dies wird durch die Generierung von Ecmascript[Int02]-Code in den Rumpf des Prototyps realisiert. Als Beispiel wird die Validierung des Wertes von `brightness` (also des Helligkeitswertes) demonstriert.

Im Falle einer Initialisierung mit einem unerlaubten Wert wird das Feld `brightness` durch ein Skript auf den Basiswert zurückgesetzt (Funktion `initialize`). Das Setzen eines unerlaubten Wertes zur Laufzeit wird ebenfalls verhindert. Stattdessen wird der letzte gültige Wert beibehalten (Funktion `set_brightness`). Bei einer Modifizierung des `brightness`-Wertes wird eine Ereignisnachricht generiert und an Eingabefelder von X3D-Knoten geleitet, die mit dem Ausgabefeld `brightness_changed` der entsprechenden Instanz von `HSBColour` verbunden wurden.

### 8.5.2 Abbildung von SSIML-Komponenten

Wie komplexe Typen können auch SSIML-Komponenten auf X3D-Prototypen abgebildet werden. Der folgende Codeausschnitt zeigt die Schnittstelle des aus der Komponente `Lamp` (siehe Abschnitt 8.9) generierten Prototyps.

```
PROTO Lamp [
  inputOutput MFString 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) {

        }
      "]
...
```

`Lamp` besitzt eine Eigenschaft eines einfachen Typs: `geometryURL`. Da diese Eigenschaft keine Beschränkungen bzgl. der erlaubten Zugriffsmethode aufweist, kann sie in ein `inputOutput`-Feld der X3D-Prototypschnittstelle übersetzt werden. Die SSIML-Komponenteneigenschaft `lightColour` hat hingegen den komplexen Typ `HSBColour`. Der SSIML-Eigenschaft `lightColor` entspricht in der Prototypschnittstelle das Feld `lightColour`, welches mit dem korrespondierenden Wert aus dem SSIML-Modell initialisiert wird. Da `lightColour` im SSIML-Modell die Zugriffsmethode *Write* besitzt, muss ein zusätzliches Eingabefeld `set_lightColour` erzeugt werden.

Der Rumpf des Prototyps enthält einen Gruppenknoten, der selbst alle anderen Knoten beinhaltet. Abgesehen von Feldern einfachen Typs ohne Zugriffsbeschränkung, wie `geometryURL`, wird für alle Elemente der Prototypschnittstelle ein zusätzlicher Skript-Knoten

Tabelle 8.5: Abbildung von SSIML-Zugriffsmethoden auf X3D-Felder

| Zugriffsmethode | Simple Type | Complex Type |
|---|---|---|
| *Init* | initializeOnly | initializeOnly |
| *Read* | initializeOnly + outputOnly | initializeOnly + outputOnly |
| *Write* | initializeOnly + inputOnly | initializeOnly + inputOnly |
| *ReadWrite* | inputOutput | inputOnly + initializeOnly + outputOnly |

im Prototyprumpf generiert. Abhängig von den Zugriffsbeschränkungen einer Eigenschaft einer SSIML-Komponente, realisieren die Skripte die Lese- und Schreiboperationen für entsprechende Feldwerte des X3D-Prototyps. Im obigen Beispiel wird das Skript `lightColourScript` für das Schreiben des modifizierbaren (aber nicht lesbaren) Wertes des Feldes `lightColour` gezeigt. Die Felder des Skriptknotens sind durch `IS`-Anweisungen mit den jeweiligen Prototypfeldern verbunden. Das Initialisierungsskriptfeld `_lightColour` wird dabei mit dem Prototypfeld `lightColour,` und das Skriptfeld `set__lightColour` mit dem Feld `set_lightColour` der Prototypschnittstelle verbunden. Während `_lightColour` zur Initialisierung der Eigenschaft `lightColour` dient, wird `set__lightColour` für dynamische Änderungen der Lichtfarbe zur Laufzeit verwendet.

Wäre `lightColour` im SSIML-Modell eine Komponenteneigenschaft mit Schreib- *und* Lesezugriff, müssten zusätzliche Felder zum Lesen des Wertes in der Prototypschnittstelle und für den Skriptknoten erzeugt werden. Tabelle 8.5 verdeutlicht noch einmal die Abbildung von Zugriffsmethoden auf X3D-Felder für einfache und komplexe SSIML-Datentypen.

Das `url`-Feld des Skriptknotens enthält die benötigten Funktionen. Die Funktion `initialize` wird durch die X3D-Laufzeitumgebung aufgerufen. Durch diese Funktion werden die Initialwerte von `lightColour` über die Funktion `generateOutput` an Knoten ausgegeben, die mit dem Skript innerhalb des Prototyprumpfes verbunden sind. Die Funktion `set__lightColour` hat eine analoge Funktion wie `initialize`. Allerdings wird sie zur Laufzeit erst nach dem Zeitpunkt der Initialisierung aktiviert, wenn der `lightColour`-Wert einer Instanz des Prototyps `Lamp` über das Feld `set_lightColour` von außen geändert werden soll.

Die generierten Codegerüste für die X3D-Prototypen müssen nachträglich vervollständigt werden. Ein Programmierer kann verschiedene Ausgabefelder für ein Prototypskript definieren und die im obigen Quellcode dargestellte `generateOutput`-Funktion implementieren, welche die Ausgabefelder anspricht. Dies ermöglicht die Konvertierung von Ausgabedaten in ein Format, welches sich für die Weiterverarbeitung eignet. Im vorliegenden Beispiel könnte der Programmierer der Methode `generateOutput` eine Routine implementieren, die den Farbwert, der im HSB-Format im Feld `_lightColour` vorliegt, in das RGB-Format konvertiert. Das Resultat der Konvertierung könnte in einem X3D-`SFColor`-Objekt gespeichert und an die mit den Ausgabefeldern des Skripts verbundenen Eingabefelder anderer Knoten gesendet werden.

Die Geometrie des Lampenobjekts im vorliegenden Beispiel (Komponente `Lamp`) kann mittels eines geeigneten Autorenwerkzeuges oder Editors in den Rumpf des Proto-

typs integriert werden. Ausgabefelder des `lightColourScript`-Knotens können nach der Konvertierung nach RGB über `ROUTE`-Anweisungen mit Farbeingabefeldern von X3D-Lichtquellenknoten oder -Materialknoten verbunden werden.

Da Farbwerte für Materialen und Lichtquellen in X3D, wie bereits angedeutet, standardmäßig im RGB-Format definiert werden, bietet es sich an, den Code für die Konvertierung eines Wertes im HSB- in das RGB-Format, z. B. als Funktion des Prototyps `HSBColour`, mit in die Codegenerierungsroutinen zu integrieren. Damit läßt sich der Implementierungsaufwand weiter verringern und Programmierfehlern wird vorgebeugt.

SSIML-Komponenteninstanzen werden auf Instanzen der entsprechenden X3D-Prototypen abgebildet. Bspw. würde `headLamp` aus Abbildung 8.9 in eine Instanz des Prototyps `Lamp` übersetzt.

Zusammengesetzte SSIML-Komponenten werden zu X3D-Prototypen transformiert, die neben X3D-Knoten Instanzen anderer Prototypen enthalten können. Bspw. enthielte ein X3D-Prototyp `ScooterMainUnit` (Abbildung 8.13) Instanzen des `ToggleLamp`-Prototyps.

Verhaltensobjekte entsprechen in X3D ebenfalls Instanzen spezieller Prototypen (vgl. Abschnitt 7.3). Ein X3D-Prototyp, der eine SSIML-Komponente repräsentiert, kann ein solches X3D-Verhaltensobjekt kapseln. Der Prototyp muss entsprechende Felder in seiner Schnittstelle zur Verfügung stellen, die mit den Feldern des X3D-Verhaltensobjektes verbunden sind. Bspw. würde der Prototyp `ScooterMainUnit` ein Eingabefeld `toggleMainLight_engineStarted` bereitstellen, welches mit einem Eingabefeld `engineStarted` des X3D-Verhaltensobjektes `toggleMainLight` verknüpft wäre. Für einen ausführlicheren Überblick über die Verhaltensbeschreibung mit SSIML sei auf Kapitel 7.1 verwiesen.

## 8.6 Verwandte Arbeiten

Die Verwendung von Software-Komponenten im Bereich interaktiver 3D-Grafik war bereits Gegenstand umfangreicher Untersuchungen. Einige dieser Arbeiten, wie NPSNET-V [CMBZ00] oder MOVE [GMP+02], konzentrieren sich auf komponentenbasierte Architekturen zur Erstellung von erweiterbaren kollaborativen und verteilten virtuellen Umgebungen. Betrachtet werden dabei typischerweise Komponenten, welche die verteilte Kommunikation zwischen verschiedenen Benutzern innerhalb einer virtuellen Umgebung unterstützen.

In der vorliegenden Arbeit liegt der Schwerpunkt allerdings auf *3D*-Komponenten, d.h. Komponenten, die sowohl 3D-Geometrie als auch Verhalten von 3D-Objekten kapseln.

Einige klassische Arbeiten beschäftigen sich mit der Kapselung und Modularisierung von Verhalten und 3D-Geometrien. Bspw. existieren im Actor/Scriptor-Animationssystem (ASAS) [Rey82] so genannte Darsteller (*Actors*), die Animations- und Verhaltensbeschreibungen kapseln. Conner et al. prägen den Begriff der *3D-Widgets* [CSH+92]. 3D-Widgets werden in Analogie zu 2D-Widgets (2D-GUI-Komponenten) als eine spezielle Art von 3D-Komponenten betrachtet, die Geometrie und Verhalten kapseln und dazu verwendet werden, andere Anwendungsobjekte zu kontrollieren oder Informationen über sie anzuzeigen.

Ansätze wie CONTIGRA [DHM02, Dac04] und 3D Beans [DG00] erlauben die Definition von beliebigen 3D-Komponenten auf Implementierungsebene. Im Gegensatz zu

CONTIGRA, welches ein XML-Dialekt zur Spezifizierung von 3D-Komponenten und 3D-Applikationen ist, basieren die 3D Beans auf der Java-Beans-Technolgie [Jav].

Die so genannten *Node Kits* des Szenengraph-APIs Open Inventor [SC92] stellten bereits 1992 einen Ansatz dar, komponentenartige, instanziierbare Module, die Teilszenengraphen kapseln und gleichzeitig eine gezielte Parametrisierung dieser ermöglichen, zu definieren. Bei den Node Kits handelt es sich um eine auf der objektorientierten Programmiersprache C++ basierende Lösung.

Ebenso kann der Prototypmechanismus von VRML [Int97] oder X3D [Int04b] als eine Variante betrachtet werden, komponentenähnliche Einheiten auf der Implementierungsebene entweder in Textform oder mit Hilfe von graphischen Autorenwerkzeugen zu spezifizieren. Bspw. unterstützt das Autorenwerkzeug Internet Scene Assembler (ISA) der Firma Parallelgraphics [ISA] das Zusammensetzen von 3D-Szenen mittels so genannter *Objects*, die Geometrie und Verhalten kapseln und intern durch VRML-Prototypen repräsentiert werden. Erstellte, aus *Objects* zusammengesetzte 3D-Szenen können mit dem Werkzeug als neue *Objects* abgespeichert werden und in anderen Szenen wiederverwendet werden.

Wie bereits in Abschnitt 8.1 verdeutlicht wurde, kann die Entwicklung von Komponenten auf der Implementierungsebene eine zeitaufwendige Aufgabe sein, insbesondere bei Komponenten, die eine komplexe innere Struktur und ein kompliziertes Verhalten besitzen. Graphische Autorenwerkzeuge erleichtern die Aufgabe der Komponentenerstellung in gewissem Maße, haben aber oft Einschränkungen hinsichtlich der Beschreibungsmöglichkeiten für 3D-Komponenten. Ebenso wird die Integration der Komponenten in externe Anwendungen von diesen Werkzeugen in der Regel nicht unterstützt.

Der in der vorliegenden Arbeit vorgestellte Ansatz, SSIML/Components, geht im Gegensatz zu den oben erwähnten Ansätzen einen anderen, neuartigen Weg, indem er die Beschreibung von 3D-Komponenten auf einem höheren Abstraktionsniveau *mittels einer visuellen Modellierungssprache* ermöglicht. Visuelle Modellierungssprachen werden bei der Komponentenentwicklung im traditionellen Software-Engineering durchaus erfolgreich eingesetzt, wie bspw. die Unified Modeling Language (UML) [Obj07c] zeigt. SSIML/Components zielt – im Gegensatz zu den verwandten Arbeiten – auf einen Einsatz bereits beim Entwurf im Vorfeld der Implementierung ab. Ebenso kann die Verknüpfung von 3D-Komponenteninstanzen mit Bestandteilen einer Rahmenanwendung spezifiziert werden. Nicht zuletzt bietet SSIML/Components Erweiterungskonzepte, wie die Erweiterung von Komponenten durch Mehrfachvererbung, die über Konzepte in Beziehung stehender Ansätze hinausgehen.

## 8.7 Zusammenfassung

In diesem Kapitel wurde verdeutlicht, warum 3D-Komponenten die Entwicklung interaktiver 3D-Anwendungen beschleunigen und die Integration von 3D-Inhalten in Anwendungen erleichtern können. Zum anderen wurde auf den Mangel an Konzepten und Werkzeugen hingewiesen, die eine Spezifikation von 3D-Komponenten auf einem abstrakteren Niveau als der Implementierungsebene ermöglichen. Aus diesem Grunde kann sich insbesondere die

Erstellung von Komponenten mit komplexen inneren Strukturen und vielen Eigenschaften auf der Implementierungsebene als zeitaufwendig erweisen.

Als Lösungsansatz für die dargestellten Probleme wurde in diesem Kapitel die visuelle Modellierungssprache SSIML/Components eingeführt, eine Erweiterung der Sprache SSIML, die in Kapitel 5 vorgestellt wurde. SSIML/Components erlaubt die visuelle Spezifikation von interaktiven 3D-Komponenten vor der Implementierung. Vorhandene Komponenten können zu komplexeren Komponenten über Mehrfachvererbung und Komposition zusammengesetzt werden. Ebenso ist eine Erweiterung von Komponenten durch das Hinzufügen neuer Eigenschaften möglich. SSIML/Components hilft auch bei der Aufteilung großer Szenen in einfacher handhabbare, einfacher wartbare und wiederverwendbare Bestandteile. Dabei finden die bereits von SSIML bekannten Mechanismen, Szenenelemente mit Anwendungslogik zu verknüpfen, weiterhin Anwendung. Im Gegensatz zu verwandten Ansätzen erlaubt SSIML/Components eine visuelle Spezifizierung von 3D-Komponenten und ihren Instanzen bereits im Vorfeld der Implementierung in der Entwurfsphase auf einem vergleichsweise hohen Abstraktionsniveau.

Zudem wurde in diesem Kapitel beispielhaft gezeigt, wie SSIML/Components-Modelle auf X3D-Code abgebildet werden können, um den Implementierungsaufwand zu verringern. Zusätzlich können die Komponentenmodelle zu Dokumentationszwecken und zur Diskussion der strukturellen Aufteilung einer 3D-Szene verwendet werden.

# Kapitel 9

# SSIML/AR

## 9.1 Einführung

Dieses Kapitel befasst sich mit der Unterstützung der Entwicklung von *Augmented Reality* - Anwendungen. Augmented Reality (AR) heißt soviel wie *Erweiterte Realität*. In einer AR-Umgebung wird die reale Welt mit virtuellen Objekten ergänzt. Zu unterscheiden davon ist die *Virtual Reality* (VR, Virtuelle Realität), bei der eine völlig künstliche Welt aus einem auf einem Computer gespeicherten Datenmodell erzeugt wird. Azuma [Azu97] charakterisiert AR als ein System mit drei grundlegenden Eigenschaften:

- AR kombiniert die reale und die virtuelle Welt.
- In AR erfolgt die Benutzerinteraktion in Echtzeit.
- In AR werden virtuelle Objekte räumlich (dreidimensional) mit realen Objekten verknüpft.

*Augmented Virtuality* (Erweiterte Virtualität), bei der reale Objekte in einer virtuellen Welt platziert werden, und *Diminished Reality* (Reduzierte Realität), bei der Informationen aus der realen Welt entfernt werden, werden von manchen Autoren als spezielle Formen von AR angesehen. Milgram und Colquhoun [MC99] prägen den Begriff *Mixed Reality* (MR, gemischte Realität) als ein Kontinuum zwischen der Realität und der Virtualität, welches auch AR enthält.

Der Begriff *Erweiterung* beschränkt sich im Kontext von AR nicht allein auf den Sehsinn. Die Erweiterung kann alle Sinne (Sehsinn, Hörsinn, Tastsinn usw.) betreffen. Bspw. wäre eine Audioanwendung denkbar, bei welcher der Benutzer einen Stereokopfhörer trägt, an dessen beiden Seiten zusätzlich Mikrofone angebracht sind. Eingehende Audiosignale könnten etwa mit künstlichen Raumklangsignalen gemischt werden, bevor sie über den Kopfhörer wiedergegeben werden. Eine Erweiterung des Tastsinns könnte bspw. die Simulation verschiedener Oberflächenstrukturen auf einer physikalischen Tischplatte realisieren.

Da AR ein vergleichsweise neues Forschungsfeld ist, wurde in den vergangenen Jahren viel Aufwand in AR-Basistechnologien wie Tracking und Rendering investiert. Hingegen

fehlt es noch immer an Konzepten und Werkzeugen, die eine effizientere Entwicklung von AR-Anwendungen unterstützen. Nach wie vor wird ein Großteil der AR-Anwendungen auf der Implementierungsebene entwickelt, wobei lediglich Lower-Level-Toolkits wie das ARToolkit [ART] zum Einsatz kommen. Diese Art der AR-Entwicklung kann zu einer zeitraubenden Tätigkeit werden, insbesondere dann, wenn umfangreichere Applikationen mit komplexeren AR-Benutzungsschnittstellen (*AR-User Interfaces*; kurz: *AR-UIs*) erstellt werden sollen. Ebenfalls wird durch die Low-Level-Programmierung die Wiederverwendung bereits vorhandener Bausteine zur Lösung von Problemen bei der AR-Entwicklung nicht unbedingt gefördert.

Wie erwähnt, sind Konzepte und Werkzeuge, die eine effizientere AR-Entwicklung ermöglichen, eher selten zu finden. In der traditionellen Softwareentwicklung wird UML als De-facto-Standard erfolgreich für einen abstrakten Entwurf vor der Implementierung eingesetzt. Allerdings ist die UML für den Entwurf von AR-Systemen nur in Grenzen geeignet. Insbesondere wäre für die Spezifikation des AR-UIs, einer der bedeutendsten Komponenten des AR-Systems, eine Erweiterung der UML sinnvoll, wenn nicht sogar notwendig. Ein Grund dafür ist, dass UML bis auf wenige Ausnahmen, wie die Akteure in einigen Diagrammtypen, keine explizite Unterscheidung zwischen realen (physikalischen) Objekten und virtuellen Objekten vorsieht. Eine solche Unterscheidung ist jedoch für die Modellierung von AR-UIs unabdingbar, wie in Abschnitt 9.1.3 noch verdeutlicht werden soll.

Neben der *Integration rein physikalischer Objekte in die Benutzungsschnittstelle* muss sich der AR-Entwickler den gleichen *Herausforderungen* stellen, die bereits in Kapitel 1 in Bezug auf interaktive 3D-Anwendungen angesprochen wurden, nämlich der *Erstellung von (möglicherweise umfangreichen) 3D-Inhalten* und deren Integration in die Gesamtanwendung. Im Fall einer AR-Anwendung müssen die virtuellen Inhalte über die Programmlogik mit realen Objekten verknüpft werden, damit ein *virtuelles Objekt korrekt in der realen Welt platziert* werden kann. Auch hier besteht die Problematik, dass *Programmcode und 3D-Inhalte fehlerfrei miteinander verdrahtet* werden müssen, obwohl sie u. U. von *verschiedenen Entwicklern* stammen, die *mit völlig unterschiedlichen Werkzeugen* arbeiten.

In diesem Kapitel wird deshalb eine Erweiterung der Modellierungssprache SSIML vorgestellt, die auf die Entwicklung von AR-Anwendungen im Allgemeinen und auf die Entwicklung von AR-UIs im Speziellen ausgerichtet ist. Diese Erweiterung wird SSIML/AR genannt. SSIML/AR wurde wie andere SSIML-Erweiterungen und SSIML selbst als MOF-konformes Metamodell definiert, welches zur besseren Integration in bestehende UML-Werkzeuge (im vorliegenden Fall MagicDraw) auf ein UML-Profile abgebildet wurde. SSIML/AR konzentriert sich auf die Modellierung von AR-Anwendungen aus dem Bereich Montage und Wartung, da dieser Fachbereich, wie im folgenden Abschnitt unterstrichen wird, zu denjenigen gehört, die die größten Potenziale für AR-Anwendungen besitzen. Weiterhin werden Konzepte und Mechanismen zur Abbildung von SSIML/AR-Modellen auf Code vorgestellt.

### 9.1.1 Anwendungsbereiche von AR

AR-Technologien wird großes Potenzial in verschiedenen (sich z. T. überlappenden) Anwendungsbereichen zugesprochen. Einer der Hauptanwendungsbereiche lässt sich durch die Begriffe *Montage und Wartung* bezeichnen. Diese Begriffe beziehen sich sowohl auf die Montage und Wartung von Maschinen als auch auf das Zusammensetzen von Gegenständen wie etwa Möbeln. So können dem Monteur durch das Einblenden virtueller Objekte in sein Sichtfeld Hinweise auf die Positionen gegeben werden, an denen Teile zu installieren sind. Beispiele für Forschungsarbeiten und Projekte aus diesem Bereich sind der *AR Furniture Assembly Instructor* [ZHBH03], ARVIKA [Fri04] und das AR-System von Boeing zum Zusammenfügen von Kabelsträngen [CMGJ99].

Ein weiterer sehr wichtiger AR-Anwendungsbereich ist die *Medizin*. Mit Hilfe von AR-Technologien könnte z.B. ein Embryo im Mutterleib [BFO92, SCT+94] visualisiert werden. Ebenso ist es möglich, chirurgische Eingriffe durch AR zu planen und zu unterstützen (vgl. z. B. [SLG+96, RBB+04, RRB+06]).

Auch der *Unterhaltungsbereich* bietet Möglichkeiten zum Einsatz von AR-Technologien. So wurden bereits einige AR-Spiele wie *AR-Quake* [TCD+02] und *The Invisible Train* [WPS04] umgesetzt.

Desgleichen wird AR im *Militärbereich* angewendet. Z. B. werden für die Piloten von Kampfjets erfasste Ziele durch die Projektion von virtuellen Objekten auf ihre Head-Up-Displays markiert. Das *Battlefield Augmented Reality-System* (BARS) [JBL+00] zielt besonders auf die Unterstützung von Soldaten bei Kampfhandlungen in urbanen Gebieten ab.

*Mobile AR-Tour-Guides* und *Navigationshilfen* führen den Benutzer zu einem bestimmten Ziel oder auf einem bestimmten Pfad über mehrere Zwischenstationen. Beispiele für derartige Guides sind Signpost [RS03], ein AR-Touristenführer für die Stadt Wien [RS04], die Touring Machine [FMHW97] oder Museumsführer [MAR]. Auch im *Automobilbereich* sind AR-Navigationsysteme Gegenstand aktueller Forschung und Entwicklung, etwa bei der Projektion von Navigationsinformationen wie Richtungspfeilen auf die Fahrzeugfrontscheibe [NPF+03, NPF+06]. Bei den AR-Tour-Guides (aber auch in anderen Anwendungsbereichen) wird häufig die *Annotierung realer Objekte* mit virtuellen Notizen genutzt, z. B. um Informationen zu Straßen, Gebäuden oder Ausstellungsstücken bereitzustellen.

In der *Architektur und Archäologie* sind AR-Technologien nutzbar, um noch nicht oder nicht mehr existente Bauwerke zu visualisieren, wie im Projekt ArcheoGuide [VIK+02], bei dem antike, zerstörte Bauwerke mittels AR-Technologien an ihrem ursprünglichen Standort in die reale Umgebung integriert werden. In der *Städtebauplanung* können mittels AR-UIs virtuelle Miniaturmodelle von Gebäuden über greifbare UIs (Tangible User Interfaces - TUIs) auf einem Tisch positioniert und ausgerichtet werden. Ein Beispiel für ein entsprechendes AR-System ist ARTHUR [BLO+04].

Das Projekt *Augmented Chemistry* [FV02] demonstiert, wie AR *zu Ausbildungszwecken* verwendet werden kann. In dieser Anwendung kann der Benutzer dreidimensionale molekulare Modelle zusammensetzen und mit ihnen interagieren.

In der *Roboter-Bewegungsplanung* können Tätigkeiten, die von Robotern ausgeführt werden sollen, mittels AR geplant und simuliert werden [Dra93, MZDG93].

### 9.1.2 Hardware und Technologien zur Realisierung von AR

Dieser Abschnitt soll einen kurzen Einblick in die AR-Basistechnologien geben. Von besonderem Interesse sind dabei die folgenden Fragen: Welche Hardware wird benötigt, um AR zu realisieren? Welche grundlegenden Möglichkeiten zur Verfolgung (dem *Tracking*) realer Objekte gibt es und welche der Möglichkeiten werden derzeit am häufigsten eingesetzt?

#### 9.1.2.1 AR-Displays

Die Überlagerung der realen Welt mit virtuellen Objekten kann auf verschiedenen Wegen erfolgen. Einige AR-Systeme verwenden herkömmliche Bildschirme als Ausgabegeräte (Desktop-PC-Monitore sowie Displays von mobilen Geräten wie Laptops, Tablet-PCs oder Handhelds), andere arbeiten mit Projektoren oder Head Mounted Displays (HMDs).

Insbesondere (aber nicht ausschließlich) bei HMD-basierter AR kommen zwei grundlegend verschiedene Hardwarearchitekturen zum Einsatz: *Optical-See-Through* und *Video-See-Through*.

Bei einem Optical-See-Through-System (Abbildung 9.1) ist der Benutzer in der Lage, seine reale Umgebung unmittelbar durch einen halbtransparenten Spiegel wahrzunehmen. Position und Blickrichtung des Benutzers werden verfolgt. Basierend auf den Trackingdaten wird eine virtuelle Szene gerendert. Diese wird in das Sichtfeld des Benutzers auf den halbdurchlässigen Spiegel projiziert, so dass der Benutzer das vom Spiegel reflektierte Bild wahrnehmen kann. Stereoskopie wird unterstützt, indem jeweils ein getrenntes Bild für das linke und das rechte Auge berechnet und entsprechend projiziert wird.

Bei einem Video-See-Through-System (Abbildung 9.2) sieht der Benutzer die Umgebung nicht unmittelbar, sondern über einen oder – bei Stereowiedergabe – zwei Monitore, auf denen gerade von einer Kamera aufgenommene Bilder angezeigt werden. Vor der Anzeige wird das aufgenommene Videobild mit der Ausgabe des 3D-Szenen-Renderers kombiniert. Ein Vorteil von Video-See-Through-Systemen ist, dass sich von der Kamera aufgenommene Bilder auch verwenden lassen, um Position und Orientierung realer Bezugsobjekte durch Bildanalyseverfahren zu extrahieren.

Video-See-Through-Systeme lassen sich auch mit herkömmlichen Displays realisieren; ihr Einsatz ist weniger auf HMDs oder Head-Up-Displays (HUDs) beschränkt, als dies bei Optical-See-Through-Systemen der Fall ist. Bei einem HMD-basierten Video-See-Through-System wird die Videokamera meist in Höhe der Augen am HMD befestigt. Es gibt sowohl HMDs, die mit einer Kamera, als auch HMDs, die mit zwei Kameras ausgestattet sind.

Sowohl Optical- als auch Video-See-Through-Systeme haben ihre Vor- und Nachteile, deren ausführlichere Diskussion an dieser Stelle zu weit führen würde. Nicht zuletzt hängt die Wahl des Systems vom Anwendungszweck ab. Für einen tiefgründigeren Einblick in diese Thematik sei auf [Azu97] und [ABB+01] verwiesen.

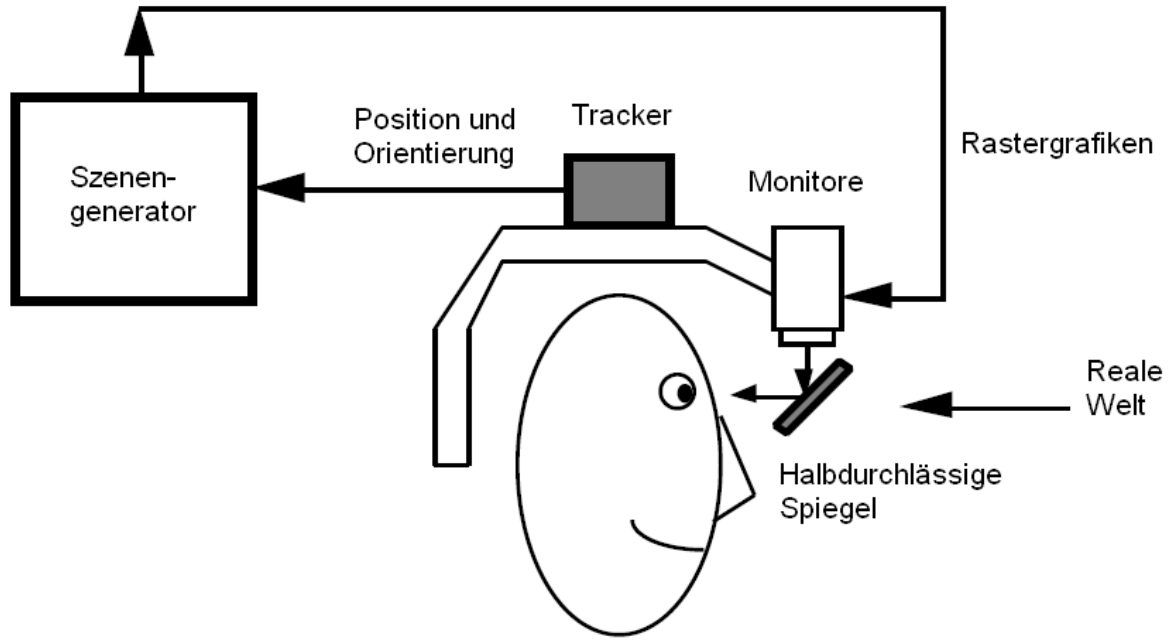



Abbildung 9.1: Optical-See-Through HMD (nach Azuma [Azu97])

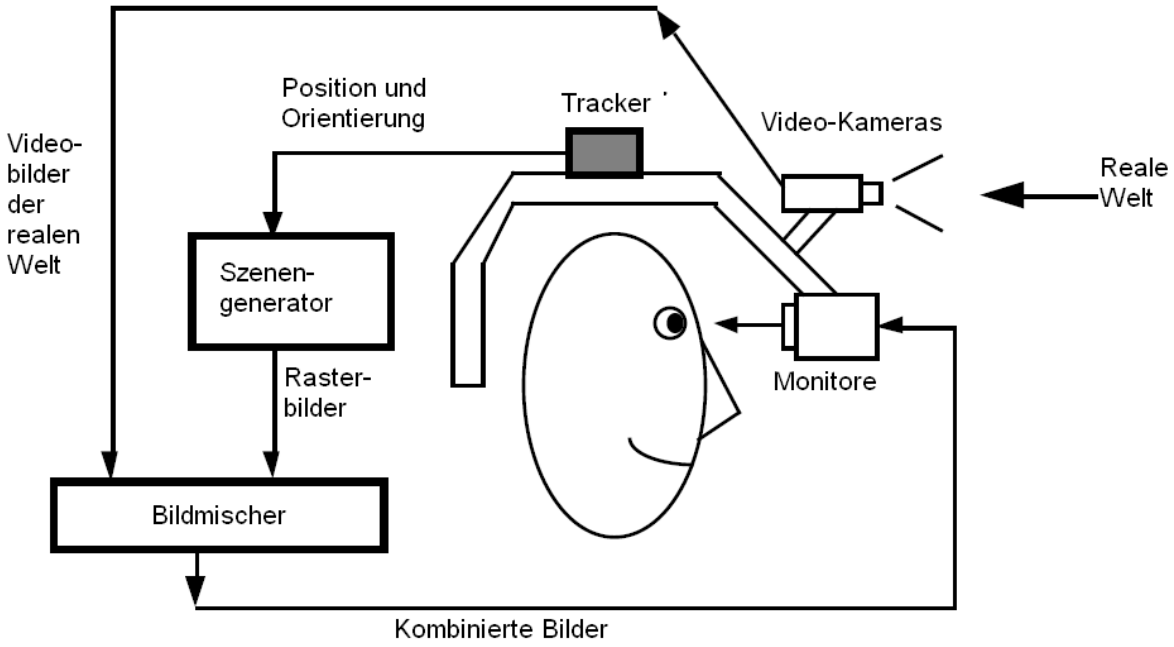



Abbildung 9.2: Video-See-Through HMD (nach Azuma [Azu97])

#### 9.1.2.2 Tracking

Um reale Objekte exakt mit den mit ihnen verknüpften virtuellen Objekten zu überlagern, muss ihre Position und Orientierung im Raum durch Trackingsysteme ermittelt werden. Neben mechanischen, magnetischen und Ultraschalltrackingverfahren hat sich insbesondere das markerbasierte Tracking etabliert. Dieses funktioniert in Systemen, die mit Kameras arbeiten. Durch Bildanalyse werden im Bild manuell angebrachte Markierungen (*Marker*) identifiziert, mit deren Hilfe eine weitere Bestimmung von Objektpositionen durchgeführt werden kann. Es wird zwischen passiven und aktiven Markern unterschieden. Aktive Marker sind bspw. Infrarot-LEDs. Passive Marker können etwa farbige Punkte oder Schwarzweißmuster sein. Anhand der Entfernung der Marker zur Kamera und der Ausrichtung der Marker (bei mehreren Markern auch durch die räumliche Relation der Marker zueinander) lässt sich auf Position und Orientierung der realen Objekte schließen. Das wohl am weitesten verbreitete AR-Framework, welches markerbasiertes Tracking unterstützt, ist das AR-Toolkit [ART]. Markerbasiertes Tracking stößt aber auch an Grenzen, z. B. wenn Marker verdeckt werden oder es nicht möglich ist, an einem Gegenstand einen Marker anzubringen (etwa bei kleineren Gegenständen). Einen vielversprechenden Ansatz stellt hier das so genannte *Natural-Feature-Tracking* bzw. *Markerlose Tracking* dar. Hierbei wird versucht, mit Hilfe fortgeschrittener Bildanalyseverfahren Orientierung und Position von Objekten völlig ohne künstlich angebrachte Marker aus den Frames eines Videostroms in Echtzeit zu extrahieren (vgl. z. B. [NY99]).

### 9.1.3 Konzepte zur Beschreibung von Elementen eines AR-UIs

Ein AR-UI enthält sowohl reale als auch virtuelle Elemente. Trevisan et al. sprechen auch von einem *Mixed Interaction Space*[1], der den *Real Interaction Space* und den *Virtual Interaction Space* umfasst [TGVM04]. Der *Mixed Interaction Space* beinhaltet alle virtuellen Elemente des *Virtual Interaction Space* und alle realen Elemente des *Real Interaction Space*, welche das AR-UI dem Benutzer zur Verfügung stellt, sowie die Beziehungen zwischen den Elementen.

Neben *rein realen* und *rein virtuellen Objekten* kann ein AR-UI auch Objekte enthalten, die sowohl eine *geometrische virtuelle* als auch eine *sichtbare reale* Komponente besitzen. Diese Objekte werden hier als *hybride Objekte* bezeichnet. Die virtuelle Komponente eines hybriden Objektes ist räumlich an dessen reale Komponente gebunden, was sich i. d. R. darin widerspiegelt, dass die virtuelle Komponente die reale stets (visuell) überlagert. Bewegt sich die reale Komponente, bewegt sich auch die virtuelle Komponente. Die virtuelle Komponente muss dabei nicht zwangsläufig exakt die gleiche Geometrie wie die reale Komponente besitzen. Bspw. kann die virtuelle Komponente auch als *Bounding Box* der realen Komponente dargestellt werden. Hybride Objekte dienen dazu, physikalische Objekte hervorzuheben oder sie mit digitaler Information visuell zu annotieren.

[1] *Interaction Space* wird verstanden als die Umgebung, die dem Benutzer zur Bearbeitung von Aufgaben bereitsteht.

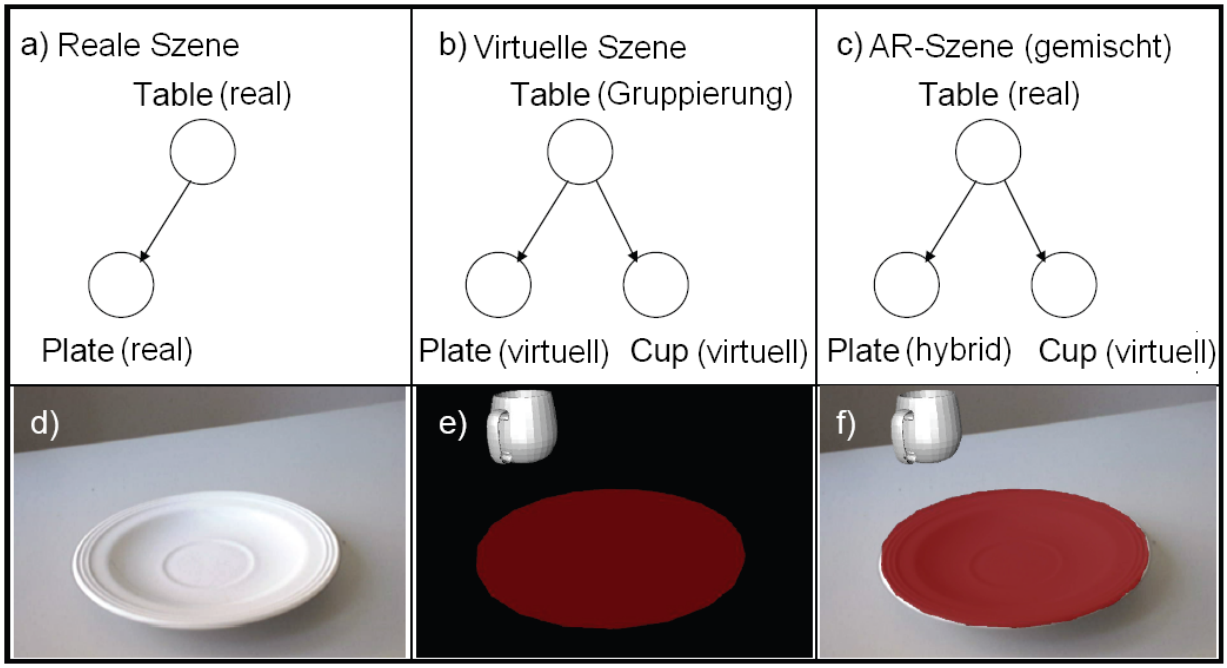



Abbildung 9.3: Zusammensetzung eines AR-Szenenmodells

Ebenfalls kann es reale Objekte geben, die zwar nicht unmittelbar mit einem konkreten geometrischen Objekt im virtuellen Raum assoziiert sind, aber trotzdem Pendants im virtuellen Raum besitzen. Ein solches reales Objekt kann im virtuellen Raum einer Gruppierung von virtuellen Objekten, also einem zusammengesetzten virtuellen Objekt entsprechen.

In SSIML/AR werden das in Kapitel 5 eingeführte Szenenmodell und das im selben Kapitel diskutierte SSIML-Interrelationenmodell um AR-spezifische Elemente erweitert. Damit lassen sich AR-UIs beschreiben. Ein Beispiel, wie ein AR-Szenenmodell aufgebaut ist, ist in Abbildung 9.3 zu sehen. Das Beispielmodell repräsentiert einen Tisch (*Table*), auf dem eine Tasse (*Cup*) und ein Teller (*Plate*) angeordnet sind. Der Tisch ist ein reales Objekt, die Tasse ist ein virtuelles Objekt und der Teller ist ein hybrides Objekt. Abbildung 9.3 (a) zeigt ein Modell der realen Szene, Abbildung 9.3 (b) zeigt das Modell der virtuellen Szene und Abbildung 9.3 (c) zeigt das erweiterte AR-Szenenmodell, das sowohl reale, virtuelle als auch hybride Objekte enthält. Der Teller besitzt eine Komponente sowohl im virtuellen als auch im realen Raum. In den Modellen der virtuellen Szene und der erweiterten Szene besitzt der Tisch keine Geometrie, repräsentiert aber ein Element zur Gruppierung des Teller- und des Tassenobjekts. Da die Tasse ein rein virtuelles Objekt ist, existiert sie auch in den Modellen der virtuellen und der erweiterten Szene als virtuelles Objekt. Der Teller ist ein hybrides Objekt und damit in allen Szenen enthalten. Die Abbildungen 9.3(d)-(f) illustrieren die Elemente des AR-UIs, die den Modellen in den Abbildungen 9.3(a)-(c) entsprechen.

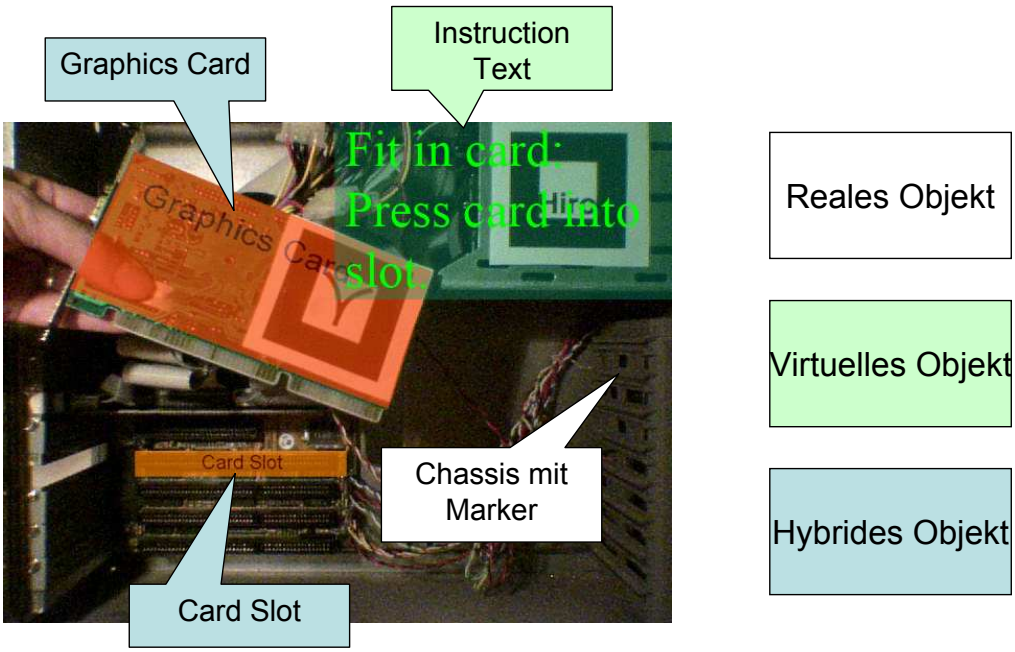



Abbildung 9.4: Verschiedene Objekttypen im Szenario *Grafikkarteninstallation*

## 9.2 Beispiel

Als Beispiel, an dem die Möglichkeiten der Modellierung mit SSIML/AR demonstriert werden, dient das bereits in Kapitel 6 vorgestellte Szenario aus dem Bereich Montage und Wartung. In diesem Szenario soll der Benutzer beim Einbau einer Grafikkarte in einen PC durch eine interaktive 3D-Anleitung unterstützt werden, die hier als AR-Anwendung umgesetzt wird. Das Taskflow-Modell zu dieser Anwendung wurde ebenfalls bereits in Kapitel 6 vorgestellt und wird in diesem Kapitel wieder aufgegriffen.

Im Grafikkarteninstallationsszenario gibt es verschiedene Objekttypen (Abbildung 9.4), die bereits im vorigen Abschnitt diskutiert wurden. Ein Beispiel für ein rein reales Objekt ist das *Chassis*. Andere Objekte, wie die Grafikkarte (*Graphics Card*) und der Kartensteckplatz (*Card Slot*) sind hybride Objekte, da sie virtuelle Überlagerungen, d.h. geometrische Repräsentationen im virtuellen Raum, besitzen. Dagegen sind eingeblendete Anweisungstexte (*Instruction Texts*) rein virtuelle Objekte. Sie sind keinen realen Objekten des AR-UIs zugeordnet und werden deshalb immer an der gleichen Stelle auf dem Display (d.h. bildschirmstabilisiert) dargestellt. Wie Abbildung 9.4 zeigt, wurden optische Marker an einigen physikalischen Objekten befestigt, um diese mittels eines passenden Tracking-Systems verfolgen zu können, und zwar an der PC-Abdeckung (*PC Cover*, nicht sichtbar in Abbildung 9.4), dem Chassis und der Grafikkarte. Positionen und Orientierungen von kleineren Objekten, wie dem Kartenschlitz, werden auf Basis von Position und Orientierung des Chassis' berechnet.

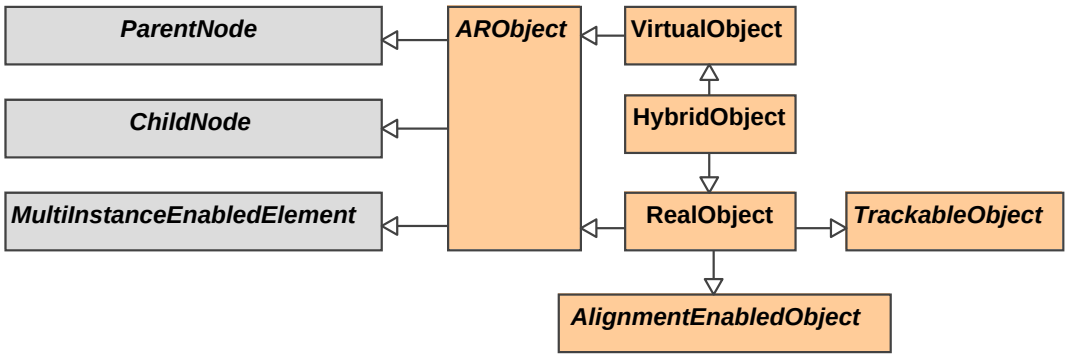



Abbildung 9.5: Ausschnitt aus dem SSIML/AR-Metamodell

Tabelle 9.1: Notation AR-spezifischer Modellelementtypen

| Knotentyp | Notation | Beispiel |
|---|---|---|
| RealObject | R<br><Objektname>(<Instanznamensliste>)?<br>(':'<Inhaltstypdefinition>)? | R<br>chassis |
| VirtualObject | V<br><Objektname>(<Instanznamensliste>)?<br>(':'<Inhaltstypdefinition>)? | V<br>instruction1:Text1 |
| HybridObject | H<br><Objektname>(<Instanznamensliste>)?<br>(':'<Inhaltstypdefinition>)? | H<br>pcCover:PCCover |

## 9.3 AR-Szenenmodell

Das AR-Szenenmodell spezifiziert die Struktur des AR-UIs. Abbildung 9.5 visualisiert die Vererbungshierarchie der neuen AR-spezifischen Metamodellelemente für Szenenmodellknoten, die aus den oben beschriebenen Überlegungen abgeleitet wurden (Farblegende s. Kapitel 5, Abbildung 5.5). Szenenmodellknoten eines AR-spezifischen Typs sind `MultiInstanceEnabledElements`, d.h. sie können zur Laufzeit des modellierten Systems mehrere Instanzen besitzen. So könnte es bspw. im Grafikkarteninstallationsszenario mehrere alternativ benutzbare Kartensteckplätze, also mehrere Instanzen eines `HybridObject`-Modellelements `cardSlot` geben. Tabelle 9.1 gibt einen Überblick über die Notation der AR-spezifischen Knoten.

Das Attribut `createCopies` einer `ParentChildRelationship`-Beziehung (s. Abschnitt 5.2.2.7), bei welcher der Kindknoten ein `RealObject`-Element ist, muss immer gelten, da ein und dasselbe reale Objekt – physisch bedingt – auch nur an einem Ort im realen Raum

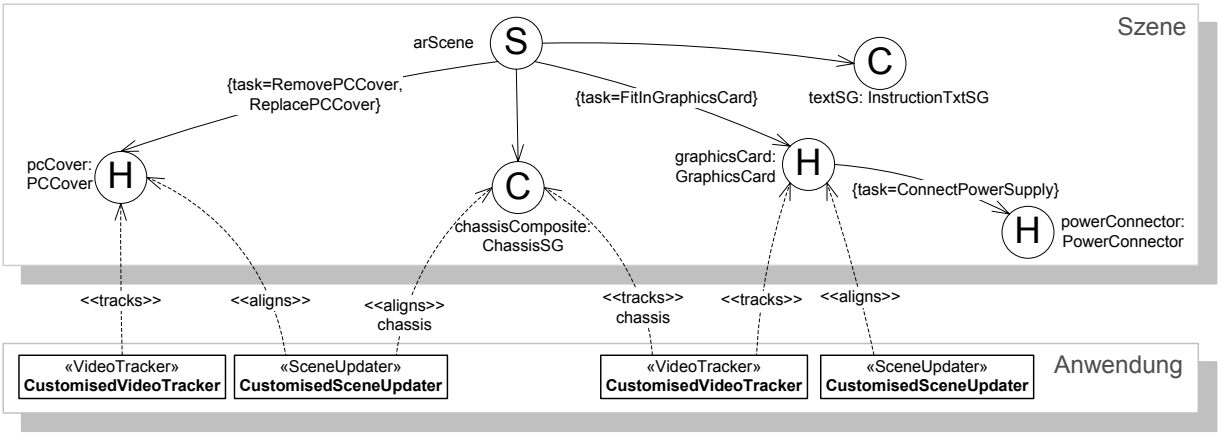
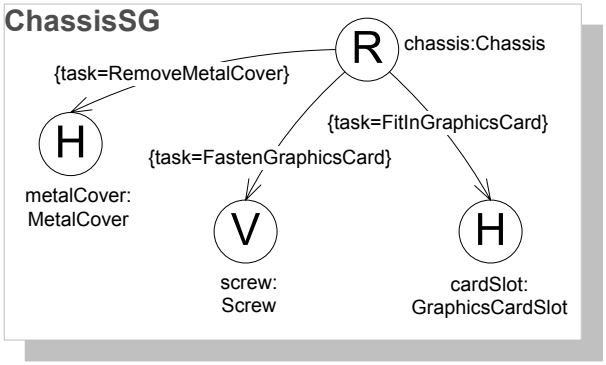



Abbildung 9.6: Beispiel eines SSIML/AR-Szenen- und Interrelationenmodells

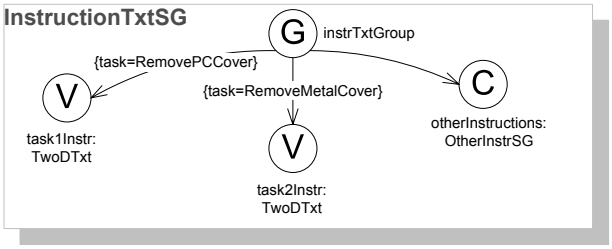
existieren kann. Mehrere parallele Transformationen eines physikalischen Objekts sind im Gegensatz zu mehreren parallelen Transformationen eines virtuellen Objekts daher nicht möglich.

In Abbildung 9.6 (oberer Teil) ist das SSIML-AR-Szenenmodell für das Grafikkarteninstallationsszenario zu sehen. Die Wurzel der Szene ist der Knoten `arScene`. Die Knoten `pcCover` und `powerConnector` (Stromversorgungsanschluss) repräsentieren die entsprechenden hybriden Objekte. Position und Orientierung des Stromversorgungsanschlusses werden relativ zur Position und Orientierung der Grafikkarte ermittelt, da sich der Stromversorgungsanschluss auf der Karte befindet. `chassisComposite` ist ein Kompositionsknoten. Die Subgraphendefinition dieses Knotens ist in Abbildung 9.7(a) zu sehen, welche den `ChassisSG`-Subgraphen zeigt. Der Wurzelknoten des Subgraphen ist der `RealObject`-Knoten `chassis`. Obwohl das Chassis nicht durch ein virtuelles Objekt überlagert wird, dient es als Gruppierungselement im virtuellen dreidimensionalen Raum. Hybride Objekte wie `metalCover` (die Metallabdeckung für den Kartensteckplatz) und `cardSlot` (Kartensteckplatz) sind fest mit dem Chassis verbunden, entweder direkt oder über andere Objekte wie das Mainboard. Damit kann ihre Position und Orientierung über die Position und Orientierung des Chassis bestimmt werden und es muss lediglich das Chassis mit einem Tracker verfolgt werden. `screw` (Schraube) ist ein rein virtuelles Objekt und dient allein dem Zweck, anzuzeigen, wo der Benutzer die physikalische Schraube einsetzen muss, um die Grafikkarte zu befestigen. Der Kompositionsknoten `textSG` kapselt taskabhängige Instruktionstexte (Abbildung 9.7(b)). Um eine taskabhängige Präsentation virtueller Inhalte zu erreichen, wurde in Kapitel 6 das Konzept der *Task-Constrained Edges* zwischen Eltern- und Kindknoten eingeführt.

Z. B. wird in dem Modell in Abbildung 9.6 spezifiziert, dass die virtuelle 3D-Repräsentation der PC-Abdeckung (`pcCover`) nur dann angezeigt wird, wenn entweder `ReplacePCCover` oder `RemovePCCover` die aktuelle Aufgabe des Benutzers ist.




(a) `ChassisSG`-Subgraph




(b) `InstructionTxtSG`-Subgraph

Abbildung 9.7: Subgraphendefinitionen des AR-Szenenmodells `arScene`

## 9.4 AR-Interrelationenmodell

Das AR-UI ist mit weiteren Komponenten des AR-Systems, wie Tracking- und Rendering-Komponenten, verbunden. Die Position und die Orientierung von virtuellen Objekten, die einen räumlichen Bezug zu realen Objekten aufweisen, können durch die Verarbeitung von Trackingdaten kontrolliert werden. In SSIML/AR werden die in SSIML bereits enthaltenen Beziehungstypen um AR-spezifische Beziehungstypen ergänzt. Abbildung 9.8 zeigt dazu einen Ausschnitt des SSIML/AR-Metamodells mit den AR-spezifischen Beziehungstypen. Konkret wurden zwei neue Beziehungstypen definiert: `TracksRelationship` und `AlignsRelationship`. Eine `TracksRelationship`-Beziehung kann zwischen einer Anwendungskomponente vom Typ `Tracker` (d. h. einem Modellelement vom Typ `ApplicationElement` und vom Subtyp `Tracker`) und einem `TrackableObject` (also einem realen oder hybriden Objekt) bestehen. Damit wird ausgedrückt, dass die Anwendungskomponente die Positions- und Orientierungswerte des Objektes, das ihr über die `TracksRelationship`-Beziehung zugeordnet ist, bereitstellt. Eine vom Element `Tracker` im Metamodell erbende Metaklasse ist `VideoTracker`, welche für videobasierte Tracker steht. Ein Beispiel für ein `VideoTracker`-Elements ist in Abbildung 9.6 unten zu sehen. Dort werden die Raumkoordinaten der PC-Abdeckung (`pcCover`) durch die Anwendungskomponente `CustomisedVideoTracker` ermittelt.

Das SSIML/AR-Metamodell ist so konzipiert, dass eine einfache Definition weiterer Tracker-Typen durch Ableitung von der abstrakten Metaklasse `Tracker` möglich ist. So könnte z. B. ein `Tracker`-Subtyp `UltrasonicTracker` eingeführt werden.

Ein so genanntes `SceneUpdater`-Objekt sorgt (zur Laufzeit) dafür, dass die von einem `Tracker` ermittelten Positionen und Orientierungen entweder auf die Position bzw. Orientierung der virtuellen Komponente eines realen – im virtuellen Raum durch eine Objektgruppierung repräsentierten – oder hybriden Objekts abgebildet werden. Um einen `SceneUpdater` mit einem `AlignmentEnabledObject` (vgl. Abbildung 9.5), also einem Element des Typs `RealObject` oder `HybridObject` zu assoziieren, wird eine `AlignsRelationship`-Beziehung eingesetzt. In Abbildung 9.6 wird z.B. der `SceneUpdater` `CustomisedSceneUp-`

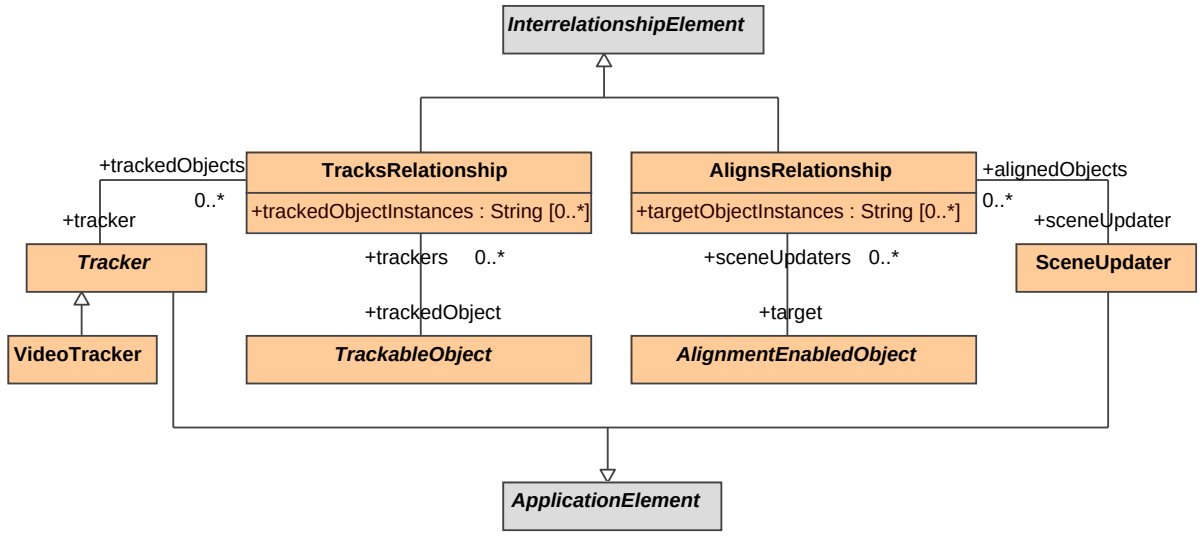



Abbildung 9.8: Die Beziehungstypen in SSIML/AR im Metamodell

`dater` gezeigt, der die mit ihm assoziierten Objekte ausrichtet. Sowohl `SceneUpdater`- als auch `Tracker`-Elemente sind *Singletons*, sie besitzen zur Laufzeit jeweils nur eine Instanz.

Oft ist ein und dasselbe Szenenobjekt gleichzeitig mit einem `Tracker`- *und* einem `SceneUpdater` assoziiert, wobei letzterer zur Laufzeit das virtuelle Pendant der realen Komponente des Szenenobjekts ausrichtet. Dies trifft z. B. für das `HybridObject`-Element `pcCover` in Abbildung 9.6 zu.

Wie bereits erwähnt, kann ein `ARObject` mehrere Laufzeitinstanzen besitzen. Über die Attribute `trackedObjectInstances` und `targetObjectInstances` einer `TracksRelationship`- bzw. `AlignsRelationship`-Beziehung kann in Form von Kompositionspfaden bestimmt werden, welche Laufzeitinstanzen eines `RealObject`-/`HybridObject`-Elements verfolgt (reale Komponenten) oder ausgerichtet (virtuelle Komponenten) werden. Wird nichts angegeben, gilt die entsprechende Beziehung für alle Laufzeitinstanzen.

Die Notation von Modellelementen der Typen `VideoTracker` und `SceneUpdater` ist in Tabelle 9.2 dargestellt.

Die Notation für Kanten der Typen `TracksRelationship` und `AlignsRelationship` entspricht der bereits in Abschnitt 5.3.2 vorgestellten Notation für SSIML-Interrelationen. Für die Bezeichnung des Beziehungstyps ist entweder *tracks* oder *aligns* anzugeben. Beispiele für `TracksRelationship`- und `AlignsRelationship`-Elemente sind in Abbildung 9.6 zu sehen.

## 9.5 Modell-Code-Abbildung

Für die Generierung von Code für ein AR-UI gelten Regeln, die sich von den Regeln der Codegenerierung für herkömmliche 3D-Anwendungen (vgl. Kapitel 5) teilweise unterschei-

Tabelle 9.2: Notation AR-spezifischer Typen von Anwendungs-Komponenten

| Elementtyp | Notation | Beispiel |
|---|---|---|
| Tracker | '<<'<Tracker-Subtyp>'>>'<br><Name> | << VideoTracker >><br>MyVideoTracker |
| SceneUpdater | '<<SceneUpdater>>'<br><Name> | << SceneUpdater >><br>MySceneUpdater |

den. Von besonderer Bedeutung ist dabei, dass eine AR-Benutzungsschnittstelle nicht nur rein virtuelle Elemente, sondern auch physikalische Elemente beinhaltet. Der generierte Code ist ebenfalls abhängig von der Tracking-Technologie, die verwendet wird, um reale Objekte zu verfolgen. Sehr häufig zum Einsatz kommt heutzutage, wie schon in Abschnitt 9.1.2.2 herausgestellt, das markerbasierte optische Tracking.

Die Codegenerierung für AR-Anwendungen soll anhand des Beispiels aus Abschnitt 9.2 demonstriert werden. Dabei wird der im Modell in Abbildung 9.6 dargestellte `VideoTracker` durch eine Tracking-Komponente realisiert, die markerbasiertes optisches Tracking unterstützt. Als Format der generierten AR-Szene soll – wie bereits in den vorangehenden Kapiteln – X3D dienen, die Anbindung an Anwendungskomponenten erfolgt in Java.

Komplexität und Umfang des aus einem AR-Interrelationenmodell generierten Codes lassen sich reduzieren, wenn `Tracker` die Koordinaten von verfolgten Objekten relativ zum Szenenursprung liefern, und nur Szenenobjekte mit Trackern und `SceneUpdater`-Objekten assoziiert sind, die direkte Kindknoten des Szenenwurzelknotens sind. Damit können die vom Tracker bereitgestellten Daten weitestgehend direkt von den `SceneUpdater`-Objekten verwendet werden, und zusätzliche Transformationsberechnungen, die andernfalls implementiert und vor dem Rendering zur Laufzeit durchgeführt werden müssten, können entfallen.

### 9.5.1 Abbildung von AR-Szenenmodellen

Prinzipiell wird für eine Instanz eines SSIML-`ARObjects` der gleiche Code wie für eine SSIML-`Object`-Knoteninstanz erzeugt (vgl. Abschnitt 5.6). Dabei werden auch für rein reale Objekte (nicht hybride Objekte) X3D-`Inline`-Knoten generiert, die auf Geometriedaten verweisen können. Rein reale Objekte besitzen zwar im virtuellen Raum im Gegensatz zu hybriden Objekten nicht unmittelbar eine geometrische Repräsentation, jedoch kann durch die Festlegung von Geometrien für reale Objekte die Anpassung der generierten 3D-Szene in einem 3D-Autorenwerkzeug visuell unterstützt werden, da reale Objekte oft einen räumlichen Bezugspunkt bei der Ausrichtung anderer Objekte im virtuellen Raum darstellen. Z. B. stellt das Objekt `chassis` im Grafikkarteninstallationsszenario einen räumlichen

Bezugspunkt für das Objekt `cardSlot` dar. Zur Laufzeit kann die virtuelle 3D-Geometrie eines realen Objektes ausgeblendet werden, z. B. indem diese volltransparent gerendert wird. Damit wird vermieden, dass das reale Objekt durch die virtuelle Geometrie verdeckt wird.

Für jedes im SSIML/AR-Modell mit einem `VideoTracker` verbundene Objekt werden zusätzliche X3D-Knoten erzeugt: Eine Transformationsgruppe, welche die räumliche Transformation des Markers enthält (Marker-Transformationsgruppe) und ein `Shape`-Knoten, der die Geometrie des Markers repräsentiert. Das für einen Marker generierte X3D-`Shape` ist ein halbtransparenter Quader mit einer Standardgröße, die später an die tatsächliche Markergröße angepasst werden kann. Das Marker-`Shape` dient ebenfalls als Orientierungshilfe bei der Anpassung der generierten 3D-Szene in einem Autorenwerkzeug und kann zur Laufzeit ausgeblendet werden.

Jeder X3D-Knoten wird entsprechend der in Abschnitt 5.6 eingeführten Regeln benannt. Damit z. B. das Verstecken der Geometrien von Markern und realen Objekten zur Laufzeit durch das AR-System, welches die 3D-Szene verwaltet, vollautomatisch durchgeführt werden kann, müssen die X3D-Knoten mit zusätzlichen Informationen versehen werden. Dafür wird jedem X3D-Knotennamen ein Suffix angehängt, das die Art des Knotens genauer spezifiziert. Das Suffix besteht aus einem Doppelpunkt und einer nachfolgenden AR-Knotenkennung. Ein Knoten, der die Geometrie eines Markers enthält, hat bspw. das Suffix `:M`.

Der nachfolgende Quellcodeausschnitt stellt die Abbildung des Knotens `graphicsCard` des Karteninstallationsszenarios in X3D dar:

```
#marker transformation group
DEF graphicsCardMarkerTrans:MT Transform {
  children [
    #marker shape
    DEF graphicsCardMarker:M Shape {
      appearance  Appearance {material  Material {...}}
      geometry Box {size 0.06 0.001 0.06}
    }
    #object transformation group
    DEF graphicsCardTrans:HT Transform {
       children
         #object group
         DEF graphicsCard:H Group {
           children
             #geometry definition
             DEF graphicsCardContent:HC
               Inline {url "GraphicsCard.x3dv"}}}]}
```

Für Task-Constrained-Edges werden in der X3D-Szene entsprechende `Switch`-Knoten eingefügt, die zur Laufzeit taskabhängig über das Knotenfeld `whichChoice` so eingestellt werden können, dass ihre Kindknoten entweder gerendert werden oder nicht (vgl. Abschnitt 6.4). Nachfolgender Codeausschnitt zeigt den für die Task-Constrained-Edge zwischen dem Knoten `chassis` und dem Knoten `screw` im SSIML/AR-Modell des Karteninstallationsbeispiels generierten Code. Wie der Ausschnitt illustriert, fließt der Name des Kompositionsbereiches (hier: `chassisComposite`), in dem ein SSIML/AR-Modellelement enthalten ist, mit in die Benennung der jeweils generierten X3D-Knoten ein.

```
...
#code for real object Chassis
#object group
DEF chassisComposite/chassis:R Group {
  children [
    #geometry definition
    DEF chassisComposite/chassisContent:RC
      Inline {url "chassisComposite/Chassis.x3dv"}
    #code for task-constrained edge
    DEF chassisComposite/chassis-chassisComposite/screw Switch {
      whichChoice  0
      choice
        #code for virtual object screw
        DEF chassisComposite/screwTrans:VT Transform {
          children
            DEF chassisComposite/screw:V Group {
              children
                DEF chassisComposite/screwContent:VC
                  Inline {url "chassisComposite/Screw.x3dv"}}}}]}
```

### 9.5.2 Abbildung von AR-Interrelationenmodellen

Wie bereits erläutert, unterscheidet sich eine AR-Anwendung u. a. darin von einer „reinen“ 3D-Anwendung, dass auch physikalische Objekte mit zur Benutzungsschnittstelle gehören. Soll ein SSIML/AR-Modell auf Code abgebildet werden, muss deshalb auch Code für Komponenten erzeugt werden, die das Tracking realer Objekte realisieren.

Als Basis für den generierten Programmcode dient ein einfaches, im Rahmen der vorliegenden Arbeit entwickeltes Java-Framework, genannt *IntegratorAR*. Die aus dem SSIML/AR-Modell generierten Klassen erweitern oder verwenden Klassen aus dem IntegratorAR-Framework. Das Framework unterstützt das videobasierte Tracking realer Objekte.

Das IntegratorAR-Framework besteht aus verschiedenen Subsystemen. Jedes Subsystem stellt eine Menge von Schnittstellen bereit, die von anderen Subsystemen benötigt werden. Für jede Schnittstelle wurde im Rahmen der vorliegenden Arbeit eine Standardimplementierung realisiert. Zum Beispiel wurde die Standard-Videotracking-Komponente mittels des jARToolkit [GRSP02] implementiert, welches einen Wrapper für das ARToolkit, eines der am weitesten verbreiteten Programmierframeworks für markerbasiertes Tracking, repräsentiert. Die Standardimplementierung der Darstellungskomponente basiert auf Java 3D. Einen Architekturüberblick des Frameworks gibt Abbildung 9.9. Die Architektur des Frameworks orientiert sich an den Empfehlungen von Reicher et al. [RMBK03], die eine generische AR-Frameworkarchitektur vorstellen. Nachfolgend werden die einzelnen Subsysteme des Frameworks und ihr Zusammenspiel genauer vorgestellt.

Das *Control-Subsystem* ist für das Sammeln von Eingabedaten verantwortlich, wie Daten einer Videokamera oder Tastatureingaben. Videodaten werden an das *Tracking-Subsystem* weitergeleitet, während andere Eingabedaten vom *Context-Subsystem* verarbeitet werden. Das *WorldModel* stellt Informationen über virtuelle und reale Objekte und Informationen über deren Zusammenhänge bereit. Es verwaltet alle Objekte der AR-Szene. Informationen über reale Objekte werden dem *Tracking-Subsystem* bereitgestellt, damit

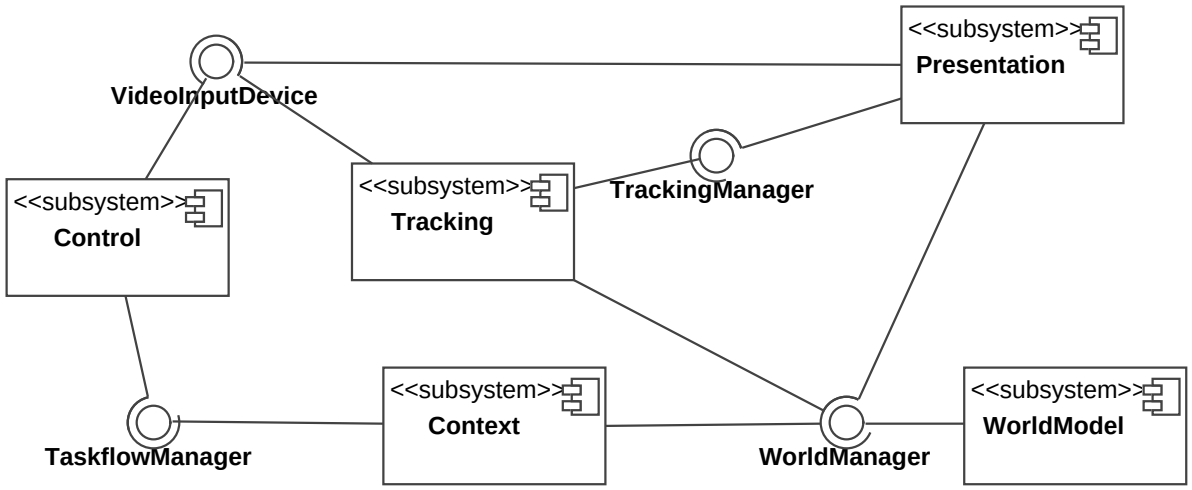



Abbildung 9.9: Das *IntegratorAR*-Framework - Architekturüberblick

dieses die beschriebenen Objekte im Videostrom finden und Ihre Positionen und Orientierungen in der realen Welt bestimmen kann. Die errechneten Positions- und Orientierungsdaten werden vom *Presentation-Subsystem* benutzt, um die virtuellen Objekte der AR-Szene korrekt in das aktuelle – vom *Control-Subsystem* stammende – Videobild zu integrieren und die resultierende Grafik darzustellen. Die Beschreibungen der darzustellenden virtuellen Objekte erhält das *Presentation-Subsystem* über das *WorldModel*. Beziehungen zwischen realen Objekten und virtuellen Objekten werden innerhalb des *Presentation-Subsystems* zur Laufzeit über `SceneUpdater`-Objekte festgehalten. Ein `SceneUpdater` kann dabei mehrere Beziehungen verwalten. Ändert ein mit einem `SceneUpdater`-Objekt assoziiertes reales Objekt seine Position und/oder seine Orientierung, sorgt der `SceneUpdater` dafür, dass das dem realen Objekt entsprechende (ggf. zusammengesetzte) virtuelle Objekt korrekt positioniert und orientiert wird.

Zudem ist das *Presentation-Subsystem* mit dem *Context-Subsystem* verdrahtet. Damit wird eine kontextabhängige – insbesondere taskabhängige – Informationsvisualisierung möglich. Eine taskabhängige Visualisierung bedeutet, dass dargestellte Informationen abhängig von der gerade zu lösenden Aufgabe des Benutzers sind (vgl. Kapitel 6). Das *Context-Subsystem* kontrolliert den Benutzerfortschritt beim Lösen einer Sequenz von Einzelaufgaben in Abhängigkeit von Ereignisdaten, die aus dem *Control-Subsystem* stammen.

Die Hauptkomponenten der einzelnen Subsysteme sowie wichtige, aus einem SSIML/AR-Modell generierte Anwendungskomponenten werden im Folgenden betrachtet.

**Beschreibung realer Objekte** Für jedes physikalische Objekt, das von einem Tracker verfolgt werden soll, wird ein Eintrag in eine Java-*Properties*-Datei generiert. Ein solcher Eintrag enthält *als Schlüssel den Namen* des zu verfolgenden Objektes und *als Wert eine Objektbeschreibung*. Die Objektbeschreibung hat ein Format, das abhängig von der verwendeten Tracking-Technologie ist. Da die vorhandene Implementierung des Frameworks aus-

schließlich videobasiertes Tracking mit dem jARToolkit unterstützt, wird standardmäßig eine Beschreibung des ARToolkit-Markers[2], der auf dem jeweiligen physikalischen Objekt anzubringen ist, generiert. Eine generierte Markerbeschreibung enthält z.B. Informationen über den Speicherort (die URL) der Markermusterdatei, welche Daten über die visuelle Charakteristik des Markers bereitstellt, sowie Informationen über die Markergröße und den Markermittelpunkt. Da die Markermusterdateien selbst derzeit nicht automatisch mit generiert werden (obwohl dies durchaus realisierbar wäre), müssen sie manuell an den jeweils referenzierten Speicherort kopiert und entsprechend benannt werden.

Neben einfachen Markern ist auch die Verwendung von so genannten Multimarkern möglich, d.h. zusammengesetzter Marker, die aus einer Gruppierung einzelner Marker gebildet werden, deren räumliche Position und Orientierung relativ zum Multimarkermittelpunkt bekannt ist. Der Vorteil eines Multimarkers ist, dass seine Position und Orientierung auch noch bestimmt werden kann, wenn alle bis auf einen seiner Teilmarker verdeckt sind. Da das jARToolkit im Gegensatz zum ARToolkit keine integrierte Unterstützung für Multimarkererkennung bietet, wurde diese Funktionalität zusätzlich – basierend auf der Einzelmarkererkennung – implementiert.

***WorldManager*** Die generierte Java-Properties-Datei mit den Beschreibungen der zu verfolgenden realen Objekte kann vom `WorldManager` (vgl. Abbildung 9.9) geladen werden, der für jeden Eintrag in der Datei eine Instanz der Framework-Klasse `RealObject` erzeugt. Der `WorldManager` lädt zudem die dem SSIML/AR-Szenenmodell entsprechende X3D-Szene, welche alle virtuellen Elemente des AR-UIs enthält, und bietet anderen Anwendungskomponenten zur Laufzeit Zugriff auf die Elemente der Szene.

***Tracker* und *TrackingManager*** Für jede `VideoTracker`-Komponente aus dem SSIML/AR-Modell wird eine Java-Klasse mit dem Namen des Trackers erzeugt, die standardmäßig die Framework-Klasse `DefaultVideoTracker` erweitert. `DefaultVideoTracker` implementiert das Interface `VideoTracker`, das seinerseits wiederum das Interface `Tracker` erweitert. Ein `Tracker` im IntegratorAR-Framework kann als eine Art Filter gesehen werden, der aus Rohdaten einer Datenquelle (z. B. aus den von einer Kamera aufgenommenen Videoframes) die Positions- und Orientierungsdaten der – entsprechend dem SSIML/AR-Modell – mit ihm assoziierten realen Objekte extrahiert. Wurde im Modell die Art des zu benutzenden Trackers (z.B. *videobasierter Tracker*) nicht genauer spezifiziert, kann lediglich eine Java-Klasse generiert werden, die das im Framework vorgegebene Basis-Interface *Tracker* implementiert. Jede generierte konkrete `Tracker`-Klasse hat in der Regel genau eine Laufzeitinstanz (d.h. ein spezieller Tracker ist ein *Singleton* [GHJV94]). Da ein Modell aber mehrere konkrete `Tracker` enthalten kann, kann es zur Laufzeit mehrere Instanzen des Schnittstellentyps `Tracker` geben.

Zusätzlich wird eine Java-Properties-Datei erzeugt, die jedem zu verfolgenden *realen Objekt* (spezifiziert durch den Objektnamen) die mit dem Objekt entsprechend dem

[2]Ein ARToolkit-Marker besitzt einen quadratischen schwarzen Rahmen, der zur Bestimmung seiner Lagedaten dient und ein Muster einschließt, das zur Identifizierung des Markers herangezogen wird

SSIML/AR-Modell *assoziierten Tracker* zuordnet. Die Tracker sind durch ihre vollständigen Klassennamen (einschließlich Paketpräfix) repräsentiert. Die Properties-Datei wird vom `TrackingManager` interpretiert, der alle Tracker verwaltet. Der `TrackingManager` lädt die benötigten `Tracker` dynamisch nach, instanziiert diese und ordnet ihnen die zugehörigen `RealObject`-Instanzen zu, welche er vom `WorldManager` durch Übergabe der jeweiligen Objektnamen anfordert. Um eine Verbindung zwischen einer Datenquelle zur Rohdatenlieferung (z.B. einer Videokamera im Falle eines videobasierten Trackers) und einem Tracker etablieren zu können, stellt der `TrackingManager` eine Methode bereit, der eine Referenz auf die Datenquelle zusammen mit dem vollständigen Klassennamen des mit der Quelle zu verbindenden `Tracker`s übergeben werden muss. Eine Datenquelle wird durch eine Instanz des Schnittstellentyps `TrackingInput` (im Framework definiert) repräsentiert. Für die Eingabe von Videodaten dient das Interface `VideoInput`, dass das Interface `TrackingInput` erweitert. Es existiert sowohl eine jARToolkit (Standard)- als auch eine Java-Media-Framework (JMF [JMF])-basierte Implementierung der `VideoInput`-Schnittstelle.

***SceneUpdater* und *Renderer*** Für jedes `SceneUpdater`-Element des SSIML/AR-Modells wird eine zugehörige `SceneUpdater`-Klasse mit dem Namen des Modellelements generiert, die von der `SceneUpdater`-Basisklasse (`DefaultARUpdater`) des AR-Frameworks abgeleitet wird. Bei der Initialisierung muss einem `SceneUpdater` sowohl eine Referenz auf den `TrackingManager` als auch auf den `WorldManager` übergeben werden. Zusätzlich lädt ein `SceneUpdater` bei der Initialisierung eine Konfigurationsdatei, welche eine Liste der Namen aller virtuellen Objekte der 3D-Szene enthält, die der `SceneUpdater` aktualisieren soll. Die Objektliste für einen `SceneUpdater` wird automatisch unter Berücksichtigung der vom `SceneUpdater` im SSIML/AR-Modell ausgehenden `AlignsRelationship`-Beziehungen generiert. Über den `WorldManager` bekommt ein `SceneUpdater` Zugriff auf die Laufzeitinstanzen der über die Liste ausgewiesenen 3D-Objekte. Wie schon gesagt, erhält ein `SceneUpdater` Informationen über Positionen und Orientierungen der realen Pendants der zu aktualisierenden virtuellen Objekte vom `TrackingManager`. Alle `SceneUpdater`-Objekte werden vom `Renderer` (der Hauptkomponente des Präsentationssubsystems) verwaltet. Der `Renderer` ruft in regelmäßigen Zeitabständen (ca. aller 20 ms) die `update`-Methoden der `SceneUpdater`-Objekte auf, um die erforderliche Neupositionierung der virtuellen Szenenelemente zu gewährleisten. Die `update`-Methode eines `SceneUpdater`s kann nach der Generierung manuell erweitert werden, z.B. um 3D-Objekte zusätzlich zu animieren.

***TaskflowManager*** Die Hauptkomponente des *Context-Subsystems* ist eine Implementierung des in Kapitel 6 vorgestellten `TaskflowManager`s, welche zur Laufzeit taskabhängig virtuelle Elemente des AR-UIs durch Manipulation der – für die *Task-Constrained-Edges* eines SSIML/AR-Modells generierten – `Switch`-Knoten der 3D-Szene ein- und ausblendet. Dafür traversiert der `TaskflowManager` das Objektmodell, das aus der XML-Beschreibung eines SSIML-Taskflow-Modells erzeugt wurde. Über die playerartige Schnittstelle des `TaskflowManager`s kann mittels der Methoden `nextTask` bzw. `previousTask`

zwischen den einzelnen Tasks navigiert werden. Für eine ausführlichere Diskussion des `TaskflowManager`s wird auf Abschnitt 6.4 verwiesen.

**Zusammenführung der generierten Klassen** Schließlich wird eine Hauptklasse mit einer Startroutine für die AR-Anwendung generiert, die alle benötigten Klasseninstanzen erzeugt, ggf. initialisiert und miteinander in geeigneter Weise verknüpft. Die erstellte Anwendung ist prinzipiell ohne Änderungen am Java-Code lauffähig. Eine nachträgliche manuelle Anpassung des generierten Codes ist aber durchaus möglich.

## 9.6 Verwandte Arbeiten

In diesem Abschnitt werden Forschungsarbeiten aufgeführt, die einen thematischen Bezug zu SSIML/AR besitzen, wenngleich auch keine der erwähnten Arbeiten alle Aspekte von SSIML/AR abdeckt.

Toolkits und Frameworks auf Implementierungsebene wie ARToolkit [ART], DWARF [BBK+01] und Studierstube [SFH+02] bieten eine geeignete Basis zur Programmierung von AR-Anwendungen. Derartige Frameworks beinhalten i. d. R. Funktionen zum Objekttracking und zur Erzeugung virtueller Overlays für reale Objekte. Da die AR-Programmierung im Normalfall auf vergleichsweise niedrigem Abstraktionsniveau erfolgt, kann die Erstellung einer nicht trivialen Anwendung sehr zeitaufwendig sein. Weiterhin ist der Einarbeitungsaufwand in einige der Frameworks aufgrund ihrer hohen Komplexität relativ groß. Oft fehlt eine umfassenden Dokumentation, da die Frameworks hauptsächlich von Akademikern für Akademiker entwickelt werden (vgl. a. [MGDB04]). Allgemein ist ein Mangel an Konzepten und Werkzeugen festzustellen, die eine strukturierte Entwicklung komplexerer AR-Anwendungen und vor allem einen Entwurf vor der Implementierung ermöglichen.

An dieser Stelle setzt SSIML/AR an. Wie bereits in Abschnitt 9.1 erwähnt, unterstützt SSIML/AR den Entwurf von AR-Anwendungen mittels visueller Modelle. Ebenso wird durch die automatische Generierung von Code aus den visuellen Modellen ein nahtloser Übergang zur Implementierung gewährleistet. Der generierte Code genügt bereits ohne manuelle Anpassungen den Anforderungen an seine Funktionstüchtigkeit und an eine sinnvolle Strukturierung.

Einige der higher-level AR-Entwicklungswerkzeuge fokussieren auf die Unterstützung von Content-Entwicklern wie Designern. Bspw. erweitert das *Designers Augmented Reality Toolkit* (DART) [MGB+03, MGDB04, MGDB05] das Multimedia-Autorenwerkzeug Adobe Director [Dir] und ermöglicht damit die Erstellung von AR-UIs durch die Verbindung von visuellem Design mit Scripting. Dies kommt der gewohnten Arbeitsweise eines Designers nahe. Ein weiteres Forschungsprojekt, dass sich insbesondere auf die Unterstützung von Designern konzentriert, ist das bereits in Abschnitt 5.7 erwähnte i4D-Framework [GPRR00]. Der i4D-Ansatz basiert auf der *Actor*-Metapher. Ein Actor ist eine Komponente mit autonomem Verhalten, die einerseits unabhängig agieren und andererseits mit weiteren Actors interagieren kann. Die Actor-Metapher existiert in einfacherer Form auch

in Multimedia-Autorenumgebungen wie Adobe Director. Die Actor-Metapher ist daher eine geeignete konzeptuelle Basis, um die Diskussion zwischen Entwicklern mit umfangreicher und geringer Programmiererfahrung, d. h. zwischen Programmierern und Designern, zu ermöglichen. i4D unterstützt Designer außerdem durch adäquate Autorenwerkzeuge, z. B. Werkzeuge zur 3D-Szenenerstellung.

Ein weiteres relevantes Forschungsprojekt ist in diesem Zusammenhang AMIRE (*Authoring Mixed Reality*) [GHP+02]. Ähnlich wie i4D bietet AMIRE ein Komponentenframework zusammen mit einem darauf abgestimmten Entwicklungsprozess. AMIRE zielt insbesondere auf die Erstellung von AR-Autorenwerkzeugen ab. Einige Werkzeuge, wie der *Authoring Wizard for Mixed Reality Assembly Instructors* [ZHBH03] wurden bereits auf Basis des AMIRE-Frameworks entwickelt. Die mittels AMIRE erstellten Autorenwerkzeuge sollen dazu dienen, auch Content-Erstellern die Entwicklung von AR-Anwendungen zu erleichtern.

Allerdings können weniger triviale AR-Anwendungen in Montage und Wartung, aber auch in anderen Bereichen, z. B. Medizin, oft nicht völlig ohne Programmierwissen erstellt werden. SSIML/AR fokussiert deshalb nicht allein auf die Unterstützung von Content-Entwicklern, sondern in gleichem Maße auch auf die Unterstützung von Programmierern. Obwohl der aus einem SSIML/AR-Modell generierte Programmcode eine grundlegende Ablaufsteuerung der Anwendung vorgibt, hat der Programmierer vielfältige Möglichkeiten zur Erweiterung und Anpassung der AR-Anwendung, z.B. durch die Integration komplizierter 3D-Animationen.

Nur wenige Forschungsprojekte setzen – wie SSIML/AR – visuelle Entwurfsmodelle im AR-Bereich ein. Eines dieser Beispiele ist das ASUR-Projekt [DSG02, DGN03], das eine grafische Notation für die Modellierung von AR-Systemen in einer frühen Stufe des Entwicklungsprozesses bietet. Der Schwerpunkt von ASUR liegt darauf, die Interaktion eines Benutzers mit einem Augmented Reality - bzw. Mixed Reality - System plattformunabhängig beschreiben zu können. Um einen Übergang zur Implementierung zu ermöglichen, kann ein ASUR-Modell werkzeuggestützt in ein ASUR-IL-Modell [GDB07] transformiert werden. Dieses spezifiziert die Software-Komponenten und ihre Beziehungen untereinander, welche zur Realisierung der im ASUR-Modell beschriebenen Interaktionen benötigt werden. Das ASUR-IL-Modell kann wiederum auf eine Menge von Komponenten der WComp-Plattform [CCT+06] abgebildet werden, welche die durch das ASUR-Modell beschriebenen Interaktionsmöglichkeiten programmtechnisch umsetzt. Im Gegensatz zu SSIML/AR berücksichtigt ASUR beim Entwurf nicht die hierarchische Strukturierung vieler AR-UIs. Daher sind die tatsächlichen Vorteile eines Einsatzes von ASUR gegenüber einer konventionellen Vorgehensweise in einigen AR-Anwendungsbereichen wie Montage und Wartung, in denen eine hierarchische Strukturierung des AR-UIs besonders häufig auftritt, schlecht abschätzbar.

Eine anderer Ansatz, der im Bereich AR visuelle Modellierung verwendet, ist die bereits in den Abschnitten 5.7 und 7.4 erwähnte *Augmented Presentation and Interaction Language* (APRIL) [Led04, LS05]. Der Schwerpunkt liegt bei APRIL – im Gegensatz zu SSIML/AR – nicht auf dem hierarchischen Design der Benutzungsschnittstelle. Außerdem ist die Intention von APRIL nicht unbedingt, eine plattformunabhängige Beschreibung des AR-UIs zu erhalten. Vielmehr ist eine APRIL-basierte Beschreibung einer AR-Präsentation speziell

für die Ausführung auf der Studierstube-Plattform [SFH+02] vorgesehen. Trotzdem teilt APRIL einige Aspekte mit SSIML/AR, wie die visuelle Modellierung von Präsentationsabläufen, die in SSIML/AR durch die Integration des Taskflow-Modells aus SSIML/Tasks beschrieben werden können, und die Übersetzung der visuellen Modelle in ein ausführbares Format.

## 9.7 Zusammenfassung

In diesem Kapitel wurde die Notwendigkeit einer strukturierten Vorgehensweise bei der Entwicklung von AR-Anwendungen diskutiert. Die hier vorgestellte neuartige Modellierungssprache SSIML/AR, eine Erweiterung von SSIML (s. Kapitel 5), stellt einen modellbasierten Ansatz zur Unterstützung einer strukturierten AR-Entwicklung dar. Insbesondere ermöglicht SSIML/AR die Modellierung von AR-UIs durch die Einführung von AR-spezifischen Modellelementen. So können z. B. physikalische Elemente bei der Modellierung der Benutzungsschnittstellenstruktur berücksichtigt werden. In Verbindung mit dem in Kapitel 6 beschriebenen Taskflow-Modell zielt SSIML/AR auf die Erleichterung der Entwicklung von AR-Anwendungen in aufgabenzentrierten Bereichen wie Montage und Wartung ab, da, wie ebenfalls gesagt, AR-Technologien in solchen Domänen große Potenziale besitzen. Ähnlich wie SSIML erlaubt SSIML/AR die Spezifizierung der Benutzungsschnittstelle in einer szenengraphorientierten Form, wobei auch die taskabhängige Visualisierung von Schnittstellenelementen einbezogen werden kann. Beziehungen zwischen Benutzungsschnittstellenelementen und anderen Anwendungsbestandteilen wie Tracking- und Rendering-Komponenten können im SSIML/AR-Interrelationenmodell festgelegt werden.

Die automatische Übersetzung von SSIML/AR-Modellen in Code dient dem nahtlosen Übergang zur Implementierung sowie der Reduzierung von Implementierungsfehlern und -kosten. Des Weiteren können die SSIML/AR-Modelle zu Dokumentationszwecken oder als Diskussionsgrundlage für ein bestimmtes AR-UI-Design dienen. Wie auch SSIML und die SSIML-Erweiterungen wurde SSIML/AR als Metamodell spezifiziert, das anschließend zum Zweck der Integration in UML-Werkzeuge auf ein UML-Profile abgebildet wurde. Eine informelle Evaluierung von SSIML/AR (und damit implizit auch eine Evaluierung der Basissprache von SSIML/AR, SSIML) wird in Abschnitt 11.3 beschrieben.

# Kapitel 10

# SSIML-Entwicklungsprozess

An dieser Stelle wird ein 3D-Entwicklungsprozess skizziert, der einen sinnvollen Einsatz von SSIML unterstützt (s. Abbildung 10.1, vgl. a. [VP05, VH06, PVH07]). In dem Prozess werden 5 Hauptentwicklerrollen unterschieden: *Software-Designer*, *3D-Entwickler*, *Programmierer*, *Framework-Entwickler* und *Transformationsdesigner*. In einem realen Projekt kann es durchaus eine detailliertere Aufteilung einiger Entwicklerrollen geben; ebenso ist es für kleinere Projekte denkbar, dass zwei Rollen in ein- und derselben Person vereint sind. Die oben getroffene Unterteilung der Rollen ist jedoch zur Beschreibung des SSIML-Entwicklungsprozesses ausreichend. Die Rollen Software-Designer und Transformationsdesigner existieren im herkömmlichen 3D-Entwicklungsprozess im Normalfall nicht; sie wurden zusätzlich eingeführt.

Der Software-Designer erstellt die visuellen SSIML-Modelle der 3D-Anwendung. Dies erfolgt in Absprache mit den anderen Entwicklern. Somit können Anregungen aller Entwickler in die Modellierung einfließen. Sind die Modelle fertiggestellt, generiert der Software-Designer Code daraus. Entsprechende Generatoren werden vom Transformationsdesigner entwickelt. Aus ein und demselben Modell wird sowohl Code für den 3D-Entwickler als auch für den Programmierer erzeugt.

Für den 3D-Entwickler wird automatisch ein so genanntes *3D-Template* (z. B. in einer Markup-Sprache wie VRML oder X3D) erzeugt (vgl. Abschnitt 5.2). Dieses beschreibt (Teil-)Strukturen der 3D-Szene und enthält benannte Platzhalter für 3D-Objekte. Das 3D-Template kann in ein 3D-Autorenwerkzeug importiert und dort mit Inhalten komplettiert werden. Somit kann der 3D-Entwickler das Template mit gewohnten Mitteln bearbeiten. Ein 3D-Template kann mehrfach zur Implementierung von verschiedenen 3D-Szenen gleicher Struktur benutzt werden. Bspw. könnte ein 3D-Template mit den Geometrien verschiedener Fahrzeugtypen ausgefüllt werden.

Das Codeskelett für den Programmierer enthält Programmcode (z. B. in Java oder C++). Der Programmierer kann Lücken im Skelett mittels eines geeigneten Programmierwerkzeuges ausfüllen.

Durch die Generierung von 3D-Template und Programmcode aus ein und demselben Modell wird die *Konsistenz* zwischen dem Anwendungscode und der 3D-Szene, die das visu-

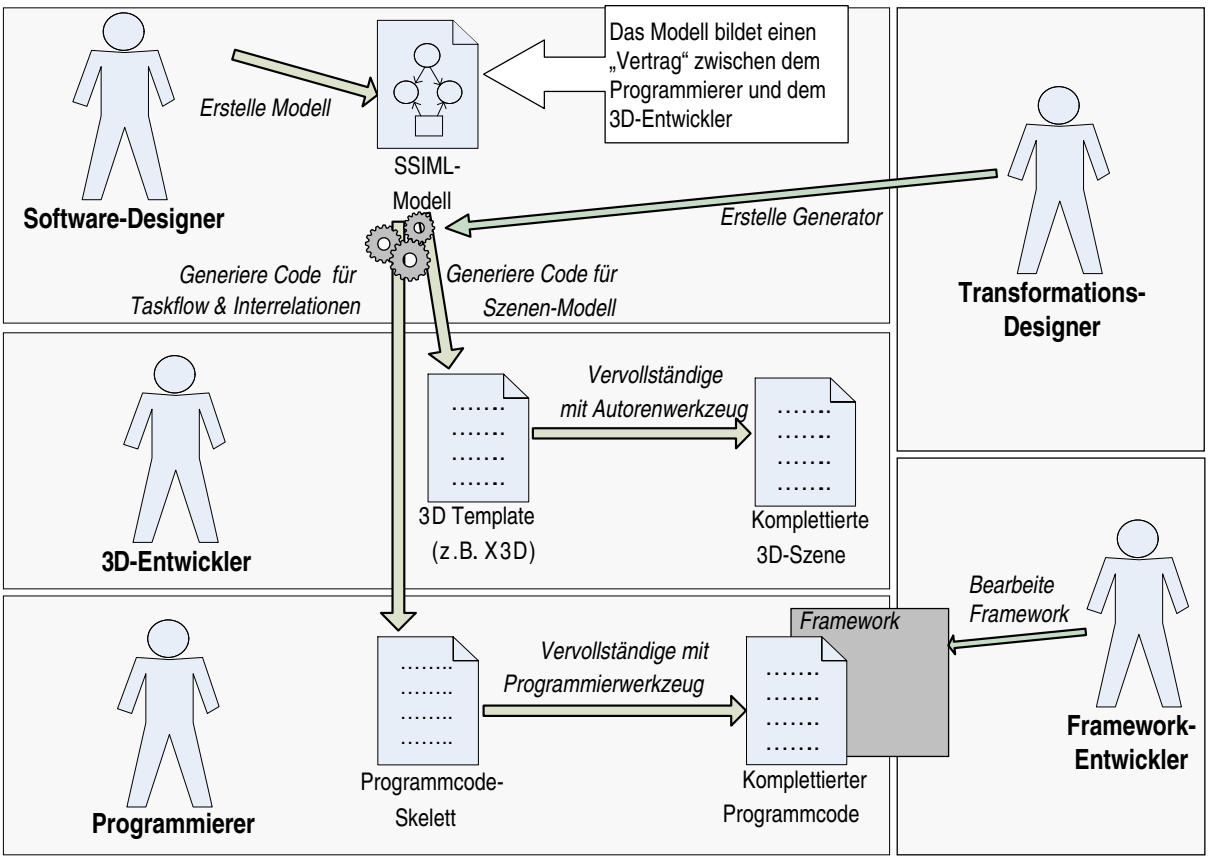



Abbildung 10.1: Der SSIML-Entwicklungsprozess

elle Erscheinungsbild der 3D-Benutzungsschnittstelle bestimmt, sichergestellt. Das Modell dient damit als eine Form eines *Vertrages* zwischen den verschiedenen Entwicklern.

Der erzeugte Programmcode setzt i. d. R. auf einem Framework (etwa einer bestimmten 3D-Engine) auf, dessen Erstellung, Anpassung und Erweiterung in der Verantwortlichkeit des Framework-Designers liegt.

## 10.1 Reverse- und Roundtrip-Engineering

Idealerweise sollte der SSIML-Entwicklungsprozess es erlauben, spätere Änderungen im SSIML-Modell auch im Code zu reflektieren sowie umgekehrt Änderungen im Code auf das Modell zurück zu übertragen. Somit würde ein *Roundtrip-Engineering* realisiert. Ebenfalls sollte es möglich sein, bereits existierende 3D-Szenen in ein SSIML-Szenenmodell zu übersetzen, bspw. um die Szene in ein vorhandenes Modell einzubinden oder um sie auf einem höheren Abstraktionsniveau zu visualisieren bzw. zu bearbeiten (*Reverse-Engineering*). Derzeit wird durch die existierenden SSIML-Werkzeuge nur *Forward-Engineering* unterstützt (s. Abschnitt 4.4). Eine umfassende Betrachtung der Möglichkeiten des *Roundtrip-*

und *Reverse*-Engineerings von 3D-Anwendungen würde den Rahmen dieser Arbeit sprengen. Im Folgenden wird allerdings ein möglicher Ansatz zur Erzeugung eines SSIML-Szenenmodells aus einer bestehenden 3D-Szene skizziert (vgl. a. [PVH07]).

**Ableitung eines SSIML-Modells aus einer existierenden 3D-Szene** Eine geeignetes Basisformat für die Generierung eines Modells aus einer vorhandenen 3D-Szene wäre ein XML-basiertes Szenenformat wie X3D. Eine XML-codierte 3D-Szene kann mittels XSLT in ein XMI-codiertes SSIML-Modell übersetzt werden. Namen und Typen von SSIML-Elementen gehen weitestgehend aus den Namen und Typen der entsprechenden konkreten Szenenelemente hervor. So kann etwa ein X3D-`Material`-Knoten in ein SSIML-`Material`-Attribut übersetzt werden. Informationen, die nicht vollautomatisch aus der 3D-Szene extrahiert werden können, müssen manuell mit in das Modell aufgenommen werden. Die Transformation einer 3D-Szene in ein SSIML-Modell sollte – bspw. durch Speicherung in einer separaten XML-Datei – auch Eigenschaftswerte von Szenenobjekten bewahren, die in einem SSIML-Diagramm u. U. nicht visualisiert werden, wie konkrete Farb-, Positions- und Orientierungswerte. Somit kann zu einem späteren Zeitpunkt eine verlustfreie Rückübersetzung eines ggf. angepassten Modells in X3D-Code oder ein anderes 3D-Format garantiert werden.

# Kapitel 11

# Evaluierung

## 11.1 Experteninterviews

Im Rahmen der vorliegenden Arbeit wurde eine informelle Befragung von sechs Experten aus dem Bereich der 3D-Anwendungsentwicklung durchgeführt. Ziel der Befragung war es, ein Feedback zum SSIML-Ansatz einzuholen und mögliche Verbesserungsvorschläge von Expertenseite zu bekommen, wobei die Experten gleichzeitig eine potenzielle Anwendergruppe des Ansatzes darstellten. Bei der Auswahl der Interviewpartner wurde der Schwerpunkt auf Personen gelegt, die Erfahrungen in der 3D-Entwicklung im industriellen Umfeld hatten. Fünf der sechs befragten Personen verfügten über solche Erfahrungen, während ein Interviewpartner entsprechende Erfahrungen im Forschungsumfeld besaß.

Die Fokussierung auf das industrielle Umfeld resultiert daraus, dass Probleme, für die SSIML einen Lösungsansatz darstellt, vorrangig in diesem Bereich auftreten. Die Entwicklung ist hier stärker durch Kosten- und Zeitvorgaben getrieben und häufiger von interdisziplinärer Zusammenarbeit geprägt als im Forschungsumfeld. 3D-Entwicklung im Forschungsbereich verfolgt i. d. R. andere Ziele als die termingerechte Fertigstellung eines Produktes für einen Kunden. Ziele einer 3D-Entwicklung im Forschungsumfeld können z. B. die Demonstration eines neuartigen Forschungsansatzes oder einer innovativen 3D-Technologie sein. 3D-Anwendungen besitzen in diesem Bereich daher häufiger einen prototypischen Charakter. Die Ansprüche an das Grafikdesign sind hier aus den genannten Gründen oft ebenfalls geringer und Programmierer sind nicht selten gleichzeitig auch 3D-Designer.

Die in den folgenden Abschnitten getroffenen Aussagen können natürlich nicht für jeden Fall als zutreffend, d. h. als allgemeingültig angesehen werden; allerdings geben sie qualitative Hinweise auf die Benutzbarkeit der Sprache SSIML und den möglichen Nutzen ihres Einsatzes.

### 11.1.1 Ablauf der Befragung

Die Befragung der Experten fand in vier Phasen statt; die erste Phase umfasste *Fragen zur Person*, *Fragen zu Vorkenntnissen im Bereich 3D-Programmierung und -Modellierung*,

*Fragen zur Vorgehensweise bei der 3D-Entwicklung* und *Fragen zu Problemen im Entwicklungsprozess*. In der zweiten Phase folgte eine kurze *Präsentation des SSIML-Ansatzes* in Form einer – mit Audiokommentaren unterlegten – Video-Präsentation. Im Einzelnen wurden der SSIML-Basisteil (Kapitel 5) und der SSIML/Entwicklungsprozess (Kapitel 10) erklärt. SSIML-Erweiterungen wie SSIML/Behaviour (Kapitel 7) und SSIML/Components (Kapitel 8) wurden nur erwähnt, aber nicht im Detail erklärt, da dies den zeitlichen Rahmen gesprengt hätte. Die dritte Phase beinhaltete *Fragen zum SSIML-Ansatz*. Zwei der Befragten nahmen an der letzten Phase teil, bei welcher sie die Möglichkeit hatten, einen kurzen *Cognitive Walkthrough* anhand der Modellierung eines kleinen Projekts durchzuführen. Bei diesem kam das UML-Werkzeug MagicDraw mit dem SSIML-UML-Profile zum Einsatz. Diese vierte Phase war optional, da der Hauptfokus der Befragung auf der Bewertung der SSIML-Konzepte und nicht auf der Bewertung der Werkzeugunterstützung für die Modellierungssprache lag.

Die Befragung der Experten erfolgte auf Basis eines zuvor erstellten Fragebogens und im Rahmen von zwei Einzelarbeiten [Jah07, Alz07], die im Kontext des SSIML-Projektes durchgeführt wurden. Vier der Interviewteilnehmer wurden in einem *Face-To-Face*-Interview befragt, zwei weitere in Telefoninterviews. Die Gespräche wurden schriftlich protokolliert und auch als Audiomitschnitte für die spätere Rekapitulation festgehalten.

### 11.1.2 Ergebnisse der Befragung

#### 11.1.2.1 Personenbezogene Fragen

Alle Befragten waren männlich und im Alter zwischen 25 und 40 Jahren. Sie stammten aus verschiedenen Bereichen der 3D-Entwicklung, wie Entertainment, Medizintechnik, CAD und Forschung. Ebenso verschieden waren die beruflichen Positionen der Interviewteilnehmer. Diese waren z. B. Werkstudent, Entwickler, mittleres Management und Geschäftsführer. Typische Aufgabenbereiche der Interviewten waren die Grafikerstellung, Programmierung und Projektleitung. Alle Befragten verfügten über Erfahrungen im Umgang mit 3D-APIs und Szenengraphen. Am häufigsten genannt wurden APIs und Szenengraphformate wie OpenGL [Sil06], Open Inventor [SC92], VRML [Int97] und 3D-Engines wie Quest3D [Que] und Shark3D [Sha]. Vier der Befragten hatten ebenfalls Erfahrung im Umgang mit 3D-Autorenwerkzeugen. Genannt wurden hier die Werkzeuge 3ds max [3ds], Maya [May] und Blender3D [Ble].

#### 11.1.2.2 Fragen zum Entwicklungsprozess

Der allgemeine Ablauf der 3D-Entwicklung entspricht der Beschreibung in Abschnitt 2.6. Besonders wurde die Wichtigkeit der Spezifikation des zu realisierenden Software-Produktes unterstrichen. Diese wird i. d. R. schriftlich festgehalten. Graphen werden zur Spezifikation ebenfalls verwendet, aber eher in informeller Form. Die UML-Notation kommt dabei selten zum Einsatz, häufiger werden Szenengraphen skizziert. Allerdings war die UML allen

Befragten bekannt. Einer der Befragten gab an, Zustandsautomaten für die Spezifikation zu benutzen.

Ein typisches Projektteam bei der 3D-Entwicklung besteht aus ca. 5 Personen, wobei die tatsächliche Zahl der Teammitglieder je nach Projekt stark differieren kann. Das größte Projekt, an dem einer der Befragten beteiligt war, umfasste 25 Personen. Kommuniziert wird in den Teams meist mündlich, in Meetings, über Chat oder kollaborative webbasierte Umgebungen wie Wikis. Allerdings ist die Protokollierung von Gesprächen oft unzureichend, so dass Informationen z. T. dann nicht vorliegen, wenn sie benötigt werden.

Es wurde bestätigt, dass Kommunikationsschwierigkeiten zwischen 3D-Designern und Programmierern, die durch unterschiedliches Vokabular und Hintergrundwissen zustande kommen, ein zentrales Problem im Entwicklungsprozess darstellen.

Es wird versucht, die Konsistenz zwischen Code und Content durch strenge Konventionen bei der Benennung von 3D-Objekten sicherzustellen. Allerdings halten sich 3D-Designer teilweise nicht an die Konventionen, wodurch eine Nacharbeit notwendig wird. Es mangelt an Software-Werkzeugen, welche die Entwickler bei der Einhaltung der Namenskonvention unterstützen.

#### 11.1.2.3 Reaktionen auf SSIML: Stärken des Ansatzes

SSIML wurde als sinnvolles Hilfsmittel bei der Kommunikation zwischen den Entwicklern und zur Festlegung des hierarchischen Aufbaus der 3D-Modelle bewertet. Ebenfalls wurde geäußert, dass das hohe Abstraktionsniveau von SSIML zu einer effizienteren Kommunikation beitragen kann. Zusätzlich wird den Entwicklern ein „Korsett“ vorgegeben, das bei der Durchsetzung von Richtlinien, bspw. bei der Einhaltung von Namenskonventionen, und der Ausschaltung von Fehlerquellen hilft. Durch die klaren Strukturfestlegungen der Modelle wird auch der Interpretationsspielraum der Entwickler reduziert. Außerdem wurde der Gedanke geäußert, dass SSIML bei der ersten Strukturierung neu zu erstellender 3D-Modelle ein hilfreiches Werkzeug darstellen würde.

Der Aufwand für Fachleute für das Erlernen der Sprache SSIML wurde als „einfach“ bis „mittel“ eingestuft, da der Ansatz auf bekannten Konzepten wie Szenengraphen aufbaut. Einer der Befragten gab an, dass er z. B. das Erlernen einer Programmiersprache oder den Umgang mit einem komplexen 3D-Autorenwerkzeug als deutlich schwerer einstufen würde.

Die Trennung von Programmcode und 3D-Inhalten wurde prinzipiell als sinnvoll erachtet, ebenso die Kapselung von Subgraphen in *Composed Nodes* oder Komponenten. Als Vorteile der Separierung von Code und 3D-Inhalten wurde angegeben, dass damit die Wiederverwendbarkeit gesteigert werden kann, indem vorhandener Code bspw. zusammen mit verschiedenen, ähnlich strukturierten 3D-Modellen benutzt wird. Ebenso wurde diese Trennung als realistisch angesehen, da auch in der Praxis eine getrennte Entwicklung von Code und 3D-Modellen erfolgen muss.

Die Kapselung von Subgraphen wurde als wichtiges Konzept angesehen, um auch komplexere Szenen effizient zu verwalten. Ebenfalls könne das Konzept zur Wiederverwendung von 3D-Objekten beitragen. Einer der Interviewteilnehmer bemerkte, dass auch 3D-APIs z. T. ähnliche Möglichkeiten der Kapselung bieten.

Weiterhin wurde die Modellierung und Darstellung von Beziehungen zwischen 3D-Objekten und Codekomponenten als hilfreich angesehen, u.a. auch deshalb, weil sich damit Zusammenhänge visualisieren lassen, die im Code nicht immer sofort erkennbar sind.

Die Möglichkeit, mit SSIML Objekte zu annotieren, wurde positiv bewertet, da in der Praxis ähnliche Vorgehensweisen gebräuchlich sind. So können etwa 3D-CAD-Modelle annotiert werden, um andere Entwickler darüber zu informieren, welche Anpassungen an dem Modell vorzunehmen sind.

Als besonders wichtig wurde die Fähigkeit der Generierung von Code aus den Interrelationenmodellen empfunden, um Konsistenz und paralleles Arbeiten zwischen den Entwicklern zu unterstützen. Ebenfalls wurde als Vorteil gewertet, dass die Generierung von Programmteilen, die Standardfunktionalitäten realisieren, dem Programmierer erlaubt, sich auf kreativere Programmieraufgaben zu konzentrieren. Als weiterer Vorteil wurde die Eignung der Kombination von Modellierung und automatischer Codegenerierung zum Rapid Prototyping gesehen. Rapid Prototyping erlaubt es wiederum, dem Kunden relativ früh im Entwicklungsprozess einen funktionsfähigen Anwendungsprototyp für eine erste Bewertung zu präsentieren.

Die Verknüpfung von Semantik mit 3D-Objekten (durch Repräsentationsbeziehungen) wurde nicht als Hauptvorteil, aber doch als sinnvolle Zusatzmöglichkeit der Modellierung betrachtet.

Alle Entwickler mit Industrieerfahrung sagten aus, sie könnten sich prinzipiell vorstellen, SSIML in Projekten einzusetzen. Ein Befragter stellte fest, dass vor einem Einsatz von SSIML natürlich die Möglichkeit bestehen müsse, die Sprache und entsprechende Werkzeuge intensiv testen zu können. Als wichtige Randbedingung für die Akzeptanz des Ansatzes wurde eine stabile Werkzeugunterstützung angegeben. Einer der Befragten meinte zudem, dass ein professionelles *Look & Feel* eines Werkzeuges ebenfalls entscheidend zu seiner Akzeptanz beiträgt. Die Befragten gaben an, dass sie SSIML bereits zu Beginn des Entwicklungsprozesses zur Software-Spezifikation einsetzen würden.

#### 11.1.2.4 Reaktionen auf SSIML: Schwächen des Ansatzes

Als mögliche Schwächen des SSIML-Ansatzes wurden insbesondere die folgenden Punkte gesehen:

- Die SSIML-Modelle könnten bei großen Anwendungen unübersichtlich werden. Ähnliche Erfahrungen gab es bei einigen Befragten bereits mit der UML.

- Der Ansatz ist nicht geeignet, Ideen an Kunden zu vermitteln, da er für Kunden ohne das erforderliche fachspezifische Wissen kaum verständlich sein dürfte.

- Es könnten sich vielleicht bislang nicht absehbare Situationen ergeben, die mit der Sprache nicht modellierbar sind.

- Die Notation ist nicht intuitiv. Sie muss erlernt werden.

#### 11.1.2.5 Verbesserungsvorschläge

Die Interviewteilnehmer wurden gebeten, Vorschläge zur Verbesserung des SSIML-Ansatzes zu unterbreiten. Viele dieser Vorschläge beziehen sich auf die Werkzeugunterstützung, einige auch auf die grafische Notation und das Meta-Modell von SSIML.

**Vorschläge zu Änderungen im SSIML-Meta-Modell** Der Begriff der *Application-Class* für Elemente, welche auf die Szene zugreifen, ist vielleicht zu allgemein, da diese Elemente eine sehr spezielle Funktion haben. Eventuell sollte der Typ dieser Modellelemente umbenannt werden.

**Vorschläge zur Verbesserung der SSIML-Notation** Unterschiedliche Vorschläge gab es zur Verbesserung der SSIML-Notation. Eine Ansicht war, dass die SSIML-Notation durch den Einsatz einer bildorientierten Symbolik verbessert werden kann. Eine derartige Symbolik sollte so gewählt werden, dass sie für den Benutzer intuitiv zu verstehen ist und er sie eventuell auch mit bereits bekannten Konzepten assoziieren kann.

Einer der Befragten unterbreitete dem entgegen den Vorschlag, die Notation sogar noch stärker an der UML zu orientieren, um Benutzern des Ansatzes, die bereits Erfahrungen im Umgang mit der UML besitzen, den Einstieg in die Modellierung weiter zu erleichtern.

Eine farbige Gestaltung der grafischen Repräsentationen der SSIML-Modellelemente anstelle einer Schwarz-Weiß-Darstellung wurde von einigen, aber nicht von allen Befragten als sinnvoll empfunden. Als möglicher Nachteil einer farblichen Darstellung wurde genannt, dass nach einem Ausdruck eines Modells die Modellelemente u. U. schlecht auf dem Papier zu unterscheiden sind.

**Anforderungen an eine SSIML-Werkzeugunterstützung** Wie bereits erläutert, ist ein sinnvoller und effizienter Einsatz der Konzepte von SSIML nicht zuletzt abhängig vom Vorhandensein einer entsprechenden Werkzeugunterstützung. Für die Modellierung und automatische Codegenerierung bspw. sind gute Software-Werkzeuge unerlässlich. Die aus der Befragung stammenden Vorschläge zur Verbesserung der SSIML-Werkzeugunterstützung sind zugleich als Anforderungen an ein zukünftiges optimiertes SSIML-Werkzeug zu verstehen und folgendermaßen beschreibbar:

- Mechanismen zur Versionskontrolle sollten vorhanden sein.
- Die Unterstützung der Synchronisation von Modellen und Code ist äußerst wünschenswert. Mit einem *Roundtrip*-Engineering würde das Rückführen von Änderungen im Code in das Modell möglich. Ebenso wird damit ein iterativer Prozess unterstützt.
- Ein SSIML-Werkzeug sollte durch die Entwickler anpassbar und erweiterbar sein.
- Ein SSIML-Werkzeug sollte zudem projektspezifisch anpassbar sein.

- Klassen, die auf die Szene zugreifen und aus dem Interrelationenmodell generiert werden, sollten einer strengen Kontrolle unterstellt sein. Es muss sichergestellt werden, dass der Programmierer die Klassen nicht beliebig verändern kann.

- Die Codegenerierung sollte verschiedene Zielformate unterstützen, vor allem aber weit verbreitete Formate, die den Anforderungen heutiger 3D-Anwendungen gerecht werden, z. B. Collada.

- Mechanismen wie automatisches Layout-Management und Kantenglättung verbessern das *Look & Feel* eines SSIML-Editors. Eine ästhetisch anmutende Benutzungsoberfläche eines Werkzeuges kann dessen Akzeptanz enorm beeinflussen, insbesondere bei kreativ orientierten Entwicklergruppen.

- Ein SSIML-Modellierungswerkzeug sollte die benutzergruppenspezifische Umschaltung der Notation oder sogar die individuelle Anpassung der Notation erlauben, um den verschiedenen Notationsvorschlägen gerecht zu werden.

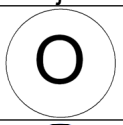
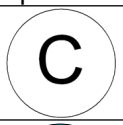
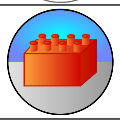
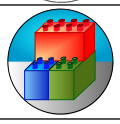
### 11.1.3 Zusammenfassung der Interviews und resultierende Arbeiten

In den Entwicklerinterviews wurde die Wichtigkeit der in Kapitel 5 angesprochenen Probleme bestätigt. Der SSIML-Ansatz wurde insgesamt positiv bewertet. Verbesserungspotenzial wurde bei der Notation gesehen, wobei die Vorstellungen hinsichtlich des konkreten Aussehens einer graphischen Repräsentation der Modellelemente auseinander gingen. Von der Mehrheit der Befragten als sinnvoll wurde die Einführung einer (weiteren) Notation erachtet, welche speziell auf die Bedürfnisse kreativ ausgerichteter Entwickler zugeschnitten ist. Diese sollte es erlauben, einen Bezug zu Bekanntem herzustellen und sich stärker an einer bildlichen Symbolik orientieren, z. B. durch Gebrauch einer auch aus 3D-Autorenwerkzeugen bekannten Symbolik. Motiviert durch die Ergebnisse der Experteninterviews wurde in der Arbeit von Jahn [Jah07] eine alternative Notation für einen Teil der SSIML-Elemente entwickelt. Diese Notation bedient sich bekannter Metaphern (in diesem Fall z. B. des Lego-Steins), um einen höheren Grad an Intuition zu erreichen (Tabelle 11.1).

Die bildorientierte Notation soll die in der vorliegenden Arbeit vorwiegend benutzte grafemorientierte Notation nicht ersetzen, sondern vielmehr ergänzen. Alzetta [Alz07] stellt in Verbindung mit der Erstellung eines umfangreicheren SSIML-Modells fest, dass die grafemorientierte Notation nach einer kurzen Lernphase ebenfalls gut benutzbar ist. Im Vergleich zu der bildorientierten Notation der Modellelemente erlaubt sie eine bessere Erkennbarkeit der Elemente nach einer Skalierung (Verkleinerung) eines Diagramms [Alz07].

In den Experteninterviews wurde weiterhin hervorgehoben, dass das SSIML-Konzept besonders in enger Kopplung mit einer stabilen Werkzeugunterstützung gewinnbringend einsetzbar ist. Da sich die vorliegende Arbeit vorrangig auf die Konzeption eines Ansatzes

Tabelle 11.1: Beispiele für alternative Notationen von SSIML-Elementen (nach Jahn [Jah07])

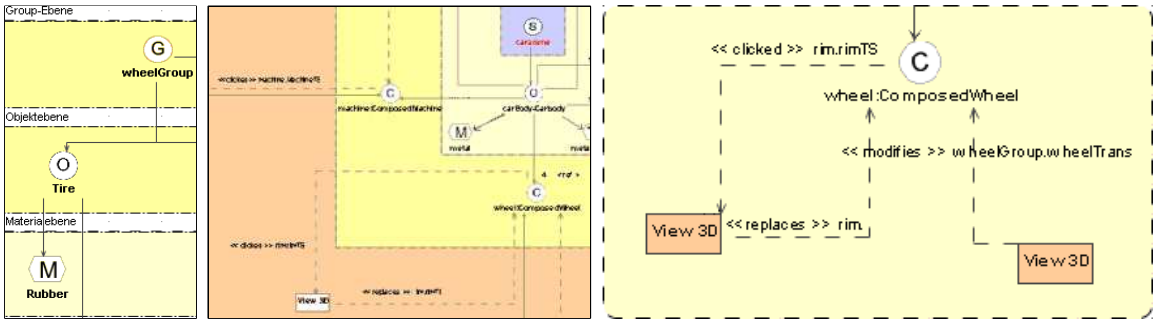
| | Object | ComposedNode |
|---|---|---|
| Grafem-orientierte Darstellung |  |  |
| Bild-orientierte Darstellung |  |  |

zur 3D-Entwicklung konzentriert, würde die Realisierung einer entsprechenden umfangreichen Werkzeugunterstützung ihren Rahmen bei weitem sprengen. Somit wurden Werkzeuge nur prototypisch realisiert, um die Machbarkeit des SSIML-Ansatzes zu demonstrieren. Allerdings wurde als ein Schritt hin zu einer verbesserten Werkzeugunterstützung in der Arbeit von Alzetta [Alz07] auf Basis der Microsoft DSL-Tools [CJKW07] ein auf SSIML spezialisierter Editor entwickelt, der das SSIML-Kernpaket unterstützt (vgl. Kapitel 5). Dieser Editor erlaubt im Gegensatz zu der bislang verwendeten UML-basierten Lösung bspw. die Abdeckung aller Constraints des SSIML-Meta-Modells. Somit wird die Erstellung nicht SSIML-konformer Modelle verhindert. Da die DSL-Tools nicht auf dem MOF-Meta-Meta-Modell basieren, musste das SSIML-Meta-Modell zunächst manuell auf ein zum Meta-Meta-Modell der DSL Tools konformes Meta-Modell abgebildet werden.

## 11.2 Einzelfallstudien

In den Arbeiten von Jahn [Jah07] und Alzetta [Alz07] wurde jeweils ein umfangreicheres Projekt unter Verwendung des SSIML-Ansatzes modelliert. In beiden Fällen kam das SSIML-UML-Profile in Verbindung mit einem UML-Editor (konkret: MagicDraw UML [Mag], Version 11.5) zum Einsatz. In der Arbeit von Jahn [Jah07] wurde eine Anwendung zur Konfiguration von Fahrzeugen ähnlich dem in Kapitel 5 vorgestellten Beispiel, allerdings deutlich umfangreicher, und in der Arbeit von Alzetta ein 3D-Fertighauskonfigurator, wie er teilweise in ähnlicher Form auf den Webseiten von Fertighausanbietern und Bauzulieferern (z. B. Schüco [Sch]) existiert, spezifiziert. Ziele der Arbeiten waren das Aufdecken von Problemen bei der Modellierung und das Unterbreiten von entsprechenden Lösungsvorschlägen. Viele der aufgetretenen Schwierigkeiten ließen sich hauptsächlich auf den Umgang mit dem Modellierungswerkzeug und Einschränkungen bei der Sprachunterstützung durch die Verwendung eines UML-Werkzeuges (z. B. eingeschränkte Möglichkeiten zur Validierung von Constraints des SSIML-Meta-Modells) zurückführen. Schwerpunkt war allerdings die Identifikation von Problemen, die sich nicht auf die Werkzeugunterstützung, sondern auf die abstrakte und konkrete Syntax (die Notation) von SSIML selbst beziehen.

In der Arbeit von Jahn [Jah07] wurden insbesondere zwei Schwächen der SSIML-Notation identifiziert. Zum einen ist es oft schwierig, in umfangreicheren SSIML-Szenen-




(a) Top-Down (b) Konzentrisch (c) Bounding Box

Abbildung 11.1: Layout-Varianten zur visuellen Strukturierung von SSIML-Modellen

und Interrelationenmodell-Diagrammen das Wurzelelement der Gesamtszene unmittelbar zu erkennen, zum anderen können die Diagramme mit steigender Komplexität an Übersichtlichkeit verlieren. Jahn [Jah07] versucht, diese Schwächen durch Anwendung verschiedener Layouts, insbesondere einer streng hierarchischen Top-Down-Anordnung (Abbildung 11.1(a)) und einer konzentrischen Elementanordnung um das Wurzelelement des Szenenmodells (Abbildung 11.1(b)), zu kompensieren. Bei beiden Layouts wurden zusätzlich z. T. farblich codierte Zonen eingeführt, die Knoten gleicher Hierarchiestufen enthalten. Die Zonen wurde teilweise in Subzonen für bestimmte Knotenattribute und für die mit Szenenelementen in Beziehung stehenden `ApplicationClass`-Elemente unterteilt (Abbildung 11.1(a)). Um auf einzelne Aspekte des Modells fokussieren zu können, wurde die Möglichkeit angedacht, einzelne Zonen ein- und ausblenden zu können. Zusätzlich wurde eine besondere farbliche Codierung des Szenenwurzelelements vorgeschlagen, um dieses von anderen Modellelementen leichter unterscheiden zu können. Ein alternativer Vorschlag, um die Strukturierung des Diagramms zu verdeutlichen, ist die Umrahmung zusammengehöriger Modellelemente durch Bounding-Boxes, z. B. die Umrahmung eines SSIML-Knotens mit allen assoziierten Attributen und Anwendungsklassen (Abbildung 11.1(c)). Als zu bevorzugendes Layout bei sehr umfangreichen Diagrammen wurde die streng hierarchische Top-Down-Anordnung der Elemente bewertet, da diese Darstellung zusammen mit den bereits erwähnten Zonen besonders gut die Struktur des Szenenmodells widerspiegelt. Dies konnte in der Arbeit von Alzetta [Alz07] bestätigt werden, in der zunächst die gleichen Probleme wie in [Jah07] identifiziert wurden. Da allerdings bei einer Top-Down-Strukturierung mit zunehmendem Modellumfang gleichsam die physikalische Höhe und Breite des Modells entsprechend anwächst, sollte ein SSIML-Werkzeug geeignete Mechanismen zur Verfügung stellen, mit denen im Modell navigiert werden kann, z. B. ein Fokus- und Kontext-Konzept, wie es einige UML-Tools, darunter MagicDraw, beinhalten (s. Abbildung 11.2).

Allgemein ist zu beachten, dass farbliche Codierungen in einem Modell so zu wählen sind, dass auch bei Graustufen- oder Schwarz-Weiß-Ausdrucken von Modellen die Lesbarkeit derselben nicht negativ beeinflusst wird (vgl. Abschnitt 11.1.2.5). Ebenso ersetzen die angesprochenen Layout-Techniken und der Einsatz farblicher Codierungen nicht die primär zum Management komplexer Modelle vorgesehenen Mechanismen des SSIML-

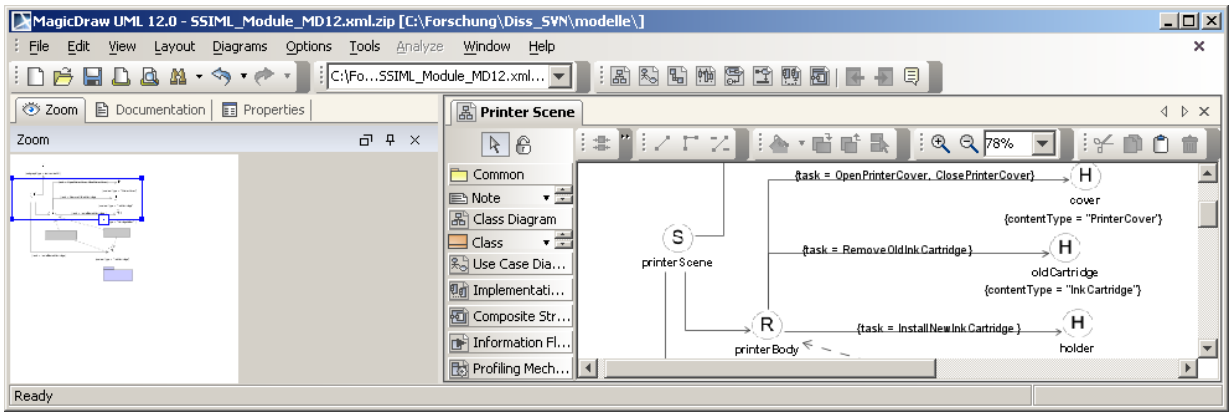



Abbildung 11.2: *Focus* (rechts) & *Context* (links) in MagicDraw [Mag]

Meta-Modells, wie Kompositionsknoten und Komponenten, die zusätzlich gezielt eingesetzt werden sollten.

## 11.3 Projektfallstudie

Um Hinweise auf den Nutzen eines praktischen Einsatzes von SSIML zu erhalten, wurde das auf der SSIML-Kernkomponente aufbauende SSIML/AR in einem Projekt im Rahmen eines Praktikums "Augmented Reality" der Lehr- und Forschungseinheit Medieninformatik an der Ludwigs-Maximilians-Universität München eingesetzt. Ferner fand als Ergänzung zu SSIML/AR die SSIML-Komponente SSIML/Tasks im Projekt Verwendung. An dem Projekt nahmen sechs Studenten der (Medien-)Informatik teil, von denen keiner zuvor Erfahrung mit der Programmierung von AR-Anwendungen oder im Umgang mit SSIML hatte. Ziel des Praktikums war die Erstellung einer Augmented Reality - Anwendung aus dem Bereich Montage und Wartung.

Neben einer Prüfung des Nutzens von SSIML i. Allg. sollte im Projekt insbesondere genauer hinterfragt werden, unter welchen Rahmenbedingungen (Projektgröße, Anwendungsbereich usw.) der Einsatz von SSIML besonders ergiebig ist und wie der SSIML-Ansatz weiter verbessert werden kann.

### 11.3.1 Praktikumsphasen

#### 11.3.1.1 Trainingsphase

Um sich entsprechende Fähigkeiten anzueignen, lösten die Praktikumsteilnehmer nach jeweils einer kurzen thematischen Einführung zunächst mathematische Aufgaben, etwa zur Multiplikation von 3D-Matrizen, und später AR-bezogene Programmieraufgaben. Die programmtechnische Grundlage zur Lösung der Programmieraufgaben stellten das AR-Toolkit [ART] und das jARToolkit [GRSP02], ein Java-Wrapper für das ARToolkit, dar. Die Themenstellungen der Programmieraufgaben hatten u. a. Bezug zum Tracking von einfachen

Markern und Multimarkern, zu 3D-Transformationshierarchien und Szenengraphen sowie zur Integration von 3D-Modellen und Programmcode in ein AR-Gesamtsystem.

#### 11.3.1.2 Projektphase

Nach dieser *Trainingsphase* folgte die eigentliche *Projektphase*. In dieser Phase wurden die Teilnehmer des Praktikums in zwei Teams mit jeweils drei Mitgliedern eingeteilt. Die Projektaufgabe für beide Teams war es, eine AR-Anwendung ähnlich der in Kapitel 9 vorgestellten Beispielanwendung zu entwickeln, die den Benutzer beim Austausch defekter PC-Komponenten unterstützt. Die Teams sollten die Aufgabe auf unterschiedlichen Wegen lösen: *Team C* mit Hilfe des ARToolkits und herkömmlichen Entwicklungstechniken, *Team SSIML* unter Benutzung von SSIML/AR und dem IntegratorAR-Framework als Basis für die Codegenerierung. Die Zuordnung der Teams zu den Entwicklungsansätzen erfolgte zufällig. Jedes Team besaß einen Betreuer mit entsprechendem Hintergrundwissen zum jeweiligen Ansatz, der bei Fragen und Problemen zur Verfügung stand.

**Einführung in die Projektphase** Zu Projektbeginn wurden jedem Team die benötigten Konzepte und Werkzeuge präsentiert. Da das Team SSIML mit dem SSIML/AR-Ansatz zuvor nicht in Berührung gekommen war, wurde für die Vorstellung von SSIML/AR und der damit verbundenen Werkzeugunterstützung (MagicDraw mit dem SSIML/AR-Profile, Saxon-XSLT-Prozessor [SAX] zur Transformation der Modelldateien in Code, 3D-Autorenwerkzeug Cosmo Worlds [Däß02, S. 51], IntegratorAR-Framework) mehr Zeit benötigt als für die Vorstellung der für Team C relevanten Konzepte und Werkzeuge. Die Einführung für Team SSIML dauerte daher vier Stunden, die Einführung für Team C hingegen nur etwa eine Stunde.

**Projektverlauf** Da die Entwicklungsansätze beider Teams sehr unterschiedlich und damit nicht direkt vergleichbar waren, wurden verschiedene Meilensteine definiert, um einen Vergleich auf Basis der erreichten Teilergebnisse in bestimmten Stadien der Entwicklung durchführen zu können. Insgesamt war die Projektphase fünf Wochen lang und enthielt vier Meilensteine:

1. Anwendungskonzeption und Entwurf,
2. AR-Benutzungsschnittstellenprototyp (korrekte Positionierung, Ausrichtung und Darstellung der virtuellen Benutzungsschnittstellenelemente),
3. erster lauffähiger Anwendungsprototyp mit Anbindung der Anwendungslogik,
4. finale Anwendung einschließlich eventuell implementierter Zusatzfeatures.

Zu jedem Meilenstein wurde ein wöchentliches Treffen der Teams mit ihren Betreuern durchgeführt. Zu den Treffen wurden die jeweiligen Teilergebnisse diskutiert und etwaige

Änderungen beschlossen, die bis zur nächsten Zusammenkunft umzusetzen waren. Zusätzlich musste jedes Team die für die Umsetzung des jeweiligen Meilensteins benötigte Zeit in Personenstunden angeben.

Am Ende des Projektes füllten alle Praktikumsteilnehmer einen Fragebogen aus. Zusätzlich wurde mit jedem Team ein informelles Interview durchgeführt.

## 11.3.2 Projektergebnisse

Zunächst ist anzumerken, dass die Ergebnisse der Bewertung von SSIML nicht als umfassend und in jeder Hinsicht gesichert anzusehen sind, u. a. wegen der relativ geringen Anzahl der Teams und Teammitglieder. Die Resultate sind aber wertvoll im Sinne richtungsgebender Hinweise bzgl. der Verwendbarkeit und des Nutzens des SSIML-Ansatzes. Insgesamt erfüllten die entwickelten Anwendungen beider Teams die in der Projektaufgabe aufgestellten Anforderungen vollständig.

### 11.3.2.1 Erzeugte Codemenge

Die Menge des produzierten Codes unterschied sich bei beiden Teams wesentlich. Team SSIML produzierte nur etwa 300 physikalische Zeilen Code (*Physical Source Lines of Code - PSLOCs*), während Team C etwa 1000 Zeilen PSLOCs erzeugte[1]. Hinzu kommt, dass ein Großteil des Codes von Team SSIML, nämlich das 3D-Template im VRML-Format sowie XML- und Java-Code zur Ablaufsteuerung, nicht per Hand erstellt, sondern aus dem Modell generiert und anschließend angepaßt wurde. Der Unterschied in den produzierten Codemengen lässt sich u. a. dadurch begründen, dass Hauptbestandteile der Anwendungslogik, wie das Taskflow-Management, bereits im IntegratorAR-Framework enthalten waren. Team C arbeitete ohne Zuhilfenahme automatischer Codegenerierung und – abgesehen vom ARToolkit – ohne zusätzliche Frameworks. Weiterhin implementierte Team C Zusatzfunktionen wie Animationen, die nicht Bestandteil der Grundaufgabenstellung waren.

### 11.3.2.2 Zeitaufwand

Das Balkendiagramm in Abbildung 11.3(a) zeigt die Zeiten der Teams in Personenstunden, die für die Realisierung der jeweiligen Meilensteine benötigt wurden. Daraus wird deutlich, dass das Team SSIML insbesondere bei den Meilensteinen 1 und 2 deutlich weniger Zeit benötigte als Team C. Zusätzlich zu den in der Abbildung dargestellten Zeiten gab Team C an, etwa 37 Personenstunden für die Implementierung von zusätzlichen, über die eigentliche Aufgabenstellung hinausgehende Funktionalitäten, wie 3D-Animationen und die automatische Task-Weiterschaltung durch die Erkennung von Markern, benötigt zu haben, während Team SSIML auf die Implementierung derartiger Features nahezu komplett verzichtete. Um

[1]Bei beiden Teams wurde der Code für die verwendeten 3D-Modelle mit den Geometriebeschreibungen nicht eingerechnet. Die 3D-Modelle wurden mit Modellierungswerkzeugen wie 3D-Studio erstellt und anschließend in das VRML-Format exportiert, da sowohl das ARToolkit als auch das IntegratorAR-Framework Funktionen zum Import von VRML anbieten.

vergleichbare Ergebnisse zu erhalten, wurden daher die Zeiten für die Implementierung optionaler Funktionalitäten in Abbildung 11.3(a) nicht berücksichtigt. Bei Betrachtung eines vergleichbaren Umfangs an realisierten Funktionalitäten, nämlich der Funktionalitäten, die in der Aufgabenstellung explizit gefordert waren, ergibt sich, dass Team C knapp den doppelten Zeitaufwand wie Team SSIML benötigte. Werden die Gesamtzeiten betrachtet, die beide Teams in die Lösung der Projektaufgabe investierten, ergibt sich die Zeitaufstellung in Abbildung 11.3(b). Danach investierte Team C im Vergleich zu Team SSIML sogar einen etwa um den Faktor 3,3 höheren Zeitaufwand in die Bearbeitung der Projektaufgabe.

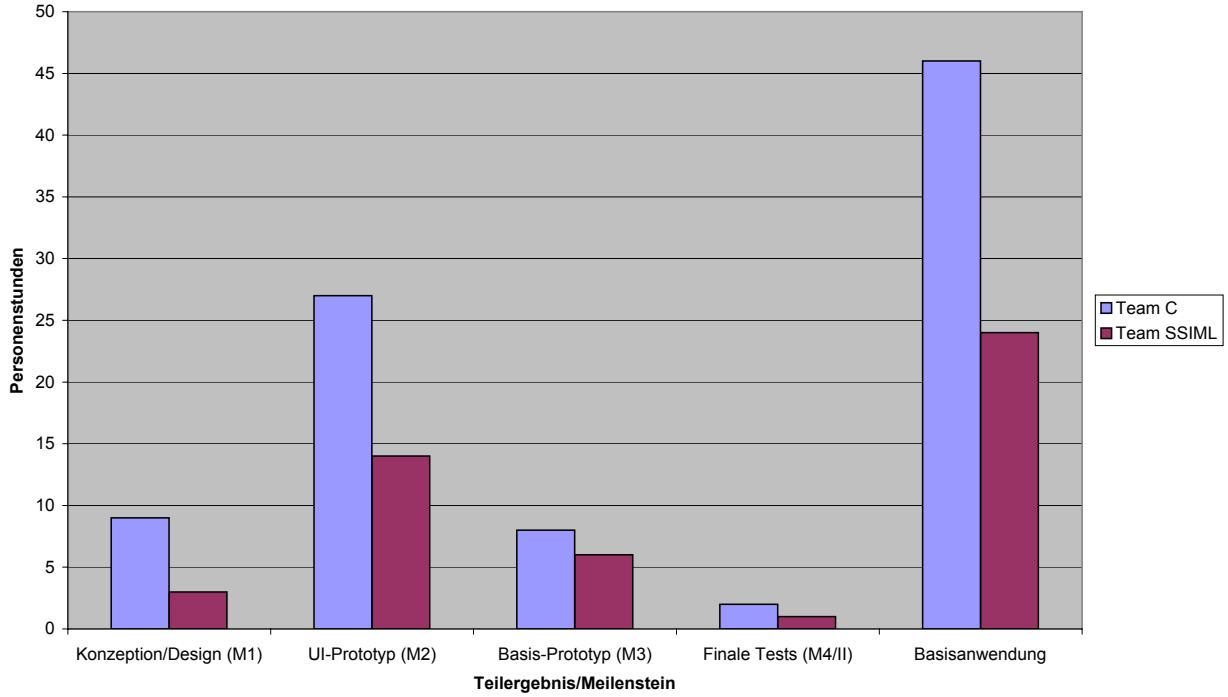
#### 11.3.2.3 Auswertung der Fragebögen

Im Folgenden sind die Ergebnisse der schriftlichen Befragung aller Teammitglieder nach Projektabschluss (Fragebögen) zusammengefasst wiedergegeben.

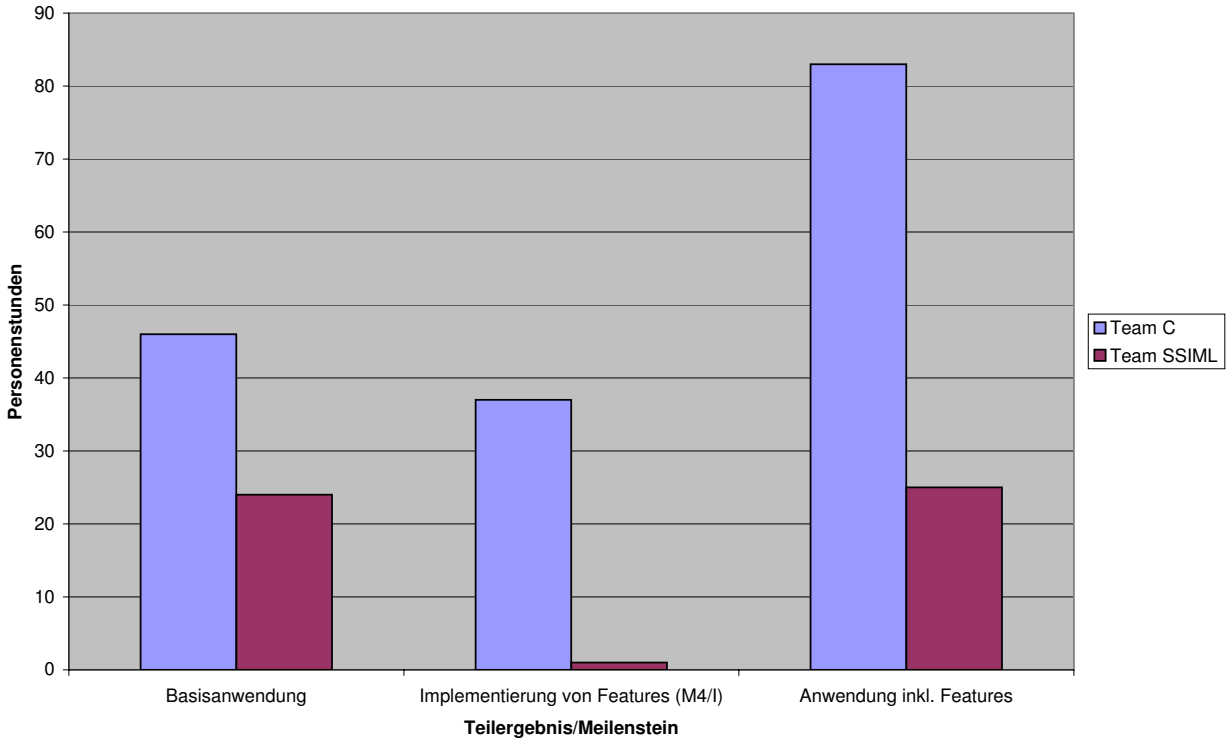
**Allgemeine Bewertung des Projekts** Alle Teilnehmer gaben an, dass sie bereits im Vorfeld des Praktikums bspw. in Vorlesungsübungen Erfahrungen mit Prinzipien des Software-Engineerings und visuellen Modellierungssprachen, insbesondere UML, sammeln konnten. Nach der Trainingsphase fühlten sich alle Projektteilnehmer in der Lage, kleine bis mittelgroße AR-Anwendungen mit C/C++ oder Java zu erstellen. Die Schwierigkeit der bearbeiteten Projektaufgabe wurde zwischen 4 und 5 auf einer Skale von 1 bis 10 (Schwierigkeitsgrad 1: Erstellung einer statischen Szene mit VRML; Schwierigkeitsgrad 10: Erstellung eines hoch-interaktiven 3D-Spiels) bewertet.

Team C fertigte in der Entwurfsphase des Projekts ein UML-Klassenmodell der zu erstellenden Anwendung an. Dies wurde auch im Nachhinein als wichtiges Hilfsmittel bewertet, um die Anwendung sinnvoll zu strukturieren. Zusätzlich gab Team C an, dass leichtere Schwierigkeiten während der Implementierung der Anwendung auftraten, die auf die verwendete Programmiersprache C++ zurückzuführen waren. Allerdings gaben die Mitglieder von Team C auch an, dass sie keine signifikanten Zeitersparnisse erwarten würden, wenn sie an Stelle von C++ Java eingesetzt hätten.

**Vorteile von SSIML/AR und der automatischen Codegenerierung** Die Mitglieder von Team SSIML sagten aus, SSIML/AR als visuelle Modellierungssprache mittlerer Komplexität empfunden zu haben. Besonders wichtig war die Bestätigung aller Mitglieder von Team SSIML, dass SSIML/AR in Verbindung mit der automatischen Codegenerierung für sie einen echten Gewinn in Hinsicht auf Zeitersparnis und Aufwandsersparnis bei der Bearbeitung der Projektaufgabe darstellte. Der Einsatz von SSIML/AR in kleineren und mittelgroßen Projekten wurde als sinnvoll befunden. Die automatische Codegenerierung wurde als wichtiges Instrument betrachtet, um vom Entwurf zur Implementierung zu gelangen. Der generierte Code wurde als wohlstrukturiert, leicht verständlich, konsistent, leicht erweiterbar und anpassbar bewertet. Bemerkenswert hierbei ist, dass Team SSIML nur minimale Änderungen am generierten Java-Code vornahm, und der Hauptaufwand bei der Implementierung in der Erweiterung und Anpassung der 3D-Szene (Einbindung der Geometrien, Anpassung der Objekttransformationen) bestand.




(a) Zeitaufwand für die Umsetzung der Basisanwendung




(b) Zeitaufwand für die Umsetzung der Gesamtanwendung mit Zusatz-Features

Abbildung 11.3: Zeitaufwandsangaben der Teams zur Erstellung der AR-Anwendungen

**Schwächen des Ansatzes** Hauptsächlich wurde am SSIML-Ansatz kritisiert, dass er bislang auf eine Vorwärtsentwicklung (*Forward-Engineering*) beschränkt ist. Es wurde unterstrichen, dass ein *Roundtrip-Engineering*, welches einen Rückfluss von Änderungen im Code in das Modell ebenso wie die Berücksichtigung von Modelländerungen im Code unterstützt, den Mehrwert von SSIML gegenüber einer konventionellen Entwicklung von 3D-Software noch einmal deutlich steigern würde. Eindeutige Aussagen, wie sich SSIML/AR für die Anwendung in anderen AR-Domänen wie Medizin oder für den Einsatz in größeren Projekten eignen würde, konnten mangels Erfahrung nicht getroffen werden. Ebenso erwartete Team SSIML keine besonderen Vorteile hinsichtlich der Wartbarkeit und Erweiterbarkeit des aus den SSIML(/AR)-Modellen generierten Codes im Vergleich zur herkömmlichen Erstellung der AR-Anwendung.

Sowohl die Mitglieder von Team C als auch die Mitglieder von Team SSIML wurden gefragt, wie sie den Zeitaufwand einschätzen würden, wenn sie ein ähnliches Projekt noch einmal realisieren müssten. Interessant ist dabei, dass Team C den Aufwand auf nur etwa 30% des aktuellen Projektes schätzte, während Team SSIML einen Zeitaufwand von etwa 45 % des aktuellen Projektes angab. Dies bedeutet, dass sich der Zeitvorteil der Benutzung von SSIML/AR gegenüber der traditionellen Entwicklung einer Anwendung zumindest verringern könnte, sobald entsprechende Erfahrungen mit früheren Projekten vorhanden sind und grundlegende Softwarebausteine zur Verfügung stehen, die in neuen Projekten wieder verwendet werden können.

#### 11.3.2.4 Ergebnisse der Team-Interviews

Gegenstand der Team-Interviews waren vor allem Fragen zum Projektverlauf und der Organisation des Projektes. Beide Teams sagten aus, dass es i. d. R. keine festgelegten Entwicklerrollen, wie Projektleiter, Programmierer und Designer, gab. Als Gründe wurden die geringe Größe des Teams und der moderate Umfang der Projektaufgabe angeführt. Meist wurde in der Gruppe gearbeitet und diskutiert. Team SSIML gab zudem an, die Technik der Paarprogrammierung (*Pair Programming*, vgl. [BA04]) angewandt zu haben. Lediglich einige spezielle Aufgaben, wie die Modellierung bestimmter 3D-Objekte mit Werkzeugen zur 3D-Modellierung, wurden innerhalb der Teams von Einzelpersonen bearbeitet. Allerdings hielten beide Teams die Einführung von Entwicklerrollen in größeren Projekten für durchaus sinnvoll. Ebenso empfanden beide Teams die die Meilensteine begleitenden Treffen als äußerst hilfreich, um Feedback zum Projektfortschritt zu erhalten.

### 11.3.3 Zusammenfassung und Diskussion

Zunächst muss noch einmal festgehalten werden, dass beide Teams die vorgegebene Projektaufgabe vollständig erfüllten. Team C entschied sich für eine aufwendigere Realisierung der Anwendung, indem bereits zu Beginn des Projekts Zusatzfunktionalitäten eingeplant wurden, die über die gegebenen Anforderungen hinausgingen. Die von Team C zusätzlich implementierten Funktionalitäten beziehen sich hauptsächlich auf die Einbindung von 3D-Animationen und die automatische Task-Weiterschaltung durch die Erkennung von

Markern. Team SSIML hingegen verfolgte das Ziel, mit möglichst wenig Aufwand die Aufgabenstellung vollständig zu erfüllen. Eine solche stark zielorientierte Arbeitweise kann sich etwa in einem industriellen Umfeld als durchaus sinnvoll erweisen, da hier vor allem ein zeitoptimiertes Arbeiten und die Erfüllung der Muss-Kriterien des Pflichtenheftes von Bedeutung sind.

Die dargestellte Situation wirft eine interessante Frage auf: Fördert ein Ansatz wie SSIML eine stärker zielorientierte Anwendungsentwicklung? Obwohl diese Frage im Rahmen der vorliegenden Arbeit nicht eindeutig bejaht werden kann, ist eine positive Antwort doch nicht unwahrscheinlich. Es kann zumindest angenommen werden, dass SSIML eine strukturierte und koordinierte Vorgehensweise bei der Softwareentwicklung unterstützt, da die Folge einzelner Schritte im Entwicklungsprozess klar vorgegeben ist. Beobachtungen des Projektverlaufs untermauern diese Annahme.

Die Lösungen beider Teams können hinsichtlich der Erfüllung der gestellten Projektaufgabe als etwa gleichwertig betrachtet werden, wobei sie jeweils Vor- und Nachteile aufweisen. Ein Hauptziel war es, die vom Benutzer zur Bearbeitung eines Tasks benötigten Informationen möglichst klar zu visualisieren und auch die Zielpositionierung zu installierender Hardware-Komponenten zu verdeutlichen. Während Team C vor allem 3D-Animationen einsetzte, um diese Anforderung zu erfüllen, bediente sich Team SSIML in erster Linie farblicher Codierungen bei der Darstellung virtueller Objekte (Abbildung 11.4). Eine weitere Teilaufgabe war die Realisierung eines Mechanismus' zur Task-Umschaltung, um nur die für den Benutzer im aktuellen Kontext relevanten Informationen visualisieren zu können. Team C implementierte eine Lösung, bei der die Umschaltung nicht explizit vom Benutzer initiiert werden muss, sondern bei der Erkennung bestimmter Marker automatisch erfolgt. Allerdings besteht hier die Gefahr, dass der Mechanismus aufgrund nicht korrekt erkannter Marker versagt. Team SSIML hingegen realisierte die Task-Umschaltung über eine Funkmaus, die dem Benutzer als Fernbedienung dient. Ein Druck auf einen Mausbutton löst den Übergang zum nächsten Task aus, während ein Druck auf den zweiten Mausbutton im Bedarfsfall zusätzlich den Rücksprung zum vorherigen Task ermöglicht. Diese Variante der Taskumschaltung, die ähnlich in einer AR-Anwendung bei Boeing [CMGJ99] umgesetzt wurde, hat zwar den Nachteil, dass ein Benutzer den Übergang zum nächsten Task explizit auslösen muss, allerdings ist sie deutlich stabiler als die von Team C umgesetzte automatische Task-Weiterschaltung und erlaubt zudem eine einfache Realisierung des angesprochenen Rücksprungs zum vorherigen Task.

Insgesamt investierte Team C zur Entwicklung der Anwendung etwa einen um den Faktor 3,3 größeren Zeitaufwand als Team SSIML. Ohne Berücksichtigung der von Team C aufgebrachten Zeit zur Realisierung der oben bereits diskutierten Zusatzfunktionalitäten war der Zeitaufwand von Team C immer noch um den Faktor 1,9 größer als derjenige von Team SSIML (vgl. Abschnitt 11.3.2.2). Dies zeigt, dass SSIML zumindest in Szenarien aus dem Bereich Montage und Wartung, in welchen sich auch das vorgegebene Projekt einordnen lässt, gewinnbringend eingesetzt werden kann. Das wurde auch durch Aussagen der Mitglieder von Team SSIML bestätigt, die SSIML/AR als sinnvolles Hilfsmittel bei der Bearbeitung der vorliegenden Projektaufgabe empfanden und gleichzeitig einen Einsatz des Ansatzes zumindest in ähnlichen kleinen bis mittelgroßen Projekten als sinnvoll


Abbildung 11.4: Screenshot der AR-Anwendung von Team SSIML: Die Steckkontakte der Grafikkarte und der entsprechende Steckplatz auf dem Mainboard sind blau markiert

erachteten. Als äußerst wichtige Punkte zur Reduzierung der Entwicklungszeit wurden die automatische Abbildung der SSIML/AR-Modelle auf fehlerfreie und untereinander konsistente Codekomponenten sowie das Vorhandensein eines Frameworks als Codebasis, das häufig benötigte Grundfunktionalitäten bereitstellt, genannt.

SSIML/AR unterstützt die schnelle Entwicklung voll funktionsfähiger 3D-Anwendungen (speziell: AR-Anwendungen) schon mit minimalen Anpassungen am generierten Programmcode (s. a. Abschnitt 11.3.2.3), wodurch eine Konzentration auf die Anpassung der 3D-Szene und die Erstellung der 3D-Inhalte möglich ist. Damit erleichtert SSIML auch denjenigen Entwicklern die Erstellung von 3D- und AR-Anwendungen, die noch keine umfangreichen Erfahrungen im Bereich der 3D- bzw. AR-Programmierung besitzen. Allerdings könnte sich mit zunehmender Erfahrung der Entwickler in ähnlichen Projekten und einer wachsenden Basis wiederverwendbarer Codebestandteile der Zeitvorteil des Einsatzes von SSIML gegenüber der traditionellen Entwicklung verringern (vgl. Abschnitt 11.3.2.3).

Als Hauptnachteil des SSIML-Ansatzes wurde allerdings angesehen, dass mit den bisher existierenden Werkzeugen nur ein Forward-Engineering möglich ist. Die Umsetzung einer Roundtrip-Engineering-Unterstützung für SSIML könnte den praktischen Nutzen des Ansatzes noch einmal deutlich steigern.

# Kapitel 12

# Zusammenfassung und Ausblick

Die vorliegende Arbeit beschäftigt sich mit Problemen bei der 3D-Anwendungsentwicklung, die vorwiegend aus einem Mangel an Konzepten und Werkzeugen zur Unterstützung der interdisziplinären Zusammenarbeit von kreativ und technisch orientieren Entwicklern resultieren. Kommunikationsschwierigkeiten zwischen Entwicklern mit unterschiedlichen Fachkenntnissen und eine rein informelle Software-Spezifikation können zu Inkonsistenzen zwischen den durch die verschiedenen Entwicklergruppen erstellten Systembestandteilen führen, die sich wiederum nachteilig auf die Entwicklungszeit eines 3D-Software-Produktes auswirken.

Zur Lösung der dargestellten Probleme wurde ein modellbasierter Ansatz gewählt. Dabei wird die Entwicklung von 3D-Anwendungen mittels verschiedener, miteinander kombinierbarer visueller Modellierungssprachen bereits in der Entwurfsphase vor der Implementierung unterstützt. Konzepte zur Abbildung der Modelle auf Code, die einen nahtlosen Übergang vom Entwurf zur Implementierung erlauben, wurden ebenfalls entwickelt. Weiterhin wurden entsprechende Werkzeuge zur Modellierung und Codegenerierung realisiert. Einige Werkzeuge weisen eine umfassende Funktionalität auf, andere wurden nur bis zu einer experimentellen Stufe entwickelt. Die bereitgestellte Werkzeugunterstützung war jedoch völlig ausreichend, um im Rahmen dieser Arbeit den Nutzen des Einsatzes visueller Modellierungssprachen bei der Entwicklung von 3D-Anwendungen auch praktisch demonstrieren zu können.

## 12.1 Kapitelzusammenfassung

Kapitel 1 führt in die eingangs in diesem Kapitel skizzierte Problematik ein. Zielstellungen der Arbeit sowie der in der Arbeit gewählte Lösungsansatz in Form der SSIML-Sprachfamilie (SSIML: *Scene Structure and Integration Modelling Language*) werden kurz vorgestellt. Zudem wird die Struktur der vorliegenden Arbeit erläutert.

Kapitel 2 gibt einen Überblick über die Charakteristiken und die Entwicklung von 3D-Anwendungen. Ebenso werden bekannte 3D-Formate vorgestellt, darunter das in der vorliegenden Arbeit als Zielformat der Codegenerierung verwendete Format X3D.

Kapitel 3 beschäftigt sich mit der modellgetriebenen und domänenspezifischen Anwendungsentwicklung. Wichtige Begriffe aus dem Bereich der modellgetriebenen Entwicklung werden definiert.

Kapitel 4 präsentiert überblicksartig die Komponenten der SSIML-Sprachfamilie: SSIML, SSIML/Tasks, SSIML/Behaviour, SSIML/Components und SSIML/AR. Diese Komponenten, welche auch den Hauptbeitrag der Arbeit darstellen, werden in den Kapiteln 5 bis 9 im Detail vorgestellt.

Kapitel 10 skizziert, wie die SSIML-Sprachen und die dazu gehörenden Werkzeuge sinnvoll in einen 3D-Entwicklungsprozess integriert werden können. Zu diesem Prozess wurden auch Entwicklerrollen definiert.

Kapitel 11 stellt eine Evaluierung des SSIML-Ansatzes vor. Vorrangig wurden dabei die Sprachen SSIML, SSIML/Tasks und SSIML/AR betrachtet. Teile der Evaluierung waren eine Expertenbefragung, zwei Einzelarbeiten und ein Teamprojekt. In den Einzelarbeiten und dem Teamprojekt wurden jeweils verschiedene Anwendungen mit SSIML bzw. SSIML/AR und SSIML/Tasks modelliert und – im Falle des Teamprojekts – anschließend implementiert. Es konnte gezeigt werden, dass SSIML ein wertvolles Werkzeug bei der Erstellung von 3D-Anwendungen sein kann.

## 12.2 Beiträge der Arbeit

In Interviews mit 3D-Entwicklern (s. Abschnitt 11.1) konnte bestätigt werden, dass ein Bedarf an Konzepten und Werkzeugen zur Lösung der eingangs in diesem Kapitel und im Detail im Einführungskapitel aufgeführten Probleme besteht.

Ein Ansatz zur Lösung der Probleme und Hauptbeitrag dieser Arbeit ist ein Satz von visuellen Modellierungssprachen, die zur Unterstützung des 3D-Entwicklungsprozesses dienen. Die Basis bildet die *Scene Structure and Integration Modelling Language* (SSIML). SSIML wurde um vier Erweiterungen ergänzt: SSIML/Tasks, SSIML/Behaviour, SSIML/-Components und SSIML/AR. Die Sprache SSIML und ihre Erweiterungen bilden die Komponenten der SSIML-Sprachfamilie.

Für alle Komponenten der Familie wurden Metamodelle definiert, die sich miteinander kombinieren lassen. Die Metamodelle wurden zum Großteil auf UML-Profiles abgebildet, um eine Integration der Teilsprachen in vorhandene UML-Werkzeuge zu ermöglichen. Bis auf die Sprache SSIML/Components, die eine sehr spezielle Notation aufweist, wurden die Sprachen so in das UML-Werkzeug MagicDraw von NoMagic integriert. Die Basissprache SSIML wurde zusätzlich in das verbreitete UML-Werkzeug Rational Rose eingebunden. Durch diese Werkzeugintegration ist es zudem möglich, SSIML-Modelle als XML-Dateien im XML Metadata Interchange - Format (XMI) zu speichern. Aus den XML-Beschreibungen kann mittels XSLT Code generiert werden. SSIML-Modelle sind aufgrund ihres vergleichsweise hohen Abstraktionsniveaus plattformunabhängig, d. h. sie können in verschiedene Zielformate und –sprachen übersetzt werden. In dieser Arbeit wurden exemplarisch X3D und Java als Zielsprachen gewählt.

Zur Modellierung der Struktur der 3D-Inhalte und der Verknüpfungen zwischen Inhalten und Anwendungskomponenten wurde die visuelle Modellierungssprache SSIML konzipiert. Neben der Möglichkeit, die Struktur von 3D-Inhalten in einer abstrakten, szenengraphorientierten Notation zu modellieren, ist als eine besondere Neuerung das Interrelationenmodell anzusehen, in dem Verknüpfungen zwischen 3D-Inhalten und Anwendungskomponenten spezifiziert werden können. Die automatische Generierung von Codegerüsten aus *einem* Interrelationenmodell sowohl für technisch orientierte Entwickler (Programmierer) als auch kreativ orientierte Entwickler (3D-Designer) ist eine Kernidee des Ansatzes; sie erlaubt es, die Konsistenz zwischen den von den einzelnen Entwicklergruppen separat und parallel zu bearbeitenden Codebestandteilen zu sichern (vgl. Kapitel 10). Die Modelle stellen damit eine Form eines Vertrages zwischen den Entwicklergruppen dar.

Der Einsatzbereich von SSIML/Tasks sind aufgabenbezogene 3D-Visualisierungen. SSIML/Behaviour erlaubt zusätzlich die Beschreibung von Verhalten und Animation von 3D-Objekten mittels UML 2-Zustandsautomaten. Eine innovative Idee von SSIML/Behaviour ist es, Animationsbeschreibungen innerhalb der Zustände darzustellen und damit dem Entwickler eine konkrete Vorstellung von einem komplexeren Animationsablauf zu verschaffen. Ebenfalls innovativ ist die Ausnutzung der Möglichkeiten von UML2-Zustandsautomaten zur Realisierung paralleler Abläufe.

SSIML/Components ermöglicht die visuelle Spezifikation von 3D-Komponenten zur Verbesserung des Managements komplexer 3D-Szenen und -Modelle. Durch integrierte Vererbungsmechanismen können neue Komponenten aus vorhandenen Komponenten abgeleitet werden. SSIML/Components geht dabei über die Möglichkeiten anderer Ansätze hinaus, indem es eine flexiblere Kombination von vorhandenen zu neuen, komplexeren Komponenten durch Mehrfachvererbung erlaubt.

SSIML/AR eignet sich für die Spezifikation von AR-Benutzungsschnittstellen (AR-UIs) und -Anwendungen. Insbesondere in Verbindung mit SSIML/Tasks ist SSIML/AR für die Beschreibung von Anwendungen aus den Bereichen Montage, Reparatur und Wartung gewinnbringend einsetzbar. Dafür wurden neue Konzepte für die Spezifizierung und Integration physikalischer und hybrider Elemente des UIs (vgl. Abschnitt 9.1.3) entwickelt.

Neben den Modellierungssprachen wurden auch ein den Ansatz begleitender Entwicklungsprozess skizziert und entsprechende Entwicklerrollen definiert. Konzepte zur Abbildung der Modelle auf Code wurden ebenfalls präsentiert. Die in Kapitel 11 vorgestellten Evaluationsergebnisse geben deutliche Hinweise darauf, dass eine Kosten- und Zeitersparnis im 3D-Entwicklungsprozess durch einen Einsatz von SSIML-Sprachkomponenten und -Werkzeugen möglich ist. Verbesserungen, die bzgl. der Realisierung einer alternativen, intuitiveren und an die Anforderungen kreativ orientierter Entwickler angepassten SSIML-Notation von Experten vorgeschlagen wurden, wurden teilweise umgesetzt (s. Abschnitt 11.2). Zudem ergab sich aus einem Einzelprojekt, in welchem SSIML eingesetzt wurde, dass die in dieser Arbeit vorwiegend benutzte grafemorientierte SSIML-Notation nach einer kurzen Lernphase ebenfalls gut benutzbar ist (vgl. Abschnitt 11.2). Damit können die in Abschnitt 1.1 formulierten Zielstellungen als erfüllt angesehen werden.

Frühere Versionen von Komponenten der SSIML-Sprachfamilie wurden bereits in Beiträgen zu internationalen Konferenzen [VP05, Vit05, Vit06b, Vit06a], einem Work-

shop [Vit06c] und zu einem Journal [VH06] vorgestellt. Die Ergebnisse der Evaluation von SSIML/AR (s. Abschnitt 11.3) wurde in einem weiteren Beitrag [Vit07] präsentiert. Der SSIML-Entwicklungsprozess (Kapitel 10) ist Teilthema von drei Artikeln [VP05, VH06, PVH07].

### 12.2.1 Wichtige Vorteile des Ansatzes

Die visuellen SSIML-Modelle können gewinnbringend eingesetzt werden

- als Kommunikationshilfe zwischen den verschiedenen Entwicklergruppen,
- zur frühzeitigen Diskussion von Alternativen bzgl. des Aufbaus der 3D-Szene (bzw. hierarchisch zusammengesetzter 3D-Objekte im Allgemeinen) und des Designs der Interaktionsbeziehungen zwischen Szene und Rahmenanwendung
- und zum verbesserten Verständnis für Zusammenhänge zwischen Komponenten der 3D-Anwendung (3D-Szene, Rahmenanwendung) durch visuelle Spezifikation.

Die Modularisierung von 3D-Inhalten, Verhaltens- und Animationsbeschreibungen durch 3D-Komponenten steigert zusätzlich die Wiederverwendbarkeit von Szenenbestandteilen. Durch eine (semi-)formale Spezifikation der 3D-Anwendung sowie einen Übergang zur Implementierung durch die Abbildung der Modelle auf Code nach definierten Regeln kann auch die Strukturierung des 3D-Entwicklungsprozesses verbessert werden. Die parallele Generierung von Code für den 3D-Entwickler und den Programmierer, also die gleichzeitige Erzeugung von Codegerüsten in einem 3D- und einem Programmcodeformat aus ein und demselben Modell trägt bei zur

- Sicherstellung der Konsistenz zwischen generierten Codebestandteilen, bspw. zwischen Benennungen von Objekten in der 3D-Szene und im Programmcode,
- Unterstützung der parallelen Entwicklung von Programm und 3D-Inhalten
- und damit zur Zeit- und Kostenersparnis bei der 3D-Anwendungsentwicklung.

Validierungswerkzeuge können bei der Prüfung von Differenzen bzgl. Struktur und Benennungen der im Modell spezifizierten und einer aus dem Modell erzeugten und nachbearbeiteten 3D-(Teil-)Szene helfen. Ein entsprechendes prototypisches Werkzeug wurde in Abschnitt 5.6.1.6 beschrieben.

Programmcode wird als so genannter Konfigurationscode (vgl. Abschnitt 3.5) für ein unterliegendes Framework oder einen Interpreter erzeugt. Damit kann die zu generierende Codemenge weitestgehend minimiert werden. In einigen Fällen kann die Programmcodeinformation sogar vollständig aus einem Modell extrahiert werden, etwa bei AR-Anwendungen aus dem Bereich Montage und Wartung. Indem die Erzeugung des Codes zur Abdeckung immer wiederkehrender Aspekte von domänenspezifischen Anwendungen automatisiert wird, wird dem Entwickler Gelegenheit gegeben, sich auf die Umsetzung der Funktionalitäten zu konzentrieren, deren Realisierung ein höheres Maß an Kreativität erfordert und die sich speziell auf das aktuelle Projekt beziehen.

## 12.3 Offene Fragestellungen und zukünftige Arbeiten

**Metamodell** Im Rahmen der Arbeit sind einige Fragen offengeblieben, die an dieser Stelle noch einmal diskutiert werden sollen. So ist es in SSIML bisher nur möglich, zu spezifizieren, *welche* Aktionen auf Objekten der Szene ausgeführt werden. In einigen Fällen kann aber auch *die Abfolge* dieser Aktionen eine Rolle spielen. So kann z. B. auf ein Objekt, welches bereits aus der Szene entfernt wurde, zu einem späteren Zeitpunkt nicht mehr zugegriffen werden. Eine Möglichkeit zur Spezifikation erlaubter Zugriffsfolgen wäre daher von Vorteil.

In Kapitel 7 wird das Verhalten von 3D-Objekten durch externe Anwendungskomponenten gesteuert, die entsprechende Nachrichten generieren. Solche Nachrichten können ihre Ursachen in Objekt-Objekt-Interaktionen, wie Kollisionen zwischen Objekten, haben. In aufgabenzentrierten AR-Anwendungen (vgl. Kapitel 9) könnte die Überwachung von Beziehungen zwischen realen und/oder virtuellen Objekten (Nähe der Objekte, Kollision, Lage der Objekte zueinander) dazu dienen, den Fortschritt eines Benutzers bei der Abarbeitung einer Folge von zusammenhängenden Teilaufgaben zu kontrollieren und den Taskflow entsprechend zu steuern. Konzepte für die graphische Spezifikation von Objekt-Objekt-Interaktionen in 3D-Szenen und deren Einbindung in die vorhandenen Metamodelle wären daher ein interessanter Gegenstand für weiterführende Untersuchungen.

Die Animationszustände in Kapitel 7 können nur teilweise benutzt werden, um Parameter von SSIML-3D-Komponenten zu verändern. Somit ist zu diesem Zeitpunkt nur eine Verwendung von `ProceduralAnimationState`-Elementen zusammen mit SSIML-Komponenten möglich. Hier wäre eine Erweiterung von SSIML/Behaviour sinnvoll, um die Kombination aller Animationszustandstypen in Verbindung mit SSIML-Komponenten zu erlauben.

**Notation** Als Reaktion auf die in Abschnitt 11.1 beschriebene Expertenbefragung wurde eine zusätzliche, bildorientierte Notation als Alternative zu der in dieser Arbeit verwendeten grafemorientierten Notation mit dem Ziel erarbeitet, eine intuitiv verständlichere Notation für die SSIML-Elemente bereitzustellen. Nach mehrheitlicher Meinung der im Rahmen der Arbeit befragten 3D-Experten muss sich die Gestaltung der Bilder (*Icons*) für Modell-Elemente nach – z. B. aus 3D-Autorenwerkzeugen – bekannten Metaphern richten und gewissen ästhetischen Ansprüchen genügen, um eine breite Akzeptanz auch bei kreativ orientierten Entwicklern wie 3D-Designern zu erreichen. Die nach diesen Grundsätzen in der Arbeit von Jahn [Jah07] entwickelte Notation deckt derzeit allerdings nur SSIML-Grundelemente ab. Eine Erweiterung um Icons für alle Modellelemente aller SSIML-Sprachkomponenten und eine umfangreichere Bewertung der Notation durch potenzielle Benutzer wäre daher sinnvoll.

**Werkzeugunterstützung** Eine stabile und umfassende Werkzeugunterstützung ist für den erfolgreichen Einsatz eines Entwicklungsansatzes wie SSIML fundamental wichtig (vgl. Abschnitt 11.1). Die im Rahmen der vorliegenden Arbeit verwendeten UML-Werkzeuge

stellten in Verbindung mit integrierten SSIML-Profiles ein Hilfsmittel dar, um zeiteffizient einen Beleg der Benutzbarkeit der vorgestellten Konzepte (*Proof-of-Concept*) erbringen zu können. Diese Vorgehensweise ist aber auch mit Nachteilen verbunden: Einerseits bestehen Einschränkungen bei der Prüfung der in den SSIML-Metamodellen definierten Zusicherungen (*Constraints*); es ist also durchaus möglich, mit einem UML-Werkzeug, in welches das SSIML-UML-Profile integriert wurde, ein Modell zu erstellen, welches nicht konform zum SSIML-Metamodell ist. Andererseits werden durch UML-Werkzeuge Grenzen bei der Einbindung spezieller konkreter Syntaxen gesetzt, wie etwa bei der Notation von Komponenten in SSIML/Components. Eine Verbesserung dieser Situation stellen DSL-Werkzeuge wie das Eclipse GMF [GMF] oder die Microsoft DSL Tools [CJKW07] dar, welche sich in letzter Zeit stark weiterentwickelt haben und die Generierung von (grafischen) Modelleditoren auf Basis eines Metamodells erlauben. Mittels der MS DSL Tools wurde daher ein auf SSIML spezialisierter Editor entwickelt, der allerdings nur die Elemente des SSIML-`core`-Paketes unterstützt. Eine Erweiterung dieses Editors um weitere SSIML-Sprachkomponenten ist daher erstrebenswert. Weiterhin erstrebenswert ist die Einbindung der in Abschnitt 11.2 angesprochenen Konzepte zum verbesserten Management komplexerer Szenen durch die Anwendung von Layout-Algorithmen und Möglichkeiten des Ein- und Ausblendens von Diagrammbestandteilen in einem spezialisierten SSIML-Editor.

Die automatische Codegenerierung aus SSIML-Modellen ist unabdingbar, um einen – im Hinblick die Entwicklungszeit und -kosten – gewinnbringenden Einsatz der SSIML-Sprachen zu gewährleisten. Es besteht Verbesserungspotenzial bei der Implementierung der Codegenerierung, die bislang nur eine Teilmenge der Elemente des SSIML-Metamodells abdeckt, darunter allerdings alle spezifischen Modellelementtypen von SSIML/AR und SSIML/Tasks. Außerdem wurden in der Arbeit hauptsächlich X3D und Java als Zielformate betrachtet. Da SSIML-Modelle plattformunabhängig sind, wäre die Entwicklung von Codegenerierungsroutinen für andere potenzielle Zielformate sinnvoll.

Als weiterer wichtiger Punkt zur nochmaligen deutlichen Steigerung des praktischen Nutzens von SSIML wurde die Unterstützung eines *Round-Trip-Engineerings* angeführt (vgl. Abschnitte 11.1, 11.3), um Änderungen im generierten Code in die Modelle zurückfließen lassen zu können und umgekehrt. Die existente Werkzeugunterstützung für SSIML erlaubt kein Round-Trip-Engineering; in Abschnitt 10.1 werden jedoch Ansätze zur Realisierung eines Round-Trip-Engineerings für 3D-Szenen skizziert. Um das Round-Trip-Engineering zu realisieren, wäre eine Verfeinerung der in Abschnitt 10.1 vorgestellten Konzepte und die Bereitstellung entsprechender Software-Werkzeuge notwendig.

**Evaluierung** Die SSIML-Komponenten SSIML/Components und SSIML/Behaviour wurden bei der in Abschnitt 11.1 vorgestellten Expertenbefragung zwar am Rande berücksichtigt, waren aber nicht deren Hauptgegenstand. Auch in den in Kapitel 11 vorgestellten Projekten, in denen SSIML-Sprachen praktisch verwendet wurden, wurden diese SSIML-Teilsprachen weniger stark eingesetzt. Aus diesem Grunde ist eine weitere Evaluierung sinnvoll, die sich auf die Sprachen SSIML/Behaviour und SSIML/Components konzentriert.

Kapitel 11 und vor allem das in Abschnitt 11.3 beschriebene AR-Projekt zeigen, dass der Einsatz von SSIML-Sprachen (und damit modellgetriebener Entwicklung) in kleinen bis mittelgroßen 3D-Entwicklungsprojekten durchaus profitabel sein kann. Allerdings konnte im Rahmen dieser Arbeit aus Zeitgründen keine klare Aussage darüber getroffen werden, bis zu welchem Maß eine Skalierung des Ansatzes möglich ist. Erst die Benutzung der SSIML-Sprachen in Verbindung mit einer entsprechenden Werkzeugunterstützung in einer Reihe sehr umfangreicher Projekte kann darüber Aufschluss geben.

# Literaturverzeichnis


[3B] *3B-Browser*. http://3b.net/browser/newhome.html, Abruf: 11.03.2008

[3ds] *Autodesk 3ds max*. http://www.autodesk.de/3dsmax, Abruf: 13.03.2008

[ABB+01] Azuma, Ronald; Baillot, Yohan; Behringer, Reinhold; Feiner, Steven; Julier, Simon; MacIntyre, Blair: Recent Advances in Augmented Reality. In: *IEEE Comput. Graph. Appl.* 21 (2001), Nr. 6, S. 34–47

[AD69] Annett, J.; Duncan, K.: Task Analysis and Training Design. In: *Occupational Psychology* 41 (1969), S. 211–221

[ADSG71] Annett, J.; Duncan, K.; Stammers, R.; Gray, M.: *Task analysis*. London : HMSO, 1971

[Ali] *Alice - 3D-Autorenwerkzeug*. http://www.alice.org, Abruf: 11.03.2008

[Alz07] Alzetta, Alexander: *Informale Evaluierung eines modell-getriebenen Ansatzes zur Entwicklung interaktiver 3D-Anwendungen*, Ludwig-Maximilans-Universität München, Diplomarbeit, 2007

[Arc] *GRAPHISOFT ARCHICAD*. http://www.graphisoft.de/produkte/gestalten-Graphisoft/archicad/, Abruf: 11.03.2008

[ART] *ARToolkit*. http://www.hitl.washington.edu/artoolkit/, Abruf: 29.02.2008

[Aut] *Autodesk Inventor*. http://www.autodesk.de/inventor, Abruf: 11.03.2008

[Azu97] Azuma, Ronald T.: A Survey of Augmented Reality. In: *Presence: Teleoperators and Virtual Environments* 6 (1997), August, Nr. 4, S. 355–385

[BA04] Beck, Kent; Andres, Cynthia: *eXtreme Programming Explained*. 2. Auflage. Addison-Wesley, 2004 (The XP series)

[Bal] *BALLView*. http://www.ballview.org/, Abruf: 12.03.2008

[BBK+01] Bauer, Martin; Bruegge, Bernd; Klinker, Gudrun; MacWilliams, Asa; Reicher, Thomas; Riss, Stefan; Sandor, Christian; Wagner, Martin: Design of a Component-Based Augmented Reality Framework. In: *Proceedings of the International Symposium on Augmented Reality (ISAR)*, 2001

[BE05] Burrows, Tony; England, David: YABLE – yet another behaviour language. In: *Web3D '05: Proceedings of the tenth international conference on 3D Web technology*. New York, NY, USA : ACM, 2005, S. 65–73

[BFO92] Bajura, Mike; Fuchs, Henry; Ohbuchi, Ryutarou: Merging Virtual Reality with the Real World: Seeing Ultrasound Imagery within the Patient. Chapel Hill, NC, USA : University of North Carolina at Chapel Hill, 1992. – Forschungsbericht

[BKLP04] Bowman, Doug A.; Kruijff, Ernst; LaViola, Joseph J.; Poupyrev, Ivan: *3D User Interfaces. Theory and Practice*. Addison-Wesley Professional, 2004

[bla] *blaxxun Plattform*. `http://www.blaxxun.com`, Abruf: 12.03.2008

[Ble] *Blender*. `http://www.blender.org`, Abruf: 11.03.2208

[Bli98] Blinn, Jimm F.: Ten more unsolved problems in computer graphics. In: *IEEE Computer Graphics and Applications* 18 (1998), Nr. 5, S. 86–89

[BLO+04] Broll, Wolfgang; Lindt, Irma; Ohlenburg, Jan; Wittkämper, Michael; Yuan, Chunrong; Novotny, Thomas; Schieck, Ava F.; Mottram, Chiron; Strothmann, Andreas: Arthur: A Collaborative Augmented Environment for Architectural Design and Urban Planning. In: *Journal of Virtual Reality and Broadcasting* 1 (2004), Nr. 1, S. 1–10

[Bre04] Breuer, Holger: *Model Driven Architecture. Ein Ansatz mit dem OpenSource Generator AndroMDA*. GRIN Verlag, 2004

[Bul] *Bullet Physics Engine*. `http://www.bulletphysics.com`, Abruf: 12.03.2008

[C2X] *C2X Collada-zu-X3D Übersetzer (Pinecoast)*. `http://www.pinecoast.com/c2x.htm`, Abruf: 13.03.2003

[CAB+00] Conway, Matthew; Audia, Steve; Burnette, Tommy; Cosgrove, Dennis; Christiansen, Kevin: Alice: lessons learned from building a 3D system for novices. In: *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM Press, 2000, 486–493

[CCT+06] Cheung, D.; Cheung, D.; Tigli, J.-Y.; Lavirotte, S.; Riveill, M.: Wcomp: a Multi-Design Approach for Prototyping Applications using Heterogeneous Resources. In: Tigli, J.-Y. (Hrsg.): *Proc. Seventeenth IEEE International Workshop on Rapid System Prototyping*, 2006, S. 119–125

[CH03] Czarnecki, Krzysztof; Helsen, Simon: Classification of Model Transformation Approaches. In: *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003

[CJKW07] Cook, Steve; Jones, Gareth; Kent, Stuart; Wills, Alan C.: *Domain Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007 (Microsoft .net Development)

[CMBZ00] Capps, Michael; McGregor, Don; Brutzman, Don; Zyda, Michael: NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments. In: *IEEE Comput. Graph. Appl.* 20 (2000), Nr. 5, S. 12–15

[CMGJ99] Curtis, Dan; Mizell, David; Gruenbaum, Peter; Janin, Adam: Several devils in the details: making an AR application work in the airplane factory. In: *IWAR '98: Proceedings of the international workshop on Augmented reality: placing artificial objects in real scenes*. Natick, MA, USA : A. K. Peters, Ltd., 1999, S. 47–60

[Col] *ColoTux - Virtuelle Koloskopie*. `http://www.colotux.de/`, Abruf: 11.03.2008

[Con97] Conway, Matthew J.: *Alice: Easy-to-Learn 3D Scripting for Novices*, Carnegie Mellon University, Diss., 1997

[CPGB94] Conway, Matthew; Pausch, Randy; Gossweiler, Rich; Burnette, Tommy: Alice: a rapid prototyping system for building virtual environments. In: Plaisant, Catherine (Hrsg.): *CHI Conference Companion*, ACM, 1994, S. 295–296

[CSH+92] Conner, Brookshire D.; Snibbe, Scott S.; Herndon, Kenneth P.; Robbins, Daniel C.; Zeleznik, Robert C.; Dam, Andries van: Three-dimensional widgets. In: *SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics*. New York, NY, USA : ACM Press, 1992, S. 183–188

[Däß02] Dässler, Rolf: *Das Einsteigerseminar VRML*. vmi Buch, 2002

[Dac04] Dachselt, Raimund: *Eine deklarative Komponentenarchitektur und Interaktionsbausteine für dreidimensionale multimediale Anwendungen*. Der Andere Verlag, 2004

[DG00] Dörner, Ralf; Grimm, Paul: 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)*. New York, NY, USA : ACM Press, 2000, S. 69–74

[DGN03] Dubois, Emmanuel; Gray, Philip; Nigay, Laurence: ASUR++: Supporting the design of mobile mixed systems. In: *Interacting with Computers* Ausgabe 15 (2003), Nr. 4, S. 497–520

[DHM02] Dachselt, Raimund; Hinz, Michael; Meissner, Klaus: Contigra: an XML-based architecture for component-oriented 3D applications. In: *Web3D '02: Proceeding of the seventh international conference on 3D Web technology*. New York, NY, USA : ACM Press, 2002, S. 155–163

[Dir] *Adobe Director*. `http://www.adobe.com/products/director/`, Abruf: 29.02.2008

[DIV] *DIVE VR-Plattform*. `http://www.sics.se/dive/`, Abruf: 12.03.2008

[DOO] *DOOM 3 (id Software)*. `http://www.doom3.com/`, Abruf: 11.03.2008

[Dot] *Microsoft .Net Framework*. `http://www.microsoft.com/net/`, Abruf: 14.03.2008

[DR03] Dachselt, Raimund; Rukzio, Enrico: Behavior3D: an XML-based framework for 3D graphics behavior. In: *Web3D '03: Proceedings of the eighth international conference on 3D Web technology*. New York, NY, USA : ACM, 2003, S. 101–ff

[Dra93] Drascic, David: Stereoscopic Vision and Augmented Reality. In: *Scientific Computing & Automation* 9 (1993), Nr. 7, S. 31–34

[DSG02] Dubois, Emmanuel; Silva, Paulo P.; Gray, Philip D.: Notational Support for the Design of Augmented Reality Systems. In: *DSV-IS*, 2002, S. 74–88

[ECM07] ECMA International: *ECMA-363: Universal 3D File Format, 4. Edition*, Juni 2007

[Ell99] Elliott, Conal: An Embedded Modeling Language Approach to Interactive 3D and Multimedia Animation. In: *Software Engineering* 25 (1999), Nr. 3, S. 291–308

[EMF] *Eclipse Modeling Framework (EMF)*. `http://www.eclipse.org/modeling/emf/`, Abruf: 14.03.2008

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

[Fir] *Fire Safety (Incredible Sims)*. `http://incrediblesims.com/News/fire-safety`, Abruf: 12.03.2008

[FLPS03] Fishwick, Paul; Lee, Jinho; Park, Minho; Shim, Hyunju: Next generation modeling I: RUBE: a customized 2d and 3d modeling framework for simulation. In: *WSC '03: Proceedings of the 35th conference on Winter simulation*, Winter Simulation Conference, 2003, S. 755–762

[FMHW97] Feiner, Steven; MacIntyre, Blair; Hollerer, Tobias; Webster, Anthony: A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment. In: *ISWC '97: Proceedings of the 1st IEEE International Symposium on Wearable Computers*. Washington, DC, USA : IEEE Computer Society, 1997, S. 74

[FN05] Favre, Jean-Marie; NGuyen, Tam: Towards a Megamodel to Model Software Evolution Through Transformations. In: *Elsevier Electronic Notes in Theoretical Computer Science* 127 (2005), Nr. 3, S. 59–74

[For] *Ford 3D-Fahrzeugkonfigurator*. `http://www.ford.de`, Abruf: 11.03.2008

[Fri04] Friedrich, Wolfgang (Hrsg.): *ARVIKA. Augmented Reality für Entwicklung, Produktion und Service*. Publicis Corporate Publishing, 2004

[FV02] Fjeld, M.; Voegtli, B.M.: Augmented Chemistry: an interactive educational workbench. In: *Proc. International Symposium on Mixed and Augmented Reality ISMAR 2002*, 2002, S. 259–321

[FvFH95] Foley, James D.; van Dam, Andries; Feiner, Steven K.; Hughes, John F.: *Computer Graphics: Principles and Practice*. 2. Auflage. Addison-Wesley Professional, 1995 (Systems Programming Series)

[GDB07] Gauffre, Guillaume; Dubois, Emmanuel; Bastide, Rémi: Domain Specific Methods and Tools for the Design of Advanced Interactive Techniques. In: Pleuss, Andreas (Hrsg.); Bergh, Jan Van d. (Hrsg.); Sauer, Stefan (Hrsg.); Görlich, Daniel (Hrsg.): *Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI)*, CEUR Workshop Proceedings, Nov. 2007

[GHJV94] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994 (Professional Computing Series)

[GHP+02] Grimm, P.; Haller, M.; Paelke, V.; Reinhold, S.; Reimann, C.; Zauner, R.: AMIRE - authoring mixed reality. In: *Proc. First IEEE International Workshop Augmented Reality Toolkit*, 2002

[GMF] *Graphical Modeling Framework (GMF)*. `www.eclipse.org/gmf/`, Abruf: 14.03.2008

[GMP+02] García, Pedro; Montalà;, Oriol; Pairot, Carles; Rallo, Robert; Skarmeta, Antonio G.: MOVE: component groupware foundations for collaborative virtual environments. In: *CVE '02: Proceedings of the 4th international conference on Collaborative virtual environments.* New York, NY, USA : ACM Press, 2002, S. 55–62

[Goo] *Google Earth.* `http://earth.google.com`, Abruf: 12.03.2008

[GPRR00] Geiger, C.; Paelke, V.; Reimann, C.; Rosenbach, W.: A framework for the structured design of VR/AR content. In: *VRST '00: Proceedings of the ACM symposium on Virtual reality software and technology.* New York, NY, USA : ACM Press, 2000, S. 75–82

[GRSP02] Geiger, C.; Reimann, C.; Stöcklein, J.; Paelke, V.: JARToolKit - A Java binding for ARToolKit. In: *Proc. First IEEE International Workshop Augmented Reality Toolkit*, 2002

[GS03] Greenfield, Jack; Short, Keith: Software factories: assembling applications with patterns, models, frameworks and tools. In: *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* New York, NY, USA : ACM, 2003, S. 16–27

[GSCK04] Greenfield, Jack; Short, Keith; Cook, Steve; Kent, Stuart: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004

[Har87] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), Juni, Nr. 3, S. 231–274

[HCK+99] Hubbold, Roger; Cook, Jon; Keates, Martin; Gibson, Simon; Howard, Toby; Murta, Alan; West, Adrian; Pettifer, Steve: GNU/MAVERIK: a micro-kernel for large-scale virtual environments. In: *VRST '99: Proceedings of the ACM Symposium on Virtual Reality Software and Technology.* New York, NY, USA : ACM, 1999, S. 66–73

[HLN+88] Harel, D.; Lachover, H.; Naamad, A.; Pnueli, A.; Politi, M.; Sherman, R.; Shtul-Trauring, A.: Statemate: a working environment for the development of complex reactive systems. In: *ICSE '88: Proceedings of the 10th international conference on Software engineering.* Los Alamitos, CA, USA : IEEE Computer Society Press, 1988, S. 396–406

[HMB03] Hendricks, Zayd; Marsden, Gary; Blake, Edwin: A meta-authoring tool for specifying interactions in virtual reality environments. In: *AFRIGRAPH '03: Proceedings of the 2nd International Conference on Computer Graphics,*

*Virtual Reality, Visualisation and Interaction in Africa*. New York, NY, USA : ACM, 2003, S. 171–180

[IC05] Ieronutti, Lucio; Chittaro, Luca: A virtual human architecture that integrates kinematic, physical and behavioral aspects to control H-Anim characters. In: *Web3D '05: Proceedings of the tenth international conference on 3D web technology*. New York, NY, USA : ACM, 2005, S. 75–83

[Int97] International Organization for Standardization: *ISO/IEC 14772-1:1997: Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML) – Part 1: Functional specification and UTF-8 encoding*, 1997

[Int98] International Organization for Standardization: *ISO 8807:1989: Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, 1998

[Int02] International Organization for Standardization: *ISO/IEC 16262:2002: Information technology – ECMAScript language specification*, 2002

[Int04a] International Organization for Standardization: *ISO/IEC 14772-2:2004: Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML) – Part 2: External authoring interface (EAI)*, 2004

[Int04b] International Organization for Standardization: *ISO/IEC 19775:2004: Information technology – Computer graphics and image processing – Extensible 3D (X3D)*, 2004

[Int05a] International Organization for Standardization: *ISO/IEC 19774:2005: Information technology – Computer graphics and image processing – Humanoid animation (H-Anim)*, 2005

[Int05b] International Organization for Standardization: *ISO/IEC 19776:2005: Information technology – Computer graphics and image processing – Extensible 3D (X3D) encodings*, 2005

[Int05c] International Organization for Standardization: *ISO/IEC 19777-1:2005: Information technology – Computer graphics and image processing – Extensible 3D (X3D) language bindings – Part 1: ECMAScript*, 2005

[Int05d] International Organization for Standardization: *ISO/IEC 19777-2:2005: Information technology – Computer graphics and image processing – Extensible 3D (X3D) language bindings – Part 2: Java*, 2005

[Int08a] International Organization for Standardization: *ISO/IEC CD 19775-2 Ed. 2:2008: Information technology – Computer graphics and image processing – Extensible 3D (X3D) – Part 2: Scene access interface (SAI)*, 2008

[Int08b] International Organization for Standardization: *ISO/IEC FDIS 19775-1:2008: Information technology – Computer graphics and image processing – Extensible 3D (X3D) – Part 1: Architecture and base components, 2. Edition*, 2008

[ISA] *Internet Scene Assembler (Parallelgraphics).* `http://www.parallelgraphics.com/products/isa`, Abruf: 29.02.2008

[J3D] *Java 3D.* `https://java3d.dev.java.net/`, Abruf: 29.02.2008

[JAB+06] Jouault, Frédéric; Allilaire, Freddy; Bézivin, Jean; Kurtev, Ivan; Valduriez, Patrick: ATL: a QVT-like transformation language. In: *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.* New York, NY, USA : ACM, 2006, S. 719–720

[Jac05] Jacobs, Scott: Visual Design of State Machines. In: Pallister, Kim (Hrsg.): *Game Programming Gems 5*, Charles River Media, 2005 (Game Programming Gems Series), S. 169–176

[Jah07] Jahn, Margit: *Qualitative Evaluierung einer graphischen Notation zur Beschreibung von 3D-Anwendungen.* Projektarbeit an der Ludwig-Maximilans-Universität München, 2007

[Jav] *Java Beans (Sun).* `http://java.sun.com/beans`, Abruf: 29.02.2008

[JBK06] Jouault, Frédéric; Bézivin, Jean; Kurtev, Ivan: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering.* New York, NY, USA : ACM, 2006, S. 249–254

[JBL+00] Julier, S.; Baillot, Y.; Lanzagorta, M.; Brown, D.; Rosenblum, L.: BARS: Battlefield Augmented Reality System. In: *NATO Symposium on Information Processing Techniques for Military Systems*, 2000

[JMF] *Java Media Framework (Sun).* `http://java.sun.com/products/java-media/jmf/`, Abruf: 19.03.2008

[KBJV06] Kurtev, Ivan; Bézivin, Jean; Jouault, Frédéric; Valduriez, Patrick: Model-based DSL frameworks. In: *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.* New York, NY, USA : ACM, 2006, S. 602–616

[Köh00] Köhler, Hans-Josef: *Codegenerierung für UML-Collaborations-, -Sequenz- und -Statechart-Diagramme*, Universität-Gesamthochschule Paderborn, Diplomarbeit, 2000

[Khr06] Khronos Group: *Collada - Digital Asset Schema Release 1.4.1 Specification*, 2006

[KWB03] Kleppe, Anneke G.; Warmer, Jos; Bast, Wim: *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003

[Led04] Ledermann, Florian: *An Authoring Framework for Augmented Reality Presentations*, Vienna Technical University, Diplomarbeit, 2004

[Loo] *Looking Glass - 3D Desktop (Sun)*. `https://lg3d-core.dev.java.net/`, Abruf: 12.03.2008

[LS05] Ledermann, F.; Schmalstieg, D.: APRIL: a high-level framework for creating augmented reality presentations. In: *Proc. IEEE Virtual Reality VR 2005*, 2005, S. 187–194

[LS06] Ludwig, Friedemann; Salger, Frank: Werkzeuge zur domänenspezifischen Modellierung. In: *Objektspektrum* 3 (2006), S. 16–20

[LW94] Liskov, Barbara H.; Wing, Jeannette M.: A behavioral notion of subtyping. In: *ACM Trans. Program. Lang. Syst.* 16 (1994), Nr. 6, S. 1811–1841

[Mag] *MagicDraw UML (NoMagic)*. `http://www.magicdraw.com/`, Abruf: 29.02.2008

[Maj05] Majorov, Alexander: Universal 3D (U3D). In: *Proceedings of Graphicon 2005*, 2005

[MAR] *Mobile Augmented Reality Quest (MARQ)*. `http://studierstube.icg.tu-graz.ac.at/handheld_ar/marq.ph`, Abruf: 19.03.2008

[MAT] *MATADOR (Medical Advanced Training in an Artificial Distributed Environment) - Projektseite*. `http://www.telenor.no/fou/prosjekter/matador/project.html`, Abruf: 12.03.2008

[May] *Autodesk Maya*. `http://www.autodesk.de/maya`, Abruf: 13.03.2008

[MB02] Mellor, Stephen J.; Balcer, Marc J.; Booch, Grady (Hrsg.); Jacobson, Ivar (Hrsg.); Rumbaugh, Jim (Hrsg.): *Executable UML. A Foundation for Model Driven Architecture*. Addison-Wesley, 2002 (Object Technology Series)

[MC99] Milgram, P.; Colquhoun, H.: A Taxonomy of Real and Virtual World Display Integration. In: Ohta, Yuichi (Hrsg.); Tamura, Hideyuki (Hrsg.): *Mixed Reality - Merging Real and Virtual Worlds*. Berlin : Springer Verlag, 1999, S. 1–16

[Med] *Media Machines Studio*. `http://www.mediamachines.com/developer.php`, Abruf: 12.03.2008

[MFCGD99] Multon, Franck; France, Laure; Cani-Gascuel, Marie-Paule; Debunne, Giles: Computer animation of human walking: a survey. In: *The Journal of Visualization and Computer Animation* 10 (1999), Nr. 1, S. 39–54

[MGB+03] MacIntyre, B.; Gandy, M.; Bolter, J.; Dow, S.; Hannigan, B.: DART: the Designer's Augmented Reality Toolkit. In: *Proc. Second IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2003, S. 329–330

[MGDB04] MacIntyre, Blair; Gandy, Maribeth; Dow, Steven; Bolter, Jay D.: 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*. New York, NY, USA : ACM, 2004, S. 197–206

[MGDB05] MacIntyre, Blair; Gandy, Maribeth; Dow, Steven; Bolter, Jay D.: DART: a toolkit for rapid design exploration of augmented reality experiences. In: *ACM Trans. Graph.* 24 (2005), Nr. 3, S. 932–932

[MMSC97] Marrin, Chris; McCloskey, Bill; Sandvik, Kent; Chin, Don: Creating Interactive Java Applications with 3D and VRML. In: *White Paper, Silicon Graphics, Inc.* (1997)

[Mon] *Escape from Monkey Island (Lucas Arts)*. `http://www.lucasarts.com/`, Abruf: 11.03.2008

[MPS02] Mori, G.; Paterno, F.; Santoro, C.: CTTE: support for developing and analyzing task models for interactive system design. In: *IEEE Transactions on Software Engineering* 28 (2002), Aug., Nr. 8, S. 797–813

[MSF] *Microsoft Flight Simulator 2004: Das Jahrhundert der Luftfahrt*. `http://www.microsoft.com/germany/games/pc/fs2004.mspx`, Abruf: 11.03.2008

[MZDG93] Milgram, P.; Zhai, S.; Drascic, D.; Grodski, J.: Applications of augmented reality for human-robot communication. In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems '93 (IROS '93)* Bd. 3, 1993, S. 1467–1472

[Nee] *Need for Speed ProStreet (Electronic Arts)*. `http://www.ea.com/nfs/`, Abruf: 11.03.2008

[NPF+03] Narzt, W.; Pomberger, G.; Ferscha, A.; Kolb, D.; Müller, R.; Wieghardt, J.; Hörtner, H.; Lindinger, C.: Pervasive information acquisition for mobile AR-navigation systems. In: *Proc. Fifth IEEE Workshop on Mobile Computing Systems and Applications*, 2003, S. 13–20

[NPF+06] Narzt, Wolfgang; Pomberger, Gustav; Ferscha, Alois; Kolb, Dieter; Müller, Reiner; Wieghardt, Jan; Hörtner, Horst; Lindinger, Christopher: Augmented reality navigation systems. Berlin, Heidelberg : Springer-Verlag, 2006, S. 177–187

[NT05] Niaz, Iftikhar A.; Tanaka, Jiro: An Object-Oriented Approach To Generate Java Code From UML Statecharts. In: *International Journal of Computer & Information Science* 6 (2005), Nr. 2

[NY99] Neumann, Ulrich; You, Suya: Natural Feature Tracking for Augmented Reality. In: *IEEE Transactions on Multimedia* 1 (1999), Nr. 1, S. 53–64

[Obj03] Object Management Group: *MDA Guide Version 1.0.1*, 2003

[Obj06a] Object Management Group: *Meta Object Facility (MOF) Core Specification, OMG Available Specification, Version 2.0*, 2006

[Obj06b] Object Management Group: *Object Constraint Language, OMG Available Specification, Version 2.0*, 2006

[Obj07a] Object Management Group: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification*, 2007

[Obj07b] Object Management Group: *MOF 2.0/XMI Mapping, Version 2.1.1*, 2007

[Obj07c] Object Management Group: *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.1.2*, 2007

[Obj08] Object Management Group: *Common Object Request Broker Architecture (CORBA) Specification, OMG Available Specification, Version 3.1*, 2008

[OV3] *Operational Views in 3D - Simulator (3Dsolve)*. `http://www.3dsolve.com/ov3d.html`, Abruf: 12.03.2008

[PAB+97] Pierce, Jeffrey S.; Audia, Steve; Burnette, Tommy; Christiansen, Kevin; Cosgrove, Dennis; Conway, Matthew; Hinckley, Ken; Monkaitis, Kristen; Patten, James; Shochet, Joe; Staack, David; Stearns, Brian C.; Sturgill, Christopher B.; Williams, George H.; Pausch, Randy F.: Alice: Easy to Use Interactive 3D Graphics. In: *ACM Symposium on User Interface Software and Technology*, 1997, S. 77–78

[Phy] *Ideal Gas in 3D (Physiksimulation).* `http://www.physics-software.com/`, Abruf: 11.03.2008

[Pie01] Pierce, Jeffrey S.: *Expanding the Interaction Lexicon for 3D Graphics*, School of Computer Science, Carnegie Mellon University, Diss., 2001

[Pol07] Pollner, Michael: *Ein Vergleich von interaktiven virtuellen Umgebungen, offline- gerenderten Animationen und textbasierten Dokumenten bei der Erstellung und Nutzung von Montage- und Reparaturanleitungen.* Projektarbeit an der Ludwig-Maximilians-Universität München, 2007

[PSB07] Pirchheim, Christian; Schmalstieg, Dieter; Bornik, Alexander: Visual Programming for Hybrid User Interfaces. In: *Proceedings of the 2nd International Workshop on Mixed Reality User Interfaces (MRUI'07), IEEE Virtual Reality 2007 Conference, Charlotte NC, USA*, 2007

[PVH07] Pleuss, Andreas; Vitzthum, Arnd; Hussmann, Heinrich: Integrating Heterogeneous Tools into Model-Centric Development of Interactive Applications. In: *Model Driven Engineering Languages and Systems* Bd. 4735, Springer Berlin / Heidelberg, 2007 (Lecture Notes in Computer Science), S. 241–255

[Que] *Quest3D Game Engine.* `http://www.quest3d.com/`, Abruf: 20.03.2008

[Rap] *RapidManual (Parallelgraphics).* `http://www.cortona3d.com/rapidmanual`, Abruf: 11.03.2008

[RBB+04] Reitinger, B.; Bornik, A.; Beichel, R.; Werkgartner, G.; Sorantin, E.: Tools for augmented reality based liver resection planning. In: *Proceedings of the SPIE Medical Imaging 2004: Visualization, Image-Guided Procedures, and Display*, 2004, S. 88–99

[Rey82] Reynolds, Craig W.: Computer animation with scripts and actors. In: *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques.* New York, NY, USA : ACM Press, 1982, S. 289–296

[RMBK03] Reicher, Thomas; MacWilliams, Asa; Brügge, Bernd; Klinker, Gudrun: Results of a Study on Software Architectures for Augmented Reality Systems. In: *Proceedings of the International Symposium on Mixed and Augmented Reality (STARS).* Tokio, Japan, Okt. 2003

[Ros] *Rational Rose Technical Developer.* `http://www-306.ibm.com/software/awdtools/developer/technical/`, Abruf: 18.03.2008

[RRB+06] Reitinger, B.; Reitinger, B.; Bornik, A.; Beichel, R.; Schmalstieg, D.: Liver Surgery Planning Using Virtual Reality. In: *IEEE Comput. Graph. Appl.* 26 (2006), Nr. 6, S. 36–47

[RS03] Reitmayr, Gerhard; Schmalstieg, Dieter: Location based applications for mobile augmented reality. In: *AUIC '03: Proceedings of the Fourth Australasian user interface conference on User interfaces 2003*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2003, S. 65–73

[RS04] Reitmayr, Gerhard; Schmalstieg, Dieter: Collaborative Augmented Reality for Outdoor Navigation and Information Browsing. In: *Proceedings of the Symposium of Location Based Services and TeleCartography*, 2004, S. 31–41

[SAAT05] Segura, Alvaro; Arizkuren, Iosu; Aranburu, Inaki; Telleria, Inaki: High quality parametric visual product configuration systems over the web. In: *Web3D '05: Proceedings of the tenth international conference on 3D Web technology*. New York, NY, USA : ACM Press, 2005, S. 159–167

[SAX] *SAXON XSLT und XQuery Processor*. `http://saxon.sourceforge.net/`, Abruf: 20.03.2008

[SC92] Strauss, Paul S.; Carey, Rikk: An object-oriented 3D graphics toolkit. In: *SIGGRAPH Comput. Graph.* 26 (1992), Nr. 2, S. 341–349

[Sch] *Schüco 3D-Fertighaus-Konfigurator*. `http://www.schueco.de/virtual_home_kampagne/`, Abruf: 20.03.2008

[Sch05] Schulmeister, Rolf: Interaktivität in Multimedia-Anwendungen. (2005). `http://www.e-teaching.org/didaktik/gestaltung/interaktiv/InteraktivitaetSchulmeister.pdf`, Abruf: 13.03.2008

[SCT+94] State, Andrei; Chen, David T.; Tector, Chris; Brandt, Andrew; Chen, Hong; Ohbuchi, Ryutarou; Bajura, Mike; Fuchs, Henry: Case Study: Observing a Volume Renderd Fetus within a Pregnant Patient. 1994 (TR94-034). – Forschungsbericht

[Sec] *Second Life (Linden Lab)*. `http://secondlife.com/`, Abruf: 11.03.2008

[SFH+02] Schmalstieg, Dieter; Fuhrmann, Anton; Hesina, Gerd; Szalavári, Zsolt; Encarnaçao, L. M.; Gervautz, Michael; Purgathofer, Werner: The studierstube augmented reality project. Cambridge, MA, USA : MIT Press, 2002, S. 33–54

[Sha] *Shark3D Game Engine*. `http://www.shark3d.com/`, Abruf: 20.03.2008

[Sil06] Silicon Graphics, Inc: *The OpenGL Graphics System: A Specification, Version 2.1*, 2006

[SLG+96] State, Andrei; Livingston, Mark A.; Garrett, William F.; Hirota, Gentaro; Whitton, Mary C.; Pisano, Etta D.; Fuchs, Henry: Technologies for augmented reality systems: realizing ultrasound-guided needle biopsies. In:

*SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 1996, S. 439–446

[SMM98] Schneider, Daniel K.; Martin-Michiellot, Sylvere: *VRML Primer and Tutorial*, 1998. `http://tecfa.unige.ch/guides/vrml/vrmlman/vrmlman.html`, Abruf: 13.03.2008

[Sofa] *3D-Sofa-Konfigurator (Demo von 3D-Scapes)*. `http://www.3d-scapes.de/referenzen/3d/jaguar/praesentation.html`, Abruf: 11.03.2008

[Sofb] *Softimage/XSI*. `http://www.softimage.com/products/xsi/`, Abruf: 13.03.2008

[STA] *Telelogic Statemate*. `http://modeling.telelogic.com/products/statemate/index.cfm`, Abruf: 18.03.2008

[Sun06] Sun Microsystems: *Java Platform, Enterprise Edition 5 (Java EE 5) Specification, Version: 5.0, Final Release*, 2006

[SV05] Stahl, Tom; Völter, Markus: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. dPunkt 2005, 2005

[SW06] Smith, Shamus P.; Willans, James S.: Virtual object specification for usable virtual environments. In: *OZCHI '06: Proceedings of the 20th conference of the computer-human interaction special interest group (CHISIG) of Australia on Computer-human interaction: design: activities, artefacts and environments*. New York, NY, USA : ACM, 2006, S. 183–190

[Swi] *SwirlX3D-Autorenwerkzeug*. `http://www.pinecoast.com/swirl3d.htm`, Abruf: 12.03.2008

[TCD+02] Thomas, Bruce; Close, Ben; Donoghue, John; Squires, John; Bondi, Phillip D.; Piekarski, Wayne: First Person Indoor/Outdoor Augmented Reality Application: ARQuake. London, UK : Springer-Verlag, 2002, S. 75–86

[TGVM04] Trevisan, Daniela G.; Gemo, Monica; Vanderdonckt, Jean; Macq, Benoit M.: Focus-based Design of Mixed Reality Systems. In: *Proc. TAMODIA 2004*, 2004, S. 59–66

[VH06] Vitzthum, Arnd; Hussmann, Heinrich: Modeling Augmented Reality User Interfaces with SSIML/AR. In: *JOURNAL OF MULTIMEDIA (JMM)* 1 (2006), Nr. 3, S. 13–22

[VIK+02] Vlahakis, V.; Ioannidis, M.; Karigiannis, J.; Tsotros, M.; Gounaris, M.; Stricker, D.; Gleue, T.; Daehne, P.; Almeida, L.: Archeoguide: an augmented reality guide for archaeological sites. In: *IEEE Computer Graphics and Applications* 22 (2002), Sept./Okt., Nr. 5, S. 52–60

[Vira] *Interaktive 3D-Produktpräsentation (Demo von 3D-Scapes).* `http://3d-scapes.de/referenzen/3d/SeikoM6000/M6000.html`, Abruf: 11.03.2008

[Virb] *Virtual Graffiti (Fraunhofer Institut für Graphische Datenverarbeitung).* `http://a4www.igd.fraunhofer.de/products/9/`, Abruf: 12.03.2008

[Virc] *Virtuelle 3D-Gallerie.* `http://www.3dvirtualgallery.net/3dvg.php`, Abruf: 11.03.2008

[Vird] *Virtuelle Demontageanleitung (Demo von Parallelgraphics).* `http://www.parallelgraphics.com/products/isa/examples/conductor/`, Abruf: 11.03.2008

[Vire] *Virtuelle Sightseeing-Tour (Demo von Parallelgraphics).* `http://www.parallelgraphics.com/products/showroom/virtual-tours/montmartre/`, Abruf: 11.03.2008

[Virf] *Virtueller Gebäuderundgang (Demo von Parallelgraphics).* `http://www.parallelgraphics.com/products/isa/examples/house/`, Abruf: 11.03.2008

[Vis] *Microsoft Office Visio.* `http://office.microsoft.com/visio/`, Abruf: 14.03.2008

[Vit05] Vitzthum, Arnd: 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, 2005, S. 159–167

[Vit06a] Vitzthum, Arnd: SSIML/AR: A Visual Language for the Abstract Specification of Augmented Reality User Interfaces. In: *Proc. of 3D User Interfaces (3DUI'06)*, 2006, S. 135–142

[Vit06b] Vitzthum, Arnd: SSIML/Components: a visual language for the abstract specification of 3D components. In: *Web3D '06: Proceedings of the eleventh international conference on 3D web technology.* New York, NY, USA : ACM, 2006, S. 143–151

[Vit06c] Vitzthum, Arnd: Towards a Structured Design of Augmented Reality Applications. In: *Proceedings of the First International Workshop on Mixed*

*Reality User Interfaces (MRUI'06), IEEE Virtual Reality 2006 Conference, Alexandria, Virginia, USA*, 2006

[Vit07] VITZTHUM, Arnd: Model-Driven and Framework-Supported Augmented Reality Development. In: *Proceedings of the 2nd International Workshop on Mixed Reality User Interfaces (MRUI'07), IEEE Virtual Reality 2007 Conference, Charlotte NC, USA*, 2007

[Voxa] *VOXEL-MAN 3D-Navigator: Inner Organs.* `http://www.voxel-man.de/3d-navigator/inner_organs/`, Abruf: 11.03.2008

[Voxb] *Voxel Man TempoSurg.* `http://www.voxel-man.de/simulator/temposurg/`, Abruf: 12.03.2008

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

[VRS] *VR-Therapie gegen Spinnen-Phobien des HITLab Washington.* `http://www.hitl.washington.edu/projects/exposure/`, Abruf: 11.03.2008

[W3D] *Webseite des Web3D-Konsortiums.* `http://www.web3d.org`, Abruf: 12.03.2008

[Wal] *Walrus - Informationsvisualisierung.* `http://www.caida.org/tools/visualization/walrus/`, Abruf: 12.03.2008

[Wel01] WELIE, Martijn van: *Task-based User Interface Design*, Freie Universität Amsterdam, Diss., 2001

[WH01] WILLANS, James S.; HARRISON, Michael D.: Prototyping Pre-implementation Designs of Virtual Environment Behaviour. In: *EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*. London, UK : Springer-Verlag, 2001, S. 91–108

[Wor98] WORLD WIDE WEB CONSORTIUM: *Synchronized Multimedia Integration Language (SMIL) 1.0 Specification*, 1998

[Wor03] WORLD WIDE WEB CONSORTIUM: *Scalable Vector Graphics (SVG) 1.1 Specification*, 2003

[Wor04a] WORLD WIDE WEB CONSORTIUM: *XML Schema Part 0: Primer Second Edition*, 2004

[Wor04b] World Wide Web Consortium: *XML Schema Part 1: Structures Second Edition*, 2004

[Wor04c] World Wide Web Consortium: *XML Schema Part 2: Datatypes Second Edition*, 2004

[Wor06] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, 2006

[Wor07a] World Wide Web Consortium: *XML Path Language (XPath) 2.0*, 2007

[Wor07b] World Wide Web Consortium: *XSL Transformations (XSLT), Version 2.0*, 2007

[WoW] *World of Warcraft (Blizzard Entertainment)*. `www.worldofwarcraft.com/`, Abruf: 11.03.2008

[WPS04] Wagner, Daniel; Pintaric, Thomas; Schmalstieg, Dieter: The invisible train: a collaborative handheld augmented reality demonstrator. In: *SIGGRAPH '04: ACM SIGGRAPH 2004 Emerging technologies*. New York, NY, USA : ACM, 2004, S. 12

[Xj3] *Xj3D Toolkit*. `http://www.xj3d.org`, Abruf: 12.03.2008

[YYGL04] Ying, Jianghui; Ying, Jianghui; Gracanin, D.; Lu, Chang-Tien: Web visualization of geo-spatial data using SVG and VRML/X3D. In: Gracanin, D. (Hrsg.): *Proc. Third International Conference on Image and Graphics*, 2004, S. 497–500

[ZHBH03] Zauner, J.; Haller, M.; Brandl, A.; Hartman, W.: Authoring of a mixed reality assembly instructor for hierarchical structures. In: Haller, M. (Hrsg.): *Proc. Second IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2003, S. 237–246

[Zün02] Zündorf, Albert: *Rigorous Object Oriented Software Development*. Habilitation, Universität Paderborn, 2002

[ZZZM05] Zhijiang, Du; Zhijiang, Du; Zhiheng, Jia; Minxiu, Kong: Virtual Reality-based Telesurgery via Teleprogramming Scheme Combined with Semi-autonomous Control. In: Zhiheng, Jia (Hrsg.): *Proc. 27th Annual International Conference of the Engineering in Medicine and Biology Society IEEE-EMBS 2005*, 2005, S. 2153–2156

# Danksagung

Während meiner Doktorarbeit haben viele mein Denken beeinflusst und bereichert, so dass es unmöglich ist, allen einzeln zu danken. Einige möchte ich aber besonders hervorheben.

Herrn Prof. Dr. Heinrich Hußmann bin ich zu Dank verpflichtet für die Betreuung meiner Arbeit und für die Bereitstellung eines Arbeitsumfeldes sowie der materiellen Bedingungen, die das Zustandekommen der Arbeit erst ermöglicht haben. Prof. Dr. Morten Fjeld danke ich für die freundliche Bereitschaft, meine Dissertation zu begutachten.

Meinen Kollegen von der Ludwig-Maximilians Universität bin ich dankbar für ihre konstruktive Kritik. Besonders meinem Kollegen Andreas Pleuß möchte ich meinen Dank für die zahlreichen fruchtbaren Diskussionen aussprechen. Auch Rainer Fink und Sybille Thomsen, die mir bei technischen und administrativen Schwierigkeiten Hilfestellung geleistet haben, sollen nicht unerwähnt bleiben.

Erwähnung verdient haben auch (ehemalige) Studenten der Ludwig-Maximilians-Universität wie Margit Jahn und Alexander Alzetta, die wertvolle Beiträge zum Gelingen dieser Arbeit geliefert haben.

Prof. Dr. Bernhard Jung und meinen neuen Kollegen von der Technischen Universität Bergakademie Freiberg möchte ich für die Freiräume danken, die sie mir zur erfolgreichen Fertigstellung meiner Dissertation geschaffen haben.

Ein weiterer Dank gilt meiner gesamten Familie für ihr Interesse und die Unterstützung im Alltag. Insbesondere danke ich meinem Vater für das Korrekturlesen. Nicht zuletzt möchte ich auch meiner Freundin für ihr Verständnis für meine Arbeit und die vielen motivierenden Worte danken.

Dank auch an alle unerwähnt Gebliebenen, die zum Gelingen meiner Doktorarbeit beigetragen haben.

# Lebenslauf

Arnd Vitzthum

**Persönliche Daten**

| | |
|---|---|
| Geburtsdatum: | 27. September 1976 |
| Geburtsort: | Dresden |

**Schulbildung, Studium und Beruf**

| | |
|---|---|
| 1983 - 1995 | Grundschule und Gymnasium in Dresden, Abschluss Abitur |
| 1995 - 1996 | Zivildienst im Krankenhaus Dresden-Friedrichstadt |
| 1996 - 2002 | Informatikstudium an der Technischen Universität Dresden, Abschluss Diplom |
| 2003 - 2008 | Wissenschaftlicher Mitarbeiter an der Lehr- und Forschungseinheit Medieninformatik der Ludwig-Maximilans-Universität München |
| seit 15.01.2008 | Wissenschaftlicher Mitarbeiter an der Professur für Virtuelle Realität und Multimedia der Technischen Universität Bergakademie Freiberg |