Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

David Asmuth

Prozedurale Erzeugung von Fahrzeugen auf Basis einer domänenspezifischen Sprache und parametrierbaren Modellen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

David Asmuth

Prozedurale Erzeugung von Fahrzeugen auf Basis einer domänenspezifischen Sprache und parametrierbaren Modellen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Jenke
Zweitgutachter: Prof. Dr. Sarstedt

Eingereicht am: 29.09.2017

David Asmuth

Thema der Arbeit
Prozedurale Erzeugung von Fahrzeugen auf Basis einer domänenspezifischen Sprache und parametrierbaren Modellen

Stichworte
Prozedurale Modellierung, Prozedurale Fahrzeuge, DSL, Domänenspezifische Sprache, Parametrierbare Modelle, Shape-Keys

Kurzzusammenfassung
Die Ansprüche und Komplexität von 3D-Modellen werden mit steigenden Rechner-Kapazitäten immer größer. Damit steigt auch der Kosten- und Zeitaufwand der für die Erstellung benötigt wird. Mit Hilfe spezieller Werkzeuge lässt sich der Aufwand senken, indem die Modelle automatisiert erzeugt werden. Dies kann man durch eine spezialisierte Sprache und veränderbaren Modell-Teilen realisieren. In diesem Dokument geht es darum eine solche Sprache zu entwerfen und einen Prototyp zu entwickeln der anhand der Sprache und Modellteile Fahrzeuge erstellt.
David Asmuth

Title of the paper
Procedural generation of vehicles based on a domain specific language and parameterizable models

Keywords
Procedural Modeling, Procedural Vehicles, DSL, Domain Specific Language, Parameterizable Models, Shape-Keys

Abstract
The standards and complexity of 3D models increase with the ongoing growth of computer processing power. This leads to increased investment of time and money for creating these assets. With the help of specialized tools, we're able to decrease these efforts by automatizing the process of modeling. This can be achieved by using a specific language and deformable partial models. This document is about designing such a language and implement a prototype that uses this language and partial models to generate vehicles.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Abbildung 1.1: Eine Collage von durch den Prototypen erzeugter Fahrzeugmodelle

Egal ob Video-Spiel oder Animationsfilm, oft werden große Mengen von 3D-Modellen benötigt die eine Heerschar von Designern entwirft. Dieser Vorgang ist zeitaufwändig und kostspielig. Die immer größeren Rechnerkapazitäten lösen das Problem nicht, sondern verschärfen es indem sie die Erwartungshaltung und Ansprüche an 3D-Modelle erhöhen. Um diesen Ansprüchen entgegen zu kommen müssten noch mehr Designer eingestellt werden, was wiederum die Kosten weiter erhöht.
Eine mögliche Lösung um ansprechende Inhalte mit minimalem Arbeitsaufwand zu erzeugen, ist die prozedurale Modellierung. Es handelt sich um prozedurale Inhalte wenn Teile des Inhalts,

oder der gesamte Inhalt automatisiert durch ein Regelsystem erstellt wird. Unterschiedliche Bereiche erfordern unterschiedliche Herangehensweisen, um die entsprechenden Inhalte prozedural zu erzeugen.
So werden oft einzelne Pflanzen in Spielen nicht mehr von einem Menschen modelliert, sondern ein Algorithmus erzeugt diesen vollautomatisch auf Basis von L-Systemen. Die dazugehörigen Texturen entstammen nicht mehr von aufwändig nachbearbeiteten Fotos, sondern werden durch die Modell-Daten, Fraktale und vordefinierter physikalischer Eigenschaften erstellt.
Im Gegensatz zu Pflanzen können Landschaften einfach aus Bildrauschen und Fraktalen erzeugter Höhenkarten generiert und gegebenenfalls nachbearbeitet werden.

Ein weiterer Bereich sind Fahrzeuge, für die ein gänzlich anderer Ansatz verfolgt werden muss. Für Fahrzeuge soll aus den vorgegebenen Fahrzeugparametern ein Fahrzeug-Modell erzeugt werden. Dazu müssen komplexe Regeln und Zusammenhänge abgebildet werden können. L-Systeme und ähnliche Ansätze sind hier ungeeignet. Dieses Problem soll ein System lösen, welches auf Basis einer spezifischen Grammatik und Parametern Fahrzeug-Modelle generiert. Somit könnte man beispielsweise einen ganzen Fahrzeugstau mit sehr vielen unterschiedlichen Fahrzeugen simulieren, ohne dass dies zu einer zeit- und kostspieligen Angelegenheit wird.

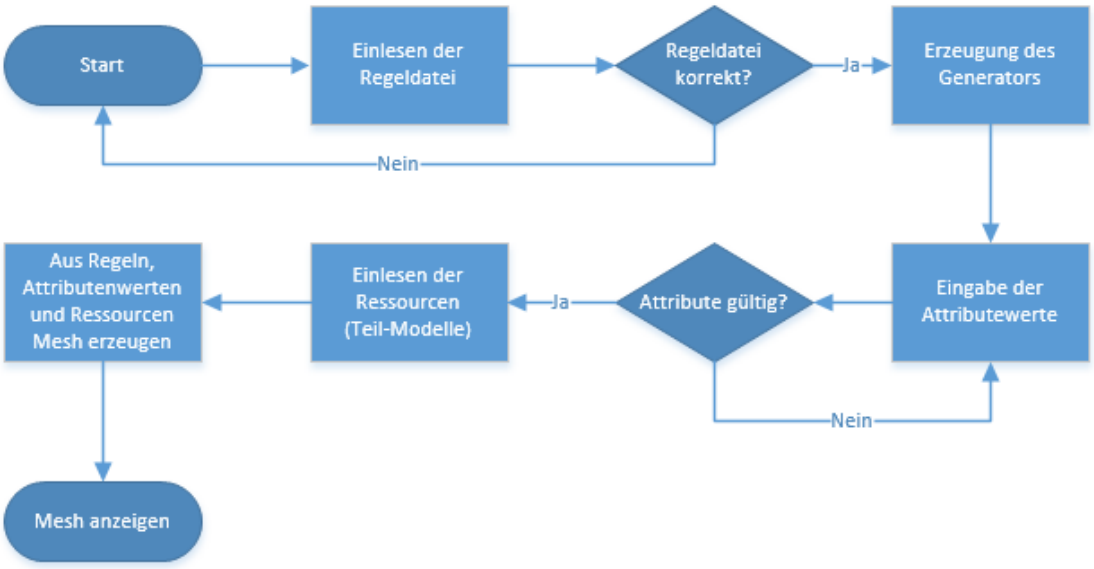

Abbildung 1.2: Ablaufdiagramm zur Erstellung eines Fahrzeugs

1.2 Zielsetzung

In dieser Arbeit soll ein System entwickelt werden, mit dem es möglich sein soll Fahrzeuge mittels Regeln, Parametern und Modellteilen zu generieren. Ziel dieses Systems ist, mit wenigen Ressourcen eine große Variation von Fahrzeug-Modellen erzeugen zu können. Dabei geben die Regeln die Struktur und die Modellteile die Form des Fahrzeugs vor. Die Parameter sollen sowohl die Struktur (Ein LKW hat keinen Kofferraum), als auch die Form (Sportfahrzeuge sind mehr rund als eckig) bestimmen. Zudem sollen über die Grammatik einfache Abhängigkeiten darstellbar sein (ein Sportwagen hat keinen Anhänger). Die Grammatik definiert die zur Verfügung stehenden Parameter. Somit können andere Regeln mit gänzlich anderen Parametern arbeiten. Die Regeln und die Modellteile sollen einfach austausch- und erweiterbar sein.
Das System soll nur die Mesh Erzeugung vornehmen. Eine prozedurale Texturierung soll zwar nicht ausgeschlossen werden, ist aber nicht Ziel dieser Arbeit.

Die Arbeit soll damit eine Grundlage schaffen auf der weitere Arbeiten aufbauen können.

1.3 Struktur der Arbeit

Die Arbeit ist in 7 Kapitel gegliedert die wie folgt aufgebaut sind:
Das erste Kapitel gibt eine Einführung in das Thema.
Darauf folgen im zweiten Kapitel Grundlagen und bisherige Arbeiten, um einen Überblick über das Thema zu verschaffen. Des Weiteren werden unter Anderem die Begriffe prozedural, domänenspezifische Sprache und Shape-Keys erläutert.
Im dritten Kapitel werden die Anforderungen an die Sprache, die Form der Modell-Daten und des Prototyps ermittelt.
Im Anschluss folgt im vierten Kapitel der Entwurf des Prototyps im Hinblick auf die Anforderung aus dem vorangegangen Kapitel.
Im fünften Kapitel erfolgt die Umsetzung des Entwurfs und Erläuterungen wie die einzelnen Komponenten funktionieren und interagieren.
Das sechste Kapitel beschäftigt sich mit der Evaluation des Prototyps anhand der Anforderungen aus dem dritten Kapitel und beinhaltet Messungen zur Leistungsfähigkeit des Prototyps.
Im siebten und letzten Kapitel findet noch einmal eine Zusammenfassung der Arbeit und ein Ausblick auf mögliche Erweiterungen statt.

2 Grundlagen

2.1 Überblick

Dieses Kapitel soll die Grundlagen vermitteln die zum Verständnis der darauffolgenden Kapitel benötigt werden. Dazu gehören domänenspezifische Sprachen um die Benutzer-Eingabe zu verarbeiten, die Shape-Keys um die Modellteile zu modifizieren und die Prinzipien von Prozeduraler Modellierung im Allgemeinen.

2.2 Domänenspezifische Sprachen

Domänenspezifische Sprachen (kurz: DSL) sind Sprachen die spezialisiert auf einen bestimmten Zweck optimiert wurden. Sie unterscheiden sich zu Universalsprachen oder auch "General Purpose Languages"(kurz: GPL) genannt, darin, dass mit ihnen nur bestimmte Problemstellungen gelöst werden können. Dies wird erreicht indem für eine Problemstellung explizite Sprach-Merkmale definiert werden und nicht benötigte Merkmale entfallen. Dadurch lassen sich mit domänenspezifischen Sprachen — obwohl weniger mächtig als eine GPL — Problemstellungen häufig eleganter und mit weniger Aufwand lösen. [AvD00, Kap. 1]
Beispiele für DSL sind SQL, eine Sprache zugeschnitten auf Datenbank-Abfragen, oder PostScript, eine Sprache zur Beschreibung von Dokumenten für Drucker und Anzeige-Geräte.
Innerhalb der domänenspezifischen Sprachen wird noch einmal zwischen einer internen DSL und einer externen DSL unterschieden. [AvD00, Kap. 6]

Interne DSL basieren auf einer Trägersprache, vorzüglich auf einer Universalsprache. Dabei wird die Syntax der Trägersprache verwendet. Dies hat den Vorteil, dass bereits zu Übersetzungszeit die DSL auf Syntax überprüft werden kann ohne das ein eigener Interpreter oder Parser benötigt wird. Hat aber gleichzeitig - je nach Trägersprache - den Nachteil, dass der Quelltext der DSL bereits schon zur Übersetzungszeit bereitstehen muss, oder mangels Ausdrucksstärke umständliche Sprachkonstrukte entstehen.

Externe DSL basieren auf keiner anderen Sprache und können somit komplett neu gestaltet

werden. Dies führt dazu, dass man sich mit der Gestaltung der Syntax und der anschließenden Verarbeitung auseinander setzen muss. Die Syntax muss dabei hinsichtlich der Anforderungen gestaltet werden und für eventuelle Erweiterungen gewappnet sein. Für die Verarbeitung von Quelltexten der DSL müssen geeignete Komponenten entworfen werden. Das können Parser, Interpreter und Compiler sein. Diese Systeme müssen nicht zwangsläufig selber geschrieben werden. Mit Hilfe von so genannten "Language Workbenches"welche untere anderem Parser-Generatoren mitbringen, lassen sich die Quellcode verarbeitenden Komponenten auch automatisch aus der Sprachdefinition erzeugen [Fow05, S. 15]. Der verarbeitete Quelltext, kann in eine weitere Zielsprache umgewandelt werden.

2.3 Prozedurale Modellierung

Die Modellierung von virtuellen Objekten kann mittels verschiedener Techniken bewerkstelligt werden. Der übliche, nicht prozedurale, Ansatz ist der, über eine Modellierungssoftware. Dabei wird das Modell mittels der von der Software angebotenen Funktionen "händisch"von einem Designer entworfen indem dieser das Mesh modifiziert und erweitert. Die Bearbeitung des Mesh muss nicht zwangsläufig explizit erfolgen, sondern kann auch beispielsweise über implizite Oberflächen-Funktionen wie Marching Cubes erzeugt werden.
Bei einem prozeduralen Ansatz wird das Model über Prozeduren erzeugt. Dies wird häufig auch Modell Synthese genannt. Es erfolgt eine Eingabe mit Parametern, die darauf hin eine vorher festgelegte Reihe von Operationen nach einem Regelwerk ausführt, die wiederum das eigentliche Mesh erzeugen. Die Regeln können auch Zufälligkeit beinhalten. Also gleiche Eingaben müssen nicht zu einer immer gleichen Ausgabe führen [Fre16, S. 4]. Dabei kann es nötig sein, dass die Eingabe eine vor- bzw. die Ausgabe einer Nachbearbeitung bedarf.
Der Übergang von nicht- zu prozeduraler Modellierung ist fließend, denn viele Funktionen innerhalb einer Modellierungssoftware haben einen prozeduralen Hintergrund. Funktionen für Weichzeichnung, Spiegelung, Derformierung und Triangulierung von Meshes basieren auf einem prozeduralen Prinzip.

2.4 Parametrierbare Modelle (Shape-Keys)

Shape-Keys, Vertexkeys, Per-Vertex Animation, Shape Interpolation, Morph-Targets oder Blend-Shapes sind spezielle Zusatzinformationen für Modelle. Obwohl je nach Software und Kontext ein anderer Name verwendet wird, beschreiben alle Bezeichnungen das selbe Prinzip: Ein Shape-Key bezeichnet eine Gruppe vom Ursprungs-Modell abweichende Vertex-Positionen.

Dazu wird der Gruppe ein Name Zugewiesen und die betroffenen Vertex-Positionen abweichend vom Ursprungs-Modell verschoben. Die neue Position ist dabei nicht absolut, sondern relativ zum Ursprungspunkt zu sehen. Der Verschiebungsfaktor, der sogenannte Offset, kann über die in dem Shape-Key definierten Vertex-Positionen hinausgehen. Es ist auch möglich einen negativen Offset zu verwenden.
Mit Hilfe mathematischer Operationen und Gewichtungen lässt sich das Ursprungs-Modell mit beliebig vielen Shape-Keys mischen. Damit stellen sie eine Alternative zu Skelett-Basierter Animationsdaten dar. Shape-Keys werden überall dort eingesetzt wo ein Skelett-System nicht sinnvoll ist.
Der häufigste Anwendungsfall in denen Shape-Keys zum Einsatz kommen sind Gesichtsanimationen [BV99]. Das Ursprungs-Modell ist eine entspannte Gesichtshaltung, die Shape-Keys stellen jeweils einen Gesichtsausdruck dar. Im Endergebnis steht nun das entspannte Gesichts-Modell mit jeweils einem Parameter für einen Gesichtsausdruck. Die Parameter können nun unabhängig gewichtet werden, sodass flüssige Übergange zwischen Gesichtsausdrücken möglich sind.[Liu09, S. 11]

Für diese Arbeit werden die Shape-Keys zur Parametrisierung von Modellteilen verwendet. Im Bezug auf Fahrzeuge bedeutet dies Beispielsweise, dass ein Reifen-Modell einen Parameter “Felgengröße“ besitzt, mit dem übergangslos das Verhältnis zwischen Reifen- und Felgengröße festgelegt werden kann.

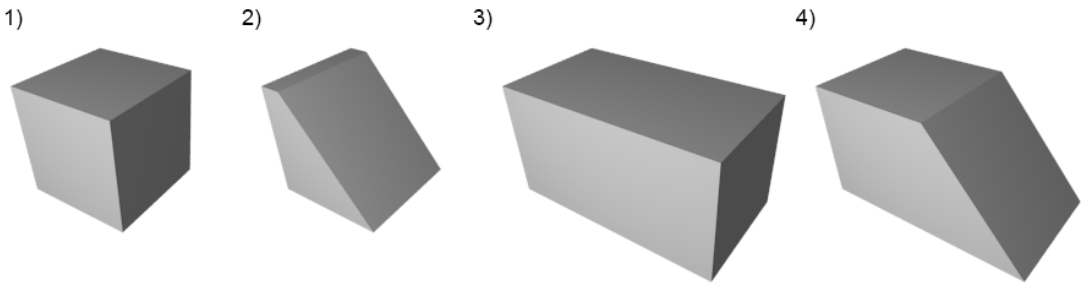

Abbildung 2.1: Grundfunktionalität von Shape-Keys. 1) Würfel als Basismodell, 2) Shape-Key mit abgeflachter Kante, 3) Shape-Key mit verlängerten Seiten, 4) Basis-Modell mit beiden Shape-Keys in voller Gewichtung

2.5 Aktuelle Entwicklungen

Wie im Kapitel 1.1 bereits angesprochen, gibt es eine Reihe von Bereichen in der Software-Entwicklung in denen prozedurale Modellierung die Erstellung von Inhalten erleichtern, oder gar komplett automatisieren. Für die Modellierung von Fahrzeugen gibt es verschiedene Ansätze. Das Grundprinzip ist bei Allen sehr ähnlich [Pap14]. Es werden Teilmodelle erstellt, und die prozedurale Anordnung erstellt aus den verschiedenen Teilmodellen dann immer unterschiedliche Gesamtmodelle. Im Folgenden werden verschiedene Arbeiten und Endprodukte vorgestellt:

A Probabilistic Model for Component-based Shape Synthesis [KCKK12] ist eine Arbeit aus der Universität Stanford. In der Arbeit werden Teilmodelle aus Bestehenden Gesamtmodellen extrahiert und mittels eines lernenden Algorithmus zu neuen Gesamtmodellen zusammengesetzt. Die Ergebnisse sehen hervorragend aus. Die Modelle sind plausibel und weisen eine hohe Diversität auf. Dies ließ sich nicht nur auf Fahrzeuge anwenden, sondern auch auf Möbelstücke, Kreaturen und Flugzeugen. Der Ansatz garantiert jedoch jedoch keine Konsistenz der Modelle. Es gib weitere Arbeiten die darauf aufbauen und ein Deep-Learning Verfahren verwenden [HKM15].

Example-Based Model Synthesis [Mer07] aus der Universität North Carolina, verwendet bestehende Modelle und Teilt diese Manuell auf Teilmodelle auf. Anschließend werden per Algorithmus Modellteile ausgewählt die konsistent genug sind um sie in wiederholenden Mustern zu einem Gesamtmodell zusammen zu setzen. Dies Funktioniert am Besten mit sich wiederholenden Strukturen, wie zum Beispiel Gebäuden bzw. Architektur im Allgemeinen. Für organische Synthese ist diese Technik nicht gut geeignet.

No Man's Sky[NMS] ist ein Computerspiel, welches vom Britischen Entwickler-Studio Hello Games[Hel] entwickelt wurde. Im diesem Spiel startet der Spieler in einer vollständig prozeduralen Galaxie mit 2^{64} Planeten, von denen viele mit prozedural erzeugter Flora, Fauna und intelligentem Leben bestückt sind. Der Spieler kann von Planet zu Planet reisen. Dazu nutzt er ebenfalls prozedural erzeugte Raumschiffe. Obwohl die unterschiedlichen Modelle keine spielerischen Auswirkungen haben, kann man hier ein festes Regelwerk erkennen: Jäger haben ein schnittiges Aussehen. Aufklärer wirken sehr grazil im Vergleich zu den eher großen, klobigen Transportern.

Durch die schiere Größe ist es unwahrscheinlich, dass Spieler jemals den exakt gleichen Inhalt

zu Gesicht bekommen und steht auch Beispielhaft dafür, welche Möglichkeiten prozedurale Ansätze eröffnen die mit klassischen Verfahren nicht erreicht werden können.

Abbildung 2.2: Verschiedene prozedurale Schiffsmodelle aus dem Spiel No Man's Sky. (Screenshot Hello Games: Hello Games[Hel] , 2014, twitter.com/nomansskynews)

Der **ShapeWright Schiffs Generator** ist ein Web-Projekt vom niederländischen Künstlers Dolf Veenvliet[vee]. Auf der seiner Seite[shi] kann ein Benutzer unter Verwendung eines Seeds ein raumschiffartiges Modell erstellen lassen. Die Regeln zur Erstellung der Modelle sind nicht sonderlich komplex und es gibt auch keine Möglichkeit das Ergebnis gezielt zu beeinflussen. Die prozedural erstellten Modelle sind dadurch nicht immer plausibel, aber sehr variabel und meist optisch ansprechend bis kurios.

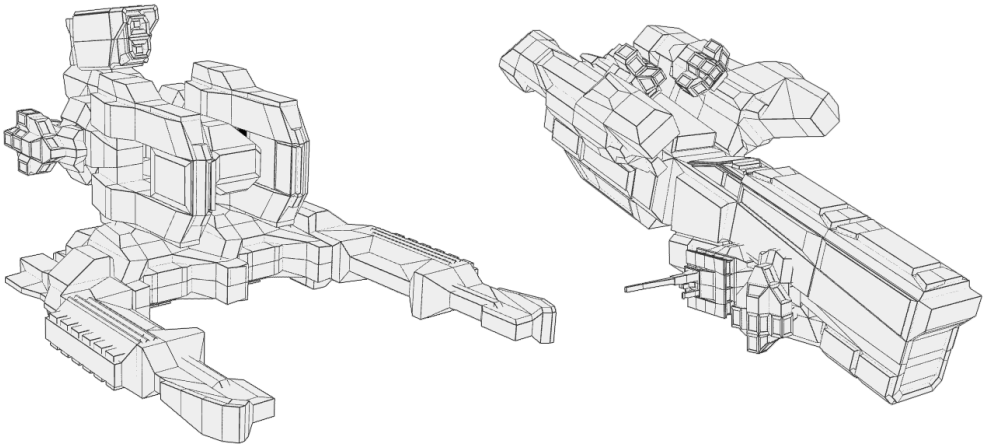

Abbildung 2.3: Auf der Website ship.shapewright.com erstellte Schiffsmodelle (Selbst erstellter Screenshot: Dolf Veenvliet[vee], 2017, ship.shapewright.com)

3 Anforderungen

In diesem Kapitel werden die Anforderungen an den Prototypen gestellt.

3.1 Anforderungen an die DSL

Die domänenspezifische Sprache sollte grundsätzlich simpel und einfach verständlich sein. Um dies zu erreichen sollte sie möglichst wenige von der natürlichen Sprache abweichenden Sprachkonstrukte verwenden. An den Stellen an denen dies nicht gelingt, sollte sie auf bekannte Sprachkonstrukte aus anderen Programmiersprachen zurückgreifen, um den Anwender möglichst nicht zu überraschen.
Das Hauptaugenmerk dieser Sprache sollen Regeldefinitionen sein, die zur Erzeugung von Fahrzeugen angewendet werden. Jede Regel sollte dabei aus mindestens einem Funktionsaufruf bestehen. Ein Funktionsaufruf kann, sofern gewünscht, wieder eine weitere Regel aufrufen. Aus diesen beiden Anforderungen ergibt sich zunächst eine kontextfreie Grammatik, die in der Chomsky-Hierarchie als Typ-2-Grammatik klassifiziert wird. Es werden jedoch noch weitere Sprachkonstrukte benötigt.
Funktionsaufrufbedingungen: Funktionsaufrufe sollen nur unter bestimmten Voraussetzungen erfolgen. Ist in einer Bedingung ein bestimmter Schwellenwert nicht erreicht, soll kein Funktionsaufruf stattfinden.
Handelt es sich bei dem Funktionsaufruf um die Modellierung und Anbringung eines Heckspoilers kann ein Logischer Ausdruck diesen verhindern, wenn es sich beispielsweise um ein Nutzfahrzeug handelt.
Funktionsaufruf-Quantifizierung: Funktionsaufrufe sollen gar nicht, einmalig oder mehrmals aufgerufen werden können. Dazu soll der Funktionsaufruf mit Werten quantifizierbar sein. Die Werte geben dabei an, wie oft die Funktion aufgerufen wird. So kann der Aufruf der genau eine Sitzreihe an das Fahrzeug anbringt, genau so oft aufgerufen werden, wie Sitzplätze gefordert werden.
Funktionsaufrufhierarchie: Es sollte möglich sein auf die Parameter der übergeordneten Funktionsaufrufe zugreifen zu können. Damit lassen sich dann Zusammenhänge in der Aufrufhierarchie abbilden.

Funktionsüberladung: Wenn Funktionen mit nahezu gleicher Funktionalität bestehen die sich nur in ihrer Parameterzahl unterscheiden, dann sollte es nur einen Funktionsaufruf geben. Die benötigte Unterscheidung entsteht durch die Anzahl der Parameter.
Mathematische Ausdrücke: Für die Aufrufbedingungen, die Quantifizierung und für die Funktionsparameter selbst, werden mathematische Ausdrücke gebraucht. Deswegen werden arithmetische und logische Operationen benötigt. Die Ausdrücke sollen die Operatorpräzedenz einhalten und evaluiert werden, bevor mit ihnen gearbeitet wird (sog. “Strict Evaluation“).

3.2 Anforderungen an die Modellteile

Um das Fahrzeug-Mesh zu erzeugen gibt es keinen prozeduralen Ansatz. Daher werden bereits fertige Mesh-Einzelteile benötigt. Ein naiver Ansatz wäre nun, die Mesh-Einzelteile passend, aber wahllos, oder nach bestimmten Regeln ‚“zusammen zu stecken“. Dies bietet aber noch keine echte Varietät und es entstehen sich schnell wiederholende Muster. Zur Vermeidung von beidem könnte man jetzt hunderte Einzelteile des selben Teil-Typs anfertigen. Zum Beispiel: 100 Variationen einer Felge, 100 Variationen einer Karosserie und so weiter. Damit löst man aber das in der Motivation 1.1 erklärte Ziel nicht. Die Kosten bzw. der Aufwand für die fertigen Fahrzeugmodelle wären nicht nennenswert gefallen. Aus diesem Grund werden verformbare Modellteile auf Basis der in 2.4 erläuterten Shape-Keys benötigt. Erstellt man für ein Einzelteile geschickt Shape-keys, lassen sich daraus viele Variationen bilden, ohne das hunderte Einzelteile erstellt werden müssen.
Für diese Art der Modellteile ergeben sich nun folgende Anforderungen:
Einheitliches Namensschema: Die Namen der einzelnen Shape-Keys müssen über alle Einzelteile hinweg einem Schema befolgen. Denn sonst besteht das Problem das die Grenzen von zwischen zwei Teilmodellen sichtbar werden (siehe 3.1) . Verändert also ein Shape-Key Vertices im Mesh die zu anderen Modellen als “benachbart“ gelten, müssen dieselben Veränderungen unter dem selben Namen in allen benachbarten Modellteilen stattfinden. Eine weitere Möglichkeit wäre es, Vertices mit direkten Nachbarn nicht mit Shape-Keys zu modifizieren.
Markierungen: Durch die Shape-Keys können sich die Positionen an denen weitere Einzelteile benötigt werden verschieben. Wenn eine solche Verschiebung stattfindet muss die neue Position ausgelesen werden können. Es werden also Markierungen benötigt, die sich mit den Shape-Keys verschieben, sodass angrenzende Einzelteile ebenfalls verschoben werden.

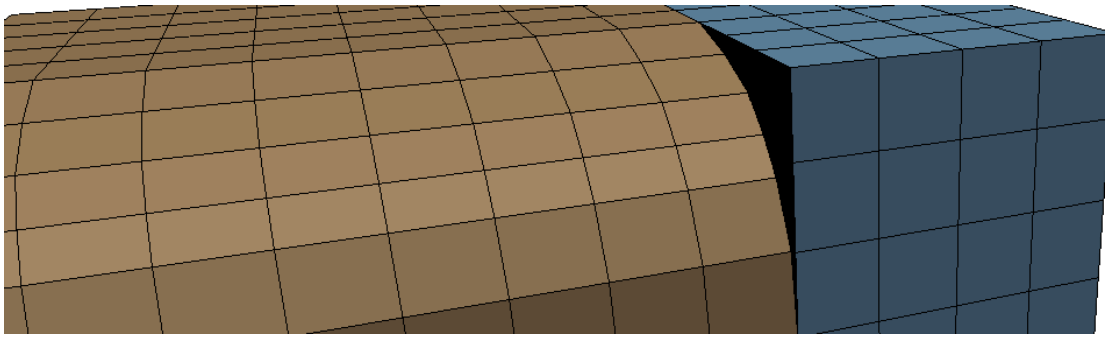

Abbildung 3.1: Durch unterschiedliche Shape-Keys verursachte Lücke zwischen zwei Einzelteilen

3.3 Anforderungen an die Dateiformate

Der zu entwerfende Prototyp arbeitet mit 2 Dateiformaten. Einem Dateiformat für die Regeldateien und einem Dateiformat für die Modellteile. Die Anforderungen an die Regeldateien lassen sich aus den Anforderungen an die DSL ableiten und bedürfen keiner weiteren Erklärung, da es sich letztendlich um einfache Text-Dateien handelt.
Die wichtigste Anforderung an das Dateiformat der Modellteile ist die Unterstützung für Shape-Keys. Werden diese nicht Unterstützt ist das Format unbrauchbar. Das Format muss die namentliche Markierung von Vertices oder ganzen Vertex-Gruppen unterstützen, um die Anforderungen aus 3.2 zu erfüllen. Zudem sollte das Format einfach gehalten sein, damit sich die Verarbeitung der Modellteile nicht zu komplex gestaltet. Wünschenswert ist ebenfalls die Lesbarkeit des Format zu debugging-Zwecken. Das Format sollte also vorzugsweise nicht in Binär sondern in ASCII-Text formatiert sein. Zuletzt sollte es sich um ein nicht patentbehaftetes, freies Format handeln, um Rechtsunsicherheiten aus dem Weg zu gehen.

3.4 Anforderungen an den Prototypen

Der Prototyp soll in der Lage sein schnell und unkompliziert neue Fahrzeuge zu erstellen. Schnell bezieht sich auf die Zeit und den Speicheraufwand die für das einlesen der Regeldatei, sowie das verarbeiten der Regeln benötigt wird. Es wäre wünschenswert, wenn der Prototyp bereits so schnell ist, dass sich abzeichnet ob das Konzept geeignet ist, um in einer Echtzeitanwendung verwendet zu werden. So könnte Beispielsweise ein Spiel realisiert werden, in dem sich der Spieler du eine endlos prozedurale Landschaft bewegt in der es endlos viele Prozedural erstellte Städte gibt in denen prozedural erzeugte Fahrzeuge fahren.

Unkompliziert bezieht sich auf die Menge der Schnittstellen die nach außen benötigt werden um um den Prototypen zu steuern. Möglichst wenig ist hier das angestrebte Ziel.

3.4.1 Anforderungen an die Benutzerschnittstelle

In der Benutzerschnittstelle soll der Benutzer zunächst eine Regeldatei auswählen und laden können. Die Benutzerschnittstelle sollte es dem Benutzer ermöglichen, alle in der Regeldatei definierten Variablen zu belegen und Bedingungen zu steuern. Dazu sollte die Benutzeroberfläche dynamisch auf Basis der ausgewählten Regeldatei erzeugt werden können und die verfügbaren Variablen und Steuerelemente grafisch präsentieren.

Abschließend sollte die Benutzerschnittstelle dem Benutzer das erzeugte Fahrzeug-Modell anzeigen.

4 Konzept

4.1 Auswahl der Grafik-Engine

Für die Umsetzung des Prototyps stehen eine Fülle an bereits bestehender Grafik- und Spiele-Engines zur Verfügung. Wichtig für die Umsetzung des Prototyp sollte ein möglichst leichter Einstieg sein. Um dies zu erreichen sollte die Grafik-Engine bereits viele benötigte Funktionen mitliefern, oder zumindest einfach um benötigte Funktionen erweiterbar sein.

Das für diese Arbeit gewählte CG-Framework von Prof. Dr. Jenke bietet auf Basis des Java-Framework eine einfache Grafik-Engine mit einem Scene-Graphen und integrierter Funktionalität für die Einbindung individueller Benutzer-Oberflächen.

4.2 Auswahl eines geeigneten Modell-Formats

Für die Modellteile muss im CG-Framework ein entsprechender Model-Loader zur Verfügung stehen, der die Mesh-Daten aus der Modell-Datei einliest und die Daten in die Datenstrukturen des CG-Frameworks überträgt. Im Folgenden werden verschiedene, häufig genutzte Modell-Container vorgestellt und auf Basis der Anforderungen ein geeignetes Container-Format ausgewählt:

Wavefront OBJ ist ein alter aber weit verbreiteter Modell-Container. Er verwendet das ASCIII-Format und ist sehr simpel gehalten. OBJ Unterstützt zwar Materialien, allerdings weder Skelett- noch Shape-Key-Animationen. Durch seine hohe Verbreitung und Einfachheit wird es von vielen Frameworks und Grafik-Engines unterstützt.

Filmbox FBX ist ein von der Firma Autodesk entwickelter, proprietärer Modell-Container. Ursprünglich zu Motioncapture-Zwecken entwickelt, ist es heute ein verbreitetes Format um Modell-Daten zwischen Anwendungen auszutauschen. FBX unterstützt Animationen (sowohl Shape-Keys als auch Skelette), Szenen, Vertex-Gruppen und Materialien. Dabei kann es entweder im ASCII-Format oder Binär gespeichert werden.

COLLADA DAE ist ein offener, von dem Industriekonsortium Khronos Group und Sony entwickelter, Modell-Container. Es wurde mit dem Ziel entwickelt ein offenes Austauschformat zwischen verschiedenen Anwendungen zu bieten. Es basiert auf der XML Syntax und ist somit nur im ASCII-Format verfügbar. COLLADA unterstützt eine sehr breite Palette von Funktionen. Darunter: Animation in Form von Skeletten und Shape-Keys, Szenen, Vertex-Gruppen und Materialien.

Die Anforderungen an das Modell-Format erzwingen die Kompatibilität von Shape-Keys. Wavefront unterstützt diese Grundsätzlich nicht und ist somit kein geeignetes Format. Filmbox FBX sowie COLLADA unterstützen beide die nötigen Shape-Keys und können im ASCII Format vorliegen. COLLADA wurde im Gegensatz zu FBX nur zum Austausch zwischen Anwendungen entworfen und wurde daher nicht auf Echtzeitverarbeitung optimiert. Da das FBX Format nicht offen ist, wird eine geschlossene Bibliothek des Entwicklers Autodesk benötigt, welche unter einer eigenen Lizenz unterliegt, die die Weitergabe der Bibliothek untersagt. Da COLLADA keiner restriktiven Lizenz unterliegt und das Format offen ist, wird für diese Arbeit das COLLADA-Format verwendet.

4.3 Entwurf des Model-Loaders

4.3.1 COLLADA Bestandteile

Da das CG-Framework von Prof. Dr. Jenke keinen COLLADA-Loader mitbringt, muss ein entsprechender COLLADA-Loader entwickelt werden. Da das vollständige COLLADA Format sehr umfangreich und Komplex ist, werden nur die essentiellen und benötigten Komponenten des COLLADA-Formats geladen:

Header
Der COLLADA-Header enthält Metadaten die insbesondere für Debugging-Zwecke hilfreich sind. In diesen Daten lässt sich ablesen mit welchem Programm die Datei erzeugt wurde und welches Koordinaten-System und Längenmaß verwendet wird.

Szenen
Szenen sind der Einstiegspunkt um zu erfahren welche Inhalte in der Datei vorliegen. Ein Szeneneintrag besteht aus sogenannten Nodes, die wiederum Geometriedaten referenzieren können.

Geometrie
Die Geometriedaten beinhalten die Daten die zur Darstellung eines Modells benötigt werden. Für das Projekt werden lediglich Vertex- Index- und Farb-Daten sowie die Oberflächen-Normalen benötigt. Welche Geometriedaten zu welchem Model gehören wird in der Szene festgelegt. Shape-Keys sind ebenfalls gewöhnliche Geometriedaten.

Controller
Innerhalb der Controller wird unter Anderem festgelegt welche Geometriedaten Shape-Keys von Szenen-Modellen sind, indem jeweils ein Eintrag pro Basis-Mesh auf alle zugehörigen Shape-Key Geometrieeinträge verweist.

4.4 Entwurf der DSL

Um die Sprache zu entwerfen, muss zunächst die Frage geklärt werden wie, bzw. womit die Sprache verarbeitet werden soll. Diese Entscheidung hat unmittelbare Auswirkungen auf die Möglichkeiten der Sprachfunktionen. Die Verarbeitung einer Sprache erfolgt in mehreren Schritten. Der erste Schritt ist die sogenannte lexikalische Analyse indem die Sprachteile, also Schlüsselwörter, Bezeichner, Konstanten und Operatoren in Tokens überführt werden.
Im Anschluss an die lexikalische Analyse erfolgt die Überführung in einen Syntaxbaum, welcher wiederum von einem Tree-Parser abgearbeitet wird.
Man könnte all diese Komponenten selbst schreiben, aber heutzutage können die in Kapitel 2 beschriebenen Parsergeneratoren diese Aufgabe übernehmen. Dabei wird eine Eingabe, die eine Beschreibung der Sprache enthält, an den Parsergenerator übergeben. Dieser erstellt automatisch aus diesen Vorgaben die Komponenten in eine gewünschte Zielsprache. Welche Zielsprachen unterstützt werden, ist vom Parsergenerator abhängig.
Im diesem Kapitel wird nun ein geeigneter Pasergernerator gewählt. Anschließend werden die benötigten Sprachfunktionalitäten entworfen und im letzten Abschnitt die einzelnen Bestandteile der Sprache erklärt.

4.4.1 Parsergenerator ANTLR

```
grammar ShapeGrammar;

shape_grammar: decoration* start_entry shape_rule+;

decoration: attr | resource;

attr: 'attr' ATTR_TYPE IDENTIFIER ('(' NUMBER ',' NUMBER (',' NUMBER)? ')')? END;

resource: RESOURCE_TYPE REFERENCE_WORD PATH END;

start_entry: 'start' REFERENCE_WORD END;

shape_rule: REFERENCE_WORD ARROW ((func|conditional_func) (',' (func|conditional_func))*) END;

func: IDENTIFIER '(' (parameter (',' parameter)*)* ')' func_quant?;

conditional_func: boolean_condition '?' func;

parameter: expr | REFERENCE_WORD;

func_quant: '*' expr;

boolean_condition: expr (CONDITION_OP expr);
```

Abbildung 4.1: Beginn einer ANTLR Grammatik-Datei zum Parsen von Regel-Dateien

Für diese Arbeit wurde ANTLR als Parsergenerator gewählt[ANTc]. ANTLR unterstützt als Zielsprache Java und bietet eine sehr gute Unterstützung für die Java Entwicklungsumgebungen Eclipse[ANTa] und IntellJ IDEA[ANTb].
Die Eingabe erfolgt bei ANTLR über eine Grammatik-Datei in der die Regeln der Sprache festgelegt werden. Die ANTLR spezifische Sprache wird einfach Grammatik genannt und erlaubt es eigene komplexe Programmiersprachen zu Entwickeln. Die Verarbeitungskette sieht dabei folgendermaßen aus:

Anlegen der Grammatik

Zunächst entwirft der Benutzer eine Grammatik die die gewünschte Sprache abbildet, mit der von ANTLR vorgegebenen Syntax, wie in 4.1 zu sehen.

Erzeugen des Java-Codes

Ist die Sprache mittels Grammatik soweit modelliert, übergibt man die Grammatik-Datei dem ANTLR Code-Generator. Dieser liest die Grammatik-Datei ein und erzeugt den zum verarbeiten der Sprache benötigten Java-Code. Der Java Code wird anschließend händisch, oder automatisiert in das eigene Java-Projekt eingefügt, welches die Sprache später verarbeiten soll. Der generierte Java-Code enthält unter Anderem einen Lexer der die Tokens bereit stellt und eine Basis-Klasse ”BaseListener” mit dessen Erweiterung man die eigentliche Verarbeitung

vornehmen kann.

Implementierung der Abarbeitung

Wie bereits im vorigen Schritt erwähnt kann man mit der Erweiterung der ”BaseListener” die eigentlicher Verarbeitung vornehmen. Es besteht ein Teil der Sprachverarbeitung darin, einen Parse-Baum abzuarbeiten. Genau dafür ist diese Klassen-Erweiterung zuständig. Daher wird diese Klasse im folgenden TreeWalker genannt, da sie über den Parse-Baum wandert und die entsprechenden Methoden zu den einzelnen Knoten im Baum aufruft. Es kann ein Methoden-Aufruf beim erreichen des Baum-Knotens als auch beim verlassen eines Knotens erfolgen. Was genau in diesem Methodenaufrufen geschieht, ist abhängig von Funktion und Einsatzzweck der Sprache.

Verarbeiten der Sprache

Nun kann zur Laufzeit der erstellten Software der Programmcode der neuen Sprache geladen werden. Eine zusätzliche Laufzeit-Bibliothek von ANTLR übernimmt hier sämtliche Zwischenschritte (Lexikalische Analyse und bauen des Parse-Baums) indem man lediglich die entsprechenden Funktionen aufruft. Letztendlich wird der Tree-Walker gestartet der die entsprechenden Methoden beim verarbeiten des Parse-Baum aufruft.

4.4.2 Sprachfunktionalitäten

```
1   attr float load_capacity (0, 1);
2   attr float power (0, 1);
3
4   model Hood                      "car_parts/Hood";
5   model Hood_Scoop                "car_parts/Hood_Scoop";
6
7   start Car;
8
9   Car ->          container(1, 1, 1, CarFront);
10
11  CarFront ->
12                  render(Hood),
13                  snap_container(1,1,1, Hood.hood_tire_left, TireLeft),
14                  snap_container(1,1,1, Hood.hood_tire_right, TireRight),
15                  power > 0.7 ? snap_container(1,1,1, Hood.hood_center, HoodScoop),
16                  container(1,1,1 CarFrontAssets),
17                  container(1,1,1 CarFrontSideAssets),
18                  load_capacity < 0.5 ? snap_container(1,1,1, Hood.hood_to_cabin, Cabin),
19                  load_capacity >= 0.5 ? snap_container(1,1,1, Hood.hood_to_cabin, Cabin_
20
21  HoodScoop ->        hood_raise_middle < 0.3 ? render(Hood_Scoop);
22  Spoiler ->          render(Spoiler);
```

Abbildung 4.2: Beispiel einer Regel-Datei (stark gekürzt)

Die Sprache lässt sich grob in drei Teile unterteilen: Attribute, Ressourcen, Regeln.

Attribute

Zunächst werden die öffentlichen Attribute für das Modell festgelegt. Diese sollen später dem Benutzer die Möglichkeit geben das Erstellung des Fahrzeugs und damit dessen Erscheinungsbild zu beeinflussen. Beginnend mit dem Schlüsselwort ”attr” folgt anschließend der Datentyp und der Bezeichner dieses Attributes. Abschließend kann optional noch geklammert ein minimaler und maximal zulässiger Wert folgen. Die Attribute “width“, “height“ und “length“ sind standardmäßig vorhanden und müssen nicht definiert werden.

Ressourcen

Ressourcen definieren und identifizieren die zur Verfügung stehenden Ressourcen in der für die Modell-Erzeugung. Dies sind in diesem Fall immer verschiedene Modell-Teile eines Fahrzeugs. Beginnend mit dem Schlüsselwort ”model” (für eine Textur-Unterstützung würde das Schlüsselwort ”texture” lauten), folgt ein Bezeichner und abschließend der relative Pfad zur Modell-Datei.

Regeln

Der Hauptteil, also die tatsächlichen Regeln der Sprache ähneln im Aufbau der einer Chomsky-Grammatik. Zunächst wird eine Start-Regel mit dem Schlüsselwort ”start” festgelegt.
Nun folgen die Regeldefinitionen. Regeln beginnen mit einem Regelnamen und einem Pfeil. Jede Regel besteht aus mindestens einem Funktionsaufruf durch Kommata getrennt. Ein Funktionsaufruf kann an eine Bedingung geknüpft sein. Diese wird, genau wie in vielen Sprachen, mit einem Konditional-Operator ”?” beschrieben. Ein Funktionsaufruf lässt sich auch quantifizieren indem, hinter dem Funktionsaufruf, eine Multiplikation stattfindet. Es gibt drei verschiedene Funktionen die aufgerufen werden können:

container() Erstellt ein Container-Objekt. Container-Objekte dienen, ähnlich dem ”div”-Boxmodell in HTML, der Positionierung von Inhalt in einem Raum. Die Dimensionen der Container werden als Parameter angegeben und können absolut sein, oder sich auf den Vater-Container beziehen. Die Positionierung der Container ist immer relativ zum Vater-Container und kann über die Ausrichtungsparameter innerhalb dessen verschoben werden. Eine Container-Funktion ruft immer eine Folgeregel auf. Alle in der Folgeregel erstellen Objekte sind somit Kind-Objekte dieses Containers. In Abbildung 4.3 sieht man einen Vater- und Kind-Container. Der Kind-Container hat 1/3 der Länge und 1/2 der Höhe des Vater Containers. Seine Ausrichtungsparameter lauten “Front“ und “Bottom“, was den Container nach vorne und unten innerhalb seines Vaters verschiebt. Um auf die Vater-Informationen zuzugreifen wird das Schlüsselwort “parent“ benötigt. “parent.height / 2“ ruft die Höhe des Vater-Containers ab und teil sie durch 2.

snap_container() Erstellt ebenfalls ein Container-Objekt, mit dem Unterschied, dass die Positionierung nicht über Ausrichtungs-Parameter erfolgt. Stattdessen wird der Funktion ein Bezugspunkt aus einer Modell-Ressource übergeben (z.B. "KotflügelRechts.RadkastenPosition"). Dadurch lassen sich zum Beispiel Räder innerhalb von Radkästen positionieren. Da die Position des Radkastens zur Zeit der Erstellung der Regel-Datei nicht bekannt ist, lässt sich somit der Bezugspunkt aus der Modell-Ressource zur Laufzeit bestimmen.
render() Erstellt ein Render-Objekt mit der im Parameter angegebenen Model-Ressource

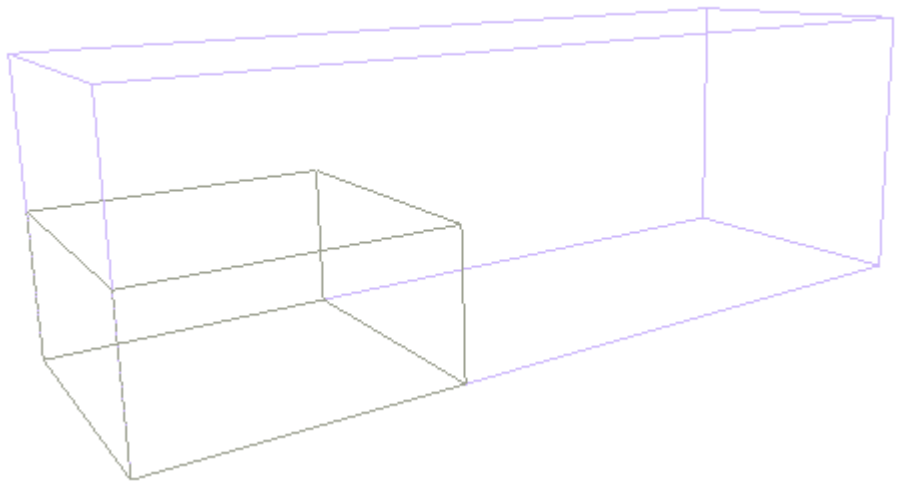

Abbildung 4.3: Beispiel eines geschachtelten Containers mit sichtbaren Containergrenzen, ohne Modelle.

4.4.3 Sprachkomponenten

Um die Sprachfunktionalitäten abzubilden werden eine Reihe von kleineren Sprachkomponenten benötigt. Im folgenden werden die Komponenten der Reihe nach mit zunehmender Komplexität erklärt. Dies ist wichtig um die spätere Verarbeitung in Java nachvollziehen zu können:
Variablen referenzieren einen numerischen Wert. Zur Unterscheidung zu Namen muss die Variable mit einem Kleinbuchstaben beginnen.
Namen sind schlicht Zeichenfolgen und repräsentieren keinen numerischen Wert. Ein Name muss mit einem Großbuchstaben beginnen, um sich von einer Variable zu unterscheiden.
Atome bilden Werte ab. Sie sind die kleinste mathematische Einheit. Ein Atom kann normalerweise entweder eine Konstante sein, oder eine Variable. Durch die benötigten Operationen

entsteht hier jedoch eine Besonderheit, sodass ein Atom auch ein geklammerter Addtions/Subtractions Ausdruck sein kann.

Arithmetische Ausdrücke sind einfache mathematische Operationen mit Atomen. Additions- und Subtraktions-, sowie Multiplikations- und Divisions-Ausdrücke lassen sich jeweils zusammenfassen.

Durch die Anforderung der Operatorpräzedenz, ergibt sich hier eine Besonderheit bei der das Atom, welches eigentlich die kleinste unteilbare Einheit darstellt, wiederum eine geklammerte Addition/Subtraktion sein kann. Diese entstehende Rekursion ist ein Standardschema zum Verarbeiten von Operatorpräzedenzen [LdR81, S. 89]. Die Grafik 4.4 verdeutlicht dieses Schema noch einmal anhand des Beispiels “1 / (2 + 3)“.

Logische Ausdrücke sind Ausdrücke aus denen boolscher Wert hervorgeht. Um die Verarbeitung möglichst einfach zu gestalten, muss eine Condition aus zwei Mathematischen Ausdrücken bestehen, die mit einem logischen Operator verbunden sind (gleich, ungleich, größer, kleiner usw.).

Parameter sind die Eingabewerte bei Funktionsaufrufen. Bei einem Parameter darf es sich entweder um einen arithmetischen Ausdruck, oder ein Namen handeln. Erstere werden für die Abmessungen der Container benötigt, Letztere für deren Ausrichtung und die aufzurufende Folge-Regel.

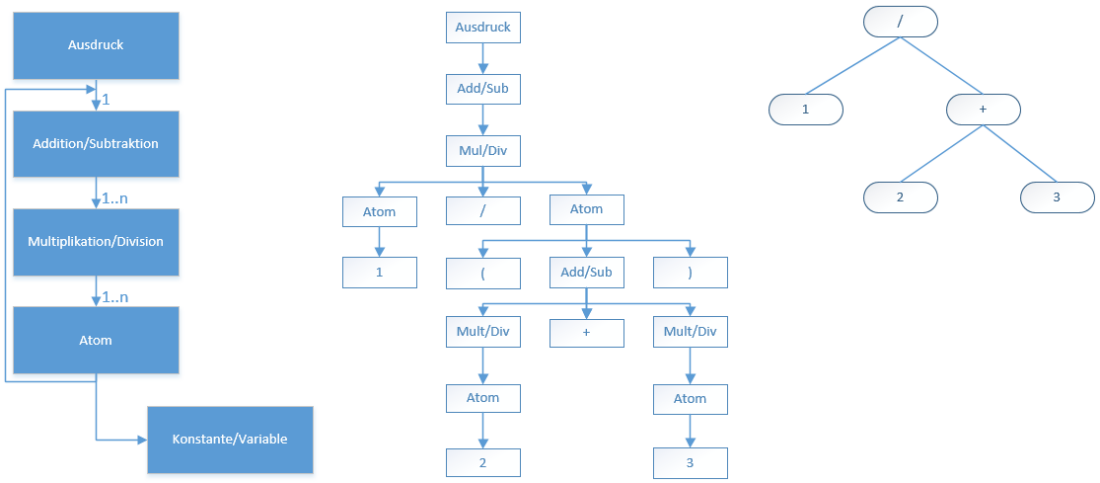

Abbildung 4.4: V. l. n. r. Die Operatorhierarchie mit den Multiplizitäten, der entstandene Parsebaum bei dem Ausdruck “1 / (2 + 3)“ und der abgeleitete Abstrakte Syntaxbaum

4.5 Entwurf des Generators

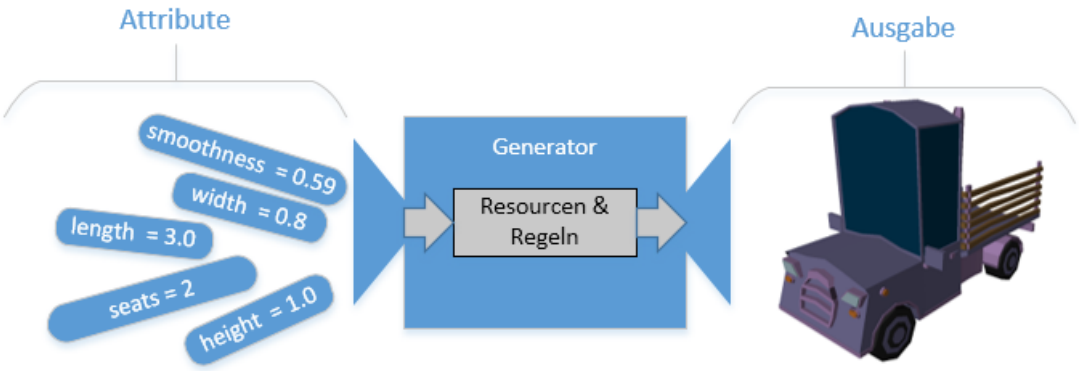

Abbildung 4.5: Arbeitsweise des Generators

Der Generator ist die zentrale Klasse dieses Projekts. Der Generator nimmt Attribute die der Benutzer, oder eine weitere Software belegt hat und verarbeitet diese unter Beachtung der Regeln und unter Verwendung der Ressourcen zu einem fertigen Fahrzeug-Modell. Die Regel-Datei wird jedoch erst zur Laufzeit von der Software eingelesen sodass das die Generator-Klasse zu Compile-Zeit ein leeres Skelett ist. Erst wenn die Regel-Datei zur Laufzeit eingelesen wurde, kann ein passender Generator erzeugt werden, der jene Regeln verarbeitet und Modelle erzeugt.
Dazu wird zunächst ein leeres ”Skelett” des Generators erzeugt und ein TreeWalker-Objekt befüllt dieses mit den Attributen, den Regeln und Ressourcen aus der Regeldatei. In Abb. 4.6 sind die einzelnen Schritte zu sehen die zum Erstellen des Generators benötigt werden.

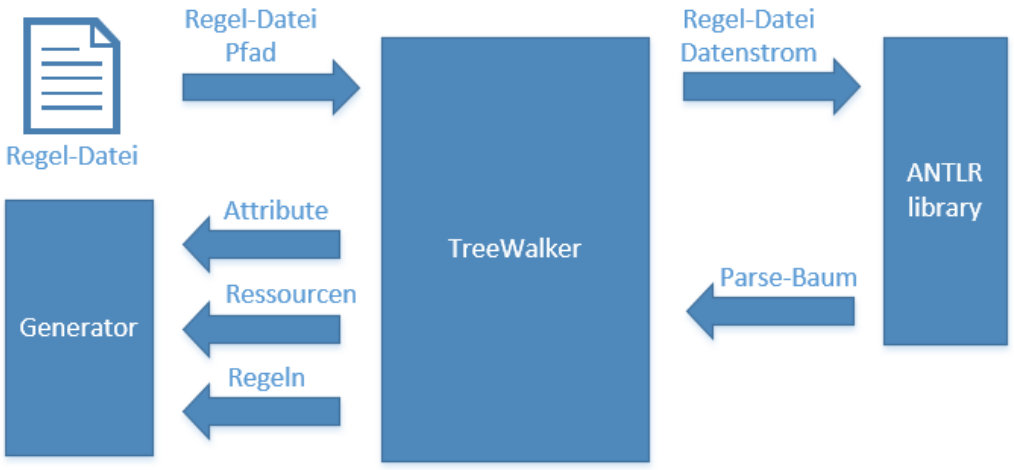

Abbildung 4.6: Verarbeitungsschritte der Regeldatei bis zum Erstellen des Generators

4.5.1 TreeWalker

Die Aufgabe des TreeWalkers ist es den Pfad der Regel-Datei entgegen zu nehmen und ein fertigen, funktionsfähigen Generator zurück zu liefern. Dazu nimmt der TreeWalker den übergebenen Pfad der Regeldatei und erstellt zu dieser ein FileStream welcher wiederum an die ANTLR Bibliothek übergeben wird. ANTLR Erzeugt daraus ein CharStream, der anschließend in die generierten Lexer und Parser übergeben werden. Letzteres erzeugt einen ParseTree der nun mit dem TreeWalker abgelaufen werden kann. Im letzten Initialisierungsschritt wird ein leerer Generator angelegt. Die Treewalker-Klasse fungiert hier als Schnittstelle nach Außen indem ihr ein Pfad zu der Regeldatei übergeben wird und ein fertiger Generator zurück geliefert wird.

Jetzt werden beim durchlaufen des ParseTrees für jeden Knoten die entsprechenden Methoden im TreeWalker aufgerufen. Zu jedem Aufruf enthält man den Kontext indem der Aufruf stattfindet. So lassen sich Informationen des aktuellen Knotens abrufen. In 4.7 wird gezeigt wie mittels der Grammatik-Regel ”start_entry” und einer Regel-Datei der Parse-Baum entsteht und für diesen Knoten das Kontext-Objekt erzeugt und belegt wird. Wird also die ”start_entry” Regel verarbeitet, wird die entsprechende Regel-Methode im TreeWalker aufgerufen. In diesem Beispiel muss der Inhalt aus REFERENCE_WORD (’Startrule”) dem Generator als Startregel-Namen übergeben werden. Damit ist der Knoten abgearbeitet. Da die Kinder des Knotens nur aus Terminalen bestehen, ist damit der gesamte Ast abgearbeitet.

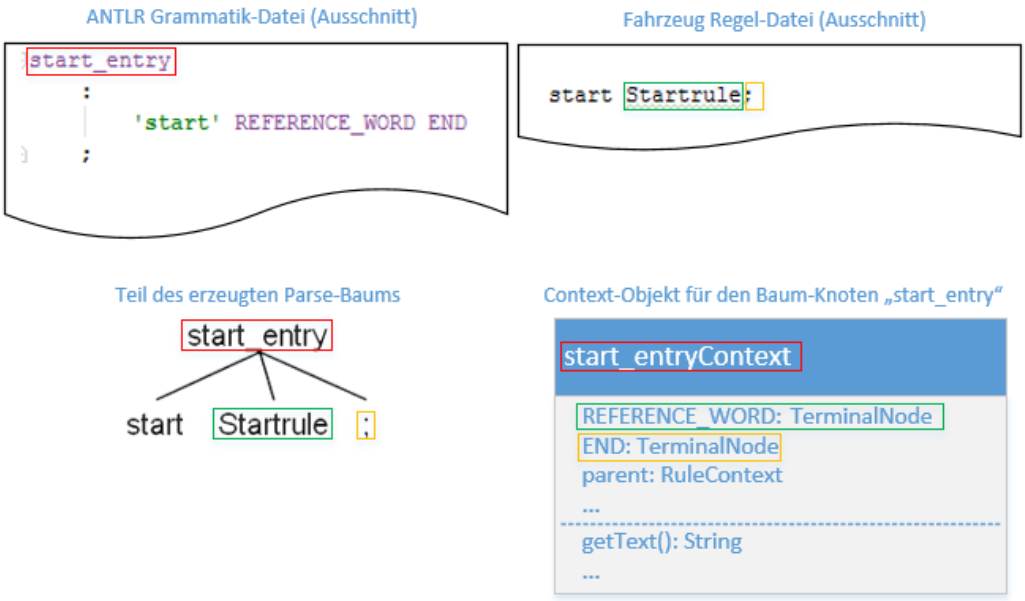

Abbildung 4.7: Erzeugung eines Kontext-Objekts auf Basis der Grammatik und Belegung auf Basis des Parse-Baums bzw. Regel-Datei

4.5.2 Übersetzung der Sprachteile in Java

Um die Regeln zur Laufzeit zu verarbeiten, müssen sie in Java-Code abgebildet werden. Dieser Schritt findet, wie bereits im Abschnitt 4.5.1 erläutert, in der TreeWalker Klasse statt. Um die einzelnen Sprachbestandteile in Java zu übersetzen, werden eine Reihe von Java-Klassen benötigt, die die Funktionalität der im Abschnitt 4.4.2 entworfenen Sprachkomponenten umsetzen. Viele Teile der dieser Sprachkomponenten lassen sich 1 Eins-zu-Eins in Java als Datenstrukturen abbilden. Wie beispielsweise das in 4.7 gezeigte Definieren der Startregel.

4.5.3 Containerbaum

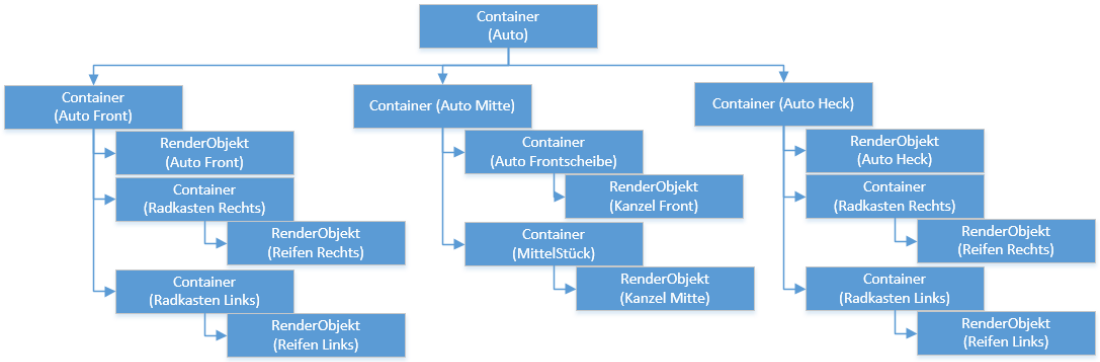

Abbildung 4.8: Beispiel für ein aus einem Regelsatz generierten Containerbaum

Das Ergebnis was der Generator liefert wird als eine Baumstruktur organisiert. Die Baumstruktur ist von Vorteil, da er sich später sehr leicht in eine Knotenstruktur für den Szenegraph übersetzen lässt und sehr gut für die rekursive Abarbeitung geeignet ist. Die Reihenfolge der Abarbeitung der Kinder ist beliebig, da es keine Abhängigkeiten unter den Kindern geben kann. Die Container sind keine sichtbaren Objekte, aus diesem Grund ist immer mindestens ein Render-Objekt in Form eines Autoteils am Ende einer Container-Hierarchie enthalten. In 4.8 ist hierzu ein Baum für ein sehr einfaches Fahrzeug zu sehen.

4.6 Prototyp

Im folgenden wird ein Prototyp entworfen der eine Regeldatei verarbeitet und mit Hilfe von Modellteilen ein Fahrzeug erzeugen kann. Die Funktionen des Prototyps sind die wesentlichen Teile beschränkt um den Entwurf kompakt zu halten.

4.6.1 Programmablauf

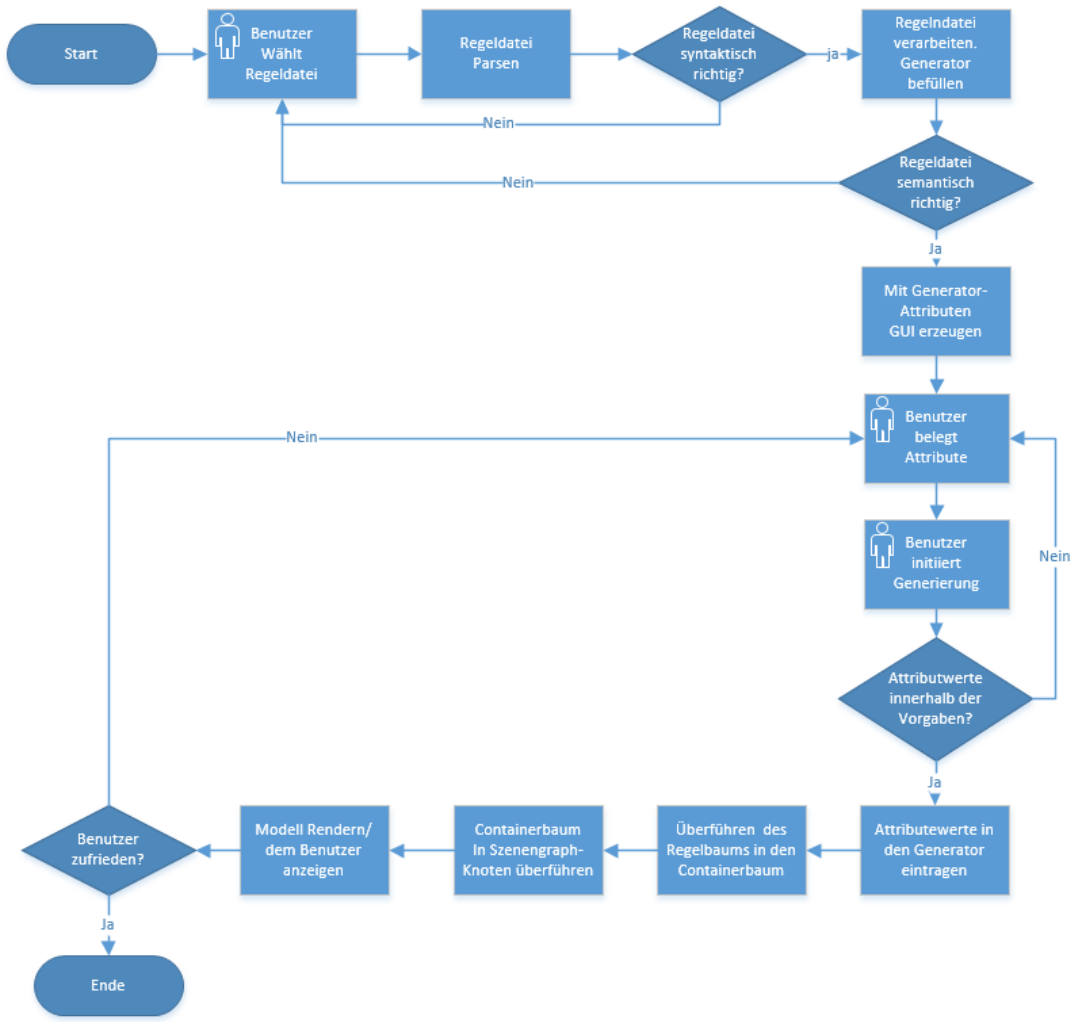

Abbildung 4.9: Ablaufdiagramm des Prototypen zur Erstellung eines Fahrzeugs

Der Programmablauf 4.9 zeigt die Nutzerinteraktion vom Start des Prototypen bis zum Erzeugen des finalen Fahrzeugmodells.
Zuerst wird dem Benutzer eine Oberfläche präsentiert, in dem er eine zuvor erstelle Regeldatei auswählt. Im Anschluss wird beim Einlesen und Parsen geprüft ob die Regeldatei syntaktisch korrekt ist. Treten Syntaxfehler auf, wird die Aktion abgebrochen und der Fehler der zum Abbruch führt wird ausgegeben. Ist die Regeldatei syntaktisch in Ordnung, wird der Generator mittels des Treewalkers mit Attributen, Ressourcen und Regeln befüllt. Zu dieser Zeit werden

auch semantische Fehler (z.B. überladene Funktion existiert nicht, falscher Datentyp) erkannt. Im Fehlerfall wird genauso verfahren, wie mit Syntaxfehlern. Ist dieser Vorgang erfolgreich abgeschlossen, kann nun eine Oberfläche aus den vom Generator zur Verfügung gestellten Attributen erzeugt werden und dem Benutzer präsentiert werden. Dieser kann nun die Attributwerte nach seinen Wünschen belegen. Ist der Benutzer fertig, kann er die Fahrzeuggenerierung anstoßen.
Zunächst wird geprüft, ob sich die Attribute innerhalb dem von der Regeldatei vorgegeben Rahmen befinden. Ist die nicht der Fall, kann der Benutzer diese Attributwerte nicht verwenden. Die Generierung wird abgebrochen. Sind die Attribute in Ordnung werden sie in den Generator eingetragen. Anschließend werden die Regeln abgearbeitet. Hier werden auch die Modellteile geladen und entsprechend den Attributen modifiziert. Die definierte Startregel ist der Beginn der Bearbeitung, der so entstehende Container-Objekt-Baum wird nun in den Generator eingetragen.
Im letzten Schritt wird dieser Containerbaum nun durchlaufen. Jeder Container stellt einen Szenegraph-Knoten dar und jedes, im einen Container enthaltene, Teil-Modell ein Render-Objekt. Dieser befüllte Szenegraph wird im Abschluss gerendert und dem Benutzer präsentiert. Ist der Benutzer unzufrieden mit dem Ergebnis kann er die Attribute weiter anpassen.

4.6.2 Prototyp Pakete

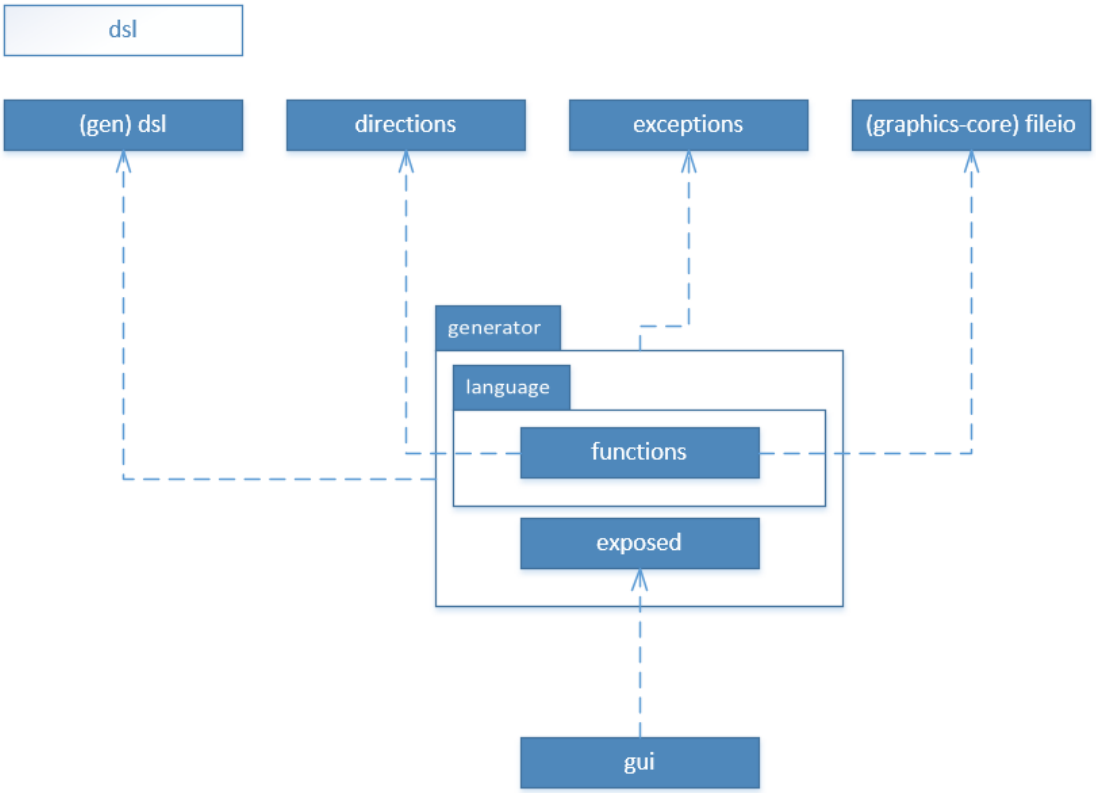

Abbildung 4.10: Paketdiagramm des Prototyps

Die Abhängigkeiten in 4.10 dargestellten Paketen in sind sehr simpel. Zur besseren Übersicht wurden die Unterpakete des *generator*-Pakets dargestellt. Pakete mit vorangestellten klammern sind nicht lokal.
“(gen) dsl“ ist das von ANTLR erstellte Paket das aus dem “dsl“-Paket erzeugt wurde. Das “dsl“-Paket enthält die Grammatik-Dateien, welche von ANTLR zur Erzeugung des “(gen) dsl“-Pakets benötigt werden. Aus diesem Grund besteht auch keine tatsächliche Abhängigkeit zu dem Rest der Pakete und ist deshalb farblich abgesetzt.
Das Paket “(graphics-core) fileio“ stellt den COLLADA-ModelLoader bereit. Um der Projektstruktur des CG-Frameworks zu folgen liegt das Paket bei den anderen Loadern im Modul “graphics core“.
Generator

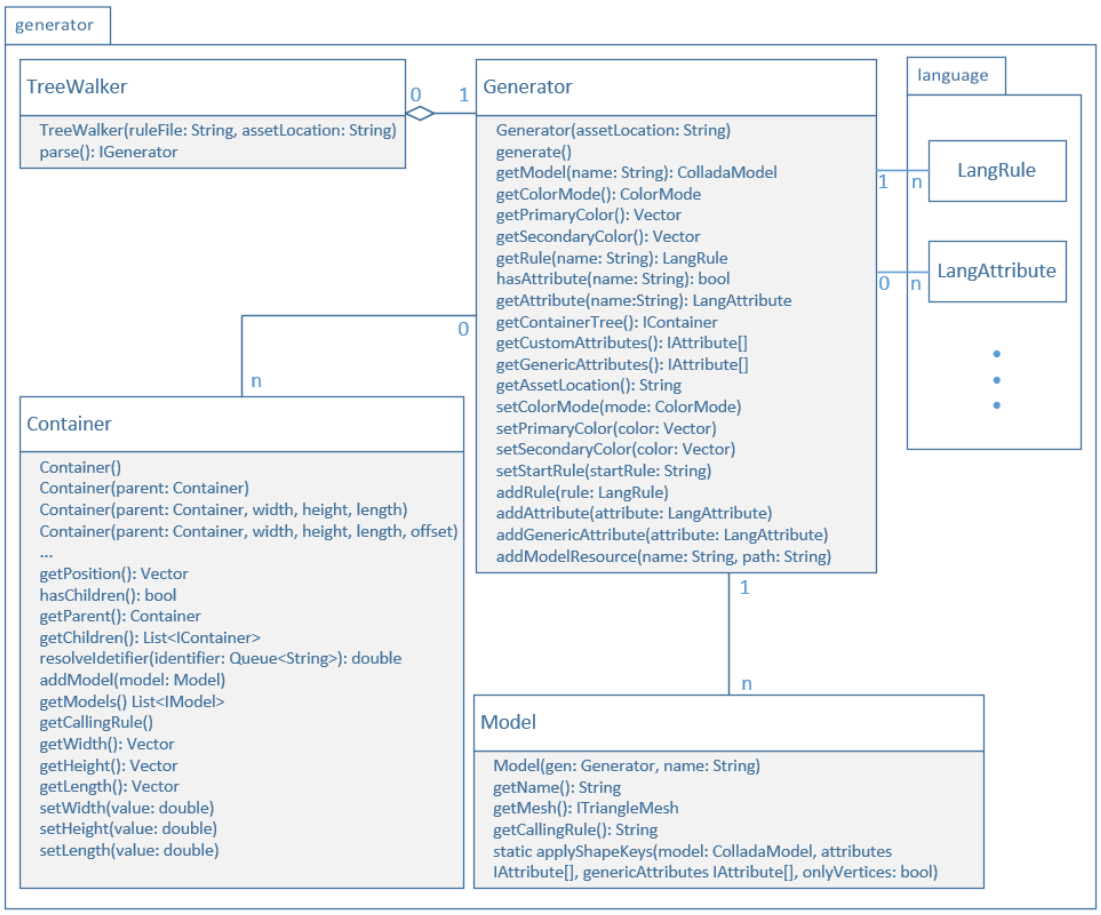

Abbildung 4.11: Klassendiagramm des “generator“ Paketes

Das *generator*-Paket enthält den gesamten Code der für die Verarbeitung der Regel-Dateien bis zum erstellen des Fahrzeugs notwendig ist. Die *TreeWalker*-Klasse erzeugt in ihrer *parse*-Methode ein neues Generator-Objekt. Dieses enthält alle Regeln und Attribute. Mit der *generate*-Methode wird das finale Fahrzeug-Model erstellt.

Language

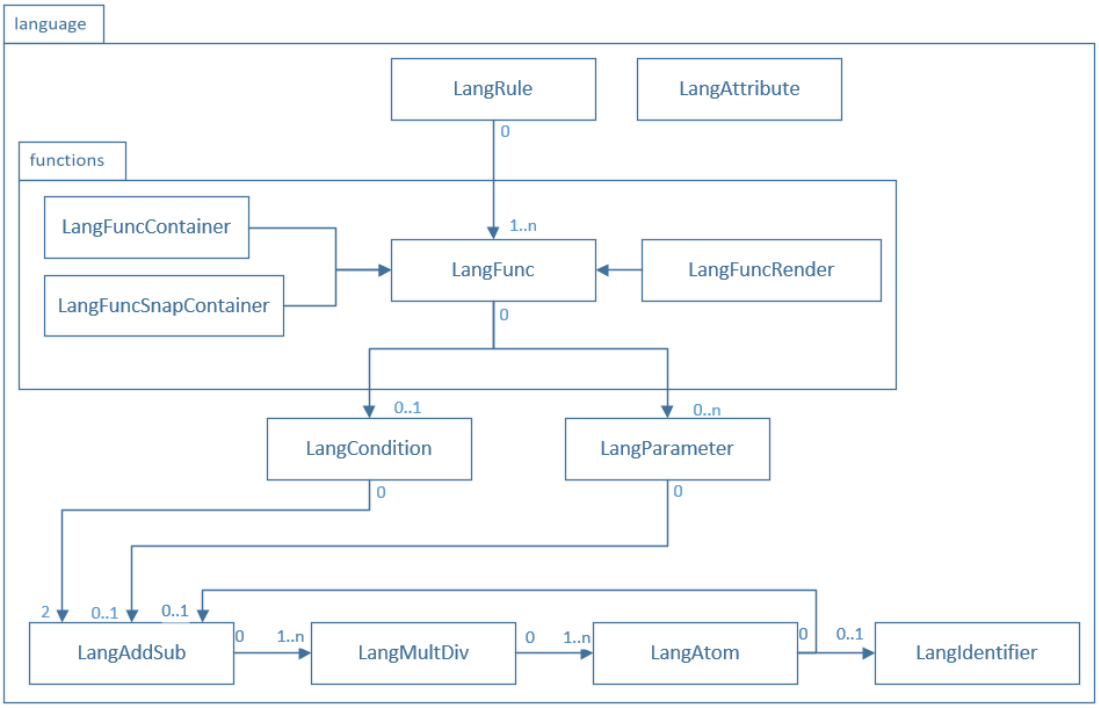

Abbildung 4.12: Klassendiagramm des “language“ Paketes und des Unterpakets “functions“

Das Language-Paket bildet die Sprachbestandteile in Form von Klassen ab. Innerhalb des *language*-Pakets befindet sich das *functions* Paket. In diesem stehen alle eingebauten Funktionen. Soll die Sprache um Funktionen erweitert werden, wird an dieser Stelle eine neue Klasse von der Basis-Klasse *LangFunc* abgeleitet. In der Basis-Klasse sind die *Condition*, die *Parameter* und die Methode zur Überprüfung der Parametertypen enthalten

Exposed

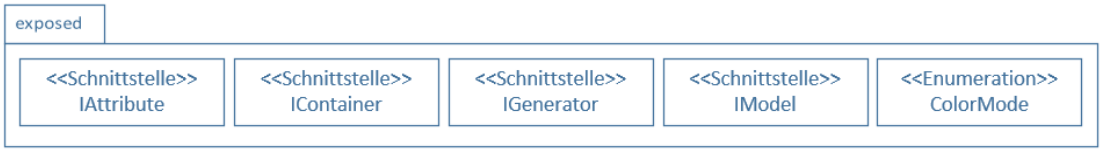

Abbildung 4.13: Klassendiagramm des “exposed“ Paketes

Das exposed-Paket innerhalb des generator-Paketes stellt die Schnittstelle nach außen bereit. Die Schnittstellen bieten die nach außen benötigten Eigenschaften und Methoden von Attributen, des Generators und Modellteilen. Diese Schnittstellen werden von der GUI verwendet und umfassen im Groben das Ändern der vom Generator exponierten Attribut-Werte, die Methode

zum Erzeugen und Auslesen eines neuen Container-Baums und dessen Modellen.

GUI

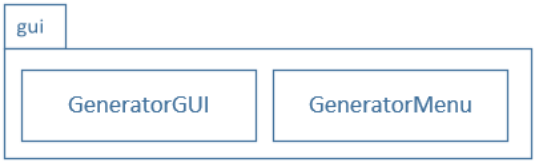

Abbildung 4.14: Klassendiagramm des “gui“ Paketes

Das GUI-Paket besteht aus zwei Klassen. Die Klasse Generator-Menü implementiert ein “Drop-Down“-Menü innerhalb des Hauptfensters des CG-Frameworks indem die Regel-Datei über einen Datei-Auswahl-Dialog geladen werden kann.
Die Zweite Klasse ist die Generator Oberfläche. Sie erzeugt aus dem mit der Regel-Datei erstellten Generator eine passende Oberfläche, die alle Attribute und Steuerungsmöglichkeiten enthält.

(gen) DSL

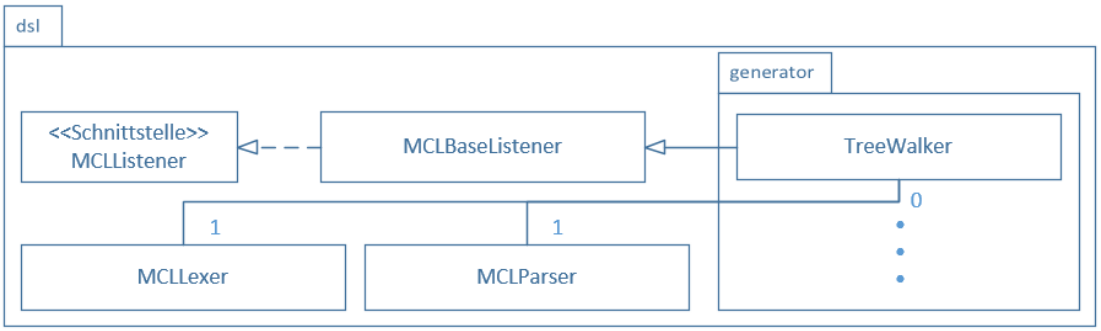

Abbildung 4.15: Klassendiagramm des von ANTLR erzeugten “dsl“ Paketes

Das “(gen) dsl“-Paket wird vollständig von ANTLR erstellt und ist in einer eigenen, vom händisch geschriebenen Programmcode abgetrennten, Paketstruktur. Das Paket umfasst die Lexer-Klasse MCLLexer, die, wie in 4.4 beschrieben, die Regeldatei in Tokens überführt, die Parser-Klasse MCL Parser die das Parsing durchführt und Tokens als Objekte bereitstellt und die Klasse BaseListener, die die Basisklasse für den Treewalker (4.5.1) implementiert. In der Parse Methode des TreeWalkers wird der Lexer und Parser aufgerufen.

Directions

Abbildung 4.16: Klassendiagramm des “directions“ Paketes

Das Paket enthält die Enums zur Verarbeitung der Ausrichtungsparameter für die Container-Struktur.

Exceptions

Abbildung 4.17: Klassendiagramm des “exceptions“ Paketes

Das Paket exceptions die Eigenen Exceptions, die beim Parsen der Regel-Datei auftreten können. Die Klasse MissingParentException ist für den Fall, dass das Schlüsselwort “parent“ verwendet wurde, obwohl kein Vater-Container existiert. Die Klasse InvalidParamCountException ist für den Fall, dass keine passende überladene Funktion für den Funktionsaufruf gefunden wurde.

(graphics-core) FileIO

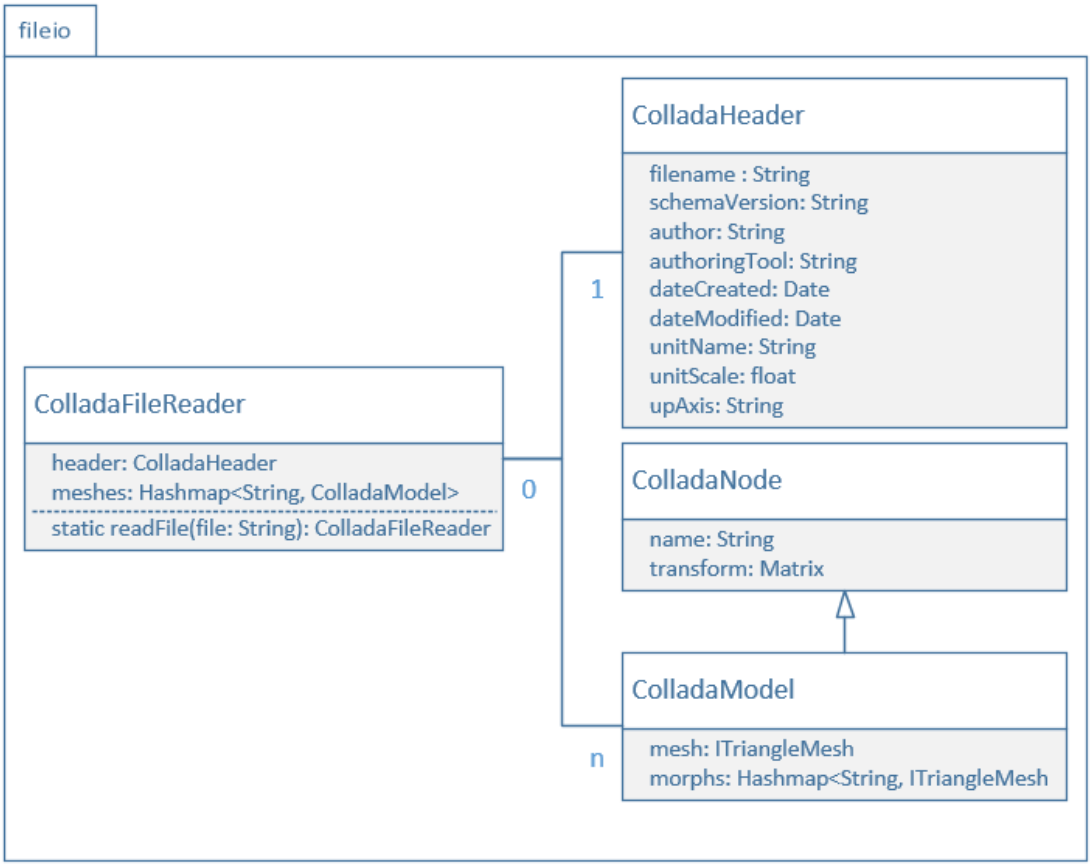

Abbildung 4.18: Aufbau des Model-Loader-Moduls innerhalb des FileIO-Pakets

Der COLLADA-Loader ist im eigentlichen Sinne kein eigenständiges Paket sondern Teil des FileIO-Paketes. Da für das Projekt aber ein COLLADA-Loader benötigt wird, wurde dies im FileIO-Paket des CGFrameworks ergänzt.
Wie 4.18 zeigt, besteht das COLLADA-Paket aus 4 Klassen. Das Einlesen und Parsen der Modell-Datei übernimmt die Klasse *ColladaFileLoader*. Beim Parsen erstellt der ColladaFileReader das Header-Objekt, welches die Header-Daten der COLLADA-Datei speichert. Anschließend werden die Nodes ausgelesen.
Für Empty-Nodes wird nur ein ColladaNode Objekt erstellt, welches den Namen und die Transformation enthält.
Für ein Geometrie-Node wird die abgeleitete Klasse ColladaModel verwendet, welches zusätzlich das Basis-Mesh und die Shape-Keys enthält. Die Shape-Keys sind über ihre Namen identifizierbar, die beim erstellen des Modells vergeben wurden.
Die Modelldaten werden all in den vom CG-Framework bereitgestellten Daten-Typ, Trian-

gleMesh übertragen. Die Klasse *ColladaFileLoader* enthält keinen öffentlichen Konstruktor, sondern instantiiert sich selbst über den statischen Aufruf von “readFile“.

5 Realisierung

In diesem Kapitel wird der im Rahmen dieser Arbeite entwickelte Prototyp vorgestellt. Der Prototyp setzt die im Kapitel 4 beschriebenen Konzepte um. Der Prototyp stellt die Mindestfunktionalität bereit, um die verwendeten Techniken und umgesetzten Konzepte anschaulich darzustellen und um einen Überblick über die Komplexität des Themas zu vermitteln.

Im Folgenden wird zunächst die Bedienung und die Aspekte der grafischen Oberfläche des Prototyps vorgestellt und erläutert. Anschließend wird im Detail auf die einzelnen Komponenten und ihre Arbeitsweise eingegangen.

5.1 Benutzeroberfläche

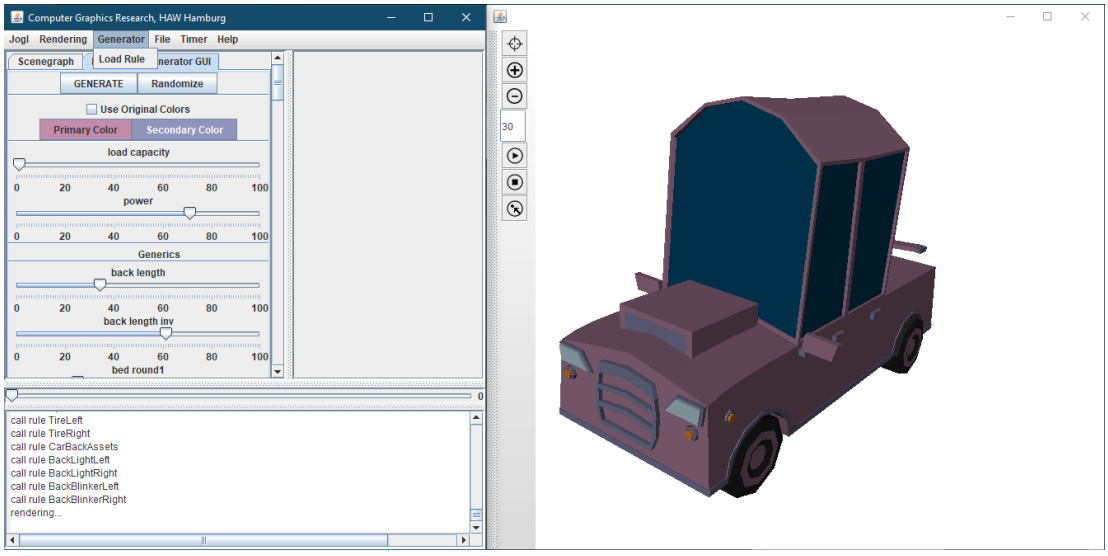

Abbildung 5.1: Die Benutzeroberfläche des Prototyps mit einem erzeugten Fahrzeug

Die grundsätzlichen Funktionalitäten der Oberfläche werden durch das CGFramework bereitgestellt. Dies umfasst wie in 5.1 zu sehen zwei Fenster. Das linke Fenster aus 5.1 zeigt das

Hauptfenster. Hier können eigene Benutzerschnittstellen eingefügt werden. Darüber hinaus gibt es einen Szenen-Graphen und einen Bereich für Textausgaben. Das Rechte Fenster aus 5.1 ist das Rendering-Fenster und stellt die Szene visuell dar. Am linken Rand sind diverse Funktionen um die Szene zu steuern.

5.1.1 Bedienung: Erstellen eines Fahrzeugs

Um ein neues Fahrzeug zu erstellen muss der Benutzer das “Drop-Down“-Menü “Generator“ öffnen. In diesem gibt es den Menüpunkt, um eine Regel-Datei zu laden. Dieser Punkt öffnet einen Datei-Auswahldialog, in dem der Benutzer eine zuvor erstellte Regeldatei auswählen kann.
Sofort nach der Auswahl wird die Regel-Datei verarbeitet und im Hauptfenster eine passende GUI Erzeugt, sofern die Regeldatei keine Fehler aufweist.

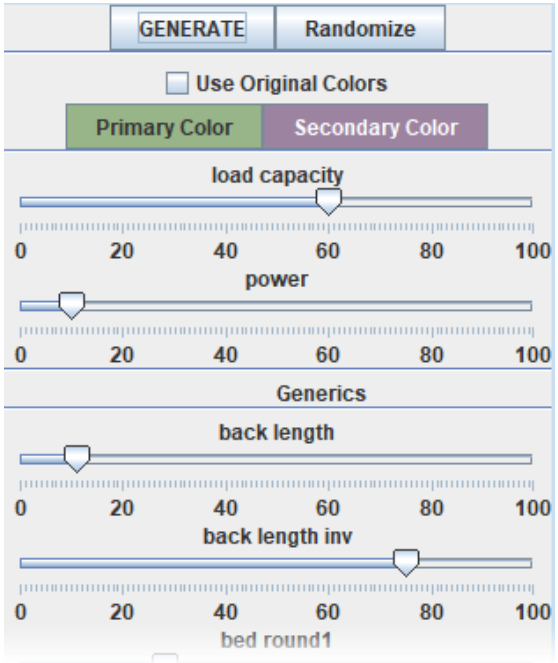

Abbildung 5.2: Eine auf Basis der Regeldatei erzeugte GUI (gekürzt)

Die GUI besteht aus 4 wesentlichen Punkten:

Generate und Randomize

Der oberste Teil der GUI 5.2 besteht aus zwei Schaltflächen. Ein Klick auf “Generate“ er-

zeugt ein neues Modell auf Basis der eingelesenen Regeln und den weiter unten getätigten Einstellungen. Die zweite Schaltfläche “Randomize“ belegt alle Einstellungen mit Zufallswerten.

Farben
In diesem Bereich kann der Farbeinstellungen für das Fahrzeugmodell treffen. Mittels einer Kontrollbox kann der Benutzer den Farbmodus bestimmten. Standardmäßig färbt der Generator alle Modellteile nach einem 2-Farb-System ein. Weiße Teile des Modells erhalten die erste Farbe. Schwarze teile des Modells erhalten die zweite Farbe. Ist ein Teil des Modells anders gefärbt bleibt diese Farbe erhalten.
Um die erste und zweite Farbe auswählen zu können, sind 2 Schaltflächen vorhanden, mit denen ein Farbauswahl-Dialog geöffnet wird. Zur besseren Übersicht werden die Schaltflächen entsprechend der Auswahl eingefärbt.

Wird “use original colors“ gesetzt, werden alle Farben aus den Modelldaten unverändert übernommen.

Benutzerdefinierte Attribute
In diesem Abschnitt werden alle Attribute angezeigt, welche in der Regel-Datei aufgeführt werden. Der in der Regeldatei angegebene Daten-Typ und der optionale Wertebereich bestimmen, ob es sich bei dem Steuer-Element um eine Textbox, einen Slider, oder eine Checkbox handelt.

Generische Attribute
Damit nicht jeder Shape-Key in die Regel-Datei als Attribut geschrieben werden muss, werden automatisch alle Shape-Keys als Generisches Attribut in der Oberfläche angezeigt. Somit lassen sich sehr feine Veränderungen an den Modellen vornehmen.

Der Benutzer nimmt nun die gewünschten Einstellungen in den Attributen und den Farben vor, oder benutzt die “Randomize“ Schaltfläche, um alle Attribut-Werte und Farben zufällig zu belegen.
Ist der Benutzer zufrieden mit den Werten wird mit der Schaltfläche “Generate“das Fahrzeug nach den Attributen und dem Regel-System der Regel-Datei zusammen gebaut und im Rendering-Fenster dargestellt.

5.2 Sprachverarbeitung

Die Treewalker-Klasse ist der Startpunkt für die Erzeugung von Fahrzeugen. Zunächst wird ihm ein Pfad zu einer Regeldatei übergeben und ein leerer Generator angelegt. Die ANTLR-Bibliothek liest die Regeldatei ein und übernimmt dabei die syntaktische Überprüfung. Tritt hier ein Fehler auf. Bricht der Vorgang ab und ANTLR teilt über eine entsprechende Exception mit, an welcher Stelle der Syntaxfehler vorlag. Ist die Regeldatei frei von Syntaxfehlern, beginnt ANTLR den erstellen Parsebaum abzuarbeiten, indem es an jedem Knotenpunkt im Parsebaum eine passende Methode im Treewalker aufruft. Wird zum Beispiel ein neues Attribut definiert, wird die *attribut*-Methode Aufgerufen. In dieser Methode kann dann über den *Context*-Parameter auf den Datentyp, den Namen und den Werten des Attributes zugegriffen werden. Es wird dann ein neues Attribut-Objekt erstellt und in einer HashMap im Generator gespeichert. Das Gleiche passiert für Ressourcen und Regeln.

Die Verarbeitung der Regeln ist noch etwas komplexer. Hier müssen Funktionen korrekt verarbeitet werden, die wiederum aus Bedingungen, Parametern und Multiplikatoren bestehen können.
Dazu werden zuerst alle Parameter nacheinander abgearbeitet. Besteht ein Parameter aus einer arithmetischen Operation, wird diese in Java abgebildet. Ist eine Aufrufbedingung vorhanden, wird der Boolsche-Ausdruck in Java abgebildet. Am Schluss wird ein Funktions-Objekt der Verwendeten Funktion erzeugt und die Parameter-Objekte als Liste übergeben. Die Aufrufbedingung und der Multiplikator wird darin ebenfalls gespeichert.
Das neue Regel-Objekt erhält anschließend die ganzen Funktions-Objekte und wird im Generator in eine HashMap mit dem Namen der Regel gespeichert.

Dies geschieht solange, bis der gesamte Parsebaum abgearbeitet ist und der nun bestückte Generator zurück gegeben werden kann.

5.3 Regelverarbeitung

Wird der Generator gestartet, wird zunächst der Name der Startregel verwendet um in der Regel-HashMap das richtige Regel-Objekt zu finden. Auf die Regel wird nun die *call()*-Methode aufgerufen und ein leeres *Container*-Objekt zurück gegeben. Ein *Container* ist ein Objekt, was weitere Container oder Modelle enthalten kann. Dies ist eine rekursive Methode, der Aufruf von *call()* genügt um das mitgegebene Containerobjekt mit der Modell- und Container-Struktur zu befüllen, wie in Kapitel 4.5.3 und Grafik 4.8 gezeigt zu befüllen.

Die *call()*-Methode des Regelobjektes ruft wiederum *call()*-Methode der in ihr enthaltenen Funktionen auf. Das Container-Objekt wird wieder übergeben. Nun findet eine semantische Überprüfung statt. Die Funktion muss die Anzahl der Parameter unterstützen und die Parameter müssen den richtigen Datentyp haben. Sind diese korrekt, werden die Parameter der Funktionen aufgelöst, indem *getValue()* auf das Parameter-Objekt aufgerufen wird. Wenn es sich um eine Container-Funktion gehandelt hat, wird nun ein neuer Container in den übergebenen Eingehängt. Enthält die Funktion als Parameter eine weitere Regel, wird diese mit dem neuen Container aufgerufen.

Dieser Prozess findet Rekursiv statt bis der Regelbaum abgearbeitet ist.

5.4 Parametrierbare Modellteile

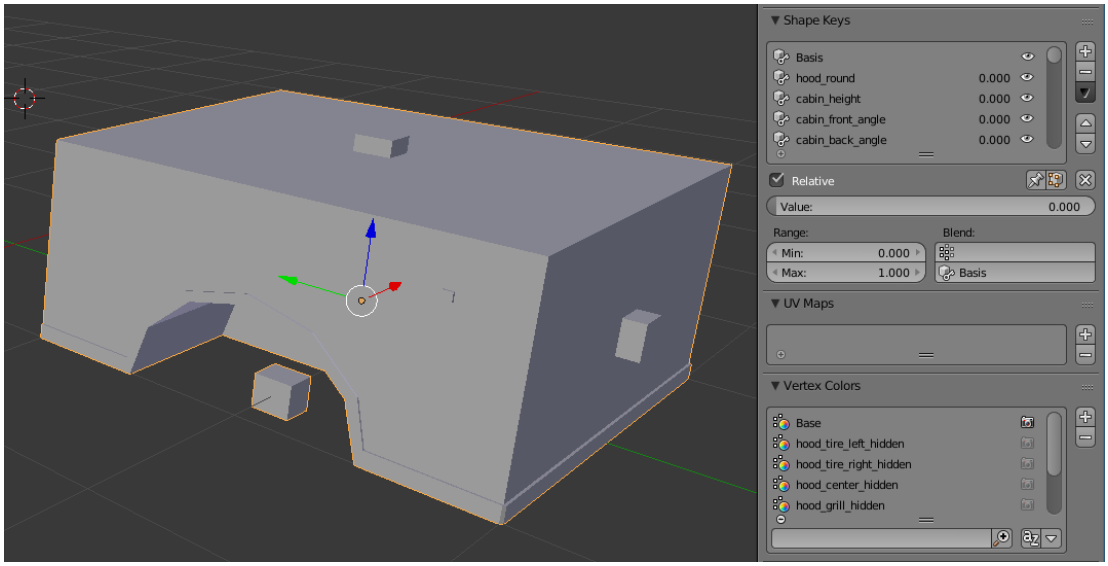

Abbildung 5.3: Screenshot aus der Modellierungssoftware Blender. Gezeigt ist eine Fahrzeugfront mit sichtbaren Ankerpunkten für weitere Teilmodelle

Um Modellteile für den Prototypen zu erstellen, ist einiges zu beachten und eine bestimmte Arbeitsfolge ein zu halten. Die Modellteile wurden mit Blender [Ble] erstellt.

Zunächst wird auf üblichem Wege ein neues Modellteil erstellt. Anschließend wird das Modell mit Ankerpunkten bestückt. Wie die Ankerpunkte aussehen ist egal. Der genaue Ankerpunkt

ergibt sich über den Mittelwert aller zum Ankerpunkt gehörigen Vertices. Somit ist es auch möglich, das Teile des Modells selber als Ankerpunkt fungieren.
Sind diese Schritte ausgeführt, darf sich die Mesh-Topologie nicht mehr verändern. Jetzt können die Shape-Keys angelegt werden.
Als letzten Schritt muss das Modell eingefärbt werden. Wird auf dem Model dir Farbe weiß oder Schwarz verwendet, können diese später durch den Prototyp als Primär- und Sekundärfarben ausgetauscht werden. Alle anderen Farben bleiben so erhalten. Durch die fehlende Unterstützung von Vertexgruppen im COLLADA-Exporter von Blender, muss für jeden Ankerpunkt ein weiterer Farblayer angelegt werden. Alle Vertices die zum Anker gehören müssen weiß sein, alle Restlichen schwarz. Dies begrenzt die Anzahl der möglichen Ankerpunkte auf 7 pro Teilmodel.
Danach ist das Modellteil vorbereitet für den Generator.

6 Evaluation

Dieses Kapitel beschäftigt sich mit der Evaluation des Prototypen. Dazu wird untersucht ob die in der Anforderungsanalyse gestellten Anforderungen vom Prototypen umgesetzt werden. Anschließend soll evaluiert werden ob sich der Prototyp in Echtzeit-Anwendungen wie in Spielen eingesetzt werden kann.

6.1 Evaluation der Anforderungen

6.1.1 Evaluation der Model Composition Language (MCL)

Im Folgenden wird die entworfene Model Composition Language in Bezug auf die Anforderungen untersucht. Die erste Anforderung aus dem Kapitel 3.1 lautet eine möglichst an der natürlichen Sprache angelehnte Sprachelemente, bzw. an bekannte Programmiersprachen übernommene Sprachelemente.
Die Attribute der Sprache werden mit *attr* abgekürzt anschließend erfolgt der Datentyp und der Name. Im Schluss folgt geklammert der Wertebereich. *attr* und die Namen der Datentypen sind bekannte Abkürzungen. *attr* findet in CSS Verwendung und *int*, *float* und *bool* sind Standardabkürzungen. Eine Schwäche in Anbetracht der Anforderungen ist der geklammerte Wertebereich, da hier nicht sofort klar ist wofür die Parameter stehen.
Die Ressourcen beginnen mit dem Schlüsselwort *model* und dem Modellteilnamen. Durch den Pfad als Letzter Parameter auf die Modelldatei ist offensichtlich, dass hier Modelteile definiert werden.
Die Regeln orientieren sich wie in der Arbeit von Herrn Watzl [Wat15] an der Chomsky-Grammatik. Dies zeichnet sich dadurch aus, dass auf der linken Seite vor dem Pfeil “->“ nur Non-Terminale stehen (der Regelname) und auf der rechten Seite des Pfeils entweder Non-Terminale (*Container*-Funktionen), oder Terminale (*model*-Funktionen), in beliebiger Reihenfolge stehen dürfen. Bedingungen vor Funktionsaufrufen werden mit einem ”?” geschrieben. Dies ist die übliche Schreibweise bei Programmiersprachen die eine Kurzform von “if“ Sprachkonstrukten haben. Funktionsaufruf-Quantifizierung erfolgt, indem hinter die Funktion einfach eine Multiplikation geschrieben wird. Es ist sofort ersichtlich, dass diese Funktion, dann ent-

sprechend oft aufgerufen wird.

Im zweiten Teil der Anforderungen von 3.1 ging es um die Speziellen Anforderungen an die Funktionen:
Funktionsaufrufbedingungen sind mit dem "?" Sprachkonstrukt gegeben. Funktionsaufruf-Quantifizierung ist vorhanden indem die Funktion einfach mit einem arithmetischen Ausdruck multipliziert wird. Mit dem "parent" Schlüsselwort kann auf die Abmessungen des übergeordneten Funktionsaufrufs zugegriffen werden, an allen Stellen an denen ein arithmetischer Ausdruck möglich ist. Damit ist eine geforderte Funktionsaufrufhierarchie vorhanden. *Container*-Funktionen müssen nicht zwangsläufig mit Ausrichtungsparametern versehen werden. Sie können auch einfach weg gelassen werden. Der Punkt der Funktionsüberladung ist damit auch gegeben. An allen Stellen an denen Werte übergeben werden, können arithmetische Ausdrücke verwendet werden. Die Operatorreihenfolge wird beachtet. Damit ist der letzte Punkt zur mathematischer Ausdrücke ebenfalls erbracht.

6.1.2 Evaluation der Modellteile

Für die Teilmodelle gab es zwei Anforderungen in Kapitel 3.2. Für die zu Testzwecken erstellen Teilmodelle wurde ein einheitliches Namensschema für die Shape-Keys verwendet. Somit konnten Shape-Keys angewendet werden, die mehrere Teile auf einmal betrafen, ohne dass es zu den in Abb. 3.1 gezeigten Problemen mit Mesh-Löchern kommt. Um unter Anderem Türgriffe, Scheinwerfer und Reifen korrekt an den jeweiligen Positionen zu befestigen, wurden farblich markierte Vertices verwendet. Damit ist die Anforderung für Mesh-Markierungen ebenfalls gegeben.

6.1.3 Evaluation der Benutzerschnittstelle

In den Anforderungen der Benutzerschnittstelle in Kapitel 3.4.1 steht, dass eine Regeldatei ausgewählt werden können soll. Mit dem in 5.1.1 beschriebenen Drop-Down Menü, ist dieser Punkt erfüllt. Im zweiten Punkt geht es darum, dass der Benutzer die Möglichkeit hat alle Variablen der Regel in der Oberfläche manipulieren können soll. Dies bedingt, das die Oberfläche je nach Regel-Datei dynamisch erzeugt werden muss. Der Prototyp erzeugt die Oberfläche dynamisch und gibt auch mit den in 5.1.1 erklärten *generischen Attributen* Zugriff auf die Variablen die nicht explizit in der Regel-Datei angegeben wurden.

6.2 Evaluation der Performance

In diesem Kapitel geht es um die Frage, wie schnell mäßig komplexe Regeldateien verarbeitet und daraus Fahrzeuge erzeugt werden können. Es soll evaluiert werden, ob sich der Prototyp auch als Middleware für Anwendungen eignet, in denen die Fahrzeuge zur Laufzeit erzeugt werden. Wichtige Merkmale sind hier benötigte Rechenzeit und Speicherbedarf. Zum einen wird getestet wie viel Ressourcen das Einlesen und Erzeugen des Generators benötigt, zum Anderen, wie viele Resourcen das Erzeugen eines Fahrzeug mit dem Generator benötigt.

6.2.1 Perfomance des Parsers

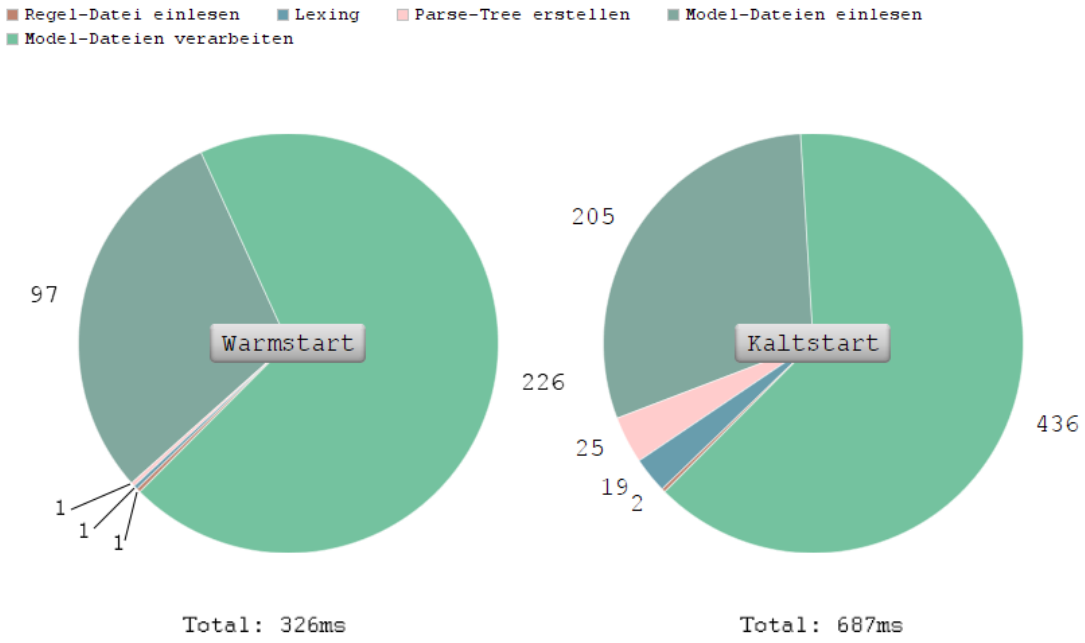

Abbildung 6.1: Dieses Diagramm zeigt die Zeit in Millisekunden, die der Parsing-Vorgang bei einer mittelgroßen Regel-Datei benötigt.

Der Parser liest die Regeldatei ein, benutzt den von ANTLR generierten Lexer und ParseTree und arbeitet anschließend den so erzeugten Parse-Baum ab. Dabei werden auch die Model-Dateien eingelesen und komplett verarbeitet um die Namen der Shape-Keys zu extrahieren, die später für die *Generischen Attribute* benötigt werden.
Die Grafik 6.1 zeigt die Zeit in Millisekunden, die benötigt wird, um die zuvor genannten

Arbeitsschritte auszuführen. Verwendet wurde eine mittelgroße Regeldatei, die 31 Teilmodelle referenziert und 43 Regeln beinhaltet. Die Zeit, die die einzelnen Arbeitsschritte im Vergleich zum gesamten Vorgang benötigen, wurden farblich hervorgehoben. Es wurde 2 Messungen gemacht. Einmal direkt nach dem Programmstart, während die erste Regeldatei eingelesen wurde (''Kaltstart", rechts) und einmal, als bereits eine Regeldatei eingelesen wurde und der Vorgang mit einer weiteren Regeldatei wiederholt wurde ("Warmstart", links). Mit diesen Informationen lassen sich 2 Dinge feststellen.
Die erste Beobachtung ist, dass der "Warmstart" nur noch knapp die Hälfte der Zeit benötigt. Dies liegt zum großen Teil an der Java-Virtual-Machine, bei der es sich um einen Just-in-time-Compiler handelt. So dass, im "Kaltstart" der Code noch nicht in hoch optimierter Form vorliegt. Zum Anderen daran, dass die Dateien bereits einmal abgerufen wurden und nun die Caching-Mechanismen der Speicherverwaltung in Kraft treten können.

Die Zweite Beobachtung ist, dass die eigentliche Verarbeitung der Regeldatei zu Java-Objekten nur sehr wenig Zeit in Anspruch nimmt. Den größten Anteil benötigt das Lesen und verarbeiten des XML-Datenstroms des COLLADA-File-Readers. Da dieser nicht auf Laufzeit optimiert wurde, sind diese Ergebnisse, auf Hinblick in Verwendung einer Echtzeitanwendung, sehr zufriedenstellend.

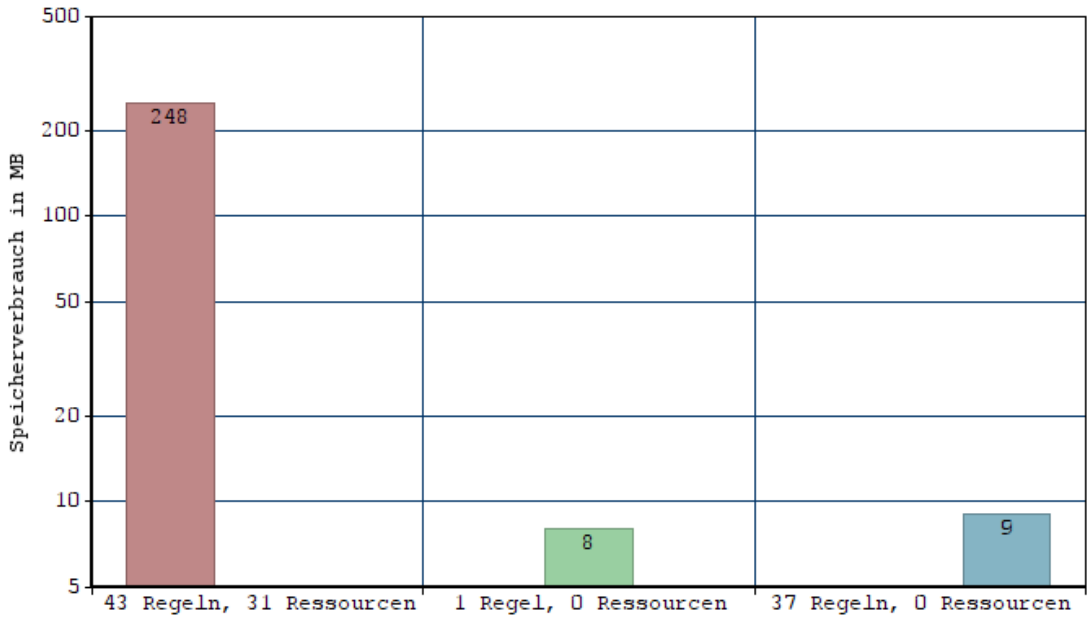

Abbildung 6.2: Der Speicherverbrauch verschiedener Regeldateien im Arbeitsspeicher

Der Speicherbedarf beträgt, für die oben genannte Regel-Datei, 248 Megabyte. Um zu Untersuchen, ob dieser Wert hauptsächlich durch die geladenen Ressourcen bestimmt wird, oder ob die Übersetzung der Regeln in Java ebenfalls signifikanten Speicherverbrauch vorweisen, wurde auch bei eine minimalen Regeldatei gemessen. In der Grafik 6.2 ist zu sehen, das eine fast leere Regeldatei lediglich 8MB Speicher verbraucht. Um zu erkennen, wie stark die Anzahl der Regeln (und damit die größe des Regelbaums) den Speicherverbrauch beeinflusst, wurde eine weitere Datei mit 37 Regeln aber ohne Ressourcen gemessen. Der Speicherverbrauch ist nur minimal gestiegen. Somit kann festgehalten werden, dass der limitierende Faktor an dieser Stelle die verwendeten Ressourcen sind und nicht die Komplexität der Regeldatei.

6.2.2 Perfomance des Generators

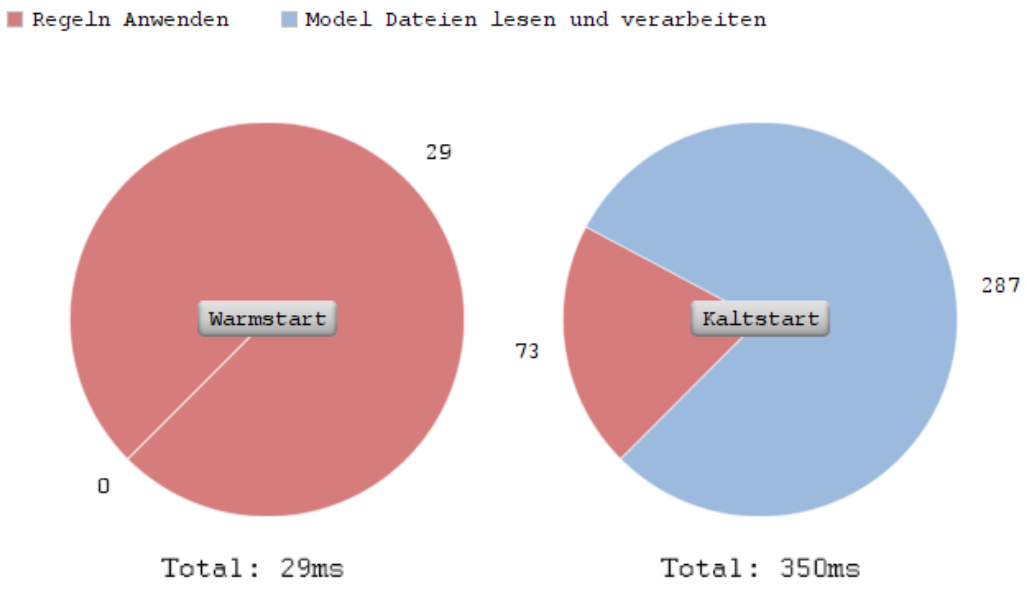

Abbildung 6.3: Dieses Diagramm zeigt die Zeit in Millisekunden, die der Generator-Vorgang bei einer mittelgroßen Regel-Datei benötigt.

Wird der Generator gestartet, wird der Regelbaum abgearbeitet. Dabei werden alle Modellteile geladen, positioniert und angepasst. Die Grafik 6.1 zeigt wie bereits im vorigen Kapitel 6.2.1 die Zeit in Millisekunden, die benötigt wird, um die zuvor genannten Arbeitsschritte auszuführen. Verwendet wurde eine mittelgroße Regeldatei, die 31 Teilmodelle referenziert und 43 Regeln beinhaltet. Die Zeit, die die einzelnen Arbeitsschritte im Vergleich zum gesamten Vorgang benötigen, wurden farblich hervorgehoben. Es wurden 2 Messungen gemacht. Einmal direkt nach dem Programmstart, während das erste Fahrzeug erzeugt wurde (''Kaltstart", rechts) und einmal als bereits ein Fahrzeug durch das selbe Generator Objekt erzeugt wurde ("Warmstart", links).
Wie bereits in Kapitel 6.2.1, zeigt sich hier wieder die Auswirkung der Java-Virtual-Machine, indem sich die Zeit der Codeausführung um über die Hälfte verkürzt, wenn der Code bereits zuvor einmal ausgeführt wurde. Die Zeit für "Model Dateien lesen und verarbeiten" ist im "Warmstart" bei 0 Millisekunden, da der Generator einen Cache für bereits geladene Modellteile besitzt. Es müssen also im weiteren Verlauf keine neuen Modellteile nachgeladen werden. Lediglich wenn durch das Regelsystem neue Modellteile hinzukommen, welche bisher noch nicht verwendet wurden, werden neue Teile vom Dateisystem in den Arbeitsspeicher geladen.

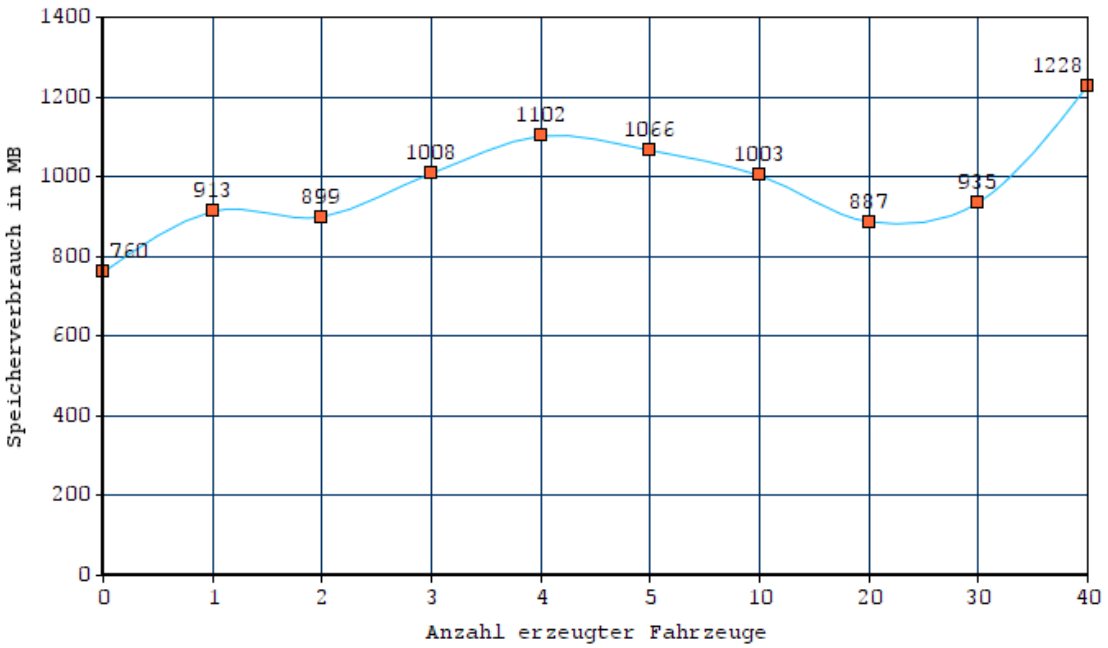

Abbildung 6.4: Der reservierte Speicher im Verlauf bei der sequenziellen Erzeugung von Fahrzeugen

Der reservierte Speicher schwankt über den Erzeugungsverlauf stark, wie man in der Grafik 6.4 ablesen kann. Dies hat damit zu tun, dass der Garbage Collector von Java den nicht mehr benötigten Speicher nicht sofort wieder frei gibt. Es zeigt sich das der Speicherverbrauch nach vielen Fahrzeugen dennoch ansteigt.
Ins Besondere dann, wenn schnell Fahrzeuge hintereinander erzeugt werden.

6.2.3 Bewertung der Perfomance

In den vorigen Kapiteln wurde die Perfomance auf hinblick der in Kapitel 3.4 definierten Anforderung untersucht. Dabei wurde festgestellt, das das Parsen der Regeldateien sehr schnell erfolgt, sofern der provisorische COLLADA-File-Reader nicht mit einbezogen wird. Für die Ansprüche einer Echtzeitanwendung sollte dies genügen, da in der Regel das Laden der Regeldateien sowieso weniger häufig erfolgt als das Abarbeiten der Regeln.
Der Generator kann im Schnitt aus einer mittelgroßen Regeldatei innerhalb von 29ms ein Fahrzeug zusammenstellen. Dies Entspricht ca. 34 Fahrzeugen pro Sekunde. Je nach Komplexität der Regeldateien können hier ganz andere Werte gemessen werden. Für das in 3.4 Beschriebene Verwendungsbeispiel als Middleware in einem Spiel, sollten 34 Fahrzeuge pro Sekunde eine ausreichende Menge sein. Der Speicherverbrauch stieg jedoch nach nach 40

erzeugten Fahrzeugen auf über 1,2GB. Dies ist eindeutig zu hoch, zumal der Speicherverbrauch nach noch mehr Fahrzeugen noch weiter einsteigen kann. Die Anforderung, dass das Konzept des Prototyps, grundsätzlich den Einsatz in einer Echtzeitanwendung erlaubt ist damit erfüllt, sofern noch eine Optimierung der Speicherverwaltung stattfindet, wie in 7.2 beschrieben.

Die Messungen wurden auf einem Intel Core i7 6700K mit 4GHz durchgeführt.

7 Fazit und Ausblick

7.1 Zusammenfassung

In dieser Arbeit wurde die prozedurale Erzeugung von Fahrzeugen behandelt. Es wurde hierfür eine domänenspezifische Sprache entworfen und mit Shape-Keys ein Modellierungs-Mechanik verwendet, die es erlaubt im Nachhinein einzelne Modellteile mittels Parametern zu verändern. Im Zusammenspiel von Sprache und parametrierbaren Modellen sollen so einfach Fahrzeugmodelle erzeugt werden können, welche den spezifischen Anforderungen entsprechen. Auch außerhalb der Erzeugung von Fahrzeugen lässt sich dieses Verfahren gut anwenden. In dieser Arbeit konnte nur ein sehr kleiner Teil der Möglichkeiten abgedeckt werden, die dieses Verfahren bietet. Mit größerer Auswahl unterschiedlicher Modellteile und komplexeren Regelsystemen, können auch weitaus differenziertere Fahrzeugmodelle erzeugt werden.
Der Vorteil in diesem Verfahren liegt die Zeitersparnis und damit auch ein enormes Kostenersparnispotential, wenn große Mengen an unterschiedlichen Modellen benötigt werden. Diese alle von Hand zu modellieren wäre deutlich aufwändiger und kostspieliger. Allerdings wächst auch der Anspruch an das Team der Modellierer. Zum Einen durch die Shape-Keys, die ein etwas anderes Vorgehen beim Modellieren benötigen. Zum Anderen wird nun ein Spezialist benötigt der die domänenspezifische Sprache beherrscht.

Im praktischen Teil der Arbeit geht es um die Evaluierung, ob das Zusammenspiel aus domänenspezifischer Sprache und parametrierbaren Modellteilen eine praktikable Methode ist, um eine Variation von Objekten, in diesem Fall Fahrzeugen, zu Erzeugen. Dazu wurde ein Prototyp entworfen, welcher das das CG-Framework von Professor Dr. Jenke verwendet, um die Benutzerschnittstelle zu implementieren. Der Prototyp soll dabei einen dazu einen groben Überblick vermitteln, welche Möglichkeiten die Verwendung von einer Domänenspezifischen Sprache und parametrierbaren Modellen bietet. Außerdem wird überprüft, ob sich das System für einen Einsatz in einer Echtzeit-Anwendung eignet, in der beispielsweise laufen neue Fahrzeuge benötigt werden, indem überprüft wird, wie viel Leistung es benötigt.

Zunächst wurde bei der domänenspezifischen Sprache versucht diese im JSON Format zu entwerfen. Somit hätte man direkt den vorhandenen JSON-Parser verwenden können. Ein eigener Parser wäre nicht nötig gewesen. Es stellte sich schnell heraus das JSON kein geeignetes Format für diesen Zweck darstellte. Zusammenhänge und Funktionsaufrufe ließen sich nicht übersichtlich abbilden. Stattdessen wurde eine eigene Syntax entwickelt. Diese verwendet, neben anderen Sprachteilen, die Chomsky-Grammatik als Herzstück für die Regelverarbeitung, wie sie auch Herr Watzl in seiner Arbeit zum Erstellen von Gebäuden verwendet hat [Wat15].

Da eine eigens Entwickelte Sprache zum Einsatz kam, wurde ein Lexer und Parser benötigt. Dazu wurde der Parser-Generator ANTLR verwendet um den benötigten Parser-Code automatisch zu erzeugen.
Im nächsten Schritt wurde das Generator-Paket geschrieben, was die eigentliche Sprachverarbeitung vornimmt. In diesem Paket befindet sich die TreeWalker Klasse, welche den von ANTLER erzeugten Parse-Baum verarbeitet. Die *Generator* Klasse stellt das fertige Objekt bereit, mit dem nachher neue Modelle erzeugt werden können. In dem Unterpaket *language* wurden alle Klassen angelegt, welche nötig sind um die entworfene Sprache in Java ab zu bilden.

Als Nächstes wurde ein Geeignetes Model-Format ausgewählt. Leider stellte sich erst später heraus, dass die verwendete Modellierungs-Software Teile des Model-Formats nicht richtig umsetzte. So konnten statt Vertexgruppen nur Farbgruppen exportiert werden. Dies bedeutete, dass maximal 7 Ankerpunkte pro Modellteil definiert werden können, an denen über die Sprache andere Teile angeheftet werden können. Da dies jedoch ausreichend war, wurde das Modell-Format beibehalten.
Da das CG-Framework das Modellformat nicht unterstützt hat, musste ein entsprechender Importer geschrieben werden, der das Modellformat einliest und an das CG-Framework weiter gibt. Da das CG-Framework wurde im gleichen Zug um Vertex-Farbfunktionalitäten erweitert.

Der Letzte Teil stellte die Benutzeroberfläche dar. Das CG-Framework wurde um einige Funktionen erweitert um eine bessere Unterstützung für höhere Auflösungen und beleuchtete Darstellung von eingefärbten Modellen zu erreichen.

7.2 Ausblick

Der hier vorgestellte Entwurf und die Umsetzung als Prototyp sind soweit vereinfacht worden, damit sie in die im Rahmen dieser Arbeit einen vertretbaren Arbeitsumpfang aufweisen. Es

gibt noch sehr viele Stellen an denen mit vergleichsweise wenig aufwand die Arbeit erweitert werden kann.

Die wünschenswerteste Erweiterung wäre eine Exportfunktion. Derzeit können die Modelle nur in der Visualisierung betrachtet werden. Da die Modelldaten bereits in der CG-Framework Datenstruktur vorliegen, ist es sehr leicht eine Exportfunktion nach zu liefern, sofern das zu schreibende Modellformat indexbasierte Vertex-Farben und Vertex-Normalen beherrscht.

Viel Verbesserungspotential besteht in der Sprache. Derzeit gibt es es nur einfach Bedingte Aufrufe, somit muss für einen alternativen Pfad der Regelverarbeitung die Bedingung ein weiteres Mal negiert aufgeschrieben werden. Auch hat sich gezeigt, dass eine Skalierung nur selten Benötigt wird, aber diese muss momentan für jeden Container angegeben werden. Die Sprache beherrscht bereits Funktionsüberladung. Würde man die Skalierungsparameter an das Ende der Parameterliste setzen, könnte man Methoden diese optional machen. Die Regel-Dateien werden mit jedem Modellteil größer. Mit einer *include* Funktionalität könnte man ganze Unterkategorien von Modellteilen in andere Regel-Dateien auslagern.

Ein sehr komplexe Erweiterung wäre die Unterstützung für Texturen. Diese müssen, um unschöne Textur-Verzerrungen an den Modellteilen zu vermeiden, prozedural erzeugt werden [RTB+92]. Beispiel für prozedurale Texturierung sind die Werkzeuge der Firma Allegorithmic [All]. Hier können sehr komplexe Materialien prozedural auf Modelle angewendet werden. Die Struktur, bzw die Form des Modells kann dabei die Texturierung beeinflussen. Zum Beispiel erscheint eine Rost-Textur nur an Kanten mit mindestens 45° Winkeln. Dieses Konzept ließe sich auch gut für diese Arbeit nutzen, um Beispielsweise ein Attribut *Fahrzeug Alter* einzuführen, welches prozedural Abnutzungserscheinungen über die Textur steuert.

Ein Weiterer sehr komplexer Punkt sind Rotationen. Derzeit gibt es kein Konzept für Rotationen und Spiegelungen, weder in der domänenspezifischen Sprache noch in der Softwarearchitektur. Für Fahrzeuge sind Rotationen weniger wichtig, da hier wenige Teile in rotierter Form vorliegen müssen. Für andere Anwendungsgebiete, zum Beispiel bei organischen Formen wie Pflanzen, können Rotationen aber sehr nützlich sein. Einfacher hingegen ist es Objekte räumlich zu spiegeln.

Die Speicherverwaltung hat noch einiges an Verbesserungspotential um in einer Echtzeitanwendung als Middleware neben einer Hauptanwendung laufen zu können.Eine Verbesserung

wäre es, weniger Objekte zu erzeugen bzw. bereits erzeugte Objekte wieder zu verwenden, damit sich weniger Arbeit beim Garbage-Collector sammelt.

Diese Arbeit hat einen Überblick über prozedurale Modellierung mittels einer domänenspezifischen Sprache und Shape-Keys gegeben. Dazu wurde ein Prototyp entwickelt und seine Eigenschaften evaluiert um ein besseres Verständnis über die Möglichkeiten dieser Vorgehensweise zu erfahren. Diese Grundlagen ermöglichen es, dass weitere Arbeiten auf dieser aufbauen.

Literaturverzeichnis

[All] Allegorithmic website. https://www.allegorithmic.com/. Letzter Zugriff: 2017-09-18.

[ANTa] ANTLR Eclipse Plugin github projekt. https://github.com/antlr4ide/antlr4ide. Letzter Zugriff: 2016-12-02.

[ANTb] ANTLR IntelliJ Plugin jetbrains website. http://plugins.jetbrains.com/plugin/7358?pr=. Letzter Zugriff: 2016-12-02.

[ANTc] ANTLR website. http://www.antlr.org/. Letzter Zugriff: 2016-12-02.

[AvD00] P. Klint und J. Visser A. v. Deursen. Domain-specific languages: An annotated bibliography. Juni 2000.

[Ble] Blender Foundation website. https://www.blender.org/. Letzter Zugriff: 2017-09-18.

[BV99] Volker Blanz and Thomas Vetter. A morphable model for the synthesis of 3d faces. pages 187–194, 1999.

[Fow05] M. Fowler. Language workbenches: The killer-app for domain specific languages? Juni 2005.

[Fre16] S. Freiberg. Procedural generation of content in video games. März 2016.

[Hel] Hello Games helloo games website. http://www.hellogames.org/. Letzter Zugriff: 2017-01-14.

[HKM15] Haibin Huang, Evangelos Kalogerakis, and Benjamin Marlin. Analysis and synthesis of 3d shape families via deep-learned generative models of surfaces. *Computer Graphics Forum*, 34(5), 2015.

[KCKK12] Evangelos Kalogerakis, Siddhartha Chaudhuri, Daphne Koller, and Vladlen Koltun. A probabilistic model for component-based shape synthesis. *ACM Trans. Graph.*, 31(4):55:1–55:11, July 2012.

[LdR81] Wilf R. LaLonde and Jim des Rivieres. Handling operator precedence in arithmetic expressions with tree transformations. *ACM Trans. Program. Lang. Syst.*, 3(1):83–103, January 1981.

[Liu09] C. Liu. An analysis of the current and future state of 3d facial animation techniques and systems. Dezember 2009.

[Mer07] Paul Merrell. Example-based model synthesis. pages 105–112, 2007.

[NMS] No Man's Sky no man's sky website. http://www.no-mans-sky.com/. Letzter Zugriff: 2017-01-14.

[Pap14] Adam Papousek. Example-based 3d-model synthesis. 2014.

[RTB+92] John Rhoades, Greg Turk, Andrew Bell, Andrei State, Ulrich Neumann, and Amitabh Varshney. Real-time procedural textures. pages 95–100, 1992.

[shi] Ship Generator website. http://ship.shapewright.com/. Letzter Zugriff: 2017-02-18.

[vee] Dolf Veenvliet website. http://www.macouno.com. Letzter Zugriff: 2017-02-18.

[Wat15] Thorben Watzl. Prozedurale modellierung von gebäuden anhand von mustern in form von shape-grammar. Mai 2015.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 29.09.2017 David Asmuth