Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Thorben Watzl

Prozedurale Modellierung von Gebäuden anhand von Mustern in Form von Shape-Grammar

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Thorben Watzl

Prozedurale Modellierung von Gebäuden anhand von Mustern in Form von Shape-Grammar

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: 27. Mai 2015

Thorben Watzl

Thema der Arbeit

Prozedurale Modellierung von Gebäuden anhand von Mustern in Form von Shape-Grammar

Stichworte

Prozedurale Modellierung, Gebäude Generierung, Stadt Generierung, Shape-Grammar, Prozedurale Gebäude Generierung

Kurzzusammenfassung

Es werden immer öfters dreidimensionale Städte benötigt. Die Städte werden dabei meist von Designern modelliert. Dies ist sehr zeitaufwendig und kostspielig. Doch die Städte können auch automatisch generiert werden. Dies ist möglich mit der Shape-Grammar. In dieser Arbeit geht es darum eine Grundlage im Bereich der Shape-Grammar zu schaffen. Dabei wird ein Prototyp entwickelt, mit dessen Hilfe die Shape-Grammar evaluiert wird.

Thorben Watzl

Title of the paper

Procedural modeling of buildings based on patterns in the form of Shape-Grammar

Keywords

Procedural modeling, Generate building, Generate City, Shape-Grammar, Procedural generation of building

Abstract

Three dimensional cities are needed more and more often. Designers model these cities most of the time. This costs a lot of money and time. But the cities can also be generated automatically. This is possible with Shape-Grammar. This thesis intends to create a foundation within the area of shape grammar. A prototype will be developed to evaluate the Shape-Grammar.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

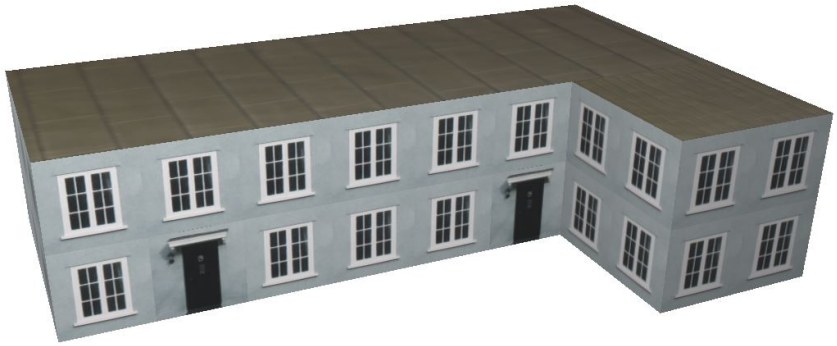

Abbildung 1.1: Mit Prototyp erzeugtes Gebäude Texturen [tex15a, Car15]

Für viele Anwendungsfälle ist es erforderlich, dass eine dreidimensionale Stadt benötigt wird. Diese sind entweder für Simulationen oder für moderne Landkarten auf den Smartphones und Navigationssystemen erforderlich. Da jedoch die Gebäude Ungleichheiten beim Aussehen und in der Struktur aufweisen, ist das modellieren von Städten eine langwierige Angelegenheit. Für die Landkarten fahren Fahrzeuge durch die Städte, um die Gebäude zu scannen. Die gescannten Daten werden durch einen Algorithmus bearbeitet, um Muster zu erkennen. Es muss die Problematik gelöst werden, aus den Daten ein Gebäude zu generieren. Bei einer Stadt bedarf es einer großen Varianz an Gebäuden. Dies wird bevorzugt von Grafikern modelliert und ist eine langwierige Angelegenheit.
Die Problematik soll mit dem Shape-Grammar System gelöst werden. Dieses System ermöglicht es, aus einer vordefinierten Grammatik und Parametern, ein Gebäude zu generieren.
Dies würde bedeuten, dass die gescannten Daten in Form der Grammatik und Parametern gespeichert werden. Diese können von dem System verwendet werden, um die

Gebäude zu generieren.

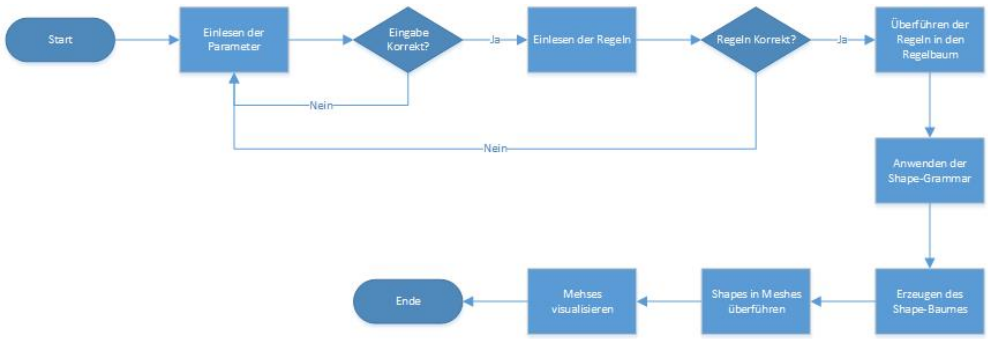

Abbildung 1.2: Diagramm zur Visualisierung des Ablaufes der Gebäudegenerierung

Es existieren weitere Einsatzmöglichkeiten, um das System zu verwenden. Eine Möglichkeit wäre die mittlerweile etablierte Spielebranche. Dort spielt es eine große Rolle, dass in einer Stadt viele verschiedene Gebäude vorkommen. Da bei Spielen die Umgebung ein wichtiger Faktor für das Spielerlebnis ist. Daher soll ein System entwickelt werden, an dem das Shape-Grammar System evaluiert werden kann.

1.2 Zielsetzung

In dieser Arbeit soll ein System entwickelt werden, mit dem es möglich sein soll, einfach ein Teil einer Stadt prozedural anhand einer Shape-Grammar zu generieren. Dabei soll eine gute Varianz an Gebäuden vorhanden sein. Die Muster und Informationen von Gebäuden sollen dabei extern geladen werden. Dies soll das Erweitern der Gebäude vereinfachen.
Dabei soll eine geeignete Form gefunden werden, um die Gebäude extern darzustellen und einzulesen. Erst einmal soll nur ein Teil einer Stadt generiert werden, da es nicht das Ziel ist, eine geeignete Datenarchitektur und Verwaltung für eine große Menge an Daten zu finden.

Es soll in dieser Arbeit eine Grundlage geschaffen werden, auf der weitere Arbeiten aufbauen können.

1.3 Aufbau der Arbeit

Im zweiten Kapitel „Grundlagen“ werden wichtige Begriffe und Inhalte erläutert. Dazu gehört die prozedurale Modellierung, dessen Verständnis benötigt wird, um die weiteren Themen zu verstehen. Anschließend wird das L-System behandelt. Danach wird die Shape-Grammar vorgestellt, welches das Kernsystem in dieser Arbeit ist. Schließlich folgt der aktuelle Stand in der Gebäudegenerierung mit dem Shape-Grammar System.
Im dritten Kapitel „Anforderungsanalyse“ werden die Anforderungen an die Software analysiert und dazu passende Lösungen vorgestellt.
Im vierten Kapitel „Entwurf“ wird der Entwurf für die Software vorgestellt und erläutert. Dabei wird auf die in der Analyse erwähnten Lösungen eingegangen und behandelt. In dem Kapitel werden zudem die verwendeten Datenstrukturen erläutert und der Entwurf des Prototyps vorgestellt. Schließlich wird die Vorgehensweise der Shape-Grammar anhand des Systems erläutert.
Im fünften Kapitel „Realisierung“ geht es um die Realisierung der Software. Zuerst wird die Realisierung des Prototyps behandelt. Des Weiteren wird die Unterteilung in die Pakete behandelt. Schließlich wird die Testdurchführung erläutert.
Im sechsten Kapitel „Evaluation“ wird die Software und die Arbeit anhand von Zeitanalyse und Performance bewertet. Außerdem wird die Shape-Grammar anhand der Software evaluiert.
Im siebten Kapitel wird die Arbeit zusammengefasst. Anschließend wird aus der Arbeit ein Fazit gezogen. Schließlich wird eine Aussicht des Themas gegeben und inwieweit die Arbeit erweitert werden kann.

2 Grundlagen

2.1 Überblick

In diesem Kapitel werden relevante Begriffe erläutert, die für diese Arbeit benötigt werden. Dazu zählen das L-System und die Methodik der prozeduralen Modellierung. Diese werden kurz erläutert, da die Begriffe benötigt werden, um die Shape-Grammar zu erklären. Am Ende des Kapitels wird der aktuelle Stand der Gebäudegenerierung vorgestellt.

2.2 Prozedurale Modellierung

Dr. Ganster und Dr. Klein erläutern die prozedurale Modellierung auf der Computergrafik Website der Uni Bonn wie folgt. Es handelt sich bei prozeduraler Modellierung um Methoden aus der Computergrafik, die automatisch aus Regelmengen dreidimensionale Objekte und Texturen erzeugen. Des Weiteren können Regeln erstellt werden, um Objekte automatisch detailreicher zu gestalten. Formen solcher Methoden zur Verwaltung der Regelmengen sind das L-System, fraktale und generative Modellierung. Entweder sind die Regelmengen fest bestimmt oder können über Parameter konfiguriert werden. Es ist möglich, dass die Regelmenge von dem Evaluierungsmechanismus getrennt ist. Bei der standardisierten Modellierung steht meist die individuelle Gestaltung einzelner dreidimensionaler Objekte im Vordergrund. Bei der prozeduralen Modellierung wird der Fokus auf die Modellierung und Verwaltung einer großen Menge von gleichartigen dreidimensionalen Objekten gerichtet. Dies bedeutet, dass komplexe und zu stark individuelle Objekte momentan nur sehr schwer durch prozedurale Modellierung generiert werden können. Diese Methode wird häufig für die Modellierung von Landschaften verwendet. Doch sie findet immer mehr Anwendungsgebiete, beispielsweise die prozedurale Modellierung von Bäumen [GK15].

2.3 L-System

Das L-System wird in dem Dokument der FU Berlin von Astrid Könnecke wie folgt beschrieben. Das Lindenmayer-System oder kurz L-System ist eine mathematische Formalisierung, die von dem theoretischen Biologen Aristid Lindenmayer 1968 vorgeschlagen wurde. Dies wird als axiomatische Theorie zur biologischen Entwicklung vorgestellt. In der Computergrafik wurde es zuerst zur Modellierung von Fraktalen und Pflanzen verwendet. Bei dem L-System handelt es sich um ein Ersetzungssystem. Dabei geht es im Grunde um das Ersetzten einfacher Objekte durch komplexere Objekte mit Hilfe von definierten Regeln. Bei der Chomsky Grammatik läuft das Ersetzungsverfahren sequenziell ab. Bei dem L-System hingegen werden die Regeln gleichzeitig und parallel auf allen Symbolen angewendet [Koe15].

Ein Beispiel des L-Systems

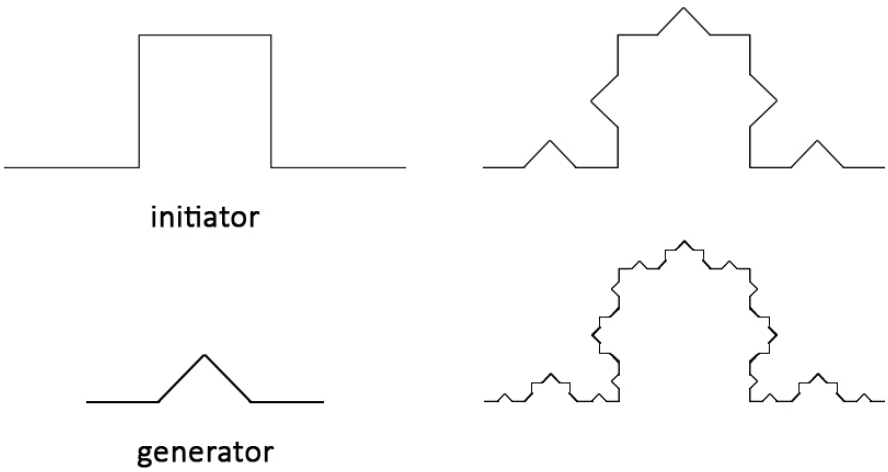

Abbildung 2.1: Ein Beispiel des L-Systems nachempfunden von Przemyslaw Prusinkiewicz und Aristid Lindenmayer[PL90]

In diesem Beispiel sieht man, dass die Kanten vom Initiator durch den Generator ersetzt werden. Im ersten Schritt werden die Kanten mit Spitzen ersetzt. Schon nach dem zweiten Schritt erkennt man, dass dem Initiator viele Spitzen hinzugefügt werden. Dies zeigt, wie man aus einem einfachen Objekt ein komplexeres Objekt erschafft. Dies geschieht durch eine simple Regel, dem Generator.

Definition

Bei einem kontextfreien L-System (0L-System) handelt es sich um ein 3-Tupel. Dieses besteht aus dem Alphabet, dem Eingabewort und eine endliche Menge an Übergangsregeln. Dabei ist zu beachten, dass auf dem Eingabewort die Übergangsregeln n-Mal angewendet werden. Dies ist jedoch nur möglich, wenn die Regeln auf dem Eingabewort angewendet werden können. Zum Beispiel, wenn für das Symbol S eine Regel $S \rightarrow abc$ existiert, wird S durch abc ersetzt [Koe15].

2.4 Einführung in die Shape-Grammar

Die Arbeit von Georg Stiny befasste sich als Erstes mit dem Thema der Shape-Grammar [SG72]. Diese Arbeit ist die Grundlage weiterer Arbeiten, wie die von Wonka [WWSR03].

In der Arbeit von Stiny wird eine Sprache für zweidimensionale Formen definiert. Hierbei geht es um das ersetzten von Formen durch andere Formen. Durch die Regeln wird definieren, welche Formen ersetzt werden. Diese Regeln bestimmen, in welcher Größe und in welchem Aussehen die Form in einer Zeichenfläche gezeichnet wird.
Die Shape-Grammar sieht der Chomsky-Grammatiken ähnlich und ist eine erweiterte Form des L-Systems. Der Unterschied zwischen den beiden Grammatiken liegt dabei, dass bei der Shape-Grammar das Alphabet aus Formen und nicht wie bei der Chomsky-Grammatik aus Zeichen besteht.
Bei der Shape-Grammar werden n-dimensionale Formen erzeugt. Bei der Chomsky-Grammatik hingegen werden Zeichenketten erzeugt.
Stiny erweitert seine Arbeit, indem er die Shape-Grammar von der zweidimensionalen auf die dreidimensionale Ebene erweitert. Somit ist es mit der Shape-Grammar möglich dreidimensionale Objekte, wie Gebäude, Autos und andere Objekte, zu erstellen [SG72].

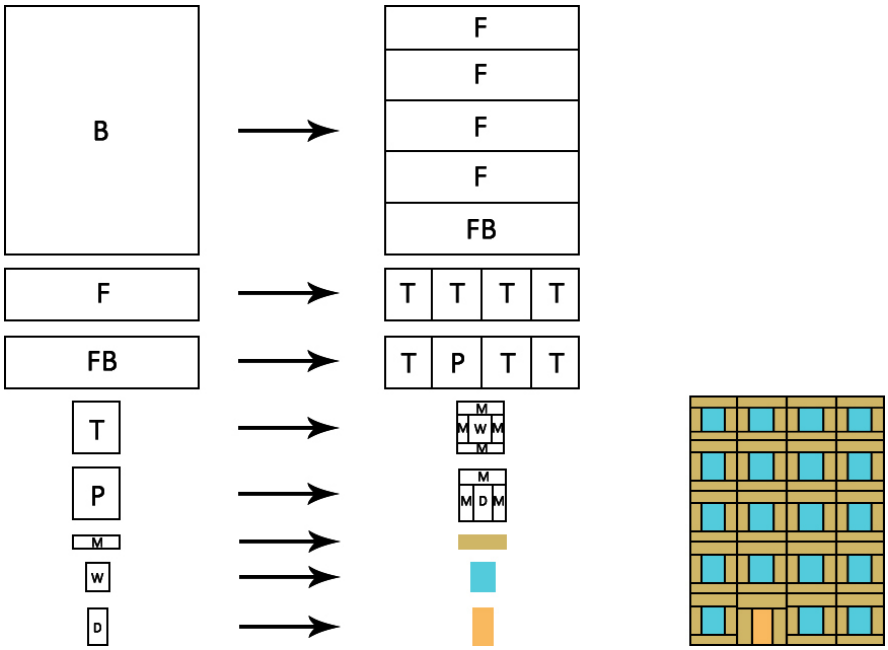

Abbildung 2.2: Ein Shape-Grammar Beispiel nachempfunden von Mark Dokter[Dok14]

Auf der Grafik ist eine simple Sammlung an Ersetzungsregeln aufgezeichnet. Die erste Regel besagt, dass die große Fläche B in mehrere kleinere F-Teile und einem FB-Teil unterteilt wird. Die darauf folgende Regel besagt, das F wiederum durch T unterteilt wird. Die dritte Regel besagt, dass FB-durch drei T-Teile und einem P-Teil ersetzt wird. Das T-Teil wird dabei durch mehrere M-Teile und einem W-Teil unterteilt. P wiederum ist durch mehrere M-Teile und einem D-Teil unterteilt. M wird durch die Wand ersetzt, W durch das Fenster und D durch die Tür. So wird durch die Regeln eine Hausfassade erzeugt.

2.5 Aktueller Stand von Gebäudegenerierung

Es gibt viele Arbeiten, die sich mit der Gebäudegenerierung beschäftigen. Viele von ihnen beschäftigen sich mit der Generierung anhand von mathematischen Grundlagen und der einfachen prozeduralen Modellierung. Auch die Shape-Grammar wird in einigen Arbeiten verwendet, doch die meisten Arbeiten konzentrieren sich dabei auf die Stadt Modellierung und bauen nur grundlegende Funktionen für die Shape-Grammar ein. Dabei sind einige Arbeiten schon sehr weit.

CityEngine von Esri [Esr15] ist eine vollständige Anwendung und kann somit schwer mit anderer Software kombiniert und verwendet werden, auch wenn die Software sehr gute Gebäude generiert. Daher ist es schwer, mit diesen Arbeiten weitere wissenschaftliche Aspekte zu untersuchen. Doch die Informationen und deren Erfahrungen sind dabei sehr wertvoll, um die Shape-Grammar zu implementieren und zu untersuchen.

3 Anforderungsanalyse

3.1 Grafik-Engine

Die Anforderung an die Grafik-Engine ist in erster Linie, einen simplen Einstieg in die Materie zu erhalten. Es soll einfach sein, dreidimensionale Objekte darzustellen. Dafür müssen Datenstrukturen existieren, welche das Verwalten und Bearbeiten von Dreiecksnetzen ermöglichen. Dieses Dreiecksnetz sollte dann an die Engine übergeben werden und diese sorgt für die Visualisierung. Es muss eine Verwaltungsmöglichkeit für Texturen vorhanden sein. Dabei soll eine Möglichkeit gegeben sein, die Textur einfach auf ein dreidimensionales Objekt zu platzieren. Die Engine soll die komplexen „Low Level“ Aufgaben übernehmen, wie zum Beispiel das Berechnen der Normalen. Dies vereinfacht den Start mit dem eigentlichen Thema der Arbeit.

3.2 Eigenschaften der Shape-Grammar

Die Shape-Grammar ist ein Ersetzungssystem basierend auf Formen. Daher ist dieses System optimal um ein Gebäude aus Regeln zu generieren.

Laut Definition ist die Shape-Grammar ein Quadrupel (VT, VM, R, I). Dabei ist VT die Menge der Terminalsymbole, VM die Menge der Variablen. R ist dabei die Menge der Regeln und I die Startform. Die Startform besteht dabei aus Elementen aus VT^* und ein Element aus VM. VT^* entspricht dabei aneinandergereihte Elemente aus VT, bei denen die Größe und Ausrichtung beliebig verändert sein darf. Gestartet wird mit der Startform I, auf der die Regeln R so lange angewendet werden, bis keine mehr passen. Die von SG erzeugte Sprache L(G) kann endliche oder unendliche Mengen von endlichen Formen darstellen [SG72].

In dieser Arbeit ist ein Quader die Startform. Durch die Startregel wird der Quader in sechs Seiten geteilt. Dadurch wird aus der dreidimensionalen eine zweidimensionale Problematik. Diese werden dann in Stockwerke unterteilt. Die Stockwerke werden wiederum in Fenster oder Tür Abschnitte unterteilt.

3.3 Anforderung an die formale Grammatik

In erster Linie soll die formale Grammatik für den Menschen leicht verständlich sein. Des Weiteren soll sich die formale Grammatik an der Chomsky Hierarchie orientieren. Genauer soll die formale Grammatik der kontextfreien Grammatik entsprechen. Die Grammatik soll es einfach machen, die Gebäude formal zu beschreiben. Dabei ist es wichtig, dass die Grammatik Funktionen zur Verfügung stellt, damit die Shape-Grammar angewendet werden kann. Diese Funktionen müssen dabei die Eigenschaften der Shape-Grammar widerspiegeln. Sie müssen es erlauben, die Split-Eigenschaften zu definieren. Dabei ist es wichtig, die Möglichkeit zu bieten, dass die Split-Größen sowie die Ersetzungen definiert werden können.

Für das Setzen von Texturen soll eine Funktion zur Verfügung gesellt werden, in der angegeben wird, in welchem Verzeichnis die Texturen liegen. Für die Veränderung der Form soll die *extrude* Funktion definiert werden. Dabei muss die Möglichkeit bestehen, die Länge zu bestimmen. Auch hier muss es möglich sein, für die Flächen Ersetzungsregeln zu definieren. Damit ein gewisser Zufallsfaktor hinzukommt, muss eine Möglichkeit gegeben werden, mit der es möglich ist, Ersetzungsregeln eine Wahrscheinlichkeit zu geben.

3.4 Anforderung Gebäudegenerierung

Die Anforderung an die Gebäudegenerierung sollte wie folgt sein. Mit ein paar einfachen Klicks soll es möglich sein, ein Gebäude zu generieren. Dies soll mit möglichst wenigen Eingaben geschehen. Diese Eingaben sollen Höhe, Länge und Breite sein. Des Weiteren die Auswahl, welches Gebäude generiert werden soll. Das Einlesen der Regeln und durchführen der Shape-Grammar soll dabei in maximal zwei Sekunden durchgeführt werden, damit der Anwender nicht lange auf das Ergebnis warten muss. Dabei ist zu beachten, dass ein überdimensioniertes Gebäude mit hoher Komplexität länger dauern kann. Die Visualisierung sollte je nach Komplexität des Gebäudes nicht länger als zwei

Minuten dauern, damit der Anwender schnellstmöglich das Ergebnis erhält. Auch hier kann es bei zu komplexen Gebäuden zu längeren Zeiten kommen. Das genaue Aussehen des Gebäudes soll dabei über die Shape-Grammar definiert und durch den Generator umgesetzt werden. Es soll möglich sein, verschiedene Formen von Gebäuden zu generieren. Dies bedeutet, dass es möglich ist, Gebäude in T-Form zu generieren. Es soll also möglich sein, die Grundstruktur frei über die Grammatik zu wählen, solange das Herausziehen im 90-Grad-Winkel zur Grundform erfolgt. Zu einem gewissen Zufallsfaktor sollen Teile des Gebäudes hinzugefügt werden. Dies soll bedeuten, dass zum Beispiel ein Fenster zu einer gewissen Wahrscheinlichkeit als Regel angewendet wird.

So wird das Problem der nicht Übereinstimmung gelöst. Die Generierung sollte möglichst mit geringem Speicherbedarf erfolgen. Damit der Prototyp auf möglichst vielen Systemen läuft.

3.5 Anforderung Stadtgenerierung

Die Anforderungen an die Stadtgenerierung sollten wie folgt definiert sein. Als Erstes soll es möglich sein, eine Stadt mit nur geringen Einstellungen und Klicks zu generieren. Dabei sollen nur die Einstellungen der minimalen und maximalen Höhe, Länge und Breite angegeben werden. Dazu soll der Anwender die Gebäuderegeln bestimmen.
Hier muss mindestens eine Regel ausgewählt werden und hat keine maximale Grenze. Zum Schluss muss noch angegeben werden, wie viele Gebäude generiert werden sollen. Als Zweites soll das Aussehen und die Größe der Gebäude unterschiedlich sein, damit die virtuelle Stadt einer richtigen Stadt ähnelt.

Als dritte Anforderung soll je nach minimalen und maximalen Werten und Anzahl der Gebäude das Einlesen der Regeln und dessen Ausführung nicht länger als fünf Minuten dauern. Die zuvor genannten Anforderungen haben den Grund, damit der Anwender mit wenig Aufwand die Stadt generieren kann. Damit der Anwender nicht zu lange auf das Ergebnis warten muss, wird oben die Zeit angegeben.

Als Viertes soll es möglich sein, 81 Gebäude zu generieren, die eine normale Komplexität haben und deren Größe durchschnittlicher Gebäude entspricht. Diese Zahl wird bestimmt, damit die Anzahl der Gebäude einer Stadt entspricht und der Speicherbedarf gering ist.

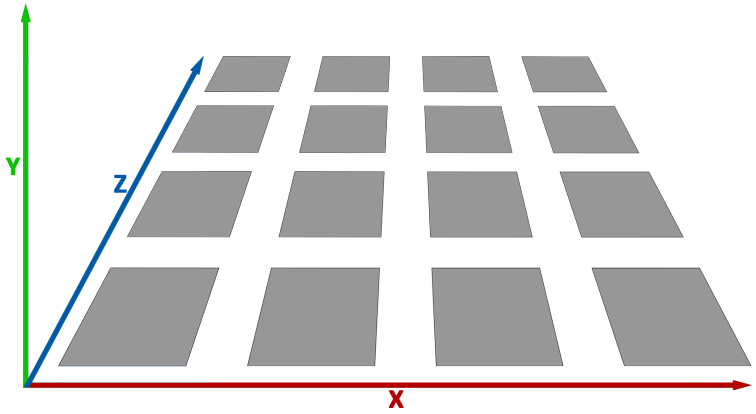

Abbildung 3.1: Beispiel der Stadtpositionierung.

Die fünfte Anforderung ist, dass die Stadt für diese Arbeit einfach aufgebaut werden soll. Es soll also kein L-System oder ähnliche Algorithmen verwendet werden, um diese zu generieren. Ein Beispiel für die Positionierung ist in Abbildung 3.1 zu sehen. Die Gebäude sollen dabei quadratisch mit Abstand zueinander generiert werden. Bei dem X-Parameter soll es sich um die Anzahl an Gebäuden pro Reihe handeln. Der Z-Parameter ist die Anzahl der Reihen, welche erzeugt werden sollen. Das Produkt vom X- und Z-Parameter ergibt die Anzahl an erzeugten Gebäuden. Genauso wie bei der Gebäudegenerierung soll die Stadtgenerierung mit wenig Speicherbedarf erfolgen.

3.6 Anforderung an den Prototyp

Die Anforderungen an den Prototypen sind wie folgt. Es soll möglichst viel implementiert werden, um die Komplexität zu erfassen und die Thematik der Shape-Grammar zu analysieren. Es soll über ein Menü möglich sein, ein Fenster zu öffnen. Dieses soll zum Erstellen von Gebäuden dienen. In dem Fenster können folgende Parameter bestimmt werden. Diese sind die Höhe, Länge und Breite sowie Position in X und Z Richtung.
Des Weiteren sollen aus dem Verzeichnis Building alle Gebäude automatisch ausgelesen werden und in dem graphical user interface(kurz GUI) angezeigt werden. Eins von diesen Gebäuden kann ausgewählt werden. Nach Betätigung einer Schaltfläche auf der GUI soll es möglich sein, die Regeln aus einer Datei auszulesen. Diese soll dann in einen Regelbaum überführt werden. Es soll dann die Split-Grammar angewendet werden, um

aus den Regeln und der Grundform den Shape-Baum zu generieren. Zum Schluss soll der Shape-Baum angezeigt und das Gebäude visualisiert werden.

4 Entwurf

4.1 Entscheidung Grafik-Engine

Es werden drei Engines vorgestellt. Es wurden diese Engines gewählt, da gewisse Erfahrungen im Umgang mit diesen vorhanden sind und eine empfohlen wurde. Dabei werden deren Vorteile erläutert und anschließend die Wahl der Engine für diese Arbeit dargelegt.

4.1.1 CgResearch

CgResearch ist ein von Professor Dr. Jenke an der HAW-Hamburg entwickeltes Grafik-Framework. Das Framework unterstützt Jogl und Jmonkey. Somit baut das Framework auf OpenGL auf und ist in der Sprache Java geschrieben. Dieses Framework ist noch sehr neu und wird ständig durch Prof. Dr. Jenke und den Studenten der HAW-Hamburg weiterentwickelt. Entwickeln kann man mit jeder Entwicklungsumgebung, die Java unterstützt. Beispiel hierfür ist Eclipse.

Es enthält alle Anforderungen, die zuvor beschrieben wurden. Des Weiteren wird es stets erweitert und bietet daher ein großes Potenzial für Forschungsthemen. Es beinhaltet für die Dreiecksnetze ein TriangelMesh Objekt. In diesem Objekt werden die Vertices und Dreiecke gespeichert. Das TriangleMesh wir dann als CgNode an den Scenengraph gehängt und visualisiert. Zur Verwaltung der Texturen bietet CgResearch den ResourceManager, welcher eine Instanz des TextureManager enthält, die man aufrufen kann, um Texturen zu verwalten.

4.1.2 XNA

XNA ist ein Framework, das von Microsoft entwickelt wurde und inzwischen nicht mehr weiter entwickelt wird. Am 30. August 2006 erschien die erste Betaversion. Am Ende des gleichen Jahres wurde die erste fertige Version veröffentlicht. Die letzte Version kam am 28. September 2011 und ist daher schon sehr veraltet. Es handelt sich bei XNA nicht

um eine reine Grafik-Engine. Es wurde entwickelt, um Computerspiele für alle Microsoft Umgebungen zu entwickeln.
Es basiert auf die hauseigene Grafik-Engine DirectX und Direct3D. Das Framework basiert auf der .Net Sprache C#. Bei XNA entwickelt man mit Visual Studio.

XNA ist eine sehr mächtige Spiele Engine. Es beinhaltet viele Features und bietet ein geeignetes Interface für sämtliche Anforderungen. Zur Visualisierung verwendet XNA eine Liste von Vertices und Indizes, welche an das GraphicDevice übergeben werden. Zur Verwaltung von Texturen verwendet das Framework die eigens entwickelte Content Pipline, in der bei Bedarf die passende Textur geladen werden kann [Mic15].

4.1.3 Unity

Unity ist ein Framework von Unity Technologies und ist wie XNA ein Framework zum Entwickeln von Spielen. Dieses Framework wird ständig weiter entwickelt. Abhängig von der Zielplattform wird Direct3D oder OpenGL verwendet. Daher erlaubt Unity die Entwicklung für sämtliche Ziel-Plattformen. In Unity kann man in C#, UnityScript und Boo entwickeln. Bei diesem Framework ist man an die für Unity extra entwickelte Entwicklungsumgebung gebunden.

Bei Unity handelt es sich wie zuvor bei XNA um eine sehr mächtige Spiele Engine. Für grafische Darstellungen bietet Unity viele implementierte Features an, die verwendet werden können. In Unity gibt es das Mesh Objekt, das zur Verwaltung der Dreiecke dient. In diesem Mesh Objekt werden die Vertices und Indizes gespeichert. Dieses Mesh wird dann in einem GameObject gespeichert und visualisiert. Die Verwaltung der Texturen läuft für den Benutzer über die Entwicklungsumgebung und wird als Texture2D verwendet [Uni15].

4.1.4 Entscheidung

In erster Linie geht es um einen schnellen Einstieg und um die wissenschaftlichen Aspekte. Daher wird das CgResearch verwendet. Es hat natürlich nicht die gesamten Features der anderen beiden vorgestellten Engines, doch es hat einige wichtige Vorzüge gegenüber den anderen zwei.

Der erste und wichtigste Vorzug ist, dass in diesem Framework die wissenschaftlichen

Arbeiten von Studenten hinzugefügt werden. Des Weiteren basieren alle wissenschaftlichen Arbeiten auf demselben Framework. Dies ermöglicht es, dass die Arbeiten einfach zusammengeführt werden können.
Dazu ist der Einstieg in diesem Framework sehr einfach und es ist schnell möglich, mit der eigentlichen Arbeit zu beginnen.
Durch den direkten Kontakt zu Prof. Dr. Jenke ist es möglich, bei Unklarheiten mit dem Framework, dies zu klären. Zudem beinhaltet das Framework alle relevanten Features, die für die Arbeit benötigt werden.

Ein weiterer Vorteil bei dem Framework ist es, dass es sich um ein Grafik Framework handelt und nicht wie bei den anderen beiden um eine Spiele Engine. Bei der Spiele Engine hat man den Game Loop. Dieser müsste bei der Entwicklung beachtet werden, da dieser zu Problemen führen kann. Durch die oben genannten Aspekte wird das CgResearch Framework verwendet.

4.2 Entwurf der Formalen Grammatik

basicrule.rule | start.rule

```
1  Building -->
2      split(f) { left, right, back : Facade | front : Front| top : Roof };
```

Abbildung 4.1: Beispiel für eine Basisregel Datei.

basicrule.rule | start.rule

```
1  import("./basic rules/basicrule.rule");
2
3  Roof -->
4      texture("./building1/textures/roof.jpg");
5
6  Facade -->
7      split(y) { 2 : Floor }*;
8
9  Floor -->
10     split(x) { 2 : WindowFacade }*;
11
12 Front -->
13     split(x) { ~ : FrontY | 4 : Extrude };
14
15 Extrude -->
16     extrude(4) { left, right, back, front : Facade | top : Roof };
17
18 FrontY -->
19     split(y) { 2 : FrontGround | ~ : Facade};
20
21
22 FrontGround -->
23     split(x) { 2 : WindowFacade | 2 : Door | 2 : WindowFacade}*;
24
25 WindowFacade -->
26     texture("./building1/textures/window.jpg");
27
28 Door -->
29     texture("./building1/textures/door.jpg");
```

Abbildung 4.2: Beispiel für eine Startregel Datei.

Die formale Grammatik wurde aus dem Programm CityEngine [Esr15] nachempfunden und für die hier entwickelte Anwendung angepasst.

Die erste Möglichkeit ist die Grammatik in XML zu erstellen. Dies lässt sich leicht parsen und verwenden. Doch dies wird immer unübersichtlicher, wenn die Grammatik komplexer wird. Um dafür eine Lösung zu finden, wird die Grammatik in Form von der Chomsky-Grammatik aufgebaut.

In den Abbildungen 4.1 und 4.2 sieht man nun den Aufbau der Grammatik. Die erste Abbildung zeigt eine der Basisregeln, die in eine für das Gebäude angepasste Grammatik geladen werden kann. Die Grammatik muss immer mit dem Variablensymbol Building beginnen, damit von der Software die Startregel erkannt wird. Nach dem Variablensymbol folgt eine bis mehrere Funktionen. Die wichtigste Funktion ist die Split-Funktion.

Nach dem Startsymbol folgt ein Split(f). f steht hierbei für Facette. Diese Funktion mit dem Parameter sorgt dafür, dass der Quader in seine Facetten unterteilt wird. Somit wird das dreidimensionale Problem zu einem zweidimensionalen Problem überführt. Als Grundform wird hierbei ein Quader verwendet, da diese Form dem Gebäude am meisten gleicht.

```
FrontY -->
    split(y) { 2 : FrontGround | ~ : Facade};

FrontGround -->
    split(x) { 2 : WindowFacade | 2 : Door[50%] | 2 : WindowFacade}*;
```

Abbildung 4.3: Beispiel für zwei split Funktionen.

Für die anderen Regeln gibt es nun verschiedene Funktionen, die zur Verfügung stehen. Zum einen gibt es die split Funktion. Split teilt die Form in X oder Y Richtung. Dabei kann man die einzelnen Teile mit Größe bestimmen. Diese werden in geschweiften Klammern geschrieben und werden mit dem vertikalen Strich getrennt. Um eine Wiederholung der Split-Regel zu erhalten, gibt es das Sternsymbol, welches am Ende der Funktion angegeben wird. Für das Angeben von relativen Größen wurde das Tilden-Symbol gewählt. Dieses wird anstatt einer Größe angegeben.

Mit der Funktion *texture* kann man einem Shape eine Textur zuweisen. Dies ermöglicht die individuelle Texturierung einzelner Shapes. Dabei ist der Parameter der Pfad zur Textur.

Für die Veränderung der Grundform gibt es die Funktion *extrude*. Hierbei gibt man die Größe an und wie beim Building Split die Regeln für die Facetten. Dabei ist zu beachten, dass diese Größe zurzeit auf die eingegebene Parameter Größe darauf addiert wird. Die Richtung, in der das Gebäude erweitert wird, hängt von der übergeordneten Facette ab. Dies bedeutet, wenn die übergeordnete Facette in X Richtung zeigt, auch das Extrudieren in X Richtung erfolgt.

Um den Regeln einen Zufallsfaktor hinzuzufügen, ist es möglich, hinter dem Regelnamen in eckigen Klammern eine Prozentzahl anzugeben. Diese bestimmt dann, wie

wahrscheinlich es ist, dass diese Regel angewendet wird. Dabei wird ein Datentyp verwendet, mit dem es möglich ist, Wahrscheinlichkeiten anzugeben. Diese werden dann zu einem gewissen Prozentsatz zurückgegeben.

4.3 Entwurf des Shape-Grammar Paket

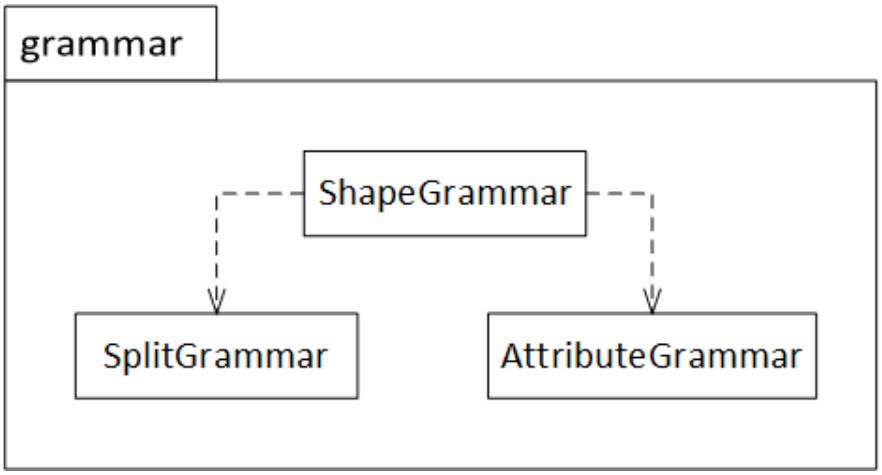

Abbildung 4.4: Aufbau des grammar Paket, nachempfunden von [WWSR03].

Das *grammar* Package ist unterteilt in drei Klassen. Die grobe Struktur wurde aus der Arbeit von Wonka nachempfunden [WWSR03].
Doch der Ablauf, wie die Klassen zusammenarbeiten und die interne Struktur funktioniert, wird für die Anwendung passend entwickelt.
Die drei Klassen dienen der Unterteilung der verschiedenen Aufgaben in der Shape-Grammar. Damit hat jede Klasse eine bestimmte Aufgabe. Dadurch ist es einfacher, bei Anpassungen die Stelle zu finden, an der die Änderungen stattfinden sollen.

4.3.1 ShapeGrammer

Die *ShapeGrammar* Klasse ist in diesem Konstrukt die Hauptklasse, die für das Zusammenspiel der anderen zwei Klassen verantwortlich ist. Diese Klasse wird aufgerufen, um die Generierung des Gebäudes zu initialisieren. In dieser Klasse werden Instanzen der *SplitGrammar* und *AttributeGrammar* Klassen erzeugt. Die *ShapeGrammar* Klasse bekommt dabei die eingelesenen Regeln übergeben und leitet diese an die *SplitGrammar* Klasse weiter. Das Ergebnis dieser Klasse wird dann an die *AttributeGrammar* Klasse

weiter gegeben. Das Ergebnis dieser Klasse wird von der *ShapeGrammar* zurückgegeben.

4.3.2 SplitGrammar

Die SplitGrammar ist die Klasse, welche das Unterteilen der Shapes übernimmt.
Dabei wird zuerst das dreidimensionale Problem auf ein zweidimensionales Problem überführt, indem der Quader in seine Seiten unterteilt wird. Diese Seiten können nun als zweidimensionale Fläche betrachtet und bearbeitet werden. In dieser Klasse werden die eingelesenen Regeln verwendet.
Die Regeln werden rekursiv durchlaufen und an den Shapes angewendet, bis keine Ersetzung mehr möglich ist. Dabei wird das Ergebnis in einer Datenstruktur gespeichert und an die ShapeGrammar Klasse zurückgegeben.

4.3.3 AttributeGrammar

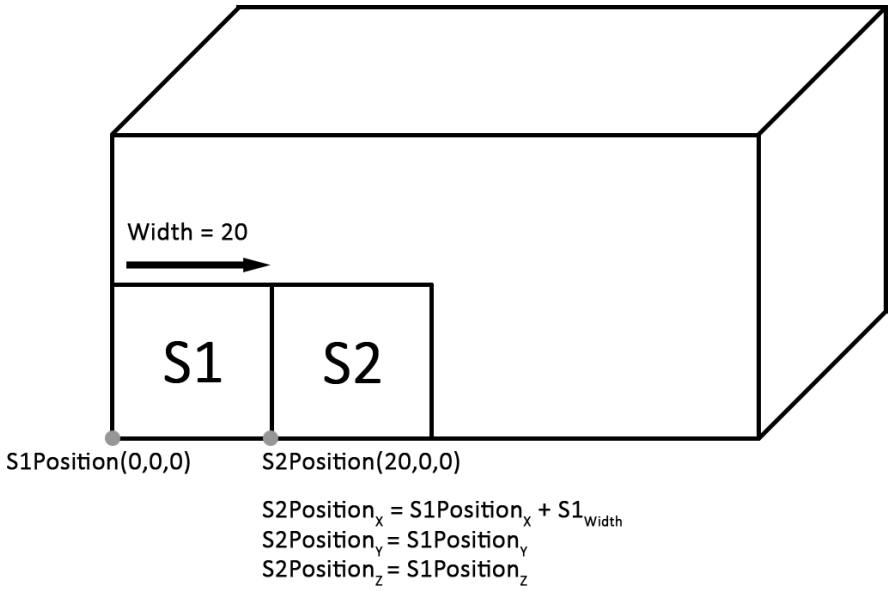

Abbildung 4.5: Beispiel für die Attributierung der Shapes.

Die AttributeGrammar Klasse ist für die Attributierung der Shapes verantwortlich. Sie sorgt für die Texturen und die Position der einzelnen Shapes. Dabei ist es wichtig, die Positionen der Shapes aus dem zweidimensionalen Bereich ins dreidimensionalen zu

überführen. Dies übernimmt diese Klasse, in dem die Position der Grundform verwendet wird (siehe Abbildung 4.5). Dabei wird die Position der Shapes rekursiv gesetzt, in dem die Länge oder Breite auf die Grundformposition addiert wird und somit wieder in den dreidimensionalen Raum überführt.

In Abbildung 4.5 ist S1Position gleich der Position der Grundform und somit die Position des ersten Shapes S1. Um die Position von S2 zu berechnen wird zuerst die Y- und Z-Position von S1Position übernommen, da das Shape S2 in Richtung der X Achse versetzt zu S1 positioniert wird. Somit muss die X-Position berechnet werden, indem die X-Position von S1 und die Breite des Shapes S1 addiert werden. Somit wird die Position von S2 berechnet. Diese Prozedur wird für jedes Shape durchgeführt. Je nach Richtung muss die Berechnung mit X, Y oder Z durchgeführt werden.

4.4 Aufbau der Ordnerstruktur

Die Ordnerstruktur im Bezug auf die Gebäude hat eine tragende Rolle in der Anwendung. Das Einhalten der Struktur ermöglicht eine Art Plug-in System. Es können weitere Gebäude hinzugefügt werden, ohne den Code zu verändern.

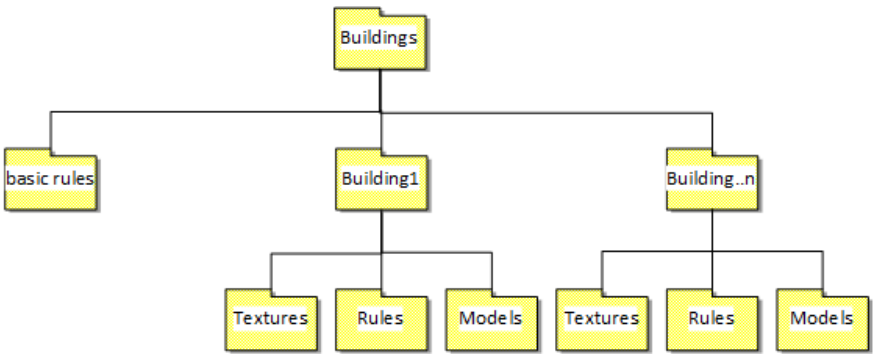

Abbildung 4.6: Beispiel der Ordnerstruktur.

Die Struktur ist in Abbildung 4.6 zu sehen. Im *Buildings* Ordner ist der *basic rules* Ordner. Dieser enthält alle Standardregeldateien. Diese können von anderen Regeln verwendet werden. Des Weiteren sind die Gebäude-Ordner in dem Verzeichnis. Die Namen sind frei wählbar und werden in der GUI angezeigt. Die jeweiligen Gebäude-Ordner müssen nun ein *Textures*, *Rules* und *Models* Ordner enthalten. In den *Textures* Ordner kommen alle Texturen, die für die Visualisierung des Gebäudes benötigt werden.

Der *Rules* Ordner ist der wichtigste. In ihm werden die Regeldateien gespeichert, die das Gebäude beschreiben. Im *Model* Verzeichnis werden die 3D-Modelle hinterlegt, welche zum Gebäude hinzugeladen werden können. Dies können Tür, Fenster, Treppen, Wände oder sogar ein ganzes Dach sein.

4.5 Entwurf verwendeter Datenstrukturen

4.5.1 Virtuelle Objekte

Virtuelle Objekte haben in dieser Arbeit eine einfache doch wichtige Rolle. Zum einen ermöglichen sie das Speichern von Daten, die für die Shape-Grammar relevant sind. Zum anderen ist der Prototyp zum größten Teil unabhängig von der Engine und muss nur an wenigen Stellen angepasst werden, wenn die Engine ausgewechselt werden soll.

VirtualDirection

Für die Richtung wurde ein Enum angelegt. Dieses Enum wird verwendet, um die Richtung eines Shapes anzugeben. Das Enum beinhaltet die sechs Richtungen. Diese sind Top, Left, Front, Right, Bottom und Down.
Dies ermöglicht es schnell zu überprüfen, in welche Richtung die Fläche zeigt. Dies ist wichtig für die Positionierung der Shapes. Des Weiteren wird die Richtung für das *extrude* benötigt. Dabei muss die neue Grundform in Richtung des Shapes extrudiert werden.

VirtualPoint

Für die Position des Gebäudes und der einzelnen Shapes wurde das *VirtualPoint* Objekt hinzugefügt. Dieses enthält die Informationen für die X, Y und Z Koordinaten und das *VirtualDirection* Objekt für die Ausrichtung. Somit wird die Position und Ausrichtung zentral an einer Stelle gespeichert. Diese können somit einfach abgerufen werden.

VirtualShape

Das *VirtualShape* Objekt dient zur Erkennung einer flachen Form die von der Shape-Grammar erzeugt wird. Diese enthält Informationen der Größe. Bei der Größe handelt es sich um die Höhe und Breite. Ein weiteres Attribut ist der Name der Form, um diese später wieder zu erkennen. Das *VirtualShapes* Objekt beinhaltet das *VirtualPoint*

um die Position und Ausrichtung zu speichern. Schließlich wird noch der Texturpfad in dem Objekt gespeichert. Somit sind alle wichtigen Informationen zur Visualisierung der Form vorhanden.

VirtualCubiod

Für die Grundform wird das *VirtualCubiod* erstellt. Dieses beinhaltet Informationen über die Position in Form des *VirtualPoint* Objektes und über die Größe in Form von Höhe, Länge und Breite.
Welche den eingegeben Parametern entsprechen. Schließlich enthält das Objekt sechs *VirtualShapes* für die sechs Richtungen. Diese werden bei Bedarf ausgelesen, wenn diese benötigt werden.

VirtualMethod

Ein weiteres virtuelles Objekt ist das *VirtualMethod* Objekt. Dieses dient der virtuellen Darstellung der Methoden in der Shape-Grammar. Es wird der Name der Methode gespeichert, um diese in der Anwendung zu erkennen. Zudem wird der Parameter der Methode gespeichert. Schließlich enthält das *VirtualMethod* Objekt die Informationen der Aktion. Diese Aktion wird mit dem angegeben Parameter ausgeführt.

4.5.2 Regelbaum

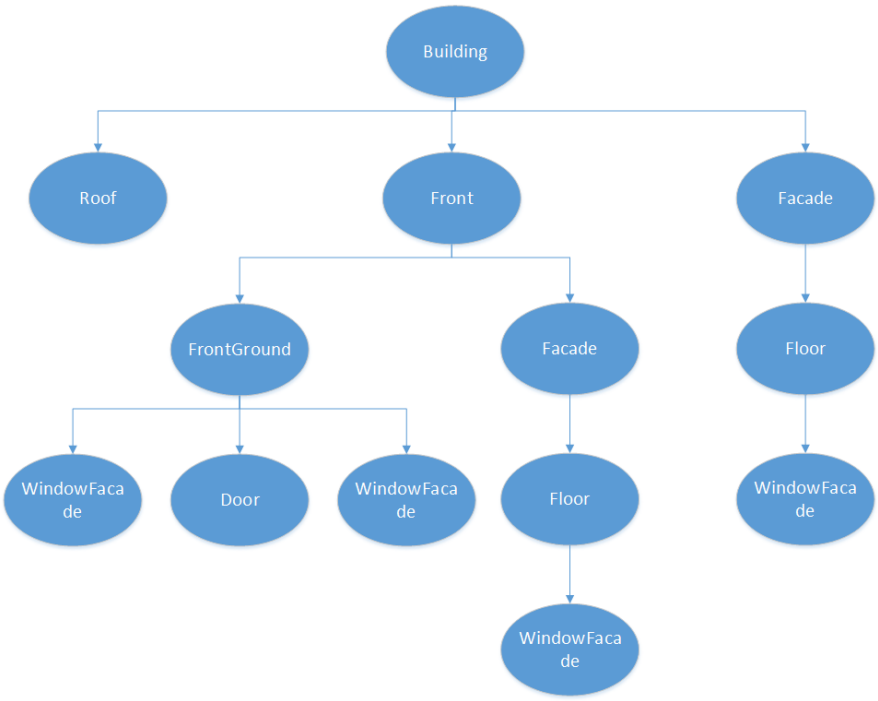

Abbildung 4.7: Beispiel einer Regelbaumes.

Für die Verwaltung der Regeln wird eine Baumstruktur verwendet. Ein einfaches Beispiel, wie der Regelbaum aussieht, wird in Abbildung 4.7 gezeigt. Aus Einfachheit und Größe wird ein simples Beispiel gezeigt.
Die Baumstruktur eignet sich am besten für die Verwaltung der Regeln. Diese Struktur hat den Vorteil, dass über den aktuellen Knoten die nachfolgenden Knoten erreicht werden. Dies erlaubt, durch Rekursion die Regeln auf die Shapes anzuwenden. Durch die Rekursion ist die Durchführung der Regeln mit wenig Code möglich. Die Baumstruktur ermöglicht den einfachen Überblick über die Ersetzungen. In der Abbildung sieht man, dass nach der Building Regel die Roof, Front und Facade Knoten folgen. Dies bedeutet, dass die Grundform durch diese ersetzt wird.
Dabei ist zu beachten, dass Knoten mit gleicher Tiefe nacheinander auf die vorherige Form angewendet werden.

4.5.3 Shape-Baum

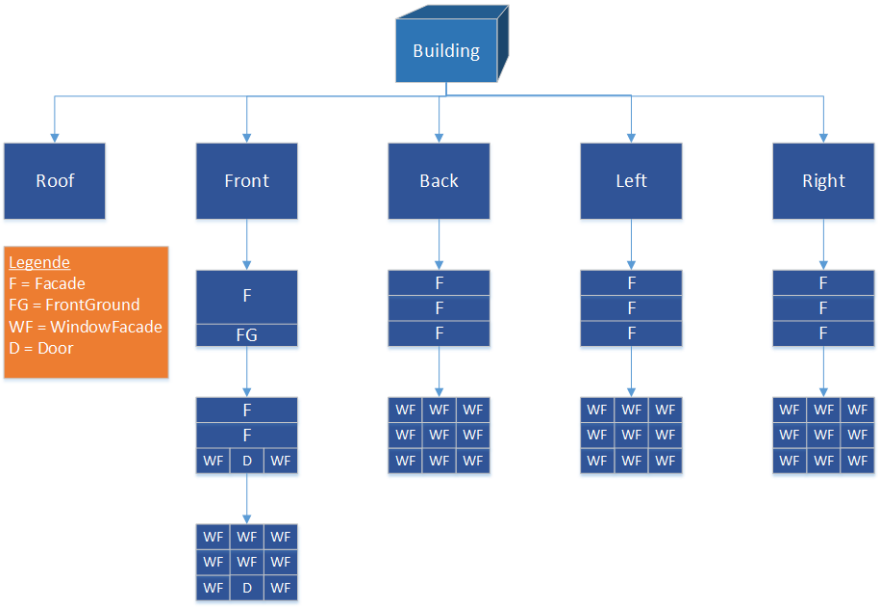

Abbildung 4.8: Beispiel eines Shape-Baumes.

Für die Verwaltung der Shapes wird ebenfalls wie bei den Regeln eine Baumstruktur verwendet. In Abbildung 4.8 wird ein Beispiel gezeigt. Hierbei ist zu beachten, dass zur Visualisierung die Endknoten des Baumes benötigt werden. Für die Bearbeitung der einzelnen Shapes wird dennoch der gesamte Baum gespeichert. Dies ist erforderlich, damit Informationen wie die Position, Größe und weitere Attribute an die Kindknoten übergeben werden können. Wenn zum Beispiel eine Änderung an einem Kindknoten vorgenommen wird, können über den Elternknoten die anderen Kindknoten angepasst werden. Dieses gilt zum Beispiel für relative Größen. Diese Größe wird erst am Ende, wenn alle statischen Größen vorhanden sind, angepasst.
Des Weiteren ermöglicht es, leichter die Vergrößerung des Gebäudes vorzunehmen.

Durch die Struktur ist es wie bei dem Regelbaum möglich, rekursiv durch die Struktur zu gehen und zu bearbeiten. Die Vorteile sind dabei wie zuvor im Regelbaum beschrieben.

4.6 Prototyp

4.6.1 Use Case

Title: Gebäude erstellen **Akteur:** Anwender **Ziel:** Ein virtuelles Gebäude aus Regeln erstellen und Anzeigen lassen

Vorbedingung:
Keine
Nachbedingung:
Der Anwender hat ein virtuelles dreidimensionales Gebäude erzeugt

Erfolgsszenario:

1. Anwender startet die Anwendung und klickt oben im Menü auf Building und dort auf New um das Fenster zum Generieren der Gebäude zu öffnen.
2. Anwender gibt Höhe, Länge, Breite und Position des Gebäudes ein.
3. Anwender wählt die Gebäuderegeln, die verwendet werden sollen, aus.
4. Anwender klick auf „Create“ um das Gebäude generieren zu lassen.
5. Anwender erhält virtuelles aus mehreren Meshes bestehendes dreidimensionales Gebäude.

4.6.2 Ablaufdiagramm

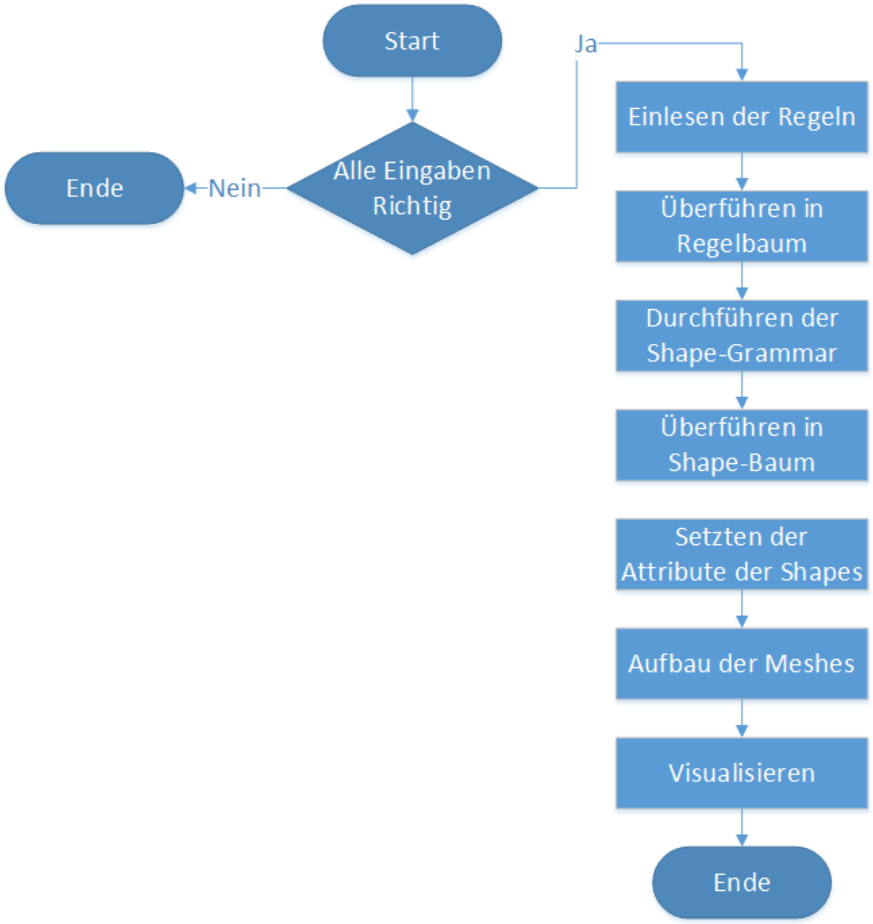

Abbildung 4.9: Ablaufdiagramm für die Generierung eines Gebäudes.

In dem Ablaufdiagramm wird gezeigt, wie der Ablauf ist, wenn der Benutzer die Erstellung des Gebäudes gestartet hat. Zuerst wird geprüft, ob die Eingabe der Parameter korrekt ist.
Wenn die Eingabe nicht korrekt ist, wird die Aktion beendet. Wenn die Eingabe korrekt ist, wird die angegebene Gebäuderegeldatei eingelesen. Die Regeln werden im nächsten Schritt in den Regelbaum überführt, damit diese in der Software verwendet werden können. Nun wird die Shape-Grammar mit dem Regelbaum und der Grundform durchgeführt. Durch diesen Schritt wird der Shape-Baum erstellt. In diesem Baum werden die Attribute der Shapes gesetzt. Im nächsten Schritt werden die Shapes aus dem Shape-

Baum zu Meshes der Gafik-Engine überführt. Im letzten Schritt werden die Meshes visualisiert. Zum Ende wird die Aktion beendet.

4.6.3 Paketdiagramm

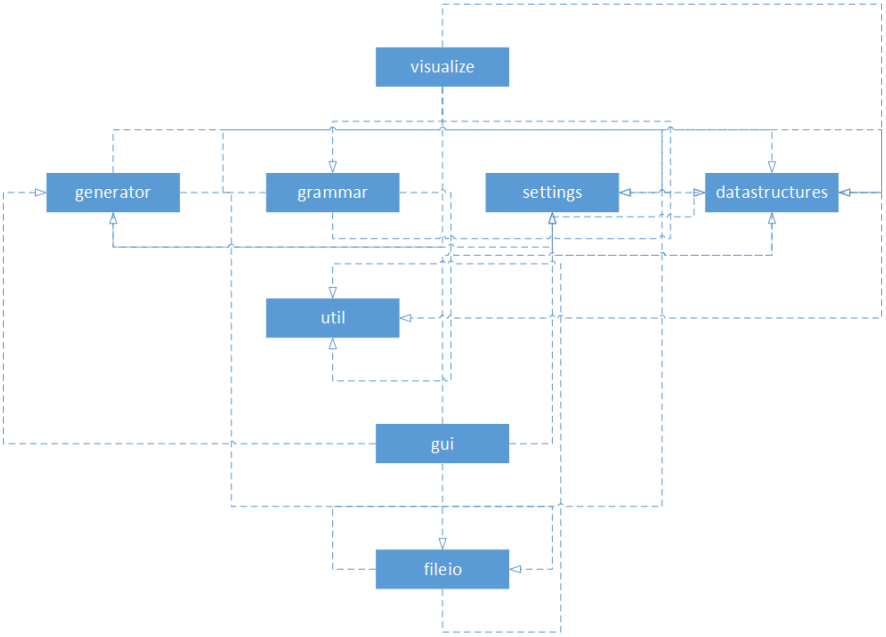

Abbildung 4.10: Paketdiagramm für den Prototyp.

Das Paketdiagramm zeigt die Abhängigkeiten der Pakete in diesem Prototypen. Dabei ist zu beachten, dass im Diagramm eine Ringabhängigkeit zwischen *datastructures* und *settings* sichtbar ist. Diese existiert in der Implementierung nicht, da das *datastructures* Subpakete enthält. Eins hat eine Verbindung zu *settings*. Die anderen werden von *settings* verwendet. Die Subpakete wurden zur Übersicht nicht im oberen Diagramm eingefügt.

Visualize

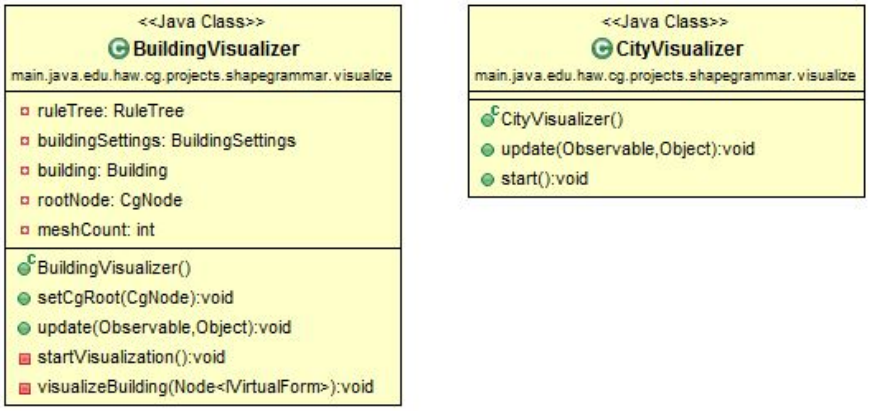

Abbildung 4.11: Klassendiagramm vom Visualize Paket.

Das *visualize* Paket ist für die Visualisierung zuständig. Dieses Paket übernimmt somit sämtliche Aufgaben, um den Shape-Baum zu visualisieren. Dies bedeutet, dass an dieser Stelle die Kommunikation mit der Grafik-Engine stattfindet. Somit müssen an dieser Stelle Veränderungen vorgenommen oder verschiedene Klassen für verschiedene Engines entwickelt werden, wenn die Engine ausgewechselt werden soll.

Generator

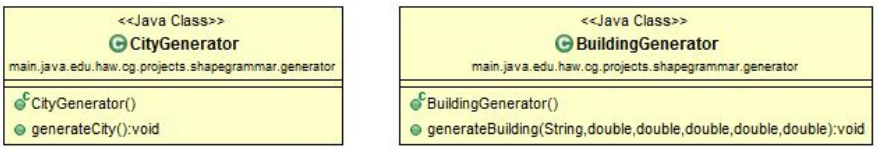

Abbildung 4.12: Klassendiagramm vom Generator Paket.

Das *generator* Paket sorgt für die Generierung der Einstellung der einzelnen Gebäude und belegt diese mit den Attributen, die beim Aufruf übergeben werden. Durch die Erstellung der Einstellungen wird die Visualisierung eingeleitet.
In diesem Paket soll es zwei Generatoren geben. Der eine für die Gebäude und ein weiterer für die Stadt.

Grammar

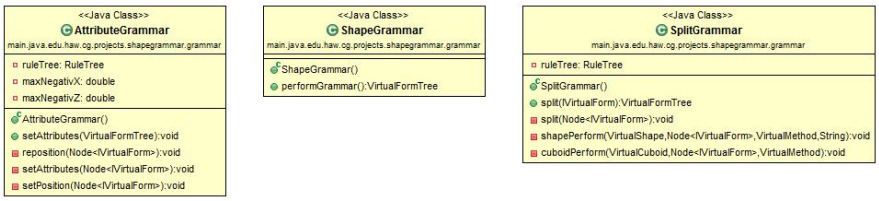

Abbildung 4.13: Klassendiagramm vom Grammar Paket.

Das *grammar* Paket sorgt für die Ausführung der Shape-Grammar. Dies bedeutet, dass in diesem Paket die drei Klassen *ShapeGrammar*, *SplitGrammar* und *AttributeGrammar* befinden. Diese übernehmen wie oben beschrieben die Durchführung der Shape-Grammar und Erstellung des Shape-Baumes.

Settings

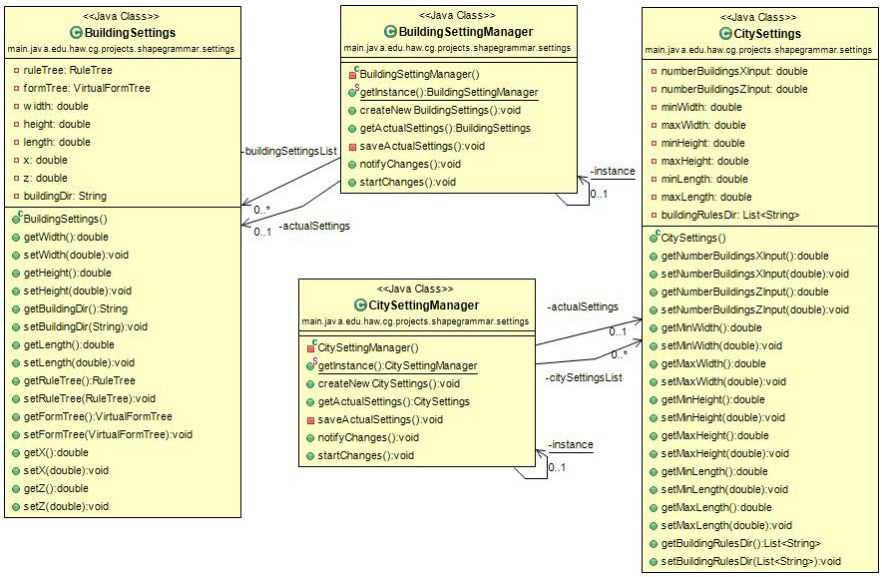

Abbildung 4.14: Klassendiagramm vom Settings Paket.

Im settings Paket befinden sich die Klassen zur Verwaltung und Speicherung der Einstellungen. Somit beinhaltet das Paket die vier Klassen BuildingSettings, BuildingSettingManager, CitySettings und der CitySettingManager.
Die BuildingSettings Klasse übernimmt das Speichern der gesamten Attribute des Gebäudes. Die BuildingSettingManager Klasse übernimmt das Verwalten der Gebäudeeinstellungen. Dabei wird das Singleton Pattern verwendet, um in dem gesamten Projekt auf die Einstellungen zuzugreifen. Des Weiteren beinhaltet die Klasse das Observer Pattern und signalisiert somit den Observern, wenn Einstellungen verändert werden.
Die anderen beiden Klassen ähneln dabei den Klassen für die Gebäude, nur das in diesem Fall die Stadtinformationen gespeichert und verwaltet werden.

Datastructures

Das datastructure Paket beinhaltet sämtliche Datenstrukturen, die in der Applikation benötigt werden.

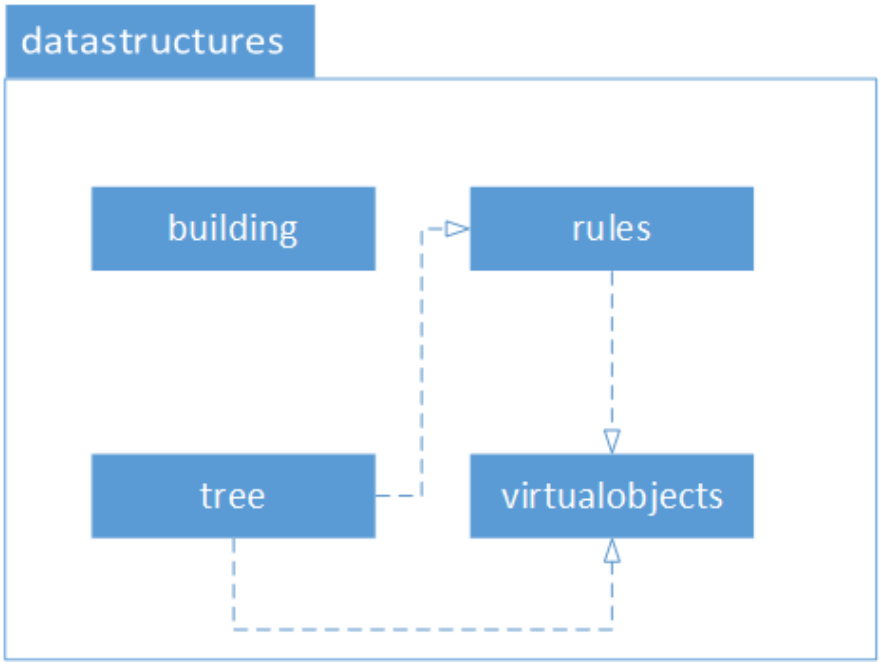

Abbildung 4.15: Klassendiagramm vom Datastructures Paket.

Das Paketdiagramm zeigt die interne Struktur des *datastructures* Pakets.
Dieses ist in vier Unterpakete unterteilt. Diese sind das *building* Paket, *rules* Paket, *tree* Paket und *virtualobjects* Paket.
Dabei beinhaltet das *building* Paket die Klasse zum Speichern der Gebäude Meshes. Diese werden dann, über eine Funktion in der Klasse, an den CgNode gehängt und somit visualisiert.
Das *rules* Paket beinhaltet die *GrammarRule* Klasse. Diese Klasse besteht aus den Attributen wie Name, der virtuellen Methode, Pfad zur Textur und dem Pfad zu dem dreidimensionalen Model.
Bei dem *tree* Paket handelt es sich um die Datenbäume. Diese sind wie weiter vorne beschrieben, der Shape-Baum und Regelbaum. Des Weiteren beinhaltet dieses Paket einen generischen Baum, der als Grundlage für die zwei zuvor genannten Bäume gilt. Die letzte Klasse in diesem Paket ist der Node, der generisch ist und von den Bäumen verwendet wird.
Das Paket *virtualobjects* beinhaltet wie im Kapitel 4.5.1 beschriebenen Klassen.

Util

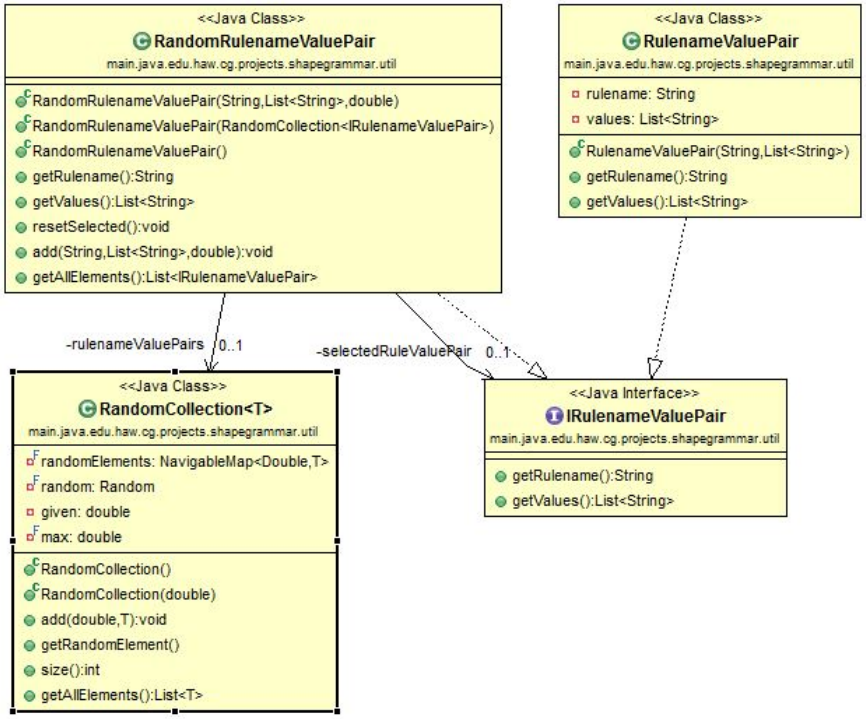

Abbildung 4.16: Klassendiagramm vom Util Paket.

Im *util* Paket sind sämtliche Utilitys Klassen enthalten, welche für die Anwendung benötigt werden. Die drei Klassen *RandomCollection*, *RulenameValuePair* und *RandomRulenameValuePair* sind dabei die Wichtigsten.

Bei *RandomCollection* handelt es sich um eine Liste, die es ermöglicht, zufällig ein Element aus der Liste zu erhalten. Dies bedeutet, wenn zum Beispiel das dritte Element aus der Liste entnommen werden soll und dieses eine 30-prozentige Wahrscheinlichkeit hat, dann wird dieses zu 30 Prozent zurückgegeben und zu 70 Prozent ein *null* Objekt. Dabei ist zu beachten, dass diese Klasse generisch ist.

Bei dem *RulenameValuePair* handelt es sich um die Klasse, die das Speichern der Regelnamen mit dessen Werten übernimmt. Dabei handelt es um die Regelnamen, welche in den geschweiften Klammern stehen.

Bei dem *RandomRulenameValuePair* handelt es sich um eine Klasse, welche die beiden oben genannten Klassen verwendet. Dies ermöglicht das Auswählen von Regeln mit Wahrscheinlichkeit.

GUI

Das *gui* Paket enthält alle relevanten Pakete, die mit dem Benutzerinterface zu tun haben.

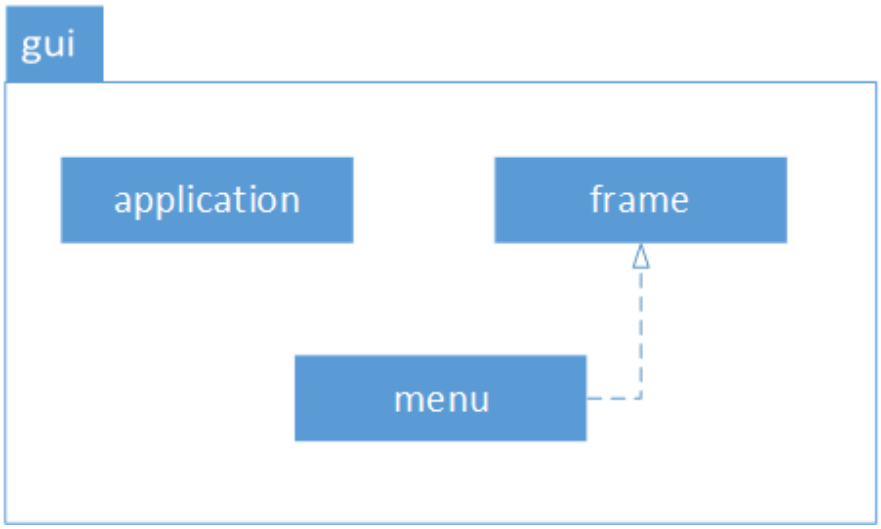

Abbildung 4.17: Klassendiagramm vom GUI Paket.

Das *application* Paket enthält Erweiterung der GUI des CgResearch Frameworks.
Das *menu* Paket enthält Erweiterungen des Menüs von der GUI des Frameworks.
Im *frame* Paket befinden sich die JFrames für die Eingabe der Daten für die Anwendung. Diese sind die GUIs für die Erstellung der Gebäude und für die Stadt.

Fileio

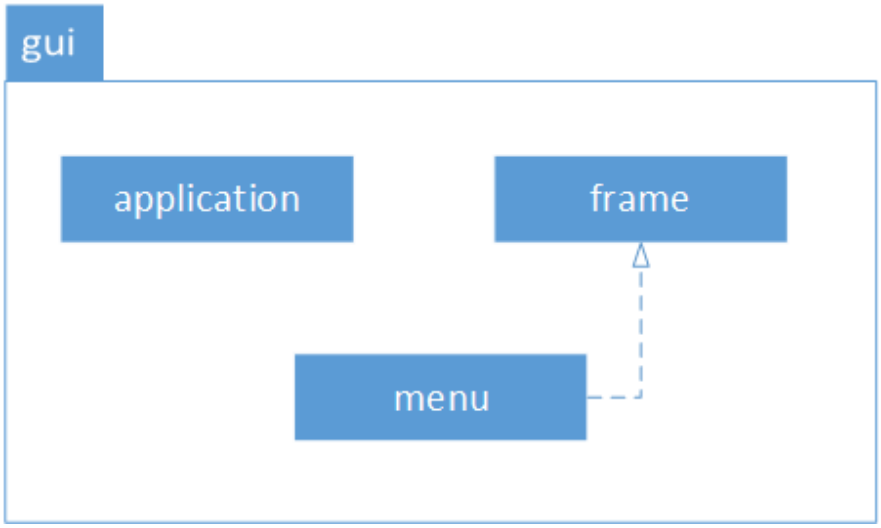

Abbildung 4.18: Klassendiagramm vom Fileio Paket.

Das **fileio** Paket sorgt für das Einlesen der oben genannten Ordnerstruktur und das Einlesen der Regeln. Die Regeln werden hier in den Regelbaum überführt.

4.6.4 Verwaltung der Gebäude Daten und Benachrichtigung

Die Verwaltung der Gebäude läuft über die Klasse *BuildingSettingManager*. Dieser speichert die Gebäude in einer Liste. Die aktuell verwendeten Einstellungen für den Zugriff werden separat gespeichert. Die aktuellen Einstellungen wechseln sich, beim Erstellen einer neuen Einstellung oder manuell durch eine Funktion.

Die *BuildingSettingManager* Klasse basiert auf dem Singleton Pattern. Somit sind die Einstellungen im gesamten Projekt verfügbar. Dies erleichtert den Zugriff auf die Einstellungen, da diese nicht durch alle Klassen durchgereicht werden müssen.

Die Benachrichtigung, ob Änderungen in den Einstellungen vorgenommen wurden, erfolgt über das Observer Pattern. Wenn eine Änderung vorgenommen wird, werden alle angemeldeten Klassen informiert.

5 Realisierung

5.1 Realisierung des Prototypen

Der Prototyp enthält alle Funktionen von der Version, die für die Evaluation der Shape-Grammar benötigt wird. Dies gibt einen leichteren Überblick über die Komplexität des Themas.
Es werden die Regeln dynamisch durch die Ordnerstruktur erkannt. Durch das Starten der Generierung eines Gebäudes wird die entsprechende Regel eingelesen und die Einstellungen in dem *BuildingSettingManager* gespeichert.

Das *visualize* Paket wird durch das Observer Pattern benachrichtigt, dass es Änderungen am Gebäude gibt, und startet die Shape-Grammar. Dabei gibt die *ShapeGrammar* Klasse den Regelbaum an die *SplitGrammar* Klasse. Des Weiteren bekommt die *SplitGrammar* die Startform, den Quader. Dieser Quader wird so lange unterteilt, bis keine Regel mehr angewendet werden kann.
Die Formen werden in einen Shape-Baum gespeichert um die Elternformen wieder zu erkennen, um bei Änderungen, über diese die anderen Kindformen zu erreichen.

Dieser Shape-Baum wird zur *ShapeGrammar* Klasse zurückgegeben und diese leitet den Shape-Baum an die *AttributeGrammar* Klasse weiter. Dort werden die Shapes attributiert. Dies bedeutet, das die Positionen der einzelnen Shapes gesetzt werden und gegebenenfalls die Texturen vom Elternknoten zum Kindknoten vererbt werden. Wenn jedoch eine Textur in den Regeln angegeben wird, wird diese verwendet. Danach werden die Shapes im Shape-Baum zu Meshes überführt, welche an den CgNode gehängt und visualisiert werden.

5.2 Realisierung der Anwendung

5.2.1 GUI Gebäude und Stadtgenerierung

Main GUI

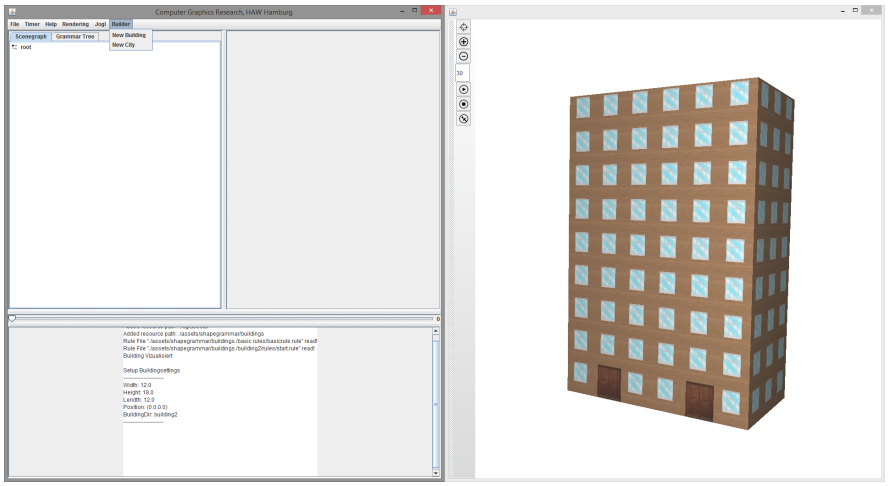

Abbildung 5.1: Abbildung der Haupt GUI. Texturen von [AGF15, mb315, tex15c, tex15b]

Die Haupt GUI wird von dem CgResearch zur Verfügung gestellt. Das rechte Fenster ist für die Visualisierung zuständig. Das Fenster bietet dabei die Steuerung in der 3D-Szene.
Das linke Fenster bietet dabei Erweiterungsmöglichkeiten. Es können weitere Bedienelemente hinzugefügt werden. Der Tab scenegraph zeigt dabei die Meshes, die an den Szenengraph hängen und visualisiert wurden.
Das Fenster enthält eine Möglichkeit zur Textausgabe für das Debuggen.

Für das Framework wurde das Menü „Builder“ und der Tab „Grammar Tree“ hinzugefügt.
Bei dem Menüpunkt „Builder“ ist es möglich „New Building“ oder „New City“ zu wählen.

Grammar Baum

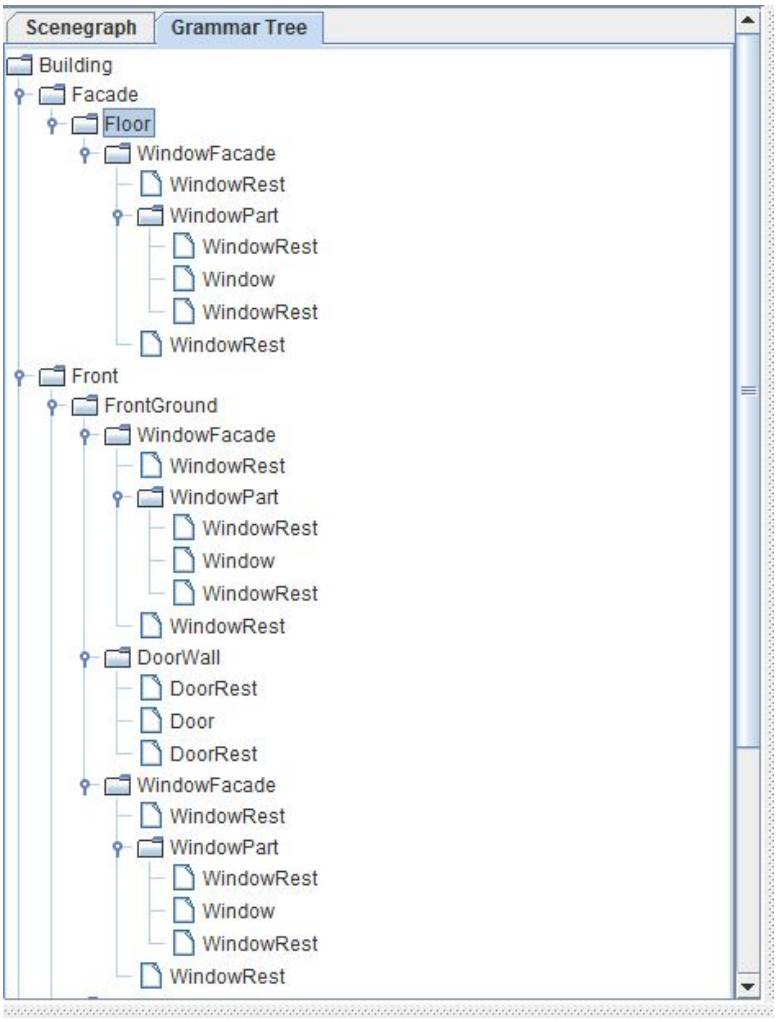

Abbildung 5.2: Abbildung der Grammar Baum GUI.

Um einen genauen Überblick über die eingelesene Grammatik zu erhalten, wird in diesem Tab der Regelbaum angezeigt. Dies ermöglicht zu überprüfen, ob in der Grammatik ein Fehler bei der Erstellung unterlaufen ist. In dieser Ausgabe wird immer die aktuelle Grammatik angezeigt, welche in dem *BuildingSettingManager* aktiv ist. Bei einem Wechsel der aktuellen Grammatik in dem Manager benachrichtigt dieser die GUI mit dem Observer Pattern, damit die GUI die Anzeige aktualisiert.

Neues Gebäude

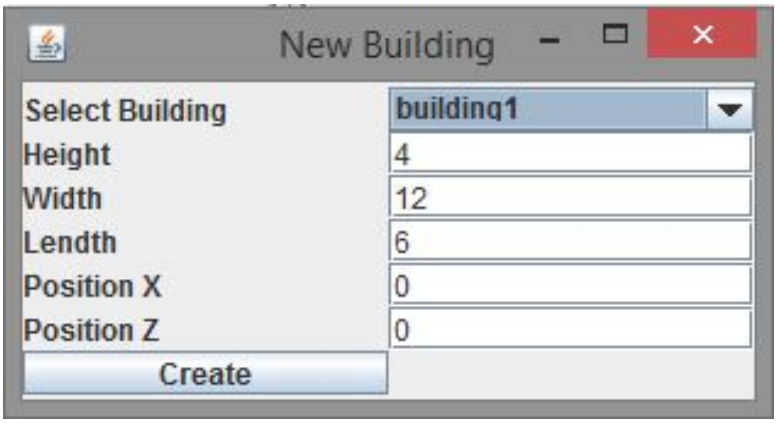

Abbildung 5.3: Abbildung der New Building GUI.

Diese GUI dient der Eingabe der Parameter für das zu generierende Gebäude. Des Weiteren wird an dieser Stelle die Generierung eingeleitet. Im ersten Feld hat der Benutzer die Möglichkeit, die Regeldatei zu wählen, welche er verwenden möchte. Als Nächstes muss die Größe bestimmt werden. Dies bieten die nachfolgenden drei Felder, in denen der Benutzer Höhe, Breite und Länge angeben muss. Schließlich muss noch die Position des Gebäudes bestimmt werden. Diese Angaben müssen bei den Feldern Position X und Position Z eingetragen werden.

Für die Generierung muss die Schaltfläche „Create“ betätigt werden. Dabei werden zuerst von der GUI die Eingaben überprüft, ob diese zulässig sind. Wenn dies nicht der Fall ist, wird in der Main GUI eine Fehlermeldung ausgegeben und die Generierung nicht gestartet. Wenn diese jedoch syntaktisch korrekt sind, wird die Generierung mit den eingegebenen Daten gestartet. Ein Beispiel eines generierten Gebäudes wird in Abbildung 5.4 gezeigt.

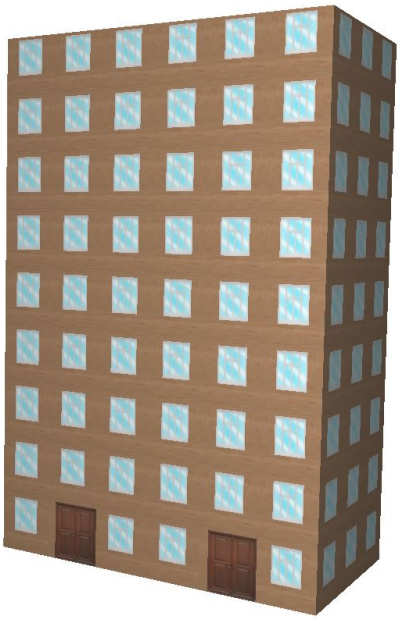

Abbildung 5.4: Ein Beispiel eines Gebäudes. Texturen von [AGF15, mb315, tex15c, tex15b]

Neue Stadt

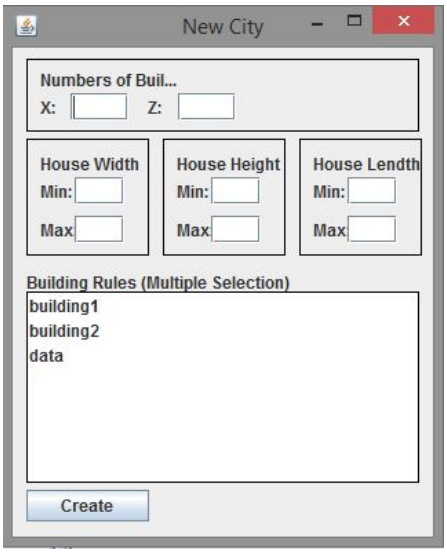

Abbildung 5.5: Abbildung der New City GUI.

Diese GUI dient der Stadtgenerierung. Die ersten zwei Eingabefelder dienen der Angabe der Anzahl an Gebäuden. Hierbei ist zu beachten, dass die Anzahl an Gebäude das Produkt von X mal Z entspricht, wie es im Kapitel 3.5 beschrieben wurde.
Die Felder in „House Width“ dienen der Eingabe der minimalen und maximalen Breite der Gebäude. Als Nächstes folgen die Felder in „House Height“. In diesen Feldern wird die minimale und maximale Höhe der Gebäude angegeben. Für die letzte Größenangabe dienen die zwei Felder in „House Length“. In diesen Feldern wird die minimale und maximale Länge angegeben. Im nächsten Schritt können nun die Regeln ausgewählt werden. Hierbei ist zu beachten, dass es möglich ist, mehrere Regeln aus zu wählen.
Schließlich kann die Generierung der Stadt durch die Schaltfläche „Create“ eingeleitet werden. Hierbei werden die Eingaben wieder überprüft. Wenn diese nicht korrekt sind, wird die Generierung nicht gestartet. Wenn diese jedoch korrekt sind, wird die Stadt generiert. Ein Beispiel für die Stadt sieht man in Abbildung 5.6.

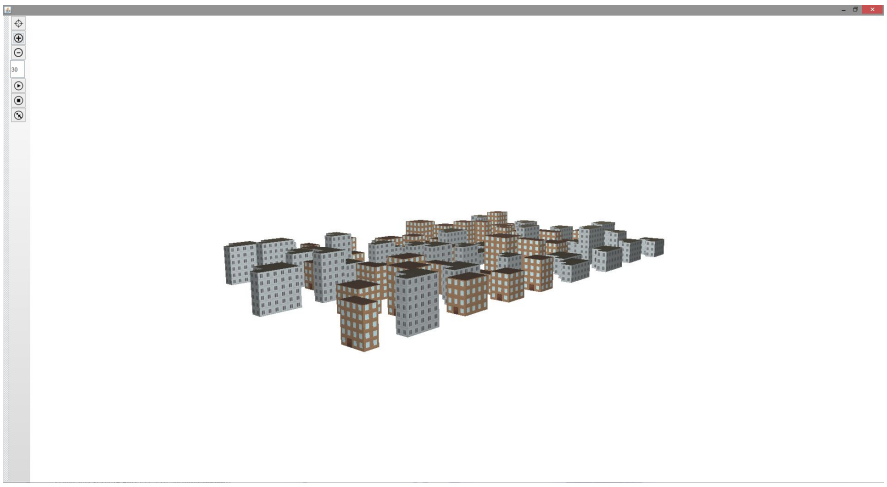

Abbildung 5.6: Beispiel einer generierten Stadt. Texturen von [tex15a, Car15, AGF15, mb315, tex15c, tex15b]

5.2.2 Einlesen der Formalen Grammatik

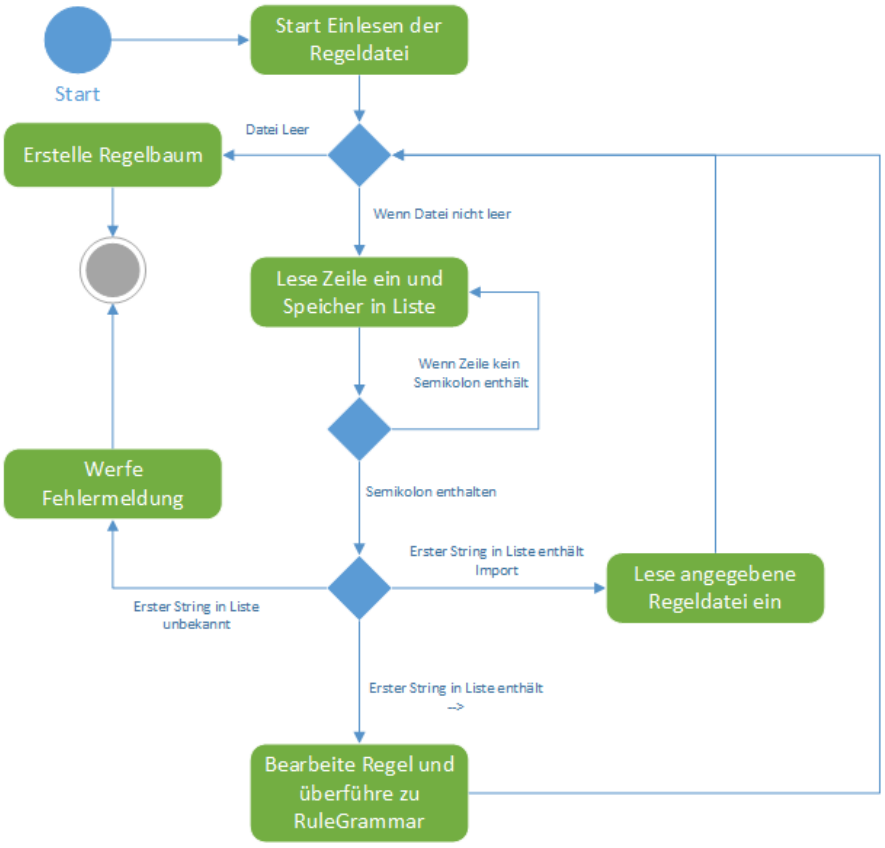

Abbildung 5.7: Aktivitätsdiagramm zum einlesen der formalen Grammatik.

Das Einlesen der Grammatik erfolgt über das Parsen der Regeldatei. Hierbei werden die Regeln bis zum Ende der Datei nacheinander eingelesen. So wird Zeile für Zeile eingelesen, bis das Terminalsymbol in einer Zeile vorhanden ist. Dabei werden die Zeilen in einer Stringliste gespeichert, um später die Zeilen separat zu bearbeiten.
Als Erstes wird, überprüft, ob die erste Zeile der eingelesenen Regel die Zeichenkette include enthält. Wenn dies der Fall ist, wird die dort angegebene Regeldatei zu erst gelesen und das Ergebnis in einer HashMap gespeichert, mit dem Namen der Regeln als Schlüssel.
Wenn die erste Prüfung fehlschlägt, wird als Zweites überprüft ob das Symbol „–>“ in der ersten Zeile enthalten ist, welches die Regel vom Namen trennt. Wenn dies der Fall ist, beginnt das Parsen der Regel in die GrammarRule. Hierbei wird als Erstes der Re-

gelname aus der ersten Zeile gelesen. Danach werden die restlichen Zeilen durchlaufen und überprüft, welche Funktion in dieser beschrieben ist. Dabei wird je nach Funktion der String geparst und bearbeitet. Die gesamten Daten werden dabei in GrammarRule gespeichert. Nachdem alle Zeilen der Regel durchlaufen wurden, wird die GrammarRule mit dem Namen in der Hashmap gespeichert.
Als Letztes wird eine Fehlermeldung ausgegeben, wenn die zuvor erwähnten Überprüfungen fehlgeschlagen sind.
Wenn das Einlesen der Regeln vollendet ist, wird die HashMap zurückgegeben und zum Regelbaum überführt.
Dies beginnt mit der Erstellung des Regelbaumes. Hierbei wird die HashMap übergeben. Als erster Schritt wird die erste Regel aus der HashMap genommen und als Root Knoten bestimmt. Danach wird seine Split Ersetzungen durchlaufen und in der HashMap über den Namen gesucht. Diese Regel wird zum Knoten erstellt und bekommt den vorherigen Knoten als Elternknoten eingetragen. Dies geschieht so lange rekursiv, bis es keine nachfolgenden Regeln existieren. Somit wird aus der Regeldatei der Regelbaum.

5.2.3 ShapeGrammar

Diese Klasse ist wie im Kapitel 4.3 beschrieben die Hauptklasse des *grammar* Paket. In dieser Klasse werden die aktuellen Gebäudeeinstellungen abgerufen und die Grundform, dem Quader erstellt. Dieser wird mit den Daten in den Einstellungen erstellt. Danach wird eine Instanz der *SplitGrammar* Klasse erstellt und die Funktion split mit dem Quader und dem Namen des Root-Knotens, aus dem Regelbaum, als Parameter aufgerufen. Das Ergebnis der *split* Funktion ist der Shape-Baum, welcher gespeichert wird.
Als Nächstes wird eine Instanz der *AttributeGrammar* Klasse erstellt und die Funktion *setAttributes* mit dem Shape-Baum als Parameter ausgeführt. Wenn die Funktion abgeschlossen ist, wird der Shape-Baum von der *ShapeGrammar* Klasse zurückgegeben.

5.2.4 SplitGrammar

Die *SplitGrammar* Klasse ist wie im Kapitel 4.3 beschrieben für das Aufteilen der Formen zuständig. Als erstes wird die Grundform als Root-Knoten an den Shape-Baum hinzugefügt. Dieser wird dann verwendet, um die Rekursion einzuleiten.
Im Ersten Schritt wird die Regel zur Form aus dem Regelbaum entnommen. Im nächsten Schritt wird überprüft, ob es sich bei der Methode in der Regel um ein *extrude*,

split mit Parameter f oder dem *split* mit Parameter X oder Y handelt.
Wenn es sich um das *split* mit dem Parameter f handelt wird der Quader je nach Definition in die Seiten unterteilt. Für jede Seite, die in der Regel steht, wird ein Knoten erstellt und an den Elternknoten angehängt.
Wenn es sich um das *extrude* handelt, wird die aktuelle Form genommen und durch die Grundform dem Quader ersetzt, mit den Daten der Form und dem Parameter. Die neue Grundform wird dabei wie beim *split* mit f parameter unterteilt.
Wenn es sich schließlich um das *split* mit dem X- oder Y-Parameter handelt, wird die Form gesplittet.
Zuerst wird der Rest definiert. Dieser ist je nach X oder Y die Weite oder Höhe der Form, da in diese Richtung gesplittet wird. Die Startposition wird auf null gesetzt. Als Nächstes werden alle Ersetzungsregeln nacheinander angewendet, solange der Rest größer null ist und der Iterator kleiner gleich der Anzahl an Ersetzungsregeln. Wenn für die Regel die Wiederholung gesetzt ist, wird der Iterator auf null gesetzt, wenn dieser der Anzahl an Ersetzungen entspricht. Somit werden die Regeln so lange wiederholt, bis keine Ersetzung mit dem Rest übereinstimmt. In der Schleife wird überprüft, ob die Form relativ ist. Wenn dies der Fall ist, wird eine Form erstellt mit je nach X oder Y, mit Höhe oder Breite null und dementsprechend die andere Größe entsprechend der vorherigen Form. Die Form wird als Knoten an den Elternknoten gehängt. Hierbei wird die aktuelle Position beibehalten.
Wenn es sich um eine Form mit fester Größe handelt, wird geprüft, ob dessen Größe kleiner ist als der Rest. Wen dies der Fall ist, wird eine Form erstellt mit der Größe der Regel und der vorherigen Form und wird als Knoten an den Elternknoten gehängt. In diesem Fall wird die Größe auf die aktuelle Position gerechnet, damit die Position der nächsten Form vorhanden ist.
Nachdem die Schleife verlassen wurde und alle Ersetzungen durchlaufen sind, wird der Rest zuerst auf die Formen mit relativer Größe verteilt. Wenn diese nicht existieren, wird der Rest auf die vorhandenen Formen verteilt. Somit stimmt die Form nicht der Regel, doch es entstehen somit keine Lücken.
Dies geschieht solange rekursiv, bis für die letzten Formen keine Ersetzungsregeln existieren. Schließlich wird der Shape-Baum zurückgegeben.

5.2.5 AttributeGrammar

Diese Klasse dient der Attributierung der Formen. Dies erfolgt wie bei der *SplitGrammar* rekursiv. Als Erstes wird die Position der Formen bestimmt. Hierbei wird wie zuvor beschrieben aus der zweidimensionalen Problematik wieder eine dreidimensionale Problematik.
Es wird überprüft, wobei es sich bei der Form handelt. Ob es eine normale Form oder die Grundform ist.
Wenn es sich um eine normale Form handelt, wird durch alle direkten Kindknoten iteriert. Je nach Richtung, welche in dem Elternknoten steht, wird die neue Position berechnet. So wird zum Beispiel, wenn man das Gebäude von vorne betrachtet, die jeweiligen x-Koordinaten und y-Koordinaten addiert. Bei der Rückseite wird die x-Koordinate subtrahiert und die y-Koordinate addiert. So werden die Formen im dreidimensionalen Raum positioniert.
Wenn es sich um die Grundform handelt, ist dieses aus dem *extrude* entstanden. Im *split* wurde die Breite und Höhe wie bei den normalen Formen vergeben und der Parameter ist die Länge. Das heißt, das je nach Richtung die Größen getauscht werden, damit das Herausziehen in die richtige Richtung geschieht. Des Weiteren wird die Gebäudegröße erhöht. Durch das Setzen der neuen Größen werden die Größen der Seiten neu berechnet. Dadurch müssen die Kindknoten dieser Grundform neu gesetzt und gesplittet werden.
Als Nächstes werden die Texturen gesetzt. Dabei wird die Elterntextur auf die Kinderknoten vererbt. Wenn in der Regel eine neue Textur vorkommt, wird diese verwendet. So werden die Shapes attributiert.

5.2.6 Visualisierung

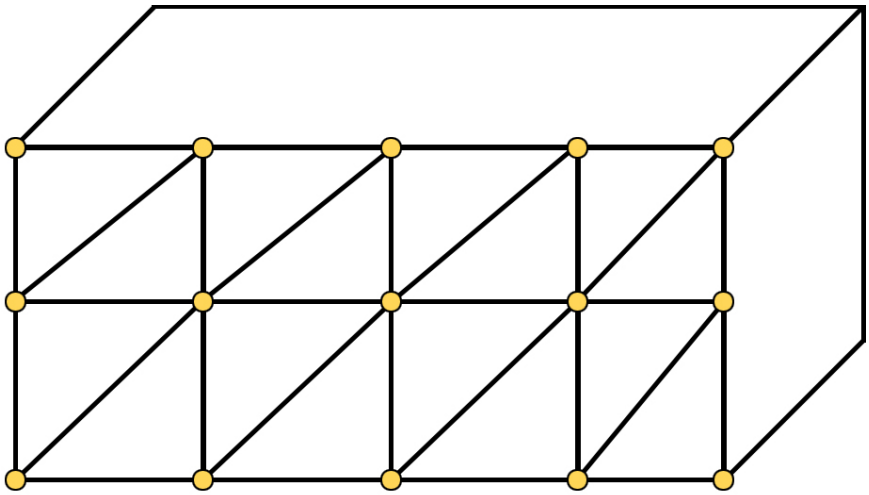

Abbildung 5.8: Beispiel aufbau eines Dreiecksnetzes.

Das *visualize* Paket ist so aufgebaut, das die von dem hier entwickelten Framework auf das Grafikframework zugegriffen wird. Dies geschieht fast nur an dieser Stelle, dadurch kann das Grafikframework schnell ausgewechselt werden.
In dieser Komponente werden die Verticies und das Dreiecksnetz für die Gebäude erstellt. Dabei wird die Komponente durch das Observer Pattern aufgerufen. Dies geschieht, wenn ein Gebäude zu den Gebäudeeinstellungen hinzugefügt wird.
In der Klasse *BuildingVisualizer* wird die *ShapeGrammar* gestartet. Der Shape-Baum wird dann bearbeitet. Es wird rekursiv durch alle Knoten iteriert. Es werden nur die Formen beachtet, welche keine Kindknoten besitzen. Diese sind das Endresultat und werden visualisiert. Je nach Position werden vier Vektoren erstellt. Dabei wird auf den Uhrzeigersinn geachtet, wegen dem Backface Culling. Aus den vier Vektoren werden vier vertices erzeugt. Diese werden verwendet um zwei Dreiecke zu generieren, welche in einem Mesh bestimmt werden. Dieses Mesh wird dem Gebäude hinzugefügt. Nachdem alle Meshes generiert wurden, werden diese an den CgNode gehängt und somit visualisiert.
Ein Beispiel wie ein Dreiecksnetz an der Vorderansicht aufgebaut ist, sieht man in Abbildung 5.8.

5.3 Tests

Es werden hauptsächlich drei Pakete getestet, welche am Fehleranfälligsten sind.
Das Paket, welches die Regeln einliest, ist das erste der drei Pakete. Hierbei ist es wichtig, dass die Regeln richtig geparst und in den Regelbaum überführt werden. Wenn hier ein Fehler geschieht, dann können unerwünschte Darstellungsfehler bei der Generierung des Gebäudes auftreten.

Das *SplitGrammar* Paket ist das zweite Paket. Hierbei ist es wichtig, das geprüft wird, ob die Unterteilung in die sechs Quader-Seiten richtig funktioniert. Zusätzlich ist es erforderlich, dass die einzelnen Seiten nach der Regel unterteilt werden. Dies ist ein Prozess, in dem sich viele Fehler einschleichen können, daher muss diese Stelle gut getestet werden. Hier muss man bedenken, dass an dieser Stelle aus dem dreidimensionalen Problem ein zweidimensionales Problem gemacht wird, um dieses leichter zu lösen.

Das *AttributeGrammar* Paket ist das dritte Paket. Dabei ist es wichtig, dass die Texturen und die Koordinaten richtig gesetzt werden. Diese müssen von zweidimensionalen in dreidimensionale Koordinaten überführt werden. Somit wird wieder aus einem zweidimensionalen Problem ein dreidimensionales Problem. Damit die Shapes an der richtigen Position gesetzt werden, ist es wichtig, dass diese Komponente richtig funktioniert. Test sind auch in diesem Fall wieder von großer Wichtigkeit.

6 Evaluation

In diesem Kapitel wird der Prototyp im Zusammenhang mit der Shape-Grammar evaluiert. Im ersten Abschnitt werden die in der Anforderungsanalyse angesprochenen Punkte untersucht. In den danach folgenden zwei Abschnitten wird die Performance der Gebäude- und Stadtgenerierung anhand von Messungen mit verschiedenen Einstellungen analysiert.

6.1 Evaluierung anhand der Anforderungsanalyse

6.1.1 Evaluierung der Formalen Grammatik

```
import("./basic rules/basicrule.rule");

Building -->
    split(f) { left, right, back : Facade | front : Front| top : Roof };

WindowPart -->
    split(x) { 0.5 : WindowRest | 1 : Window | 0.5 : WindowRest };

Window -->
    texture("./building2/textures/window.jpg");
```

Abbildung 6.1: Auflistung einiger Regeln.

Zuerst werden die Anforderungen an die formale Grammatik untersucht, welche im Kapitel 3.3 erläutert werden.
Im ersten Punkt geht es um die Lesbarkeit der Grammatik. Dies erfolgt durch die konsequente Trennung von Regel- und Variablensymbolen mit dem „–>“ Symbol. Des Weiteren werden die Regelblöcke getrennt verfasst. Dabei wird ein Block mit einem Semikolon abgeschlossen. Die Funktionen der Regeln werden dabei in einzelnen Zeilen mit einem Tabulator Abstand eingerückt. Somit wird mit der Grammatik eine hohe Lesbarkeit erreicht.
Die nächste Anforderung ist die Ähnlichkeit der Chomsky-Grammatik. Da die Gram-

matik der kontextfreien Sprache aus der Chomsky-Hierarchie ähnelt, ist dies gegeben. Dies folgt aus der Tatsache, dass auf der linken Seite der Grammatik nur Variablen vorkommen und auf der rechten Seite Terminalsymbole, sowie Variablen vorhanden sind. Die Anforderung für das Bestimmen der Textur wird mit der Funktion *texture* erfühlt. Dabei ist der Parameter der Pfad zur Textur. Des Weiteren wurde die *extrude* Funktion für das Herausziehen von Quadern aus der Grundform hinzugefügt. Dabei erfüllt der Parameter die Voraussetzung der Größenangabe. Mit der *split* Funktion wird die Eigenschaft der Shape-Grammar erfühlt.
Somit erfühlt die Grammatik sämtliche beschriebene Anforderungen.

6.1.2 Evaluierung Gebäudegenerierung

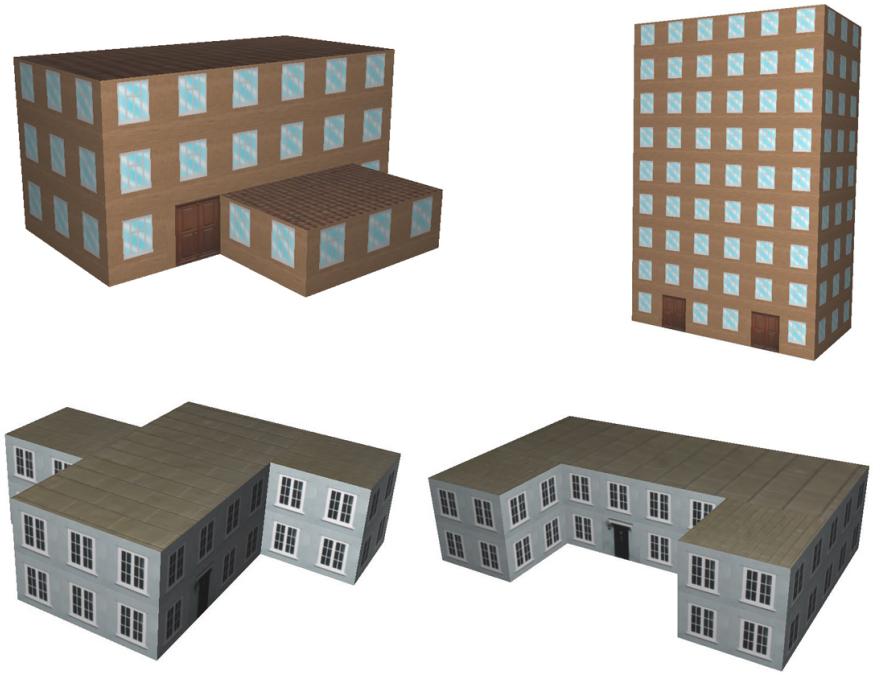

Abbildung 6.2: Beispiel verschiedene Hausformen. Texturen [tex15a, Car15, AGF15, mb315, tex15c, tex15b]

In der ersten Anforderung geht es um die geringe Anzahl an Klicks und Eingaben welche benötigt werden, um ein Gebäude zu generieren. Um ein Gebäude im Prototyp zu generieren, benötigt der Benutzer maximal sechs Maus-Klicks. Hierbei wurde das Wechseln der Eingabefelder mitgezählt. Für die Eingaben werden nur die Breite, Länge, Höhe

und die Regel benötigt. Somit ist die Anforderung der geringen Klicks und Eingaben erfüllt.
In der nächsten Anforderung geht es um die Veränderung der Grundform. Mit der *extrude* Funktion können Blöcke aus der Grundform gezogen werden. Somit ist es möglich, verschiedene Formen zu erzeugen. Dadurch sind Formen, wie dem einfachen Quader, der L-Form bis hin zur U- und T-Form, möglich. Dies erlaubt somit eine individuelle Bestimmung der Grundform, solange dies im 90-Grad-Winkel geschieht. Ein Beispiel von den Formen sieht man in Abbildung 6.2.
In der nächsten Anforderung geht es um die genaue Umsetzung der Grammatik, sodass diese sich im Gebäude widerspiegelt. Die Grammatik wird genau eingelesen und umgesetzt. Dies geschieht rekursiv und der Reihe nach, sodass es zu keiner Abweichung kommt.

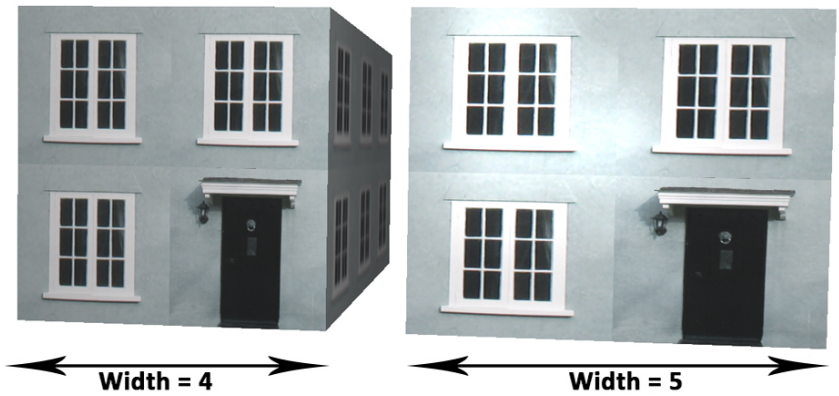

Abbildung 6.3: Beispiel der Verarbeitung der Restgröße. Texturen [tex15a, Car15]

Die Restgröße wird so verarbeitet, wie es in der Anforderung beschrieben wurde. Ein Beispiel wird dabei in Abbildung 6.3 gezeigt. Bei dieser Abbildung kann man erkennen, dass die rechte Seite leicht in die Breite verzehrt wird. Dadurch wird kenntlich gemacht, dass der Rest auf die Formen verteilt wurde. Die Perfomance-Anforderungen werden im Kapitel 6.2 behandelt.

6.1.3 Evaluierung Stadtgenerierung

In der ersten Anforderung geht es um die minimale Anzahl an Maus-Klicks, welche zur Erzeugung der Stadt benötigt werden. Hierbei sind es bei der Auswahl einer Regel maximal elf Klicks. Je nach Auswahl der Gebäude-Regeln können diese mehr werden. Dabei wird das Wechseln der Eingabefelder mitgezählt. Somit hält sich die Anzahl in Grenzen der Anforderung. Des Weiteren geht es bei der Anforderung um wenige Eingaben, welche benötigt werden, um die Stadt zu generieren. Hierbei werden, wie in der Anforderung beschrieben, die minimalen und maximalen Größenangaben, sowie die Auswahl der Regeln und die Anzahl an Gebäuden benötigt.
Für die zweite Anforderung, dass die Gebäude eine große Varianz aufweisen sollen, werden zwischen der minimalen und maximalen Größe Zufallszahlen erzeugt. Somit haben die Gebäude zufällige Größen. Außerdem werden die Regeln zufällig bestimmt. Dies bedeutet, je mehr Regeln verwendet werden, desto besser wird das Ergebnis. Die Erfüllung der Anforderung ist somit abhängig von der Anzahl ausgewählter Regeln. Bei der Positionierung wurde die Methodik verwendet, welche in der Anforderung in Kapitel 3.5 beschrieben wird. Somit werden die Gebäude in Reihen positioniert. Die

Perfomance wird im Kapitel 6.3 behandelt.

6.2 Performance Gebäudegenerierung

In diesem Abschnitt werden die Geschwindigkeit und die Speicherkosten des Prototyps bei der Gebäudegenerierung betrachtet. Dabei werden Testläufe mit verschiedenen Einstellungen durchgeführt. Dabei wird die Anzahl an Meshes betrachtet. Somit können diese Aspekte evaluiert werden.

6.2.1 Zeitaufwand

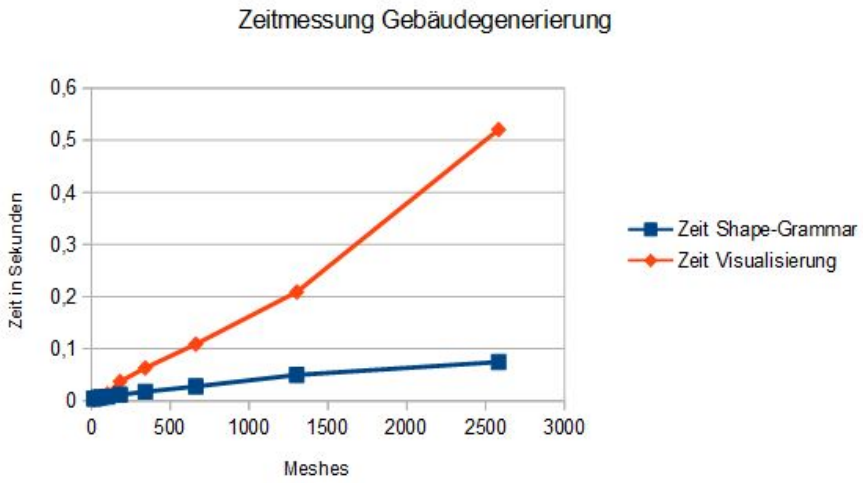

Abbildung 6.4: Diagramm von der Zeitmessung der Gebäudegernerierung.

In diesem Bereich wird der Zeitaufwand der Gebäudegenerierung betrachtet. Dabei ist zu beachten, dass die Zeit in Sekunden angegeben ist und ein Mesh aus genau zwei Dreiecken besteht. Es wurde dabei die Ausführung der Shape-Grammar und der Visualisierung getrennt betrachtet. Für diese Analyse wurde ein Diagramm erstellt, welches in Abbildung 6.4 gezeigt wird.
In diesem Diagramm sieht man, das am Anfang bei wenig Meshes die Zeit verschwindend gering ist. Dabei wird deutlich, dass die Visualisierung ab 200 Meshes beim Zeitaufwand deutlich höher ist, als bei der Ausführung der Shape-Grammar. Selbst bei Meshes über 2500 ist die Zeit noch deutlich unter einer Sekunde. Somit ist die Anforderung, die bei zwei Sekunden liegt, bei weiten erfühlt.

6.2.2 Speicherbedarf

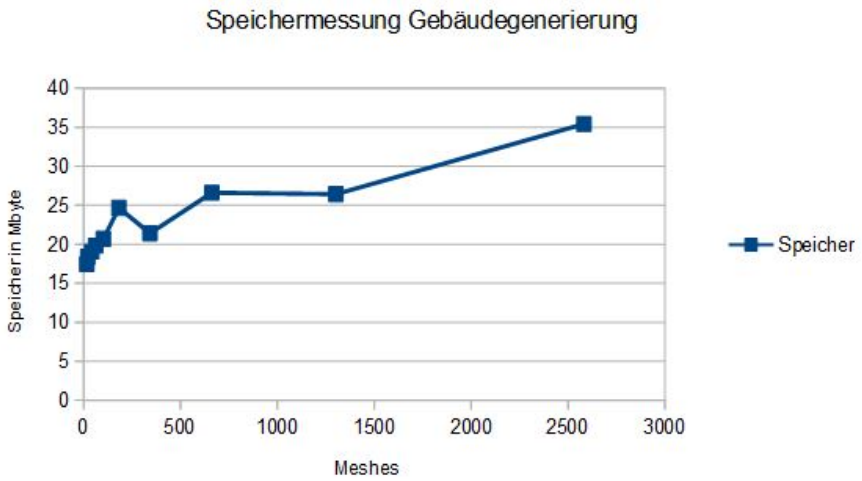

Abbildung 6.5: Diagramm von der Speichermessung der Gebäudegernerierung.

In diesem Abschnitt wird der Speicherbedarf betrachtet. Hierbei wird der Speicher in MByte angegeben. Dabei kann man in Abbildung 6.5 erkennen, dass der Speicher nur geringfügig zunimmt. Die Einbrüche, welche im Diagramm vorkommen, liegen dabei an die Garbage Collection von Java. Trotz dieser Einbrüche wird deutlich, dass der Speicherbedarf relativ konstant zunimmt. Da dieser bei über 2500 Meshes knapp über 35 MByte beträgt und somit gering ist, ist die Anforderung welche in Kapitel 3.4 gestellt wurde, erfüllt.

6.3 Performance Stadtgenerierung

In diesem Abschnitt werden die Geschwindigkeit und die Speicherkosten des Prototyps bei der Stadtgenerierung betrachtet. Hierbei werden wie in Kapitel 6.2 Testläufe mit verschiedenen Einstellungen durchgeführt. Somit kann die Performance bei der Stadtgenerierung evaluiert werden.

6.3.1 Zeitaufwand

Bei der Stadtgenerierung werden zwei Diagramme für die Zeitmessung erstellt. Einmal zum Verhältnis der Meshes und einmal zum Verhältnis der Anzahl an Gebäuden. Somit

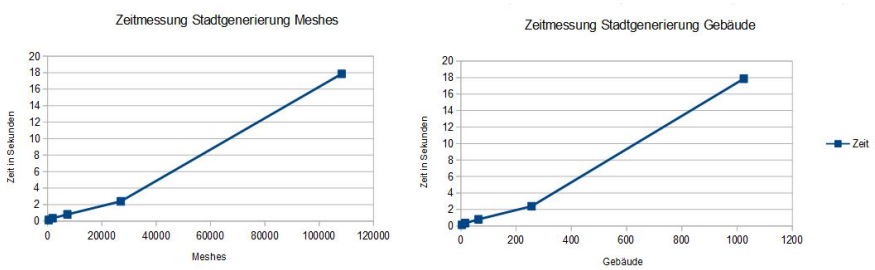

Abbildung 6.6: Diagramm von der Zeitmessung der Stadtgenerierung.

erhält man einen guten Überblick über den Zeitaufwand.
Hierbei kann man besser als bei der Gebäudegenerierung sehen, dass die Zeit relativ konstant zunimmt. Da in den Anforderungen 81 Gebäude als Maßstab gegeben sind, ist dieses Ergebnis sehr beeindruckend. Dabei kann man erkennen, dass knapp 250 Gebäude nur knapp zwei Sekunden benötigen. Somit ist die Anforderung erfüllt. Doch bei Gebäudezahlen über 1000 dauert die Generierung knapp 18 Sekunden.

6.3.2 Speicherbedarf

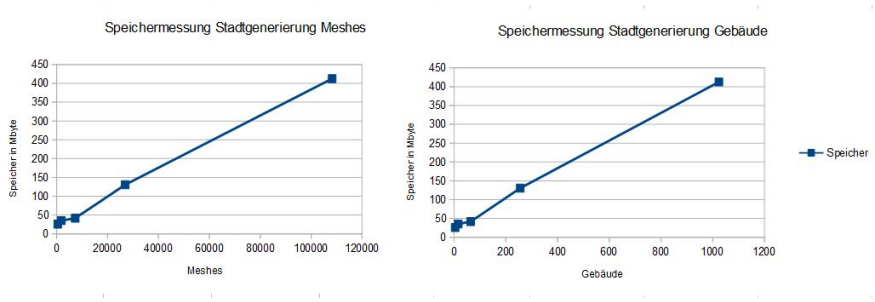

Abbildung 6.7: Diagramm von der Speichermessung der Stadtgenerierung.

Bei dem Speicherbedarf ist es ähnlich wie bei dem Zeitaufwand für die Stadtgenerierung. Hierbei wird deutlich, dass auch der Speicherbedarf konstant zunimmt. Dabei beträgt der Speicherbedarf bei über 1000 Gebäuden schon knapp über 400 MByte. Da es jedoch in dieser Arbeit darum geht, ein Teil einer Stadt zu generieren und nicht die ganze Stadt, ist die Anforderung erfühlt. Denn wir betrachten hier wieder die Anforderung von 81 Gebäuden, bei denen der Speicherbedarf sehr gering ist.

7 Fazit und Ausblick

7.1 Zusammenfassung

In dieser Arbeit wurde die Thematik der prozeduralen Generierung von Gebäuden behandelt. Dies basiert auf der Shape-Grammar. Hierbei geht es um die automatische Generierung eines Gebäudes, welches zuvor in einer Form von Mustern dargestellt wird. Dabei wird klar, dass diese Methodik eine große Möglichkeit bietet, jedoch in dieser Arbeit nur ein geringer Teil abgedeckt wird. Es wird bis jetzt nur einfache Gebäude generiert und in Form eines Stadtteils dargestellt. Doch dies zeigt schon beeindruckende Szenen.
Somit bietet diese Form von Modellierung ein großes Potenzial. Denn es ist ein sehr großer Aufwand, Städte komplett von Hand zu modellieren, wenn die Häuser noch dazu eine große Varianz aufweisen sollen. Wenn diese Methodik immer besser wird und somit die dreidimensionalen Städte immer realistischer werden, wird diese Form von Modellierung die Oberhand gewinnen. Denn es wird kein gesamtes Team aus Designern benötigt, welche die Stadt modellieren. Es werden nur noch wenige Spezialisten, die sich mit der Software und der Grammatik auskennen, benötigt.
Die Shape-Grammar ähnelt dabei der Chomsky-Grammatiken und basiert auf dem L-System. Doch anders, wie beim L-System, erfolgen die Ersetzungen sequenziell. Somit bleibt der Überblick erhalten und beim Ersetzen können so keine Fehler entstehen. Die Shape-Grammar ist daher ein Ersetzungssystem und basiert auf Formen. Dies bedeutet, das Formen durch Formen ersetzt werden, wie es in der Grammatik beschrieben wird.

Im praktischen Teil der Arbeit geht es um die Evaluation der Shape-Grammar. Dabei wird ein Prototyp entwickelt, mit dem es möglich ist, Gebäude über eine Grammatik automatisch zu generieren. Für die Visualisierung wird das Framework CgResearch von Professor Dr. Jenke verwendet. Mit dem Prototyp wird überprüft wie viel Leistung die Shape-Grammar benötigt. Des Weiteren können die Möglichkeiten, welche die Shape-Grammar bietet, überblickt werden.

Hierbei wurde als Erstes eine geeignete Form für die Grammatik gesucht. Dabei war der erste Gedanke, die Grammatik in Form von XML zu gestalten, da diese leicht geparst werden kann. Doch die Übersicht nahm bei hoher Komplexität der Grammatik stark ab. So wird die Chomsky Form für die Grammatik gewählt. Dadurch wird das Parsen komplexer, doch die Lesbarkeit für den Anwender ist dabei stark verbessert.

Im nächsten Schritt wurde die Ordnerstruktur definiert. Diese dient dem Einlesen der Gebäude. Durch diese Struktur ist es einfach möglich, die Gebäude wie in einem Plug-in System hinzuzufügen. Somit können die Regeln und Gebäude einfach erweitert und geändert werden.

Im danach folgenden Schritt wurde das Paket erstellt, welches für das Einlesen der Grammatik zuständig ist. Hierbei handelt es sich um das Einlesen von Strings und deren Untersuchungen.

Im darauf folgenden Schritt wurde das Shape-Grammar Paket aufgebaut. Dies besteht aus drei Klassen. Der *ShapeGrammar*, *SplitGrammar* und die *AttributeGrammar* Klasse. Die *ShapeGrammar* ist dabei die Hauptklasse, welche die anderen beiden steuert. Die *SplitGrammar* übernimmt dabei das Unterteilen der Formen und somit das Ersetzen dieser. Dabei ist die Grundform ein Quader, welches in seine Seiten unterteilt wird. Hierbei wird aus dem dreidimensionalen Problem fürs Erste ein zweidimensionales Problem. Die *AttributeGrammar* ist dabei für die Attributierung der Formen verantwortlich. In dieser Klasse werden die Formen im dreidimensionalen Raum positioniert. Somit wird das zweidimensionale Problem wieder zum dreidimensionalen Problem überführt. Des Weiteren werden an dieser Stelle die Texturen vergeben, welche in der Grammatik definiert sind.

Schließlich folgt die Visualisierung. Diese sorgt für die Darstellung des Gebäudes, welches zuerst von der Shape-Grammar generiert wird.

7.2 Ausblick

Es gibt im Prinzip mehrere Möglichkeiten, auf dieser Arbeit aufzubauen. Die eine Möglichkeit wäre es, den Prototypen in mehreren Aspekten zu erweitern und zu verbessern.

Dies könnte zum einen das Einlesen der Grammatik sein. Dies geschieht im Prototyp sehr einfach. Dies könnte zum Beispiel mit Antlr von Terence Parr[Ter15] verbessert werden. Mit dieser Software wäre es möglich, die Grammatik zu definieren. Dabei werden Java Klassen generiert, welche zum einen die Grammatik auf Richtigkeit überprüfen und zum anderen das Erstellen des Regelbaums übernehmen, wenn dies so definiert wird. Somit könnte ein Editor für die Grammatik in den Prototypen eingebaut werden und die Erstellung der Grammatik erleichtern. Ein weiterer Vorteil ist es, dass die Datei vor dem Lesen auf Richtigkeit überprüft werden kann.
Eine zweite Verbesserung wäre es, dass komplexere Formen über die Grammatik erstellt werden können. Mit diesen Formen könnte man auf das Einlesen von dreidimensionalen Dateien verzichten. Somit könnten noch schönere Gebäude mit vielen Details generiert werden.

Um die Stadtgenerierung zu verbessern, könnte eine geeignete Datenstruktur mit Verwaltung entwickelt werden. Diese könnte dafür sorgen, dass nur Gebäude in einem bestimmten Radius angezeigt werden. Sämtliche Gebäude, welche sich außerhalb des Radius befinden, könnten extern gespeichert werden und nur bei Bedarf geladen und visualisiert werden. Dies würde ermöglichen, dass eine gesamte Stadt generiert werden kann. Des Weiteren könnte die Stadtgenerierung parallelisiert werden. Dies würde es ermöglichen, dass mehrere Gebäude gleichzeitig generiert und platziert werden. Zudem könnte man die Positionierung der Gebäude mit dem L-System vornehmen, sodass die Gebäude wie in einer richtigen Stadt platziert werden. Dies würde das Aussehen der Stadt erheblich verbessern.

Die Erweiterung der Oberfläche wäre eine Optimierung, sodass es möglich ist, das Gebäude über die Maus zu vergrößern oder zu verkleinern. Somit könnten an dem generierten Gebäude die Maße verändert werden, um ein optimales Resultat zu erhalten.

Der Prototyp könnte zum Beispiel mit Gebäudedaten vom Landesamt für Vermessung verwendet werden. Die Daten könnten abgerufen werden und die Gebäude anhand der Maße generiert werden. Somit könnten Städte generiert werden, welche einer echten Stadt ähneln, auch wenn nur die Position und die Maße der Gebäude stimmen.
Dazu könnte ein Scanner entwickelt werden, der für den Prototyp die Grammatik anhand der Fassaden erzeugt. Somit könnten Gebäude aus der realen Welt in die virtuelle Welt übertragen werden.

Schließlich könnte eine Software entwickelt werden, welche anhand von Regeln die Grammatik prozedural generiert. Somit könnte eine große Auswahl an Gebäuden erzeugt werden. Somit wäre es noch effizienter, eine Stadt zu generieren, welche eine große Varianz an Gebäuden aufweist.

In dieser Arbeit ging es um die Erschaffung einer Grundlage der Generierung von Gebäuden anhand einer Shape-Grammar und deren Evaluierung. Dabei wurde ein Prototyp entwickelt, um dies zu prüfen und die Komplexität abzumessen. Somit können viele Arbeiten auf dieser aufbauen.

Literaturverzeichnis

[AGF15] AGF81. Eine türtextur vom gebäude von agf81 cc by 3.0. "http://agf81.deviantart.com/art/Door-Texture-30-288791840", März 2015. (letzter Abruf am 1.3.2015).

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* xford University Press, New York, 1977.

[Bil96] Hillier Bill. *Space Is The Machine: A Configurational Theory Of Architecture.* Cambridge University Press, 1996.

[Car15] Caroline. Bild von einem haus aus dem die texturen für das gebäude erstellt wurden von caroline cc by 2.0. "https://flic.kr/p/88Y5DQ", Februar 2015. (letzter Abruf am 5.2.2015).

[Dok14] Mark Dokter. *Deriving Shape Grammars on the GPU.* Institute for Computer Graphics and Vision, 2014.

[DSW94] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages : Fundamentals of Theoretical Computer Science.* Academic Press, 1994.

[Dua02] José Pinto Duarte. *Towards the mass customization of housing: the grammar of Siza's houses at Malagueira.* PhD thesis, MIT School of Architecture and Planning, 2002.

[Esr15] Esri. Cityengine. "http://www.esri.com/software/cityengine", 2015. (letzter Abruf am 13.2.2015).

[Fle87] U Flemming. *More than the sum of its parts: the grammar of queen anne houses. Environment and Planning B 14, 323–350.* 1987.

[GK15] Dr. Björn Ganster and Prof. Dr. Reinhard Klein. Prozedurale modellierung - uni-bonn. "http://cg.cs.uni-bonn.de/de/projekte/prozedurale-modellierung/", März 2015. (letzter Abruf am 5.2.2015).

[HF03] Sven Havemann and Dieter W. Fellner. *Generative Mesh Modeling*. PhD thesis, TU Braunschweig, 2003.

[Koe15] Astrid Koennecke. Lindenmayer systeme - fu berlin. "http://www.inf.fu-berlin.de/lehre/WS10/ProSem-ThInf/LindenmayerZsf.pdf/", Januar 2015. (letzter Abruf am 16.1.2015).

[LJD+01] Legakis, Justin, Julie Dorsey, , and Steven Gortler. *Feature-based cellular texturing for architectural models*. In Proceedings of ACM SIGGRAPH 2001, ACM Press, E. Fiume, Ed., 309–316, 2001.

[mb315] mb3d.co.uk. Eine fenstertextur vom gebäude von mb3d. "http://www.mb3d.co.uk/mb3d/Doors_and_Windows_Seamless_and_Tileable_High_Res_Textures_files/Window_05_H_CM_1.jpg", März 2015. (letzter Abruf am 1.3.2015).

[Mic15] Microsoft. Xna. "https://msdn.microsoft.com/dn629515", 2015. (letzter Abruf am 25.01.2015).

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990.

[SG72] George Stiny and James Gips. *Shape Grammars and the Generative Specification of Painting and Sculpture*. C V Freiman (ed.) Information Processing 71, Amsterdam, 1972.

[Ter15] Terence Parr. Antlr. "http://www.antlr.org/", 2015. (letzter Abruf am 15.04.2015).

[tex15a] texturelib.com. Erste dachtextur vom gebäude texturelib.com. "http://texturelib.com/texture/?path=/Textures/roof/roof_0020", Februar 2015. (letzter Abruf am 5.2.2015).

[tex15b] texturelib.com. Wandtextur vom gebäude texturelib.com. "http://texturelib.com/texture/?path=/Textures/brick/modern/brick_modern_0085", März 2015. (letzter Abruf am 1.3.2015).

[tex15c] texturelib.com. Zweite dachtextur vom gebäude texturelib.com. "http://texturelib.com/texture/?path=/Textures/roof/roof_0085", März 2015. (letzter Abruf am 1.3.2015).

[Uni15] Unity Technologies. Unity. "https://unity3d.com/", 2015. (letzter Abruf am 25.01.2015).

[WWSR03] Peter Wonka, Michael Wimmer, Francois Sillionn, and William Ribarsky. *Instant Architecture*. ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2003, 2003.

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

Hamburg, 27. Mai 2015 Thorben Watzl