HAW HAMBURG

Bachelorarbeit

Mathias Holtmeier

Beispiel-getriebene prozedurale Gebäude-Generierung unter Anwendung einer Shape-Grammar

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Mathias Holtmeier

Beispiel-getriebene prozedurale Gebäude-Generierung unter Anwendung einer Shape-Grammar

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Bettina Buth

Eingereicht am: 18.08.2020

Mathias Holtmeier

Thema der Arbeit

Beispiel-getriebene prozedurale Gebäude-Generierung unter Anwendung einer Shape-Grammar

Stichworte

Prozedurale Generierung, Shape-Grammar, formale Grammatik, Gebäude-Generierung, Stadt-Generierung, Dreiecksnetz

Kurzzusammenfassung

Diese Arbeit befasst sich mit der prozeduralen Generierung von Gebäuden. Hierfür wird eine formale Grammatik auf Basis von geometrischen Formen, die Shape-Grammar, verwendet. Die theoretischen Grundlagen für diese Grammatik, deren Funktionen und Aufbau werden in dieser Arbeit erläutert. Es wird an einem bestimmten Baustil gezeigt, dass mit geringem Aufwand viele Gebäude eines Typs mit hoher Varianz generiert werden können.

Mathias Holtmeier

Title of Thesis

Example driven procedural generation of buildings based on Shape-Grammar

Keywords

procedural generation, Shape-Grammar, formal grammar, generation of buildings, generation of cities, triangle mesh

Abstract

This work deals with the procedural generation of buildings. A formal grammar based on geometric shapes, the Shape-Grammar, is used for this. The theoretical basis for this grammar, its functions and structure are explained in this work. It is shown on a certain architectural style that many buildings of one type with high variance can be generated with little effort.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Prozedurale Generierung wird häufig dazu verwendet um Modellierungsprozesse zu automatisieren und wird heute in den unterschiedlichsten Bereichen eingesetzt. In der Computergrafik werden so unter anderem Pflanzen, Gebäude und Landschaften erstellt. Die Computerspiel-Branche bedient sich dieser Technik beim Leveldesign, um zum Beispiel die Umgebung innerhalb eines Spieles zu generieren. Darüber hinaus gibt es bereits Softwareanwendungen (zum Beispiel CityEngine [1]) im Bereich der Stadtplanung, die es ermöglichen 3D-Modelle von Städten inklusive deren Gebäude und Straßen sowie die umgebene Landschaft zu erstellen.

Mit dem Ansatz der prozeduralen Generierung sollen bei der grafischen Erstellung Zeit und Kosten eingespart werden, indem auf aufwändige manuelle Modellierung verzichtet wird.

Viele wissenschaftliche Arbeiten in diesem Bereich beschäftigen sich mit der Generierung von Gebäuden und Städten. Die zugrunde liegende Grammatik ist die sogenannte Shape-Grammar. Aktueller Stand der Technik ist das Framework CGA Shape [8] mit dessen Hilfe sehr detailliert Gebäude und Städte erstellt werden können und dessen theoretische Konzepte die Basis für diese Arbeit darstellen.

Diese Arbeit soll die Frage untersuchen, inwieweit die Generierung von mehreren Gebäuden in Form eines Stadtviertels in einem bestimmten Baustil unter Anwendung einer Shape-Grammar möglich ist. Ziel der Arbeit ist es ein Systems zu entwickeln, welches ein Stadtviertel im Stil der Lübecker Kaufmannshäuser vom 12. bis zum 19. Jahrhundert automatisch generiert. Im Rahmen dieses Projekts soll ein bestehendes System erweitert werden, sodass mehrere Gebäude anhand von vorgegebenen Grundrissen im oben genannten Stil gleichzeitig generiert werden. Das System soll zudem zwei Hauptaufgaben erfüllen. Zum Einen soll durch Verwendung von Regeln und auf Grundlage eines Axioms ein Gebäude erstellt werden. Zum Anderen sollen mehrere Gebäude auf einmal erzeugt werden, um die Varianz der generierten Gebäude zu veranschaulichen. Der Fokus liegt

dabei auf der Erstellung von Gebäuden, das heißt weitere Details wie Straßen und Bäume werden nicht generiert. Außerdem wird bei der Gebäudegenerierung der Schwerpunkt auf die Darstellung der Volumina gelegt, Details wie Texturen sind zu vernachlässigen.

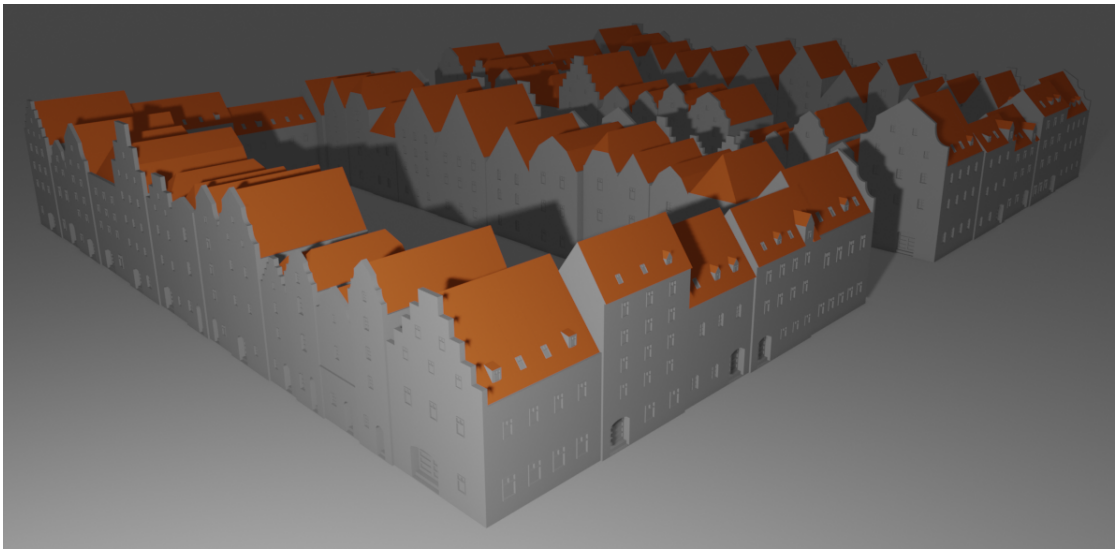


Abbildung 1.1: Eine mit dem System generierte Stadt

Die Arbeit besteht aus sieben Kapiteln. Im zweiten Kapitel werden die grundlegenden Techniken zur prozeduralen Generierung erklärt, definiert und anhand von Beispielen erläutert. Zudem wird im dritten Kapitel ein Überblick über bereits vorhandene Arbeiten zum Thema prozedurale Gebäudegenerierung geschaffen. Im vierten Kapitel Konzept erfolgt zunächst eine theoretische Einführung in die dem Projekt zugrunde liegenden Konzepte und Funktionen. Zudem werden die funktionalen Anforderungen an die Gebäude- und Stadtgenerierung beschrieben und erklärt. Zuletzt erfolgt im Rahmen des Abschnittes ein Überblick über die notwendigen Software-Komponenten. Das fünfte Kapitel beschäftigt sich mit der Umsetzung der im vierten Kapitel beschriebenen Anforderungen. Hierbei werden notwendige Anpassungen der bereits vorhandenen Shape-Grammar erläutert und anhand von Beispielen erklärt, wie die einzelnen Kaufmannshäuser erstellt wurden. Zudem wird auf die Stadtgenerierung eingegangen und wie der Benutzer anhand einer GUI die Software bedienen kann. Innerhalb des sechsten Kapitels Evaluation erfolgt eine Bewertung des erstellten Systems anhand der zuvor gestellten Anforderungen und einer Performance-Messung. Im siebten und letzten Kapitel wird ein Fazit der Arbeit gezogen und ein Ausblick über eine mögliche weitere Verwendung der Arbeit gegeben.

2 Grundlagen

In diesem Kapitel werden grundlegende Techniken zur prozeduralen Generierung erklärt, definiert und anhand von Beispielen erläutert.

2.1 Kontextfreie Grammatik

2.1.1 Definition

Eine kontextfreie Grammatik ist ein 4-Tupel $G = (V, \sum, R, S)$ [14], wobei

1. V eine endliche Menge an Variablen ist,
2. $\sum$ eine endliche Menge (disjunkt von V) von Terminalsymbolen ist,
3. $R \subset V \times (V^* \cup \sum^*)$ eine endliche Menge von Ersetzungsregeln ist und
4. $S \in V^+$ das Axiom ist.

2.1.2 Beschreibung

Eine Grammatik besteht aus einer Menge von Ersetzungsregeln, die sequentiell ausgeführt werden. Jede Regel besteht aus einer Variablen V und einer Zeichenkette ω getrennt von einem Pfeil: $V \rightarrow \omega$. Die Zeichenkette ω besteht aus Variablen und Terminalsymbolen. Nachfolgend stehen die Regeln einer beispielhaften kontextfreien Grammatik G_1.

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow T$$

Eine Variable einer kontextfreien Grammatik ist das Axiom, im Beispiel G_1 ist das die Variable A. Die Variablen sind $V = (A, B)$ und die Terminalsymbole sind $\sum = (0, 1, T)$. Die Folge der Ersetzungen, um eine Zeichenkette von Terminalsymbolen zu erhalten, heißt Herleitung. Die Herleitung in der Grammatik G_1 für die Zeichenkette 000T111 ist:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000T111$$

Diese Technik lässt sich auch auf grafische Elemente übertragen, wie in Abbildung 2.1 dargestellt. Der Initiator entspricht dem Axiom und in jedem Herleitungsschritt werden die einzelnen Linien durch den Generator ersetzt.

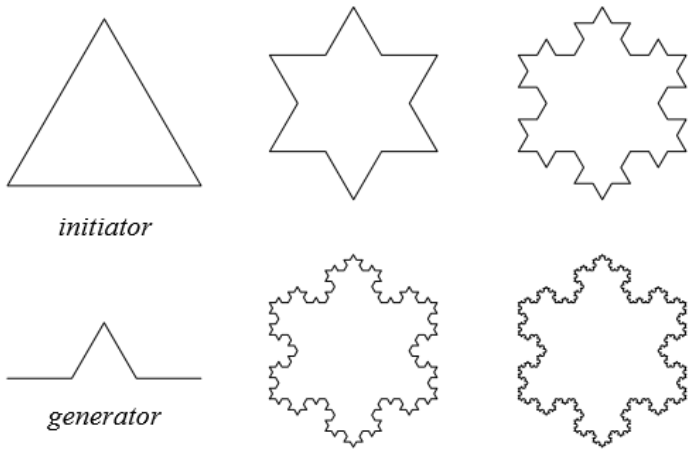


Abbildung 2.1: Konstruktion einer Kochschen Schneeflocke [12]

2.2 L-Systeme

2.2.1 Definition

L-Systeme sind ein 3-Tupel $G = (V, \omega, P)$ [12]

- V ist das Alphabet des L-Systems.
- $\omega \in V^+$ ist ein nichtleeres Wort, das Axiom.
- $P \subset V \times V^*$ ist eine endliche Menge von Ersetzungsregeln.

2.2.2 Beschreibung

L-Systeme wurden anfangs als eine mathematische Theorie zur Pflanzenentwicklung konzipiert. Entwickelt wurde diese von dem Biologen Aristid Lindenmayer. In der Computergrafik wurde diese Theorie erweitert um Pflanzen zu modellieren [12]. Das zentrale Konzept von L-Systemen ist das Ersetzen. Eine initiale Form ist gegeben und wird mit einer Menge von Ersetzungsregeln verändert (siehe Abbildung 2.1). Dieses Konzept des Ersetzens ist ähnlich wie in den kontextfreien Grammatiken. Im Unterschied zu kontextfreien Grammatiken werden die Ersetzungsregeln jedoch parallel und nicht sequentiell ausgeführt. Zudem gibt es ein weiteres Konzept der L-Systeme, welches in Bereichen der Computergrafik angewendet wird. Dieses Konzept basiert auf der Interpretation eines Strings als sogenannte Turtle-Grafik (siehe Abbildung 2.2).

Die Befehle für die Turtle-Grafik sind:

F : *einen Schritt vorwärts gehen und eine Linie malen*

f : *einen Schritt vorwärts gehen ohne eine Linie zu malen*

$+$: *Drehung nach links um den Winkel* δ

$-$: *Drehung nach rechts um den Winkel* δ

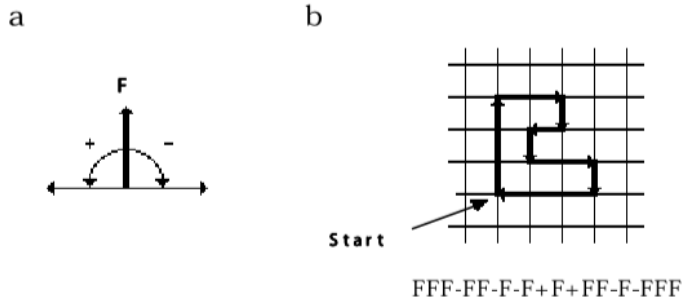


Abbildung 2.2: (a) Turtle-Grafik Interpretation der String Symbole: F,+,-. (b) Beispiel-Interpretation eines Strings. Die Erhöhung des Winkels beträgt 90°. Anfangs ist die Turtle nach oben ausgerichtet. [12]

2.3 Shape-Grammar

2.3.1 Definition

Eine Shape-Grammar besteht aus vier Komponenten $G = (S, L, R, I)$: [15]

1. S: einer endlichen Menge an Shapes,
2. L: einer endlichen Menge an Symbolen,
3. R: einer endlichen Menge an Shape-Rules (Produktionsregeln) in der Form $\alpha \rightarrow \beta$, wobei α eine Shape mit Symbol in der Menge $(S, L)^+$ und β eine Shape mit Symbol in der Menge $(S, L)^*$ ist,
4. I: einer initialen Shape, die eine Shape mit Symbol in der Menge $(S, L)^+$ ist.

2.3.2 Beschreibung

Shape-Grammar wurde 1971 von George Stiny und James Gips entwickelt [16]. Diese wird für die Generierung von Figuren (Shapes) im zwei- und dreidimensionalen Raum verwendet. Wie bei L-Systemen können die Produktionsregeln oder auch Shape-Rules parallel ausgeführt werden. Die Symbole sind IDs, die eindeutig jeder Shape zugeordnet werden können. Über die Symbole werden Abhängigkeiten unter den Shapes definiert, zum Beispiel nach Ausführung der Regel $\alpha \rightarrow \beta$ ist β eine Subshape von α. Subshapes sind, durch eine Transformation im geometrischen Raum entstandene, von der initialen Shape (mehrfach) abgeleitete Shapes.

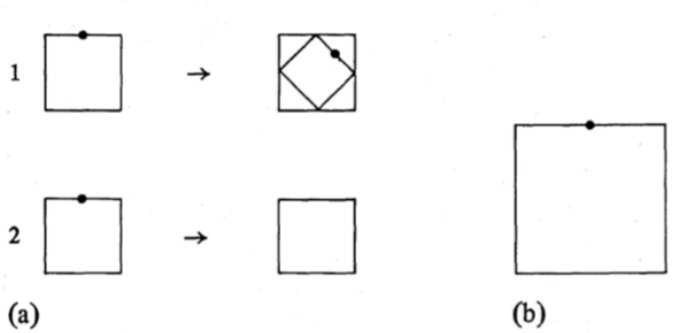


Abbildung 2.3: Eine einfache Shape-Grammar, die Quadrate in Quadraten einfügt. (a) Shape-Rules (b) initiale Shape [15]

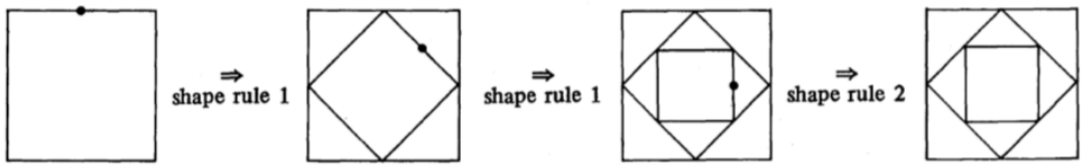


Abbildung 2.4: Generierung einer Shape mit der Shape-Grammar aus Abbildung 2.3 [15]

In Abbildung 2.3 (a) sieht man zwei Regeln, die auf eine Shape angewendet werden können. Die erste Shape-Rule produziert Shapes auf die wiederum weitere Shape-Rules angewendet werden können, um so zusätzliche Shapes abzuleiten. Die zweite Shape-Rule produziert dagegen eine Shape, auf die keine Shape-Rule zutrifft und damit terminiert die Generierung. In Abbildung 2.4 werden die Regeln aus der Abbildung 2.3 (a) angewendet und führen zu transformierten Shapes. Dabei ist die Shape nach Anwendung einer Shape-Rule die jeweilige Subshape der Shape ($Shape \rightarrow Subshape$). Die abgebildete Shape ist nur eine mögliche Darstellung, unter Anwendung der zwei Regeln können andere Shapes generiert werden.

3 Stand der Technik

Im Folgenden wird ein Überblick über die Techniken zur Generierung von Gebäuden gegeben. Die einzelnen Grammatiken werden mit Hilfe von Beispielen erklärt. Vor allem wird CGA Shape detaillierter erläutert, weil Teile dieser Arbeit auf den Konzepten der Grammatik basieren.

3.1 Prozedurale Modellierung von Städten

Parish und Müller entwickelten 2001 eine Methode, um Gebäude in einem urbanen Umfeld zu generieren [11]. Aus mehreren Karten als Eingabe wird mit Hilfe von L-Systemen zuerst ein Straßennetz generiert. Hierfür wurden die L-Systeme um Methoden erweitert, die globale Ziele berücksichtigen und die Komplexität der Produktionsregeln verringern. Das Straßennetz wird anschließend in einzelne Grundstücke unterteilt und auf den Grundstücken werden Häuser und ihre Fassaden generiert. Der Fokus liegt auf der Generierung einer großen Anzahl von vielfältigen, jedoch geometrisch simplen, Gebäuden.

3.2 Split-Grammar

Split-Grammar wurde 2003 von Wonka et al. [19] zur Generierung von urbanen Gebäuden eingeführt. Der Unterschied zu anderen Grammatiken ist die Beschränkung auf bestimmte Produktionsregeln. Im Gegensatz zur Shape-Grammar können bei einer Split-Grammar durch das Unterteilen aus einer Shape mehrere neue Shapes entstehen. Weiterhin können Shapes auch transformiert werden, zum Beispiel von einem Rechteck in einen Quader. Da diese Grammatik speziell für die Generierung von Gebäuden entwickelt wurde, braucht sie nur wenige spezifische Produktionsregeln. Durch diese Vereinfachung eignet sich Split-Grammar für eine automatische Regelauswahl. Für bestimmte Formen

von Shapes (zum Beispiel Rechteck) kann nur eine bestimmte Menge von Regeln (zum Beispiel Split) angewendet werden.

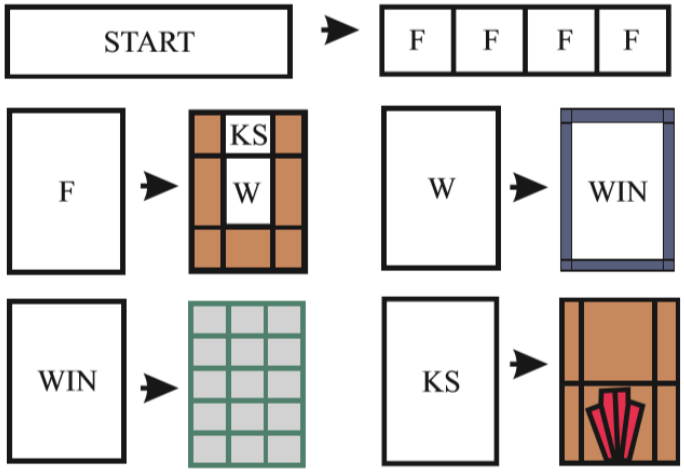


Abbildung 3.1: Die Regeln für eine einfache Split-Grammar. Die farbigen Elemente sind Terminalsymbole. [19]

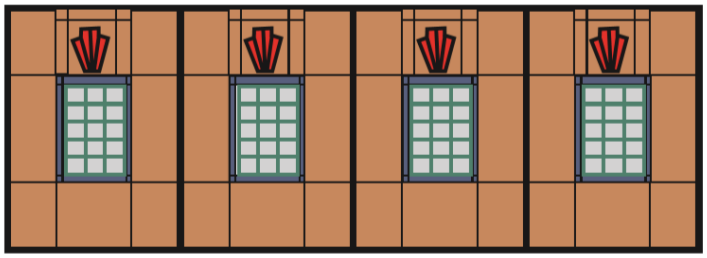


Abbildung 3.2: Das Resultat der Split-Grammar mit den Regeln aus Abbildung 3.1. [19]

.

In Abbildung 3.1 wird das rechteckige Startsymbol mit einer Split-Rule in mehrere Fassadenelemente unterteilt. Die Fassaden werden ihrerseits mit einer anderen Split-Rule unterteilt. Das wird so lange fortgeführt bis die Elemente nicht mehr weiter unterteilt werden können, weil sie Terminalsymbole sind.

3.3 CGA Shape

Computer Generated Architecture (CGA) Shape ist eine Erweiterung von Split-Grammar und wurde 2006 von Müller et al. [8] an der ETH Zürich entwickelt. Später entstand aus

CGA Shape und weiteren Arbeiten CityEngine [1], eine Anwendung zur Stadtplanung. Die Notation der Grammatik und die Regeln, um auf den Shapes Additionen, Skalierungen, Rotationen und Translationen auszuführen, sind inspiriert von L-Systemen, wobei die Produktionsregeln sequentiell abgearbeitet werden. Um Gebäude zu generieren wird zuerst das grobe Volumen des Modells erstellt, danach werden Fassaden strukturiert und am Ende Details für u.a. Fenster und Türen hinzugefügt. Die Split-Rule aus Split-Grammar wurde übernommen und weiterhin der Repeat-Split und der Component-Split eingeführt. Die Produktionsregeln haben folgende Struktur:

$$id : predecessor : cond \rightarrow successor : prob$$

- id: Die ID für die Regel.
- predecessor: Die von successor zu ersetzende Shape.
- cond: Eine logische Bedingung, die wahr ergeben muss, damit die Regel ausgeführt wird. (optional)
- successor: Die Shape(s), die den predecessor ersetzt (ersetzen).
- prob: Die Wahrscheinlichkeit mit der die Regel ausgeführt wird. (optional)

Folgendes Beispiel zeigt die mögliche Anwendung der Split-Rule:

$$1 : fac \rightarrow Subdiv(\text{"}Y\text{"}, 3.5, 0.3, 3, 3, 3)\{floor|ledge|floor|floor|floor\}$$

Diese Regel unterteilt eine Fassade in verschiedene Stockwerke (siehe Abbildung 3.3 links). Die Shape *fac* (*predecessor*) wird durch Shapes für die Flächen von vier Stockwerken und einem Vorsprung ersetzt, die sogenannten *successor*. Der erste Parameter "Y" beschreibt die Achse im Koordinatensystem an der die Split-Rule ausgeführt werden soll. Die weiteren Parameter geben die Größe der einzelnen Stockwerke und des Vorsprungs an. Der Nachteil dieser Regel ist, dass sie nur für eine Fassade mit der Höhe von 12,80 Metern funktioniert. Darum kann diese Regel alternativ mit relativen Größen verwendet werden, um diese für variable Größen zu nutzen:

$$1 : floor \rightarrow Subdiv(\text{"}X\text{"}, 2, 1r, 1r, 2)\{B|A|A|B\}$$

In diesem Beispiel kann das Stockwerk zum Beispiel zwölf oder zehn Meter breit sein (siehe Abbildung 3.3 rechts) und die Regel kann man in beiden Fällen benutzen.

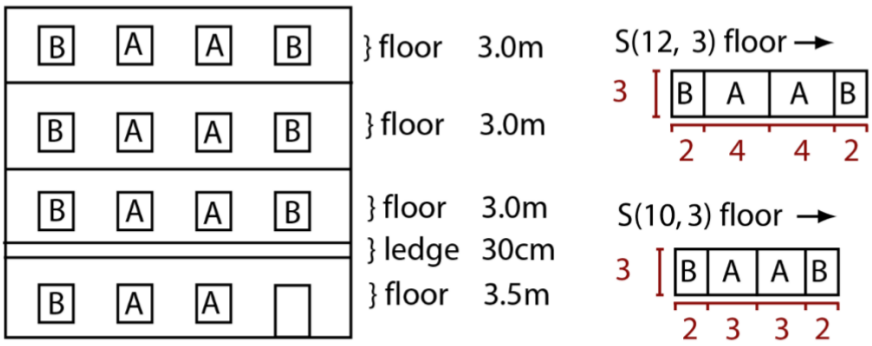


Abbildung 3.3: Links: Ein einfaches Fassadendesign. Rechts: Ein Split mit relativen Werten, hier genutzt für die oberen drei Stockwerke. [8]

Die eingeführte Repeat-Split-Rule stellt eine Erweiterung der Split-Rule dar, welche um einen Parameter für die Anzahl der Wiederholungen ergänzt wurde:

$$1 : floor \rightarrow Repeat(”X”, 2)\{B\}$$

Die Größe der einzelnen Segmente wird dann je nach der Breite der *predecessor* Shape (hier *floor*) angepasst (Segmentbreite = Breite *floor*/2). Weiterhin wurde mit dem Component-Split noch eine Regel eingeführt:

$$1 : cube \rightarrow Comp(”faces”)\{square\}$$

In diesem Beispiel wird eine dreidimensionale Shape (Würfel) durch mehrere zweidimensionale Shapes (Quadrate) ersetzt. Diese Regel kann auch für das Ersetzen einer Shape durch ihrer Kanten ("edges") und Ecken ("vertices") verwendet werden.

Mit CGA Shape können sehr detaillierte Gebäude und Städte erstellt werden. Ein gutes Beispiel ist hier Rome Reborn [3]. Da wird ein antikes Rom von 320 v. Chr. anhand von archäologischen Funden rekonstruiert. Weiterhin wurde CGA Shape auch für die Rekonstruktion von Gebäuden der Maya in Xkipché [7] verwendet. Diese Grammatik ist aktueller Stand der Technik und stellt die Basis für das Framework dieser Arbeit dar, welches in den folgenden Kapiteln näher ausgeführt wird.

Im Jahr 2015 wurde CGA++ [13] auf Basis von CGA Shape entwickelt. Mit CGA++ sind ein direkter Zugriff aus der Grammatik auf den Shape-Tree und Operationen auf Gruppen von Shapes möglich. Dies vereinfacht die Koordination in der Beziehung mehrerer Shapes.

3.4 Weitere Arbeiten

3.4.1 Skizzieren in prozeduraler Generierung

Um das Modellieren in prozeduraler Generierung zu vereinfachen wurden Ansätze entwickelt, die Gebäude mit Hilfe von Skizzen des Benutzers generieren. Einen Ansatz haben Nishida et al. [10] 2016 entwickelt. Dort werden mit Hilfe von maschinellem Lernen verschiedene Formen von den einzelnen Shapes (zum Beispiel Dächer oder Fassaden) generiert. Der Benutzer skizziert zum Beispiel ein Dach und es werden ihm mit Hilfe einer Strukturerkennung verschiedene Dachformen vorgeschlagen, zwischen denen er dann auswählen kann.

Ein anderer Ansatz ist der von Kelly und Wonka [5]. Hier wird der Fokus auf die Generierung von architektonisch komplexen Elementen gelegt. Der Benutzer skizziert einen Grundriss und ein oder mehrere Profile des Gebäudes. Durch das Verwenden verschiedener Profile können zum Beispiel komplexe Dächer mit verschiedenen Steigungen an jeder Seite generiert werden. Um Dächer und andere Formen aus einem komplexen Grundriss zu generieren, wird der Straight Skeleton Algorithmus [4] verwendet und weiter entwickelt. Mit diesem werden in Grundrissen (planaren Polygonen) durch das Finden von Mittelachsen Dachfirste und -grate berechnet.

3.4.2 Prozedurale Modellierung von statisch korrekten Gemäuern

Mit der Modellierung von statisch korrekten Gebäuden haben sich 2009 Durand et al. [18] beschäftigt. Der Fokus liegt hierbei auf gemauerten Gebäuden wie Kathedralen oder Steinbrücken. Die Auswahl der Parameter, zum Beispiel die Höhe eines Bogens, ist durch physikalische Bedingungen eingeschränkt. Weiterhin werden die Produktionsregeln eines Gebäudes solange mit dem Gradientenverfahren optimiert bis es statisch korrekt ist, erst danach wird es generiert. Diese Modelle können dann in physikalischen Simulationen verwendet werden.

3.4.3 Prozedurale Generierung von urbanen Grundstücken

Um auf einer vorgegebenen Karte Grundstücke zu generieren, haben Tom Kelly et al. [17] eine Methode entwickelt, die auch in CityEngine Verwendung findet. Ihr Ansatz um Flä-

chen in Grundstücke zu unterteilen, basiert auf den zwei Algorithmen Oriented Bounding Boxes (von Parish und Müller eingeführt [11]) und dem Straight Skeleton Algorithmus [2]. Durch die Einstellung einiger Parameter, wie zum Beispiel der minimalen Grundstücksbreite, kann die Generierung für den Anwendungsfall konfiguriert werden.

3.4.4 Prozedurale Modellierung von Fassaden aus Bildern

Weitere Arbeiten beschäftigen sich mit der Generierung von Fassaden aus Bildern unter Anwendung einer Split-Grammar. Eine der früheren Arbeiten erschien in dem Jahr 2007 von Wonka et al. [9]. Zuerst wird das Eingabebild in Stockwerke und einzelne Elemente, wie zum Beispiel Fenster, unterteilt. Danach werden diese Elemente selber unterteilt und mit vorhandenen Shapes verglichen. Am Ende werden Regeln für die Fassade extrahiert, welche manuell bearbeitet werden können.

Eine aktuellere Ausarbeitung von Nishida et al. [10] bietet diese Möglichkeit auch für komplexe Fassaden. Weiterhin werden semantische Regionen erkannt wie zum Beispiel der Eingangsbereich eines Gebäudes.

3.4.5 Visuelle Bearbeitung von Shape-Grammar

Im Gegensatz zu den meisten Ansätzen von prozeduraler Modellierung mit Grammatiken wird in der Arbeit von Lipp et al. [6] anstatt in einer Textdatei die Regeln zu bearbeiten, das Modell in einer GUI bearbeitet. Der Benutzer kann Änderungen lokal an einem einzelnen Element oder in ganzen semantischen Regionen ausführen.

4 Konzept

4.1 Die grundlegenden Konzepte der Shape-Grammar

Als Eingabe werden ein Axiom und eine Grammatikdatei benötigt. Ein passendes Axiom für die Grammatikdatei aus der Abbildung 4.1 ist zum Beispiel "Lot,1.2,0.8". Dieses Axiom beschreibt eine rechteckige Grundfläche mit der ID *Lot*, der Länge 1.2 und der Breite 0.8. In der Grammatikdatei muss es eine Regel geben, die die ID des Axioms als Vorgänger hat (siehe Abbildung 4.1 Zeile 9). Diese Regel wird auch als Start-Regel bezeichnet.

```
1  Variables:
2  buildingHeight = [0.8;1.0];
3  floorHeight = buildingHeight/2;
4  windowHeight = 0.15;
5  doorHeight = windowHeight+0.2;
6  roofSlope = 40;
7
8  Rules:
9  Lot --> extrude(buildingHeight) Building
10 Building --> component_split("side_faces"){Facade,Facade,Front,Facade} component_split("top"){Top}
11 Front --> split("X",floorHeight,1r){GroundFloor,Floor}
12 Facade --> split("X",floorHeight,1r){Floor,Floor}
13 GroundFloor --> split("Z",1r,0.2,1r,0.2,1r){Wall,DoorSegment,Wall,WindowSegment,Wall}
14 Floor --> repeated_split("Z",0.35){FloorSegment}
15 FloorSegment --> split("Z",1r,0.2,1r){Wall,WindowSegment,Wall}
16 WindowSegment --> split("X",1r,windowHeight,1r){Wall,Window,Wall}
17 Window --> detail("window") Unused
18 DoorSegment --> split("X",doorHeight,1r){Door,Wall}
19 Door --> detail("door") Unused
20 Top --> split("X",1r,0.05){RoofSegment,GableSegment}
21 RoofSegment --> roof("hip",roofSlope) Roof:0.5
22 RoofSegment --> roof("line",roofSlope) Roof:0.5
23 GableSegment --> gable(roofSlope) Gable
```

Abbildung 4.1: Beispiel einer Grammatikdatei

Die Grammatikdatei besteht aus einem Abschnitt für die Variableninitialisierung, der eingeleitet wird durch den String "Variables:" und einem Abschnitt für die Regeln, eingeleitet durch den String "Rules:". Die Syntax für die Variableninitialisierung ist folgende:

```
name = value;
```

Der Name ist frei wählbar und der Wert kann eine Zahl im Double- oder Integer-Format, eine Variable, ein Intervall oder eine arithmetische Operation zweier Variablen beziehungsweise Zahlen sein. Als arithmetische Operationen werden die Grundrechenarten Addition, Subtraktion, Multiplikation und Division unterstützt. Das Intervall hat die folgende Struktur:

```
[lowerBoundary;upperBoundary]
```

Die untere und obere Grenze des Intervalls können Zahlen im Double- oder Integer-Format oder Variablen sein. In dem Intervall wird eine Zufallszahl generiert, die in der Grammatik verwendet wird.

Der Regeln haben folgende Struktur:

```
predecessor-id : cond --> {successor-symbols}:prob
```

- predecessor-id: ID der von successor-symbols zu ersetzenden Shape
- cond: logische Bedingung, die wahr ergeben muss, damit die Regel ausgeführt wird (optional)
- successor-symbols: Symbol(e) der Shape(s), die die Shape mit predecessor-id ersetzt (ersetzen)
- prob: Wahrscheinlichkeit mit der die Regel ausgeführt wird (optional)

Mit Hilfe von Regeln kann eine ID durch eine Menge von Symbolen ersetzt werden. Ein Symbol kann entweder eine ID oder eine Shape-Operation sein. Eine Regel hat immer einen Vorgänger und einen oder mehrere Nachfolger. Ein Beispiel für eine Regel kann wie folgt aussehen:

```
GableSegment --> gable(40) Gable
```

Nach Ausführung der Regel wird die Shape mit der ID *GableSegment* unter Anwendung der Shape-Operation *gable(40)* durch die Shape mit der ID *Gable* ersetzt. Eine Shape-Operation kann eine Shape verändern oder sie durch andere Shapes ersetzen. Die Shape-Operation Gable ersetzt eine zweidimensionale Shape (Rechteck) durch eine dreidimensionale Shape in Form eines Dreiecksgiebels, die das Rechteck als Grundfläche hat.

Zudem können Shape-Operations Parameter besitzen. In dem Beispiel ist der Wert des Parameters 40 und repräsentiert die Steigung des Dreiecksgiebels in Grad. Durch die Verwendung von Regeln können weitere Shapes mit IDs aus der Grammatik abgeleitet werden.

4.1.1 Shape und Shape-Tree

Jede Shape hat eine ID und einen Scope. Der Scope einer Shape ist ein lokales Koordinatensystem, das den Bereich der Shape im globalen Koordinatensystem einer Grammatik festlegt. Die Abbildung 4.2 zeigt in hellgrau hinterlegt den Scope einer grauen Shape. Das lokale Koordinatensystem besteht aus einem Punkt P und der Ausdehnung der Shape in Richtung der x-, y- und z-Achse. Der Punkt P ist die Position des Scopes im globalen Koordinatensystem der Grammatik und gleichzeitig der Ursprung des lokalen Koordinatensystems der Shape.

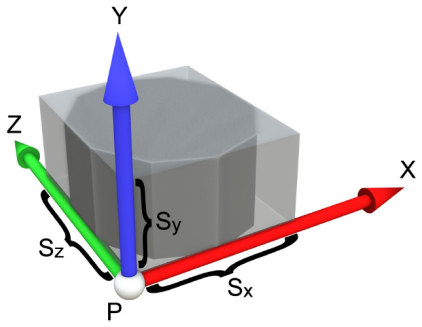


Abbildung 4.2: Der Scope einer Shape [8]

Shapes haben Beziehungen untereinander, so können Shapes mehrere Child-Shapes haben. Diese Beziehungen werden in einem Baum dargestellt, dem so genannten Shape-Tree. Alle Shapes mit Ausnahme der ersten Shape an der Wurzel des Baumes besitzen eine Parent-Shape.

Die folgenden fünf Regeln erstellen einen beispielhaften Shape-Tree (siehe Abbildung 4.3):

```
Lot --> extrude(0.5) Building
Building --> component_split("side_faces"){Back,Side,Front,Side}
             component_split("top"){Top}
Top --> split("X",1r,0.05){RoofSegment,GableSegment}
RoofSegment --> roof("line",40) Roof
GableSegment --> gable(40) Gable
```

Hierbei ist die Shape mit der ID *Lot* die Wurzel des Baumes. Es handelt sich um einen rechteckigen Grundriss aus dem ein Quader mit der ID *Building* erstellt wird. Der Quader wiederum wird in seine Seitenflächen und die obere Fläche aufgeteilt. *Building* hat somit fünf Child-Shapes, von denen nur die obere Fläche mit der ID *Top* anschließend weiter verwendet wird. Diese Fläche wird in zwei weitere Flächen aufgeteilt, auf denen dann ein Dach und ein Giebel erstellt werden.

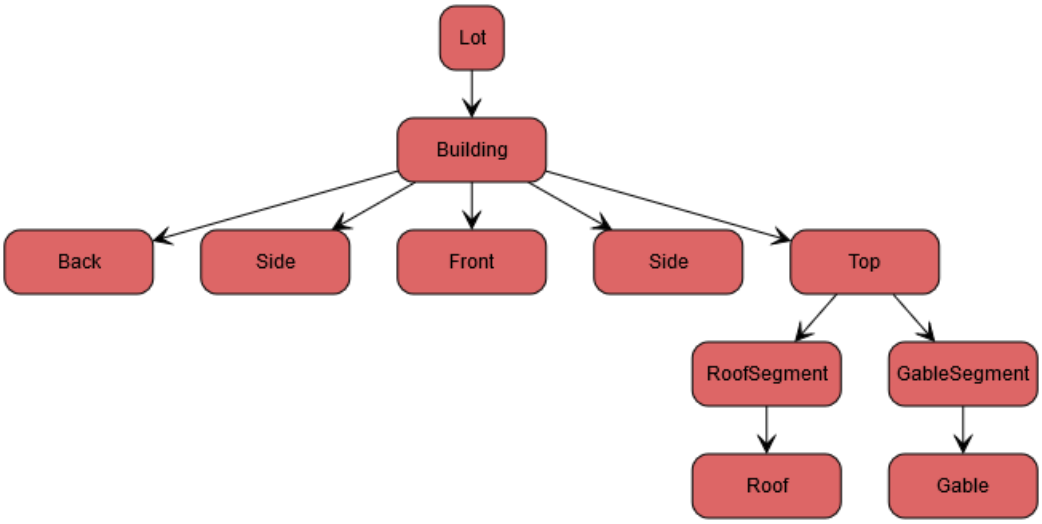


Abbildung 4.3: Beispiel eines Shape-Trees

4.1.2 Grundlegende Shape-Operations

Für die Generierung eines Gebäudes mit Hilfe einer Shape-Grammar werden mindestens die Shape-Operations Extrude, Component-Split und Split benötigt.

Unter Anwendung der Shape-Operation Extrude wird am Beispiel eines Gebäudes aus einem rechteckigen Grundriss ein Quader erstellt. Ein Beispiel für den Aufruf von Extrude ist *extrude(0.5)*. Der Parameter steht hierbei für die Höhe des von der Grundfläche extrudierten Quaders.

Mit Hilfe des Component-Split lassen sich aus einer Shape die einzelnen Flächen extrahieren, um darauf weiter Regeln anzuwenden. Ein Beispiel hierfür ist folgendes:

```
Building --> component_split("side_faces"){Back,Side,Front,Side}
             component_split("top"){Top}
```

Der Component-Split kann unter anderem auf einen Quader angewendet werden. Mit dem Parameter *"side_faces"* werden die Seitenflächen extrahiert, mit dem Parameter *"top"* die obere Fläche. Auf der oberen Fläche kann dann zum Beispiel ein Dach mit der Shape-Operation Roof erstellt werden. Auf den Seitenflächen können Fassaden generiert werden. Dazu werden die Seitenflächen zum Beispiel mit der Shape-Operation Split in Stockwerke unterteilt:

```
Front --> split("X",0.3,1r){GroundFloor,SecondFloor}
```

Hierbei wird eine zweidimensionale Shape mit der ID *Front*, in diesem Fall eine Seite eines Gebäudes, in zwei ebenfalls zweidimensionale Shapes mit den IDs *GroundFloor* und *SecondFloor* unterteilt. Dies geschieht hier in Richtung der X-Achse des Scopes der Shape mit der ID *Front*. Die anderen beiden Parameter bestimmen die Höhe der einzelnen Stockwerke. Es können absolute und relative Werte angegeben werden. Wenn die Shape mit der ID *Front* in Richtung der X-Achse die Höhe 0.5 hat, ergibt sich für die Shape mit der ID *SecondFloor* eine Höhe von 0.2.

Eine Unterform der Shape-Operation Split ist Repeated-Split:

```
SecondFloor --> repeated_split("Z",0.3){FloorSegment}
```

Hierbei wird die Shape mit der ID *SecondFloor* so oft wie möglich in Segmente mit einer Ausdehnung in Z-Richtung von 0.3 unterteilt. Bei einer Ausdehnung der Shape mit der ID *SecondFloor* in Z-Richtung von zum Beispiel 0.6 werden zwei Segmente mit einer Ausdehnung von 0.3 erstellt.

4.1.3 Bedingungen und Wahrscheinlichkeiten

Um aus einer Regeldatei Gebäude mit einer Varianz zu erzeugen, gibt es Regeln mit Bedingungen oder Wahrscheinlichkeiten. Ein Beispiel für eine Regel mit einer Bedingung ist folgende:

```
Front : height <= 1.2 --> split("X",2r,1r){Floor1,Floor2}
```

In diesem Fall wird die Regel nur ausgeführt, falls die Gebäudehöhe kleiner oder gleich 1.2 ist. Wenn die Regel ausgeführt wird, wird die ID *Front* durch die beiden IDs *Floor1* und *Floor2* ersetzt.

Darüber hinaus können Regeln mit einer bestimmten Wahrscheinlichkeit angewendet werden. Ein Beispiel für Regeln mit Wahrscheinlichkeiten sind diese zwei Regeln zur Erstellung von bestimmten Dachtypen:

```
RoofSegment --> roof("hip",roofSlope) Roof:0.5
RoofSegment --> roof("line",roofSlope) Roof:0.5
```

Hier wird aus der Shape mit der ID *RoofSegement* mit fünfzigprozentiger Wahrscheinlichkeit entweder ein Walm- oder ein Satteldach erstellt. Die Wahrscheinlichkeiten der Regeln für eine ID müssen aufsummiert immer 100 Prozent ergeben.

4.2 Anforderungen an die Stadtgenerierung

Die optischen Anforderungen an die Stadt- und Gebäudegenerierung wurden größtenteils aus dem Gestaltungsleitfaden für Architekten des Projekts "Gründungsviertel"[1] hergeleitet. Ziel des Projekts ist die Bebauung eines im Zweiten Weltkrieg zerstörten Quartiers in der Lübecker Altstadt nach historischem Vorbild.

Das System muss mehrere Gebäude gleichzeitig generieren, wobei die zugrundeliegenden Grundrisse dabei fest vorgegeben sind. Innerhalb des Stadtviertels gibt es Rippenstraßen, bei denen der Giebel der Häuser zur Straße zeigt, und Querstraßen, wo die Dachfläche

[1] Gestaltungsleitfaden "Gründungsviertel":
https://www.gruendungsviertel.de/downloads.html?file=files/inhalte/downloads/broschueren/GV_Der%20Gestaltungsleitfaden.pdf,
abgerufen am 08.04.2020

Abbildung 4.4: Blick auf die Straße Große Petersgrube in Lübeck[2]

der Häuser parallel zur Straße verläuft. Bei beiden Straßentypen ist der Straßenrand geschlossen bebaut, das heißt es gibt keine Lücken zwischen den einzelnen Häusern.

Die Generierung der einzelnen Gebäude, die im folgenden Punkt beschrieben werden, erfolgt zufällig aus den möglichen Eigenschaften der Gebäuden.

4.3 Anforderungen an die Gebäudegenerierung

Die Gebäude werden mit Fokus auf die volumetrische Darstellung generiert. Zusätzliche Details wie zum Beispiel Fassaden werden in dieser Arbeit nicht betrachtet. Die vorgegebenen Grundrisse der Giebelhäuser sind schmal und lang.

[2]Uli Harder / CC BY-SA (https://creativecommons.org/licenses/by-sa/2.0)

4.3.1 Gebäudetyp 1: Haus in der Rippenstraße

Der Gebäudetyp wird in die drei Geschossteile Erdgeschoss, Obergeschoss und Giebeldreieck unterteilt. Das Obergeschoss kann aus einem bis drei und das Giebeldreieck aus ein bis zwei einzelnen Stockwerken bestehen. In allen Geschossteilen werden Fenster generiert, wobei diese sich je nach Geschoss in Größe und Form unterscheiden. Ein festes Kriterium ist, dass die Fenster mit der Fassade abschließen oder leicht zurückgesetzt sind. Diese haben im Allgemeinen ein Verhältnis von Höhe zu Breite von mindestens 150 Prozent. Ausnahmen hierfür gibt es im Erdgeschoss bei großen Schaufenstern und im Giebeldreieck für kleine Fenster. In diesen Fällen reicht ein minimales Verhältnis von Höhe zu Breite von 120 Prozent.

(a) Attika[3] (b) Schweifgiebel[3] (c) Treppengiebel[3]

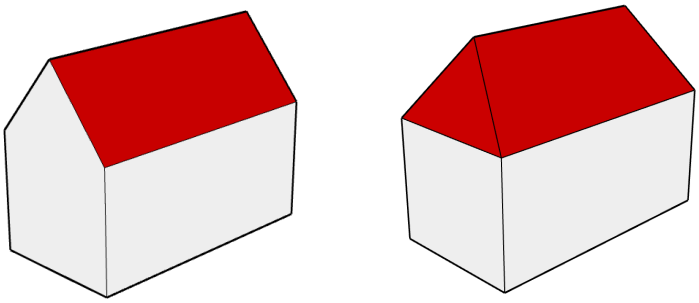
Abbildung 4.5: Die verschiedenen Giebelarten für Gebäude in Rippenstraßen

Im Erdgeschoss wird eine hohe Tür mit einem Sockel generiert, die zudem leicht zurückgesetzt sein muss. Das Erdgeschoss hat eine minimale Höhe von 4,50 Metern und kann in seltenen Fällen durch einen Sims von den anderen Stockwerken getrennt sein.

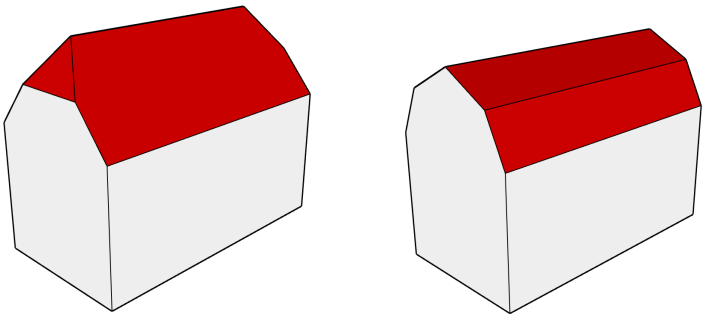
[3]Lemke, Martin / CC BY-SA (https://creativecommons.org/licenses/by-sa/4.0)

Im Obergeschoss sind alle Fenster gleich groß und haben den gleichen Abstand zueinander. Der Abstand der Fenster zur Außenkante der Fassade muss größer sein als der Abstand untereinander. Außerdem müssen die Fenster in den verschiedenen Stockwerken in einer Linie übereinander angeordnet sein.

Das Giebeldreieck muss ein Treppen- oder Schweifgiebel sein. Anstatt eines Giebeldreiecks ist auch ein horizontal angeschlossener Giebel (Attika) möglich, auf dem ein flacher Dreiecksgiebel aufgesetzt sein kann (siehe Abbildung 4.5). Das Giebeldreieck muss die Dachfläche leicht überragen und die Fenster müssen kleiner als im Obergeschoss sein. Der Treppengiebel kann drei bis fünf Stufen haben, wobei die mittlere Stufe gleich breit oder breiter als die restlichen Stufen sein muss. Der Schweifgiebel besteht hingegen aus einem oder zwei Schweifen.



(a) Sattel- und Walmdach



(b) Krüppelwalm- und Mansarddach

Abbildung 4.6: Dachtypen

Es gibt vier verschiedene Dachtypen (siehe Abbildung 4.6): Satteldach, Walm-, Krüppelwalm- und Mansarddach. Das Satteldach soll am häufigsten auftreten, gefolgt von Krüppelwalm- und Mansarddach. Das Walmdach wird nur mit dem horizontal abge-

schlossenen Attika-Giebel kombiniert. Krüppelwalm- und Mansarddach müssen vom Giebel verdeckt werden.

4.3.2 Gebäudetyp 2: Haus in der Querstraße

Das Gebäude besteht aus den Geschossteilen Erdgeschoss, Obergeschoss und Dachgeschoss und besitzt im Gegensatz zu Gebäudetyp 1 keinen Schmuckgiebel. Es muss ein steiles Satteldach haben, was an den Seiten einen leichten Überhang hat. Weiterhin kann der Gebäudetyp auf der Dachfläche einen Zwerchgiebel (ein fassadenbündiger, kleiner Giebel), Gauben oder Dachfenster haben. Gauben und Dachfenster haben mindestens einen Meter Abstand zur Traufe. Untereinander müssen sie und der Zwerchgiebel einen Abstand von 1,50 Metern und zur Außenseite einen Abstand von einem Meter haben. Weiterhin gibt es eine festgelegte Maximalbreite der Gauben und Zwerchgiebel von drei Metern. Die Geschossteile Erdgeschoss und Obergeschoss besitzen die gleichen Anforderungen wie Gebäudetyp 1. Der Gebäudetyp ist in der Stadtgenerierung traufständig ausgerichtet, das heißt mit der Dachfläche parallel zur Straße.

4.3.3 Gebäudetyp 3: Eckhaus

Das Eckhaus stellt einen Kombination aus dem Gebäudetyp 1 und Gebäudetyp 2 dar. Die Giebelseite zur Rippenstraße entspricht dem Gebäudetyp 1 und der Seitenflügel zur Querstraße dem Gebäudetyp 2. Das Dach muss wie beim Gebäudetyp 2 ein steiles Satteldach mit Dachfenstern, Gauben oder Zwerchgiebeln sein.

4.4 Anforderungen an die Software

Das System hat zwei Hauptaufgaben. Es soll aus einer Grammatikdatei und einem Axiom ein Gebäude erstellen und zudem soll aus einer Stadtdatei eine Stadt generiert werden. Die Grammatik- und Stadtdateien sollen in einem Graphical User Interface (GUI) angezeigt werden. Zudem muss es möglich sein diese Dateien in dem GUI zu bearbeiten und das jeweilige Gebäude oder die Stadt muss angezeigt werden. Die Generierung soll möglichst schnell erfolgen, aber das System muss keine Echtzeit-Anforderungen erfüllen. Das Gebäude soll man von allen Seiten in der grafischen Ausgabe begutachten können. Ebenso soll es die Option geben, die Kamera durch die generierte Stadt zu bewegen.

4.5 Aufbau des Systems

Das System besteht aus den drei Hauptkomponenten Grammatik, Grafik-Engine und dem GUI. In dieser Arbeit wird auf ein bestehendes Projekt der Gruppe Computergrafik HAW Hamburg aufgebaut. Dadurch ist die Programmiersprache mit Java und die Grafik-Engine mit JMonkey vorgegeben. Die grundlegenden Funktionen zur Erstellung von Gebäuden sind in der Grammatik schon verfügbar und werden um weitere Operationen erweitert, um die Anforderungen an die Generierung der Giebelhäuser zu erfüllen. Des Weiteren ist das System noch um die Funktionalität zur Darstellung mehrerer Häuser zu ergänzen.

4.6 Konzept für die Mesh-Generierung

4.6.1 Dreiecksnetze

Es gibt mehrere Arten von Polygonnetzen, die am häufigsten verwendeten sind Dreiecksnetze. Diese werden in diesem System zur Darstellung der Flächen verwendet. In einem Dreiecksnetz werden die Knoten, die eine Form definieren, miteinander durch Kanten verbunden. In Abbildung 4.7 ist beispielhaft ein Gebäude mit seinen Knoten und seinem Dreiecksnetz zu sehen. Mit Dreiecksnetzen können viele zweidimensionale Formen gut dargestellt werden. Rundungen lassen sich mit Dreiecksnetzen nicht darstellen, aber sie können, zum Beispiel in Kombination mit den im nächsten Abschnitt beschriebenen Bézierkurven, approximiert werden.

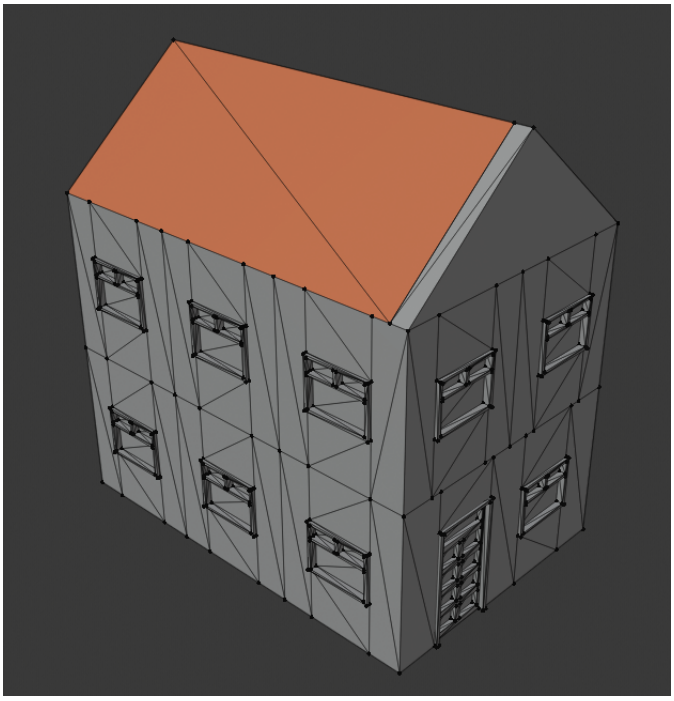


Abbildung 4.7: Dreiecksnetz von einem Gebäude

4.6.2 Bézierkurven

Für die Darstellung von Rundungen in einem Polygonnetz gibt es verschiedene Möglichkeiten. Oft werden diese durch eine aus Basisfunktionen bestehende Approximation beschrieben. Es gibt u.a. Hermite- und Bézierkurven. Für letztere wurde sich entschieden, weil Hermitekurven nur in kubischer Form vorkommen. In diesem Fall werden aber nur Kurven 2. Grades benötigt. Bézierkurven[4] werden mit folgender Formel beschrieben:

$$p(t) = \sum_{i=0}^{n} c_i \cdot B_i^n(t)$$

Wobei $B_i^n(t)$ die Basisfunktionen, n der Grad und c_i die Kontrollpunkte der Bézierkurven sind. Die Basisfunktionen sind die Bernsteinpolynome im Intervall von 0 bis 1 mit dem Grad n. Für die Verwendung einer Bézierkurve zweiten Grades ergibt das folgende Formel:

$$B_i^2(t) = (c_0 - 2c_1 + c_2)t^2 + (-2c_0 + 2c_1)t + c_0 \ , \ t \in [0, 1]$$

[4]Bézierkurve: https://de.wikipedia.org/wiki/B%C3%A9zierkurve, abgerufen am 30.07.2020

4.7 Entwurf des Dateiformats für die Stadt

Zum Speichern und Einlesen der Stadt wurde die Sprache XML (Extensible Markup Language) ausgewählt. Die Hauptgründe dafür sind, dass sie hierarchisch strukturiert und von Menschen lesbar ist. Die hierarchische Struktur ist in Abbildung 4.8 erkennbar. Das Root-Element wird mit dem Starttag *<city>* eingeleitet. Der dargestellte Ausschnitt beinhaltet zwei Gebäude (Starttag *<building>*, Endtag *</building>*), die wiederum aus XML-Elementen zum Beispiel für die Länge und Breite der Gebäude bestehen. Das Endtag *</city>* fehlt in der Darstellung, weil es sich nur um einen Ausschnitt einer Stadtdatei handelt.

```
<city>
    <building>
        <grammarFileName>shape_grammar/eckhaus_links.grammar</grammarFileName>
        <length>1.3</length>
        <width>1.2</width>
        <offset>
            <x>-0.22</x>
            <y>0.0</y>
            <z>-2.3</z>
        </offset>
        <rotation>0</rotation>
        <name>Braunstr. 8</name>
    </building>
    <building>
        <grammarFileName>shape_grammar/rippenstr_haus.grammar</grammarFileName>
        <length>1.4</length>
        <width>0.5</width>
        <offset>
            <x>-0.3</x>
            <y>0.0</y>
            <z>-1.1</z>
        </offset>
        <rotation>0</rotation>
        <name>Braunstr. 10</name>
    </building>
```

Abbildung 4.8: Ausschnitt aus der Stadtdatei gruendungsviertel.xml

4.8 Auswahl des Parsers für die Stadt

Es gibt verschiedene Möglichkeiten XML zu parsen, zwei der möglichen Varianten werden nachfolgend vorgestellt.

SAX (Simple API for XML) ist ein ereignisorientierter Parser. Bei jedem XML-Sprachkonstrukt (zum Beispiel Starttag, Endtag, Kommentar) wird ein Ereignis ausgelöst, dass direkt danach verarbeitet werden muss. SAX ist ein schneller und einfacher Parser, hat aber den Nachteil, dass nach dem Parsen Informationen verloren gehen, die nicht verarbeitet wurden.

Der DOM Parser (Document Object Model) speichert die eingelesenen Daten in einer Baumstruktur. Das Root-Element fungiert als Wurzelknoten für den Baum und alle XML-Elemente, die das Root-Element enthält, werden dessen Kindknoten. Der Baum bildet die Struktur des XML-Dokumentes ab. Auf diesen Baum kann intern zugegriffen und weiter bearbeitet werden.

Für das Projekt ist es wichtig, dass die XML-Datei immer komplett eingelesen wird. Ein Zugriff auf die Struktur der Datei ist hingegen nicht notwendig. Deswegen und aufgrund seiner Geschwindigkeit fällt die Entscheidung auf den SAX Parser.

5 Realisierung

5.1 Aufbau der Grammatik

Die Shape-Grammar besteht aus den Hauptklassen *Parser, ShapeGrammar, Evaluator, ShapeTree* und *MeshGenerator* (siehe Abbildung 5.1). Die Klasse *ShapeGrammar* verwaltet *Parser, Evaluator* und den *MeshGenerator*. Bei ihrer Ausführung findet der nachfolgende Ablauf statt. Im *Parser* wird die Grammatikdatei und das Axiom eingelesen. Daraus werden Regeln und Variablen erstellt, die in der Klasse *Grammar* gespeichert werden. In dieser wird auch eine Liste aller Shape-Operations verwaltet, die in der aktuellen Shape-Grammar vorhanden sind. Danach wird im *Evaluator* unter Verwendung der Regeln, Variablen und Shape-Operations aus der Klasse *Grammar* ein Shape-Tree erstellt. Aus dem Shape-Tree wird mit Hilfe des *MeshGenerator* ein Dreiecksnetz generiert. Dieses kann nun unter Verwendung einer Grafik-Engine dargestellt werden.

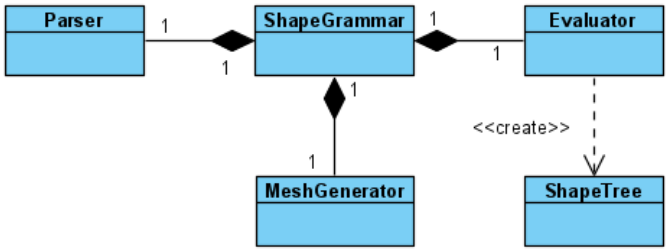


Abbildung 5.1: Klassendiagramm von *ShapeGrammar* und ihren wichtigsten Komponenten

5.2 Erweiterung der Shape-Grammar

Die in Abschnitt 4.1 beschriebenen Konzepte waren teilweise in ihrer Grundfunktion zur Gebäudegenerierung bereits implementiert. Um eine höhere Varianz bei den generierten

Gebäuden zu erzielen, wurden folgende Funktionalitäten im Rahmen dieser Arbeit hinzugefügt. Es wurden Variablen implementiert, die in Intervallen eine Zufallszahl generieren können. Außerdem können die arithmetischen Grundrechenarten mit ihnen durchgeführt werden. Weiterhin wurden die Regeln um Bedingungen und Wahrscheinlichkeiten erweitert. Zudem wurde ein lesender Zugriff auf die Länge und die Breite des Axioms aus der Grammatik implementiert.

5.3 Aufbau der Shapes

Shapes werden aus IDs und Shape-Operations erstellt. In Abbildung 5.2 ist ein Klassendiagramm von der abstrakten Klasse *Shape* und deren Beziehung zu konkreten Shapes dargestellt. Als Beispiel für eine konkrete Shape dient hier die Klasse *RoofShape*.

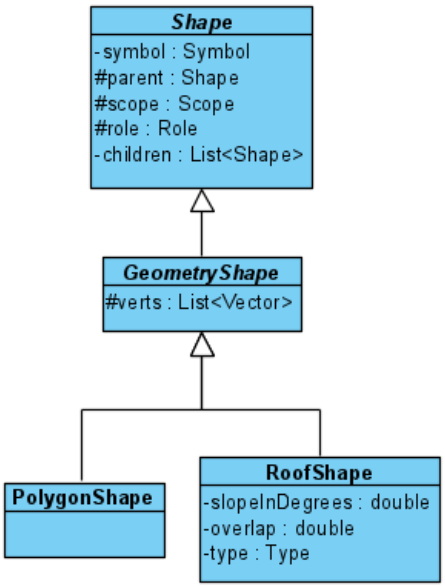


Abbildung 5.2: Klassendiagramm von *Shape*

Aus folgender Regel entsteht eine *RoofShape*:

```
RoofSegment --> roof("hip",40) Roof
```

In diesem Fall hat das Feld *symbol* den Wert *"Roof"*. Die Variable *parent* ist die Shape, die den String *"RoofSegment"* als *symbol* hat. Diese Shape muss eine *PolygonShape* sein, da die *RoofShape* nur aus einer *PolygonShape* erstellt werden kann. Jede Shape hat einen

scope, ein lokales Koordinatensystem, das den Bereich der Shape im globalen Koordinatensystem einer Grammatik festlegt. Die Variable *role* dient dazu, um Abhängigkeiten der Shapes untereinander zu modellieren. Es gibt zum Beispiel die Rolle *"Roof"*, um bei der Mesh-Generierung jede Shape mit dieser Rolle mit roter Färbung zu erstellen. Eine Shape kann auch Kinder haben, dies wäre der Fall wenn auf die *RoofShape* ein Component-Split ausgeführt wird. Die abstrakte Klasse *GeometryShape* verwaltet nur die Knoten der Shape. Von ihr sind alle Shapes, die in einem Koordinatensystem darstellbar sind, abgeleitet. Die Klasse *PolygonShape* ist die einfachste darstellbare Form einer Shape. Sie repräsentiert alle zweidimensionalen Shapes, wie zum Beispiel ein Rechteck. Als Beispiel für eine komplexere Shape ist die Klasse *RoofShape* in der Abbildung 5.2 dargestellt. Diese ist eine dreidimensionale Shape und in ihr werden die spezifischen Eigenschaften, die zur Erstellung eines Daches nötig sind, verwaltet. Diese sind *slopeInDegrees* für die Neigung und *overlap* für den Überhang des Daches. Mit *type* kann zwischen den verschiedenen Dacharten gewählt werden.

5.4 Implementierung neuer Shapes

In der vorhandenen Shape-Grammar der Gruppe Computergrafik HAW sind bereits einige Shapes wie zum Beispiel *PolygonShapes* oder *PrismShapes* vorhanden, welche in diesem Projekt Anwendung finden. Die Shape-Operations, die zur Unterteilung von Fassaden notwendig sind (Split- und Repeated-Split-Operation), wurden ebenfalls übernommen. Zur Umsetzung der Anforderungen an die Lübecker Kaufmannshäuser wurde die Shape-Grammar, um weitere Shape-Operations und daraus resultierende neue Shapes, ergänzt. Das schon vorhandene Satteldach wurde erweitert, sodass es auch mit einem Überhang generiert werden kann. Weiterhin wurde ein Component-Split für diese Dachform hinzugefügt. Es wurden außerdem drei neue Dachformen implementiert: Mansard-, Walm- und Krüppelwalmdächer.

Zudem wurde eine Shape, die einen Dreiecksgiebel generiert, erstellt. Auf diese kann wiederum ein Component-Split ausgeführt werden, um den Dreiecksgiebel unter anderem mit Giebelornamenten (Schweif- und Treppengiebel) zu versehen.

Weitere Shapes, die noch zur Realisierung der Kaufmannshäuser notwendig sind, sind die Gauben und der Zwerchgiebel. Von den Gauben wurden zwei Formen realisiert, die

Flachdach- und die Giebelgaube. Für den Zwerchgiebel und die Gauben wurden auch jeweils ein Component-Split implementiert, damit diese an der Front mit Fenstern versehen werden können.

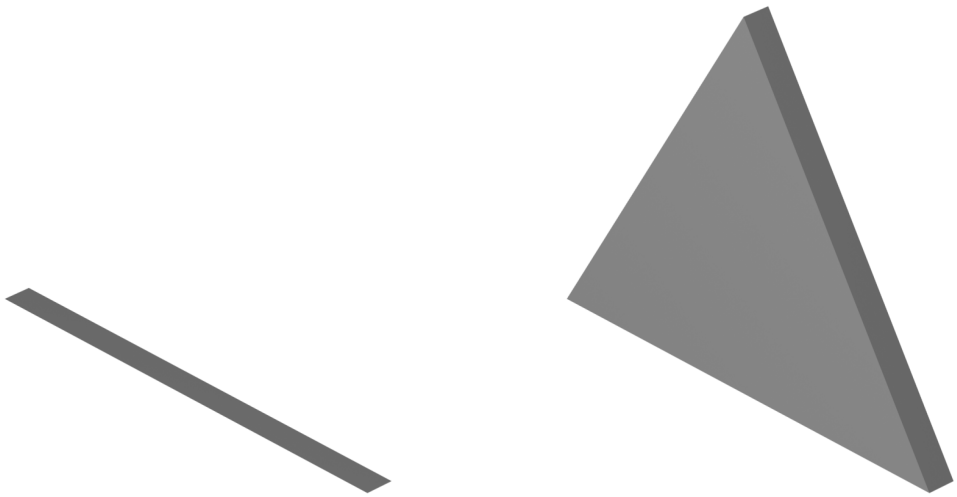
Um die in den Anforderungen festgelegten zurückgesetzten Fenster und Türen zu implementieren, wurden auch hier zwei neue Shapes realisiert. In den folgenden Unterabschnitten wird die Implementierung einiger der genannten Shapes genauer beschrieben.

5.4.1 Umsetzung eines Giebels

Für die Generierung eines neuen Giebels sind mehrere Schritte notwendig. Als Erstes wird die Shape-Operation Gable ausgeführt und erstellt aus einer rechteckigen Grundfläche vom Typ *PolygonShape* einen Dreiecksgiebel (siehe Abbildung 5.3). Die Gable-Operation hat folgende Syntax:

```
gable(slope) id
```

Es wird ein Dreiecksgiebel mit der gegebenen Neigung *slope* erstellt. Der Parameter *slope* muss einen Wert zwischen 0 und 90 Grad haben.



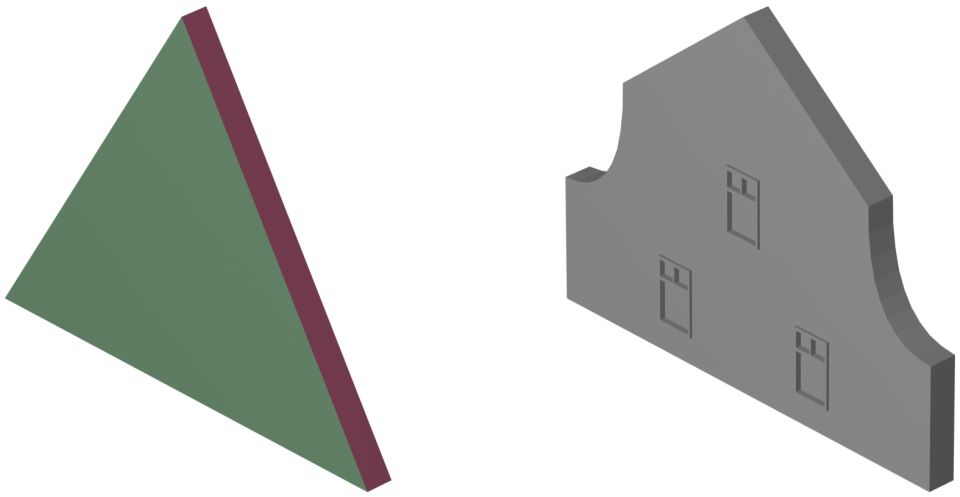
(a) Giebelsegment (b) Dreiecksgiebel

Abbildung 5.3: Die ersten beiden Schritte zur Erstellung eines Giebels

Aus dem Dreiecksgiebel werden unter Anwendung der Shape-Operation Component-Split die einzelnen Seitenflächen extrahiert. Dies wird in der Abbildung 5.4a gezeigt. Zur Verdeutlichung wurden die einzelnen Shapes mit Falschfarben dargestellt. Für die Operation wird der folgende Befehl verwendet:

```
Gable --> component_split("side_faces"){Tail,Front,Tail,Back}
```

Es entstehen vier *PolygonShapes* auf denen weitere Operationen ausgeführt werden können. In diesem Beispiel wird auf der *PolygonShape* mit der ID *Front* eine Fassade mit Fenstern generiert und auf den *PolygonShapes* mit der ID *Tail* ein Schweifgiebel erstellt.



(a) Giebel nach Component-Split (b) Schweifgiebel

Abbildung 5.4: Die letzten Schritte zur Erstellung eines Schweifgiebels

Die Syntax für die Shape-Operation, um einen Giebel mit Ornamenten zu verzieren, ist folgende:

```
gable_ornament(type,noOfSteps) id
```

Der Parameter *type* gibt die Art des Ornamentes an, entweder "tail" für einen Schweif oder "steps" für Treppen. Mit *noOfSteps* wird festgelegt wie oft das Schweif- oder Treppenelement wiederholt wird. In diesem Beispiel wird ein Giebel mit zwei Schweifelementen erstellt (siehe Abbildung 5.4b).

Die zugrunde liegende Shape-Operation hierfür ist die folgende:

```
Tail --> gable_ornament("tail",1) TailOrnament
```

Generierung des Netzes für den Giebel

Für die Darstellung aller Shapes wird ein Dreiecksnetz verwendet. Das Netz für die Darstellung der Schweife wird mit Hilfe von Bézierkurven zweiten Grades erstellt (siehe Abbildung 5.5). In diesem Bild ist die Unterteilung des Dreiecksgiebels in verschiedene Stockwerke und deren weitere Segmentierung für die Generierung der Fenster gut erkennbar. Alle Flächen werden in Dreiecke unterteilt, um sie darstellbar zu machen. Beim Schweifgiebel fällt so auf, dass er in Treppenform aufgebaut ist, aber auf der ersten Stufe mit Hilfe von zwei Bézierkurven eine Rundung erstellt wurde.

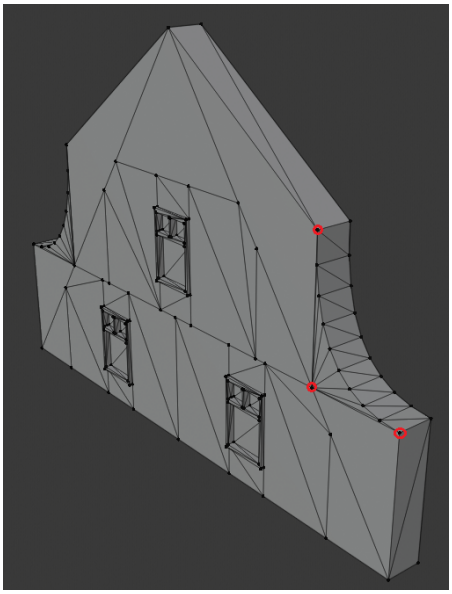


Abbildung 5.5: Schweifgiebel mit Dreiecksnetz

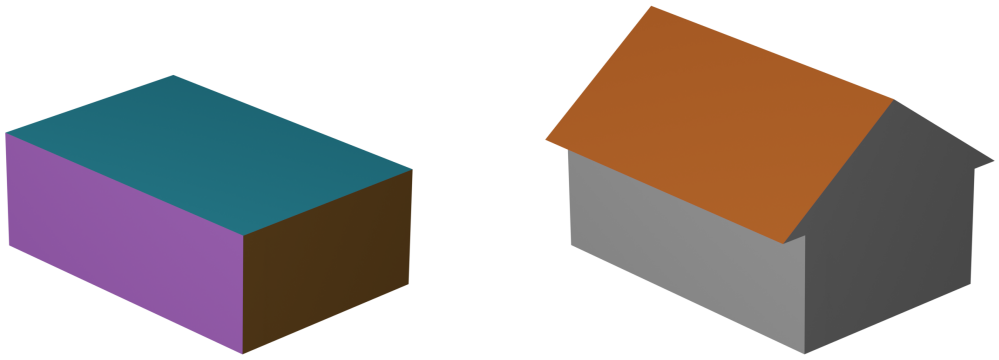
In der Abbildung 5.5 sind die drei Kontrollpunkte einer Kurve markiert (von unten rechts nach oben links: c_0, c_1 und c_2). Wie man sieht werden die Kontrollpunkte c_0 und c_2 interpoliert und c_1 wird approximiert. Hier wurde die Bézierkurve in 9 Segmente mit gleichem Abstand unterteilt. Je kürzer die Länge dieser Segmente ist, desto genauer wird die Kurve approximiert. Aber der Rechenaufwand steigt mit jedem Segment an.

5.4.2 Umsetzung eines Daches mit Zwerchgiebel

Wie in den Anforderungen beschrieben, ist ein Zwerchgiebel ein fassadenbündiger, quer zur Hauptdachfläche angeordneter kleiner Giebel. Bei einem Dach mit Überhang durchbricht er die Dachfläche. Hier wird beschrieben wie die Kombination aus Dach mit Überhang und fassadenbündigem Zwerchgiebel realisiert ist.

Umsetzung eines Daches mit Überhang

Aus einer rechteckigen Grundfläche wird ein Satteldach mit Überhang generiert. In dem Beispiel in Abbildung 5.6 wurde diese Fläche durch einen Component-Split bei einem Quader generiert. Zur Verdeutlichung wurden die einzelnen Shapes mit Falschfarben dargestellt.



(a) Quader mit Dachsegment (b) Dach mit Überhang

Abbildung 5.6: Die Schritte zur Erstellung eines Dachs mit Überhang

Die Operation zur Erstellung eines Daches mit Überhang hat folgende Syntax:

```
roof(type,slope,overlap) id
```

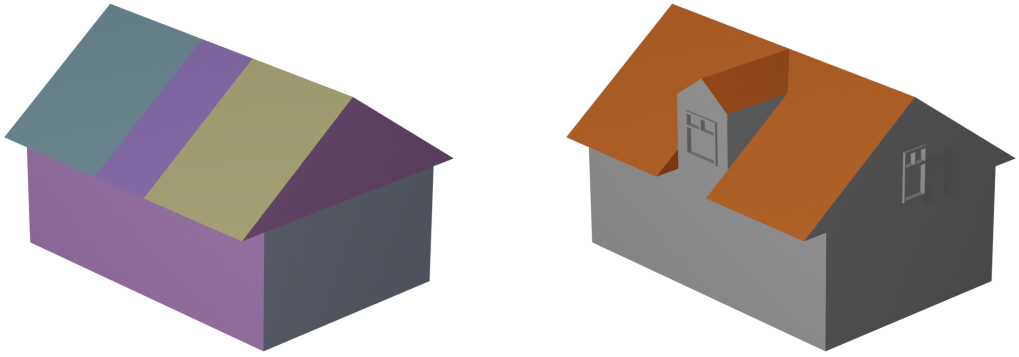
Der Parameter *type* legt den Dachtypen fest, *slope* die Neigung und *overlap* den Überhang des Daches. In dem Beispiel wird die Operation wie folgt aufgerufen:

```
Top --> roof("line",40,0.1) Roof
```

Um auf dem Dach nun einen Giebel zu generieren, muss dieses erst in seine Seitenflächen mit Hilfe eines Component-Split unterteilt werden. Danach wird die Seite auf der der Zwerchgiebel generiert werden soll und die dazugehörige Unterseite des Überhangs in einzelne Segmente unterteilt. Dies erfolgt mit der Split Operation. Das Ergebnis dieser Operationen ist in Abbildung 5.7a dargestellt. Auch hier wurden zur Verdeutlichung die einzelnen Shapes mit Falschfarben dargestellt. Der folgende Code-Abschnitt zeigt diese Operationen:

```
Roof --> component_split("side_faces"){Back,Side1,Front,Side2,
         Low1,Low2}
Side1 --> split("Z",1r,0.25,1r){Wall,DormerSegment,Wall}
Low1 -->  split("Z",1r,0.25,1r){Wall,Invisible,Wall}
```

Die beim Component-Split generierten Flächen mit den IDs *Low1* und *Low2* sind die unteren Flächen des Dachüberhangs und sind aus der Perspektive in Abbildung 5.7a nicht sichtbar. Mit den beiden Split Operationen wird eine Dachfläche und die dazugehörige Unterseite in die gewünschte Breite des Zwerchgiebels unterteilt.



(a) Dach nach Component-Split (b) Dach mit Zwerchgiebel

Abbildung 5.7: Die Schritte zur Erstellung eines Zwerchgiebels

Umsetzung des Zwerchgiebels

Um auf einem Dach mit Überhang einen fassadenbündigen Giebel zu erstellen, muss der Dachüberhang auch dem Zwerchgiebel als Parameter übergeben werden. Dies geschieht im Beispiel aus Abbildung 5.7b mit folgenden Werten:

```
DormerSegment --> dormer("wall",0.1) Dormer
```

So wird der Giebel fassadenbündig generiert und anschließend muss die Unterseite des Daches entfernt werden. Dies erfolgt mit der Operation Remove, die die Shape und die Kinder dieser Shape entfernt:

```
Invisible --> remove
```

Der Zwerchgiebel wird danach mit einem Component-Split bearbeitet, damit auf der Front ein Fenster generiert werden kann.

5.4.3 Realisierung von Tür und Fenstern

Die Fenster und die Tür wurden anders als die anderen Shapes nicht ausschließlich prozedural generiert, sondern ihre Form wurde in Blender[1] erstellt. Blender ist eine 3D-Grafiksuite, die es unter anderem ermöglicht Modelle in 3D zu erstellen. Die Tür und die Fenster wurden in Blender erstellt, weil auf diesem Wege sehr detaillierte Shapes mit geringem Aufwand generiert werden können. Dies konnte nur gemacht werden, weil aus ihnen keine neuen Shapes generiert werden sollen.

Abbildung 5.8: Die beiden Fenstertypen

In Abbildung 5.8 ist links das normale und rechts das zurückgesetzte Fenster zu sehen. Das zurückgesetzte Fenster und eine ebenfalls zurückgesetzte Tür wurden mit Blender erstellt und im Wavefront OBJ Format[2] exportiert. Beim Verwenden in der Shape-Grammar werden diese durch Skalierung für das Rechteck, auf dem sie generiert werden sollen, angepasst. Ein Beispiel für eine Regel, die eine zurückgesetzte Tür generiert, ist folgendes:

[1] https://www.blender.org/, abgerufen am 30.07.2020
[2] https://de.wikipedia.org/wiki/Wavefront_OBJ, abgerufen am 17.07.2020

```
DoorSegment --> detail("set_back_door") Door
```

Die Shape mit der ID *DoorSegment* muss dabei eine rechteckige Shape (*PolygonShape*) sein. Der Parameter *"set_back_door"* bestimmt dabei den Typ der *DetailShape*. Weitere mögliche Parameter sind *"window"* für das normale und *"set_back_window"* für das zurückgesetzte Fenster.

5.5 Stadtgenerierung

Zur Verwaltung von mehreren Gebäude und zur Erstellung der Stadt werden neue Datenstrukturen benötigt. Diese wurden wie in Abbildung 5.9 umgesetzt. Für die Generierung eines Gebäudes wird eine Grammatikdatei, aus der die Regeln eingelesen werden, und das Axiom (Länge und Breite des Grundrisses) benötigt. Damit man Gebäude auch an verschiedenen Positionen und Ausrichtungen darstellen kann, muss es auch einen Offset und eine Rotation geben.

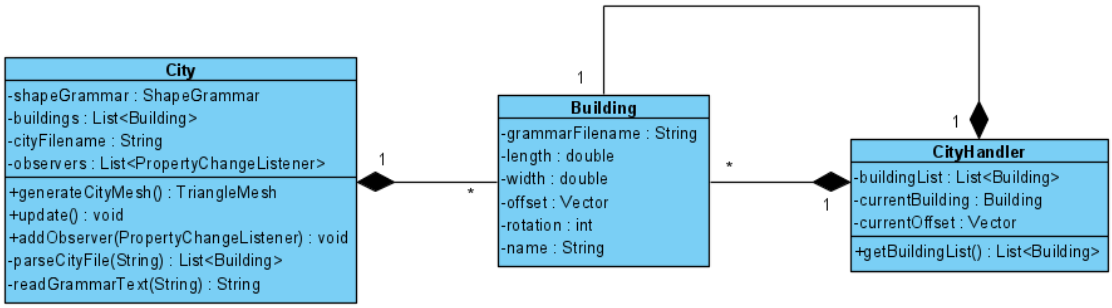


Abbildung 5.9: Klassendiagramm des Packages City

In der Klasse City werden mehrere Gebäude in einer Liste verwaltet. Diese werden aus einer XML-Datei unter Anwendung des SAX Parsers und dem CityHandler gelesen. Während der Stadtgenerierung werden alle Gebäude durchlaufen und jeweils die Dreiecksnetze unter Anwendung der Shape-Grammar erstellt. Diese werden dann zu einem gesamten Dreiecksnetz für die Stadt zusammengeführt.

5.6 Realisierung der Benutzeroberfläche

Die grafische Benutzeroberfläche besteht aus einem Eingabefenster und einem Fenster für die Ausgabe des generierten Gebäudes oder der Stadt. Das Eingabefenster ist in die Registerkarten *Grammar Editor, Settings, Axiom-Editor* und *City-Editor* unterteilt. Der *Axiom-Editor* ist für dieses System nicht weiter relevant. Der *Grammar Editor, Settings* und der *Axiom-Editor* sind im Projekt der Gruppe Computergrafik HAW bereits vorhanden, der *City-Editor* wurde hinzugefügt. In Abbildung 5.10 sieht man den *Grammar Editor* und das Ausgabefenster mit dem generierten Gebäude. In dem *Grammar Editor* hat der Benutzer direkten Zugriff auf das Axiom und auf die ausgewählte Grammatikdatei. Beides kann bearbeitet und es kann aus verschiedenen Grammatikdateien gewählt werden. Mit einem Klick auf den Button *Update* wird das ausgewählte Gebäude aus der Grammatikdatei und dem Axiom generiert und in dem Ausgabefenster ausgegeben.

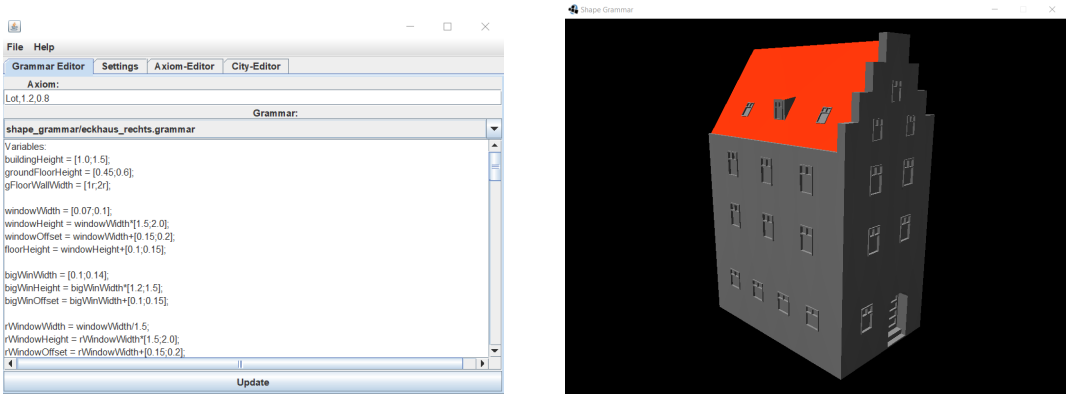


Abbildung 5.10: Ein- und Ausgabefenster für ein Gebäude

Im Tab *Settings* können verschiedene Einstellungen für die grafische Ausgabe vorgenommen werden. Unter anderem gibt es die Einstellung *showScope* um den Scope der einzelnen Shapes eines generierten Gebäudes anzuzeigen. Die Einstellung *useFalseColors* generiert ein Gebäude mit zufälligen Farben für jede Shape, was sehr hilfreich ist um Split- oder Component-Split Operationen zu evaluieren.

In der Registerkarte *City-Editor* (siehe Abbildung 5.11) kann man aus verschiedenen Stadtdateien wählen. Die ausgewählte Datei kann der Benutzer bearbeiten und somit können einzelne Gebäude verändert, hinzugefügt oder entfernt werden. Auch hier wird mit einem Klick auf den Button *Update* die ausgewählte Stadt generiert und im Ausgabefenster ausgegeben.

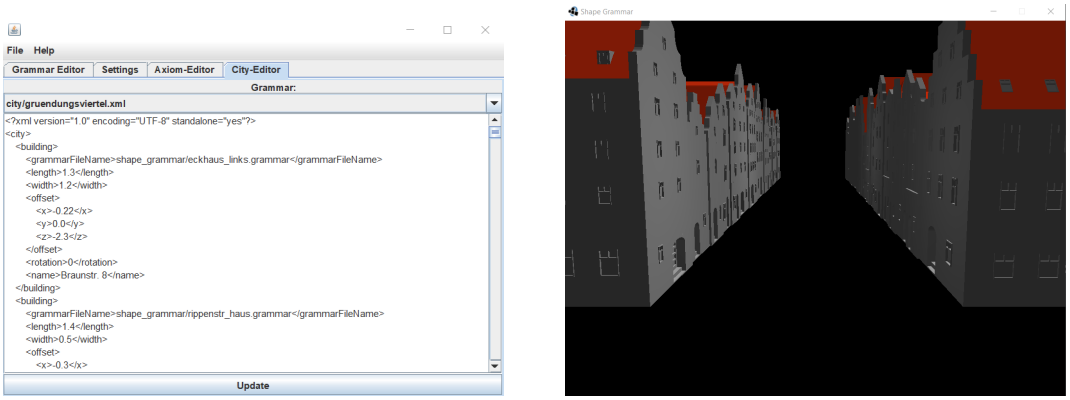


Abbildung 5.11: Ein- und Ausgabefenster für eine Stadt

Des Weiteren gibt es unter dem Reiter *File* die beiden Optionen *Export building to OBJ* und *Export city to OBJ*, die ein Mesh für ein Gebäude oder eine Stadt generieren und dieses dann im Wavefront OBJ Format exportieren. Damit können generierte Gebäude und Städte in anderen 3D-Grafikprogrammen weiterverwendet werden.

5.7 Tests

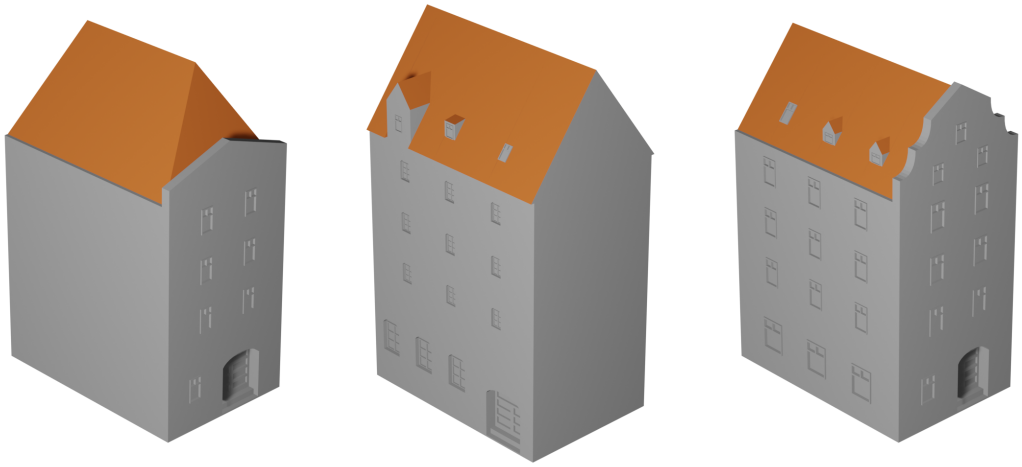
Um die korrekte Funktionsweise des Systems zu überprüfen, werden dessen Hauptkomponenten *Parser, Evaluator* und *MeshGenerator* getestet. Im Fall des *Parsers* wird kontrolliert, ob das Axiom, die Regeln und die Variablen richtig eingelesen werden. Im Rahmen des Tests des *Evaluators* wird ein Shape-Tree erstellt und dessen Korrektheit überprüft. Bei der Mesh-Generierung wird die grundlegende Datenstruktur getestet. Zudem wurde die Funktionalität der Bezierkurve geprüft.

Die für diese Arbeit erstellten Shapes und die dazugehörigen Shape-Operations werden nur visuell überprüft, da das Testen bei grafischen Darstellungen sehr komplex und in dieser Arbeit nicht zielführend ist.

6 Evaluation

6.1 Evaluierung der Gebäudegenerierung

Alle Gebäude können gemäß den im Abschnitt 4.3 formulierten Anforderungen generiert werden (siehe Abbildung 6.1). Eine Einschränkung ist, dass die Gebäude nur aus rechteckigen Grundrissen generiert werden können. Dies fällt bei den Lübecker Kaufmannshäusern nicht negativ auf, da deren Grundrisse in der Regel annähernd rechteckig sind. Um auch komplexere Grundrisse zu ermöglichen, müssten bei der Dachgenerierung Algorithmen wie zum Beispiel der Straight Skeleton Algorithmus [2] implementiert werden.



(a) Rippenstraßenhaus (b) Querstraßenhaus (c) Eckhaus

Abbildung 6.1: Die verschiedenen generierten Gebäudearten

Für die Generierung des Rippenstraßenhauses sind 17 Variablen und 50 Regeln notwendig. Die meisten dieser Regeln sind Split-Regeln, damit zum Beispiel die Fensterabstände zur Gebäudewand und untereinander eingehalten werden können. Für die anderen Gebäudetypen wurden eine ähnliche Anzahl an Variablen und Regeln verwendet. Es wurde

viel mit Regeln, die mit Wahrscheinlichkeiten ausgeführt werden, gearbeitet. Dadurch wird bei jeder Ausführung derselben Grammatikdatei ein Gebäude im selben Stil, jedoch mit unterschiedlichen Eigenschaften, erstellt.

Bei der Realisierung des Querstraßenhauses muss bei der Generierung eines Zwerchgiebels, wie in Abschnitt 5.4.2 beschrieben, gleichzeitig die Unterseite des Dachüberhangs entfernt werden. Dieses ist vom Benutzer des Systems vorzunehmen, wobei eine automatische Entfernung der Unterseite hier wünschenswert wäre. Diese Automation würde komplexere Beziehungen unter den Shapes erfordern, da die Unterseite und die Oberseite des Daches zu dem Zeitpunkt der Erstellung des Zwerchgiebels zwei unabhängige *PolygonShapes* sind.

Das Eckhaus wurde in zwei Grammatikdateien aufgeteilt, da nur für die in der Stadt sichtbaren Seiten der Gebäude Fassaden generiert werden. So hat das Rippenstraßenhaus zum Beispiel nur eine Fassade an Vorder- und Rückseite. Bei dem Eckhaus werden je nach Art der Ecke an der linken oder rechten Seite und an der Front eine Fassade generiert. Beim Eckhaus konnte kein Dachüberhang realisiert werden, weil der Überhang sonst in den Bereich des benachbarten Haus eindringt. Des weiteren wurde beim Eckhaus kein horizontal angeschlossener Giebel (Attika) verwendet, weil dieser nicht mit einem Satteldach kombiniert werden soll.

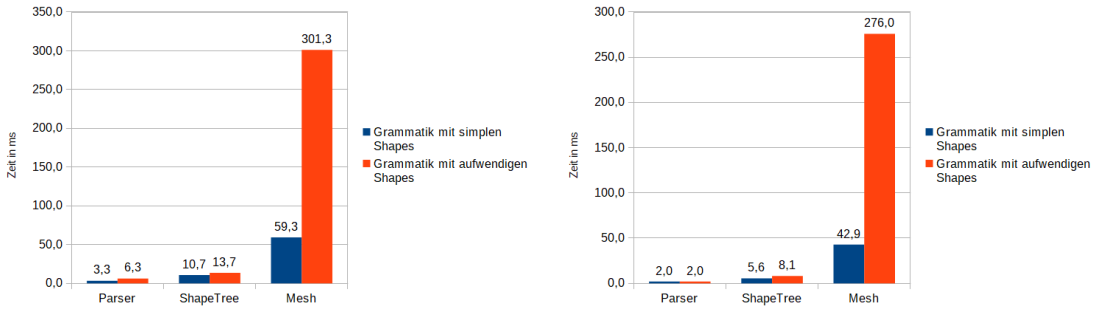
6.2 Evaluierung der Stadtgenerierung

Alle im Abschnitt 4.2 formulierten Anforderungen konnten erfüllt werden. Die Stadt wurde Straßen des Projekts "Gründungsviertel" in Lübeck nachempfunden, wobei die Grundrisse der Gebäude insoweit vereinfacht wurden, dass diese genau und nicht nur annäherungsweise rechteckig sind. Die Positionierung der Gebäude ist für den Anwender relativ aufwendig, da diese mit dem Abstand vom Koordinatenursprung angegeben ist. Hier wären auch andere Ansätze möglich gewesen, wie zum Beispiel die Grundrisse in Bildform als Stadtplan zu importieren. Dieser alternative Ansatz wurde nicht weiter verfolgt, da es nicht Ziel des Projekts ist und der Fokus auf der Gebäudegenerierung liegt.

6.3 Performance der Generierung

6.3.1 Gebäudegenerierung

Zur Auswertung der Performance bei der Gebäudegenerierung wurde diese in drei Phasen unterteilt: das Parsen der Grammatikdatei, die Erstellung des Shape-Tree und die Mesh-Generierung. Alle Messungen erfolgten mit dem Prozessor Intel Core i7-3630QM mit 2.40GHz.



(a) Erster Durchlauf (b) Wiederholter Durchlauf

Abbildung 6.2: Zeiten für die Durchläufe bei der Gebäudegenerierung

In Abbildung 6.2 sieht man den Unterschied zwischen dem ersten Generieren eines Gebäudes und dem wiederholten Generieren. Es wurden zwei Grammatiken getestet, die sich nur in ihren Fenstern unterscheiden (siehe Abbildung 5.8). Eines der Fenster hat ein sehr aufwendiges Mesh und das andere ein einfacheres. Der Unterschied zwischen erstem und wiederholtem Durchlaufen ist selbst bei der Grammatik mit den aufwendigen Fenstern mit einer Differenz von ca. 35ms nur gering. Die deutlichsten Zeitunterschiede treten bei der Mesh-Generierung auf. Je komplexer das Gebäude beziehungsweise die darin verwendeten Shapes, desto länger dauert die Generierung des Gebäudes. Die Anzahl der Regeln hat nur einen minimalen Einfluss auf die Gesamtzeit, da das Parsen und die Erstellung des Shape-Tree sehr schnell erfolgt.

6.3.2 Stadtgenerierung

Bei der Stadtgenerierung wurde der zeitliche Verlauf von zehn Durchläufen gemessen (siehe Abbildung 6.3). Hierbei wurden 64 Gebäude in einer Durchschnittszeit von ca. elf Sekunden generiert. Die hohen zeitlichen Abweichungen bei den einzelnen Durchläufen

hängen auch hier von den Fenstern ab. Je mehr Gebäude komplexe Fenster und damit auch ein komplexes Mesh haben, desto länger ist die Gesamtzeit beim Generieren der Stadt.

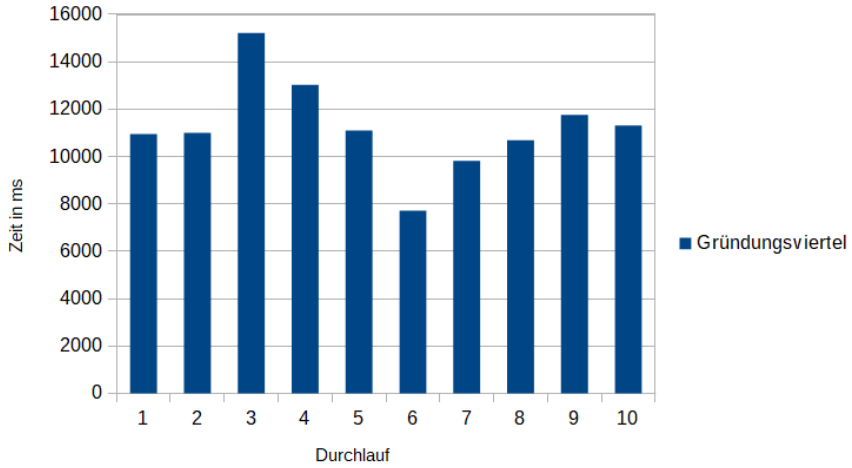


Abbildung 6.3: Zeiten für die Stadtgenerierung

7 Fazit und Ausblick

Diese Arbeit beschäftigt sich mit der Erweiterung einer Shape-Grammar zur Generierung von Gebäuden in einem bestimmten Baustil. Es soll untersucht werden, inwieweit ein System entwickelt werden kann, um ein Stadtviertel im Stil der Lübecker Kaufmannshäuser vom 12. bis zum 19. Jahrhundert automatisch zu generieren.

Dieses Ziel und die Anforderungen an die Gebäude und das Stadtviertel konnten umgesetzt werden. Dafür wurden einer bestehenden Grammatik neue Shapes und Shape-Operations hinzugefügt. Unter anderem sind Dächer, Giebel, Gauben und Fenster, die dem Baustil der Lübecker Kaufmannshäuser entsprechen, implementiert worden. Viele der implementierten Shapes können mit Hilfe der Shape-Grammar weiter bearbeitet und verwendet werden. Nur einige Dachformen, die Fenster und die Tür sind Terminal-Shapes, aus denen keine neuen Shapes generiert werden können.

Auf Basis des implementierten Systems sind mögliche Weiterentwicklungen in Bezug auf die Shapes denkbar. Der Giebel, der aktuell mit Schweifen oder Treppen verziert werden kann, könnte so zum Beispiel mit beliebigen Ornamenten für andere Baustile verwendet werden. Um die Dächer in anderen Architekturstilen mit nicht rechteckigen Grundrissen zu verwenden, müsste die Realisierung dieser angepasst werden. Dies ist unter Anwendung des Straight Skeleton Algorithmus [4] möglich. Des Weiteren können zusätzliche Details hinzugefügt werden, wie beispielsweise Schornsteine oder Regenrinnen. Zudem können die Gebäude um detailliertere Texturen für Dächer und Fassaden erweitert werden. Mit dieser Detailtiefe können die Gebäude noch realistischer dargestellt werden.

Über das implementierte Benutzerinterface kann die Grammatikdatei ausgewählt und bearbeitet werden, um so die Eigenschaften eines Gebäudes zu verändern. Eine mögliche Erweiterung des Benutzerinterfaces der Anwendung wäre zum Beispiel, dass als alternative Eingabeform Shapes oder Fassaden skizziert werden und die ähnlichste Shape ausgewählt wird (siehe Nishida et al. [10]).

Weiterhin wurde die Shape-Grammar um Variablen mit Zufallsintervallen und die Möglichkeit Regeln unter Bedingungen und mit Wahrscheinlichkeiten auszuführen, erweitert. Dadurch wird ermöglicht, dass ein Gebäudetyp mit einer Grammatik in einer hohen Varianz dargestellt werden kann. So wurden mit vier Haustypen und ihren jeweiligen Grammatiken 64 Häuser generiert und kein Gebäude sieht genau wie das andere aus.

Es wurde eine Möglichkeit entwickelt um mehrere Gebäude auf einmal zu generieren. Dazu werden Gebäude und ihre Eigenschaften, wie unter anderem die Position und ihre Grammatik, im XML-Format eingelesen. Dies kann dahin gehend erweitert werden, dass die Grundrisse aus einer Karte generiert werden [17] oder das aus einem System an Straßen automatisch Grundstücke erstellt werden.

Zusammenfassend lässt sich festhalten, dass der Einsatz einer Shape-Grammar insbesondere zur prozeduralen Generierung von Gebäuden und Städten sinnvoll ist. Es können automatisiert mehrere Gebäude in einem Baustil generiert werden und es muss nicht in Detailarbeit jedes Gebäude einzeln erstellt werden. Somit stellt dieser Ansatz eine gute Alternative für Anwendungsfälle, bei denen eine Generierung vieler Gebäude notwendig ist, dar. So sind diverse praktische Einsatzmöglichkeiten im Bereich der Stadtplanung oder für 3D-Simulationen denkbar und zum Teil bereits umgesetzt. Die vorliegende Arbeit bestätigt somit die unter Kapitel 3 beschriebenen Arbeiten und Projekte in diesem Kontext.

Literaturverzeichnis

[1] *ArcGIS CityEngine*. https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview. – abgerufen am 08.08.2020

[2] Aichholzer, Oswin ; Aurenhammer, F. ; Alberts, David ; Gärtner, Bernd: A Novel Type of Skeleton for Polygons. In: *J. UCS* 1 (1995), 01, S. 752–761

[3] Dylla, Kimberly ; Frischer, Bernard ; Mueller, Pascal ; Ulmer, Andreas ; Haegler, Simon: Rome Reborn 2.0: A Case Study of Virtual City Reconstruction Using Procedural Modeling Techniques. In: *Computer Graphics World* 16 (2008), 01

[4] Felkel, Petr ; Obdrzalek, Stepan: Straight Skeleton Implementation. In: *Proceedings of Spring Conference on Computer Graphics*, 1998, S. 210–218

[5] Kelly, Tom ; Wonka, Peter: Interactive Architectural Modeling with Procedural Extrusions. In: *ACM Trans. Graph.* 30 (2011), April, Nr. 2. – URL https://doi.org/10.1145/1944846.1944854. – ISSN 0730-0301

[6] Lipp, Markus ; Wonka, Peter ; Wimmer, Michael: Interactive Visual Editing of Grammars for Procedural Architecture. In: *ACM Trans. Graph.* 27 (2008), August, Nr. 3, S. 1–10. – URL https://doi.org/10.1145/1360612.1360701. – ISSN 0730-0301

[7] Müller, Pascal ; Vereenooghe, Tijl ; Wonka, Peter ; Paap, Iken ; Van Gool, Luc: Procedural 3D Reconstruction of Puuc Buildings in Xkipché. In: *Proceedings of the 7th International Conference on Virtual Reality, Archaeology and Intelligent Cultural Heritage*. Goslar, DEU : Eurographics Association, 2006 (VAST'06), S. 139–146. – ISBN 3905673428

[8] Müller, Pascal ; Wonka, Peter ; Haegler, Simon ; Ulmer, Andreas ; Van Gool, Luc: Procedural Modeling of Buildings. In: *ACM Trans. Graph.* 25

(2006), Juli, Nr. 3, S. 614–623. – URL https://doi.org/10.1145/1141911.1141931. – ISSN 0730-0301

[9] Müller, Pascal ; Zeng, Gang ; Wonka, Peter ; Van Gool, Luc: Image-Based Procedural Modeling of Facades. In: *ACM Trans. Graph.* 26 (2007), Juli, Nr. 3, S. 85–es. – URL https://doi.org/10.1145/1276377.1276484. – ISSN 0730-0301

[10] Nishida, Gen ; Garcia-Dorado, Ignacio ; Aliaga, Daniel ; Benes, Bedrich ; Bousseau, Adrien: Interactive sketching of urban procedural models. In: *ACM Transactions on Graphics* 35 (2016), 07, S. 1–11

[11] Parish, Yoav I. H. ; Müller, Pascal: Procedural Modeling of Cities. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : Association for Computing Machinery, 2001 (SIGGRAPH '01), S. 301–308. – URL https://doi.org/10.1145/383259.383292. – ISBN 158113374X

[12] Prusinkiewicz, Przemyslaw ; Lindenmayer, Aristid: *The Algorithmic Beauty of Plants*. Berlin, Heidelberg : Springer-Verlag, 1996. – ISBN 0387946764

[13] Schwarz, Michael ; Müller, Pascal: Advanced Procedural Modeling of Architecture. In: *ACM Trans. Graph.* 34 (2015), Juli, Nr. 4. – URL https://doi.org/10.1145/2766956. – ISSN 0730-0301

[14] Sipser, Michael: *Introduction to the Theory of Computation*. Second. Course Technology, 2006. – ISBN 7111173279 9787111173274

[15] Stiny, G: Introduction to Shape and Shape Grammars. In: *Environment and Planning B: Planning and Design* 7 (1980), Nr. 3, S. 343–351. – URL https://doi.org/10.1068/b070343

[16] Stiny, George ; Gips, James: Shape Grammars and the Generative Specification of Painting and Sculpture, 01 1971, S. 1460–1465

[17] Vanegas, Carlos A. ; Kelly, Tom ; Weber, Basil ; Halatsch, Jan ; Aliaga, Daniel G. ; Müller, Pascal: Procedural Generation of Parcels in Urban Modeling. In: *Comput. Graph. Forum* 31 (2012), Mai, Nr. 2pt3, S. 681–690. – URL https://doi.org/10.1111/j.1467-8659.2012.03047.x. – ISSN 0167-7055

[18] Whiting, Emily ; Ochsendorf, John ; Durand, Frédo: Procedural Modeling of Structurally-Sound Masonry Buildings. In: *ACM Trans. Graph.* 28 (2009), Dezember, Nr. 5, S. 1–9. – URL https://doi.org/10.1145/1618452.1618458. – ISSN 0730-0301

[19] Wonka, Peter ; Wimmer, Michael ; Sillion, François ; Ribarsky, William: Instant Architecture. In: *ACM Trans. Graph.* 22 (2003), Juli, Nr. 3, S. 669–677. – URL https://doi.org/10.1145/882262.882324. – ISSN 0730-0301

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Beispiel-getriebene prozedurale Gebäude-Generierung unter Anwendung einer Shape-Grammar

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original