Hochschule für Technik, Wirtschaft und Kultur Leipzig (FH)
Fachbereich Informatik, Mathematik und Naturwissenschaften

Helmholtz-Zemtrum für Umweltforschung UFZ
Department Umweltinformatik

Prozedurale Erzeugung von Modellen für die interaktive Visualisierung von Stadtgebieten der Gründerzeit

Masterarbeit
von

B.Sc. Lars Bilke

Betreuer: Prof. Dr.-Ing. habil. Dieter Vyhnal (HTWK Leipzig)
Dr. Björn Zehner (UFZ Leipzig)

Lars Bilke
An der Linde 5A
04420 Markranstädt

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Markranstädt, den 16. Januar 2009

____
Lars Bilke

Inhaltsverzeichnis

Abbildungsverzeichnis **4**

Tabellenverzeichnis **6**

1 Einführung und Zielstellung **8**
1.1 Prozedurale Modellierung . . . 9
1.2 Bisherige Arbeiten auf dem Gebiet . . . 10

2 Software-Architektur **12**
2.1 Szenendefinition - CityData-Klasse . . . 14
2.1.1 Allgemeiner Aufbau der Szene . . . 14
2.1.2 XML-Format zur Szenendefinition . . . 14
2.1.3 Implementierung der CityData-Klasse . . . 17
2.2 Regelsystem - RuleProcessor-Klasse . . . 22
2.2.1 Formale Grammatiken . . . 22
2.2.2 Regelsystem-Grammatik . . . 22
2.2.3 XML-Format zur Regeldefinition . . . 24
2.2.4 Implementierung der RuleProcessor-Klasse . . . 27
2.3 Ressourcenverwaltung - CityResourceManager-Klasse . . . 32
2.3.1 Ressourcen . . . 32
2.3.1.1 3D-Modelle . . . 32
2.3.1.2 Texturen . . . 32
2.3.1.3 Beleuchtungsmodelle mittels OpenGL Shading Language . . . 35
2.3.1.4 Materialien . . . 41
2.3.2 Implementierung der CityResourceManager-Klasse . . . 42
2.4 Gebäudegenerator - BuildingGenerator-Klasse . . . 46
2.4.1 Einfache Gebäudemodelle mit Fassadentexturen . . . 46
2.4.2 Prozedurale Gebäudemodelle . . . 46
2.4.3 Dächergenerieung . . . 46
2.4.4 Implementierung . . . 47

3 Anwendungen **54**
3.1 Facade Editor . . . 54
3.1.1 Einführung . . . 54
3.1.2 Bedienung . . . 55
3.1.2.1 Die Oberfläche . . . 55
3.1.2.2 Hauptmenü . . . 55

3.1.2.3 Werkzeugleiste 56
3.1.2.4 Regelfenster 57
3.1.2.5 Eigenschaftenfenster 58
3.1.2.6 3D-Fenster 58
3.1.3 Implementierung 59
3.2 CityGeneratorCL 62
3.2.1 Bedienung 62
3.2.2 Implementierung 63
3.3 Exportmöglichkeiten der generierten Modelle 64
3.3.1 Einführung 64
3.3.2 Implementierung 64
3.3.3 Ergebnis 65

4 Erstellung von Regeln und Ressourcen für Gründerzeithäusermodelle 68
4.1 Einführung in die Architektur der Gründerzeit 68
4.2 Erstellen von 3D-Geometrien in Cinema 4D 69
4.3 Erstellen von Normalen- und Specular-Texturen 72
4.4 Erarbeiten eines Regelsets 72
4.4.1 Regeln zur Fassadenstrukturierung 72
4.4.2 Regeln für Fassadenkacheln 76
4.4.3 Regeln für Verzierungen 84

5 Beispiel eines prozedural generierten Stadtgebiets 92

6 Vergleich zwischen manueller und prozeduraler Modellierung 95
6.1 Vorstellung des City3D-Projektes 95
6.2 Die City3D-Stadtszene mit dem CityGenerator erstellen 95
6.2.1 Importieren von Daten aus OpenStreetMap 96
6.2.2 Generieren der OpenStreetMap-Daten 97
6.3 Vergleiche 98

7 Ausblick 102
7.1 Problemfelder 102
7.2 Möglichkeiten zur Weiterentwicklung 102

A Koordinatensysteme 106

B Kompilieren des Visual Studio Projekts und Begleit-CD 107

Literaturverzeichnis 110

Abbildungsverzeichnis

1.1 Visualisierungszentrum des UFZ mit einer Echtzeit-Stadtvisualisierung (siehe auch Kapitel 6), Foto: B. Zehner . . . 8
1.2 Aus einem Grundriss und einigen Parametern prozedural erzeugtes Gebäudemodell . . . 10
1.3 Mithilfe von Lindenmayer-Systemen prozedural generierte Pflanzenmodelle . . 11

2.1 Übersicht aller CityGenerator-Klassen im UML-Diagramm . . . 12
2.2 Schematische Darstellung der Szenenobjekte . . . 14
2.3 XML-Schemadefinition der Szenendatei . . . 15
2.4 UML-Diagramm der CityData-Klasse und zugehörige Klassen . . . 17
2.5 UML-Diagramm der zur CityData-Klasse gehörigen Hilsfklassen . . . 18
2.6 Blockunterteilungsalgorithmus nach Müller aus [Müller 01] . . . 19
2.7 Ergebnis der straßenparallelen Gebäudeblock-Zerlegung . . . 21
2.8 Schematische Darstellung der Regeltypen . . . 24
2.9 XML-Schemadefinition der Regeldatei . . . 25
2.10 UML-Diagramm der Ruleprocessor-Klasse und zugehörigen Klassen . . . 27
2.11 Exemplarischer Ablauf einer Split-Regel . . . 29
2.12 Vue Infinite Optionen zum Export von Cubemaps . . . 33
2.13 Vertikale, kreuzförmige Cubemap-Anordnungen . . . 33
2.14 ATI CubeMapGen-Optionen . . . 34
2.15 Horizontale, kreuzförmige Cubemap-Anordnungen . . . 34
2.16 Visualisierung des Phong-Beleuchtungsmodells: Die Lichtfarbe ist Weiß, die ambiente und diffuse Farbe ist jeweils Blau und die spiegelnde Farbe ist Weiß. Quelle: [Wikipedia 09] . . . 35
2.17 Links: Vektoren zur Licht- und Reflexionsberechnung, Rechts: Vektoren zur Berechnung des Umgebungslichtes und der Reflexion . . . 37
2.18 Links: Vektoren zur Lichtberechnung, Rechts: Basisvektoren des Tangentenraums 39
2.19 UML-Diagramm der BuildingGenerator- und der CityResourceManager-Klasse 42
2.20 VRML2-Export-Struktur aus Cinema4D . . . 43
2.21 Bestandteile eines Materials implementiert als ChunkMaterial . . . 45
2.22 Schematische Darstellung eines Giebeldaches und eines flachen Mansarddaches 46
2.23 Zwei Fassadentexturen, die je nach Größe der Fassade und der Etagenhöhe oft gekachelt werden . . . 48
2.24 Beispiel zur Texturkoordinatengenerierung der einfachen Fassaden mit zweifacher Kachelung in vertikaler und vierfacher Kachelung in horizontaler Richtung 49

3.1 Das Facade Editor-Programmlayout . . . 54
3.2 Übersicht aller FacadeBuilder-Klassen in einem UML-Diagramm . . . 59

3.3 UML-Diagramm der MainWindow-Klasse und zugehörige Klassen 61
3.4 UML-Diagramm der FacadeBuilder-, RulesTreeCtrl-Klasse und zugehörige Klassen 62
3.5 UML-Diagramm der CityBuilder-Klasse 63
3.6 In 3D-Modellierungsanwendungen exportierte Häusermodelle 67

4.1 Typische Gründerzeitarchitektur im Leipziger Waldstraßenviertel, Rechts: Detailaufnahme des Eingangsbereiches 68
4.2 Weitere Gründerzeitstilhäuser im Leipziger Waldstraßenviertel, Rechts unten: Nahaufnahme von Stuckverzierungen 69
4.3 Ein Fensterrahmen-Modell in Cinema 4D mit seitlicher Detailansicht 70
4.4 Texturkoordinatengenerierung des Fensterrahmen-Modells in Cinema 4D . . . 71
4.5 3D-Geometrien mit zugewiesenen Materialien in Cinema 4D 71
4.6 Alle mit Cinema 4D erstellten Geometrien 73
4.7 Shadermap-Oberfläche mit Vorschaufenster 74
4.8 Möglichkeiten zur vertikalen Unterteilung einer Fassade 74
4.9 Verschiedene Möglichkeiten zur horizontalen Anordnung der Fassade 76
4.10 Verschiedene Möglichkeiten zur Anordnung der Erdgeschosskacheln 81
4.11 Verschiedene Möglichkeiten zur Anordnung der Eingangsbereichkacheln 84
4.12 Verschiedene Möglichkeiten zur Anordnung von Ornament-Elementen 85
4.13 Verschiedene Möglichkeiten für Balkonmodelle 88
4.14 Die Dachrinne mit Fallrohr 91

5.1 Links: Luftbildaufnahme des Gebietes um die Etkar-André-Straße, Quelle: [Google 09], Rechts: OpenStreetMap-Daten des Gebietes mit den gelb markierten, erstellten Gebäuden 92
5.2 Prozedural erzeugtes Modell einiger Häuserblöcke um die Etkar-André-Straße 93
5.3 Oben: Foto mit Blick auf die Häuser der Etkar-André-Straße und eine zugehörige Fassade, Mitte: Prozedural erzeugtes Modell, das in Cinema 4D gerendert wurde, Unten: Dasselbe Modell in VRED als Echtzeitvisualisierung 94

6.1 Der vom City3D-Projekt visualisierte Stadtbereich ist hell hervorgehoben, Quelle: 95 96
6.2 Links: Daten aus OpenStreetMap-Website exportieren, Rechts: JOSM-Fenster mit den geladenen OSM-Daten der Eisenbahnstraße 97
6.3 Links: Die bereinigten OSM-Daten der Eisenbahnstraße und Luftbildaufnahmen im Hintergrund, Mitte: hinzugefügte Gebäudeblöcke, Rechts: hinzugefügte Gebäudegrundrisse 98
6.4 Die City3D-Stadtszene in VRED, Oben: manuell modelliert, Unten: prozedural erstellt 99
6.5 Detailansicht eines Straßenzuges im City3D-Projekt, Oben: manuell modelliert, Unten: prozedural erstellt 100

7.1 CityGenerator Workflow-Übersicht 103
7.2 Oben: Normales Rendering, Mitte: mit Screen Space Ambient Occlusion-Rendering, unten: Ambient Occlusion-Textur 105

A.1 Koordinatensysteme 106

Tabellenverzeichnis

2.1 Übersicht der durchzuführenden Transformationen der aus CubeMapGen exportierten Bilder und deren Entsprechung im ENU-Koordinatensystem 34
2.2 Übersicht Modeltypen 43
2.3 Übersicht Texturenverzeichnisse 44
2.4 Übersicht Materialzusammensetzungen 44

Kapitel 1

Einführung und Zielstellung

Zukünftige Projekte am Visualisierungszentrum des Helmholtz-Zentrum für Umweltforschung UFZ (siehe Abbildung 1.1) werden vielfach die Visualisierung von Landschaften und die Darstellung von städtischen Umgebungen beinhalten, welche gerade im Raum Leipzig oft aus der Gründerzeit stammen. Hierbei kommt es eher auf die Gesamtwirkung der Umgebung an (wie z.B. wirkt es, wenn innerhalb der Gründerzeitbebauung einzelne Gebäude abgerissen werden), als darauf, die echten Gebäude originalgetreu nachzumodellieren.

Abbildung 1.1: Visualisierungszentrum des UFZ mit einer Echtzeit-Stadtvisualisierung (siehe auch Kapitel 6), Foto: B. Zehner

Im Rahmen eines Praktikums sind am UFZ von Miguel Fonseca und Michael Vieweg erste Versuche unternommen worden, Stadtlandschaften darzustellen und es wurde die Möglichkeit geschaffen, exemplarisch leichte Veränderungen daran zu visualisieren. Die verwendeten Modelle für die Häuser wurden jedoch von Hand erstellt und sind sehr einfache Geometrien, bei denen die Fassaden mithilfe von Texturen dargestellt werden. Beleuchtungsmodelle lassen sich hierauf nur sehr begrenzt anwenden und der Blick entlang der Straßenzüge wirkt unrea-

listisch weil z.B. die Fassaden flach sind und keinerlei Einsprünge aufweisen. Eine genauere Modellierung von einzelnen Gebäuden mit Stuckelementen und Erkern würde jedoch einen zu hohen Aufwand bedeuten.

Im Rahmen dieser Masterarbeit[1] sollen Anwendungen geschaffen werden, mit denen Gebäude, als auch ganze Stadtgebiete, prozedural aus einfachen Bausteinen, wie Fassadenelementen mit Türen und Fenstern, interaktiv erstellt werden können. Die geschaffenen Gebäude und Stadtgebiete werden mithilfe eines Eingabedatenformats erzeugt und sind aufgrund ihrer prozeduralen Natur durch Ergänzung mit zufällig erzeugten Merkmalen variierbar. Die Gebäude sollen sich stilistisch an Leipziger Gründerzeithäusern orientieren. Die Stilvorgabe soll exemplarisch die Möglichkeiten der Anwendungen verdeutlichen. Hierbei soll jedoch ein universelles System zur Häusergenerierung geschaffen werden, womit auch Modelle von anderen Häusertypen, beispielsweise Plattenbauten, generiert werden können. Eine Randbedingung dabei ist, dass die erzeugten Modelle im Visualisierungszentrum des Helmholtz-Zentrums für Umweltforschung interaktiv lauffähig sein sollen und daher entsprechend optimiert werden müssen. Auf dem Rechencluster des Visualisierungszentrums kommt das Szenengraphensystem *OpenSG* zum Einsatz, das es ermöglicht, mehrere Rechner (*Clusternodes*) von einem Anwendungsrechner zur Berechnung der 3D-Grafik anzusteuern. Die Clusternodes berechnen die ihnen zugeteilte Ansicht der 3D-Szene unabhängig voneinander, aber in Synchronisation mit dem Anwendungsrechner. Des Weiteren soll eine Möglichkeit erarbeitet werden, die erstellten Gebäude in einem allgemein gültigen Austauschformat exportieren zu können.

1.1 Prozedurale Modellierung

Frei aus [Ebert 98] übersetzt, ist eine „prozedurale Technik im Allgemeinen ein Algorithmus, der eine computergenerierte Sache, z.B. 3D-Geometrien und Texturen, als eine Abfolge von Anweisungen beschreibt, die diese Sache erzeugen. Im Gegensatz zur herkömmlichen Erzeugung solcher Daten, werden nicht hoch komplexe Daten mit allen Details benötigt, sondern diese werden in einer Funktion oder in einem Algorithmus abstrahiert. Die vielen kleinen Details einer Marmortextur zum Beispiel, müssen nicht mehr aufwendig gespeichert werden, sondern sind Teil des Algorithmus und werden bei Bedarf erzeugt. Die Anweisungen können parametrisiert werden, um so eine Vielfalt an Ergebnissen zu erhalten. Wichtige Eigenschaften des Ergebnisses werden mit einem Parameter versehen. Dadurch muss der Benutzer dieser Techniken nicht die zugrunde liegenden und evtl. sehr komplexen Algorithmen verstehen, sondern nur wissen, wie ein Parameter sich auf das Endergebnis auswirkt.“

Für ein einfaches Beispiel zur prozeduralen Generierung von Häusern würde dies bedeuten, dass man aus einem Rechteck (einfache Startdaten), das den Gebäudegrundriss darstellt, und verschiedenen Eigenschaften, wie z.B. Gebäude-, Etagen- und Dachhöhe (als Parameter des Algorithmus) eine quaderförmige Gebäudegeometrie mit aufgesetztem Dach erzeugt. Das 3D-Modell als Ergebnis des Algorithmus könnte z.B. eine entsprechend dem Etagenhöhen-Parameter oft gekachelte Foto-Textur eines Gebäudestockwerkes und ein Dach mit der im entsprechenden Parameter gesetzten Höhe, wie in Abbildung 1.2 dargestellt, sein.

Prozedurale Algorithmen basieren oft auf formalen Grammatiken, die eine formale Sprache beschreiben. Die bekanntesten prozeduralen Algorithmen sind die *Lindenmayer*-Systeme, die in der Computergrafik zur Erzeugung von Fraktalen und Modellierung von Pflanzen eingesetzt

[1] Der vorliegende Text ist auf Basis des Latex-Templates zu [Gockel 08] erstellt.

Abbildung 1.2: Aus einem Grundriss und einigen Parametern prozedural erzeugtes Gebäudemodell

werden (siehe Abbildung 1.3). Eine kurze Einführung in formale Grammatiken wird in Kapitel 2.2.1 gegeben.

1.2 Bisherige Arbeiten auf dem Gebiet

Die wichtigsten und fortgeschrittensten Arbeiten auf dem Gebiet kommen von einer Gruppe rund um den ehemaligen ETH Zürich-Studenten Pascal Müller. Sie haben das Programm *CityEngine* entwickelt, welches von der prozeduralen Modellierung von einem Straßennetzwerk über die Unterteilung der Gebäudeflächen bis hin zu komplett prozedural erzeugten Städten einen großen Funktionsumfang bietet (siehe [Inc. 08]). Zu Beginn der Masterarbeit war das Programm noch nicht auf dem Markt und es gab auch keine Hinweise, dass ein solches in nächster Zeit veröffentlicht werden würde. Die Gruppe beschäftigt sich seit etwa 7 Jahren mit der Thematik und es sind bereits zahlreiche Publikationen veröffentlicht wurden. In seiner Diplomarbeit [Müller 01] beschreibt Müller die Generierung des Straßennetzwerkes und zeigt erste Ansätze zur Generierung der Gebäude. Hierbei kommen erweiterte Lindenmayer-Systeme (kurz: L-Systeme) zum Einsatz. [Parish 01] fasst die Arbeit in Kurzform zusammen.

Abbildung 1.3: Mithilfe von Lindenmayer-Systemen prozedural generierte Pflanzenmodelle

In [Wonka 03] stellt Wonka die so genannte *Split Grammar* vor, durch die Häuserfassaden aus einzelnen Blöcken wie Fenster, Türen und Verzierungen zusammengesetzt werden können. Dabei stellt eine Fassade ein Nichtterminalsymbol dar, das wieder in weitere Nichtterminalsymbole unterteilt wird. Zum Schluss werden alle Symbole in Terminalsymbole überführt, die dann die eigentlichen 3D-Geometrien bilden. In [Müller 06b] geht es um die prozedurale Modellierung von Gebäuden und die *CGA Shape*-Grammatik wird vorgestellt, die auf der Grammatik von Wonka aufbaut und zusammen mit parametrischen und stochastische L-Systemen im Programm *CityEngine* zum Einsatz kommt. Müller und Wonka beschäftigen sich in [Müller 07] mit der bildbasierten Modellierung von Fassaden. Hierbei werden Fassadenmerkmale aus Fotos von echten Fassaden extrahiert und darauf basierend die entsprechende Grammatik abgeleitet, mit der das Gebäude prozedural rekonstruiert werden kann.

In [Greuter 03] geht es ebenfalls um die Echtzeit-Generierung und -Darstellung von prozedural erzeugten Städten. Die Häuser werden jedoch lediglich flach texturiert und fallen somit nicht sehr detailliert aus. [Kelly 06] gibt einen gewissen Überblick über prozedurale Techniken zur Generierung von Gebäuden und Städten und vergleicht verschiedene bestehende Ansätze miteinander. Außerdem wurde in dem Kinofilm *King Kong* von *Peter Jackson* das New York der 1930er Jahre durch eine Software namens *CityBot* von *WETA Digital* prozedural erstellt. Jedoch ist die Software nicht verfügbar und es finden sich keine weiteren Informationen dazu.

Aktuelle Arbeiten zeigen erweiterte Möglichkeiten zur interaktiven Straßennetzgenerierung [Chen 08] und zur interaktiven und visuellen Bearbeitung von prozedural erzeugten Häuserfassaden [Lipp 08] auf.

Kapitel 2

Software-Architektur

Wie im UML-Diagramm in Abbildung 2.1 zu sehen, gliedert sich die in C++ geschriebene Software im Wesentlichen in vier Hauptbestandteile. Die *CityData*-Klasse implementiert Funktionen und Datenstrukturen zum Erstellen und Speichern von Stadtszenen (siehe Kapitel 2.1.3). Die Klasse *RuleProcessor* erzeugt aus architektonischen Regeln eine abstrakte Repräsentation von Fassadenmodellen (siehe Kapitel 2.2.4).

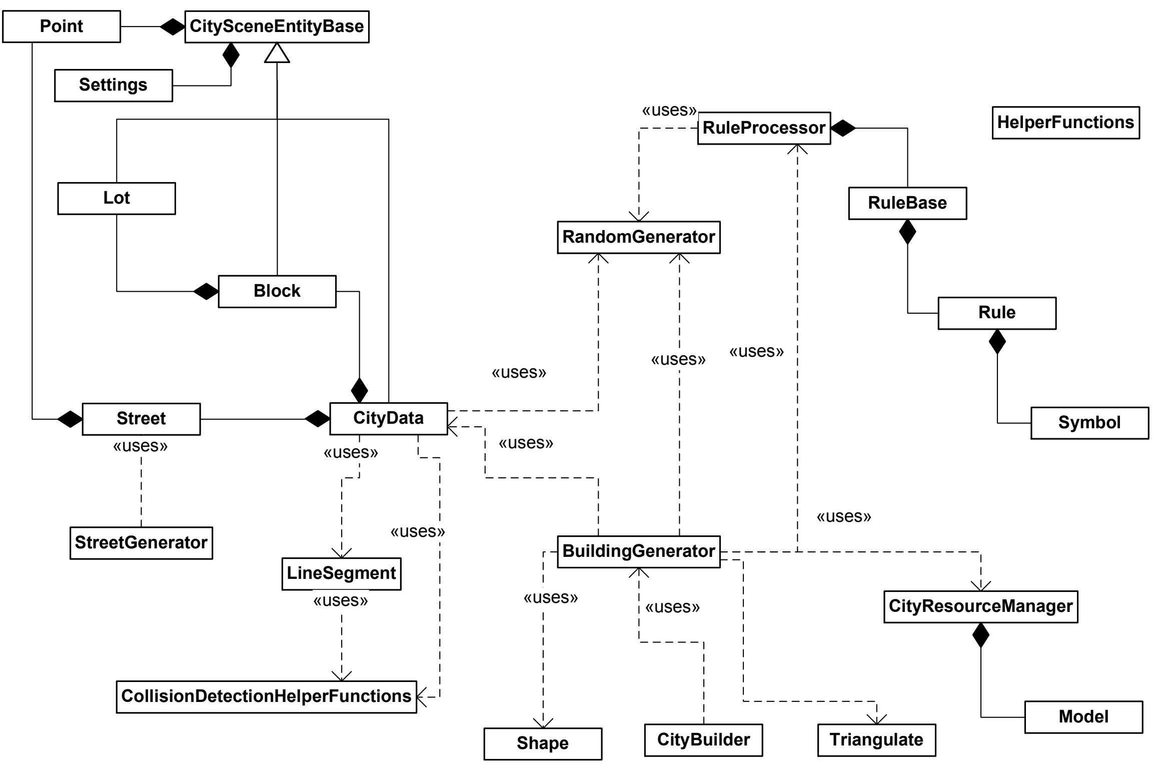


Abbildung 2.1: Übersicht aller CityGenerator-Klassen im UML-Diagramm

Die *CityResourceManager*-Klasse lädt, verwaltet und stellt die zur Erzeugung von Gebäudemodellen benötigten Resourcen, wie Texturen und 3D-Geometrien, bereit (siehe Kapitel 2.3.2). Die Klasse *BuildingGenerator* verbindet die Funktionalitäten der zuvor genannten

Klassen und erzeugt schließlich die 3D-Modelle der Häuser (siehe Kapitel 2.4.4). Im jeweiligen Kapitel ist ein detaillierteres UML-Diagramm der einzelnen Klassen zu finden. Aufgrund der beschränkten Skalierbarkeit der einzelnen Klassen-Diagramme wurden bei einigen die Funktionsparameter entfernt, da die Diagramme sonst nicht übersichtlich dargestellt werden konnten. Dies ist jedoch im jeweiligen Diagramm vermerkt. In den folgenden Implementierungsabschnitten werden wichtige Programmcodeteile erläutert.

2.1 Szenendefinition - CityData-Klasse

2.1.1 Allgemeiner Aufbau der Szene

Eine Stadtszene soll aus Gebäudeblöcken bestehen, die ggf. von Straßen umgeben sein können. Innerhalb eines Gebäudeblocks können sich Gebäudegrundrisse befinden. Sowohl Straßen und Blöcke, als auch Grundrisse werden durch Punkte im dreidimensionalen Raum definiert. Blöcke und Grundrisse werden durch mindestens 3 nicht-kollineare Punkte definiert. Außerdem wird festgelegt, dass Punkte für Grundrisse in einer horizontalen Ebene liegen (die gleiche Höhenkoordinate haben). Eine Straße wird durch einen Anfangs- und einen Endpunkt definiert. Eine Szene besteht also aus einer Menge an Punkten, die von den Szenenobjekten referenziert werden (siehe Abbildung 2.2).

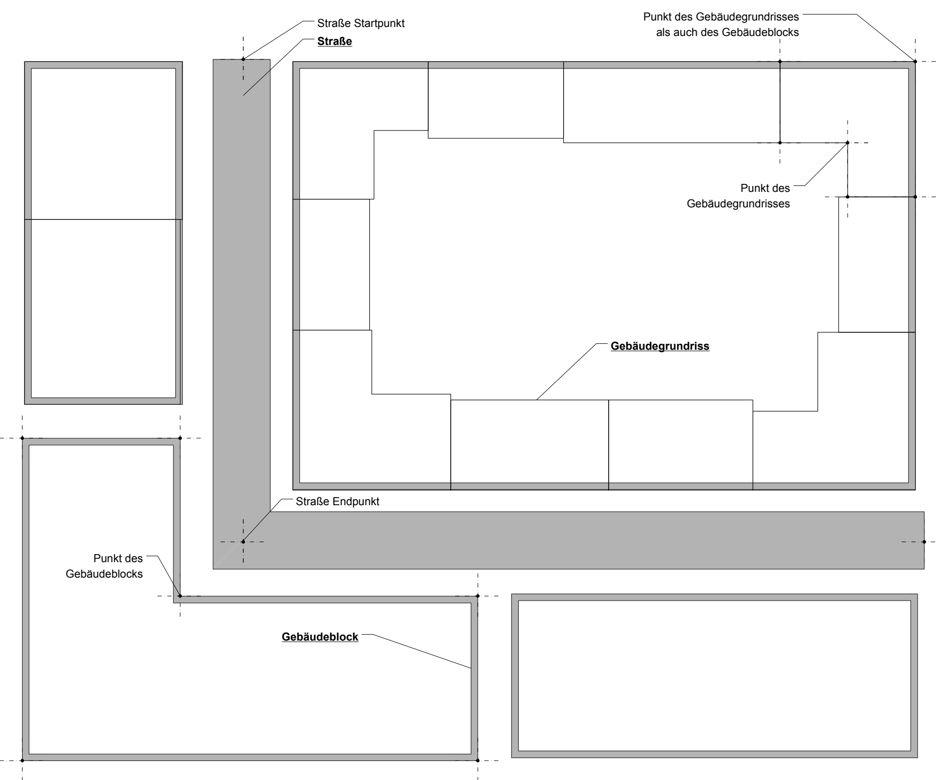


Abbildung 2.2: Schematische Darstellung der Szenenobjekte

2.1.2 XML-Format zur Szenendefinition

Nachfolgend wird das entwickelte Szenenformat zur Definition von Stadtszenen vorgestellt. Als grundlegende Form wurde XML aufgrund seiner guten Lesbarkeit für Menschen und die einfache maschinelle Verarbeitung durch frei erhältliche Parser gewählt. XML als Grundlage ermöglicht es, auch später noch Änderungen und Ergänzungen am Format vorzunehmen. So wäre es denkbar, später weitere Elemente von Stadtmodellen, z.B. Bepflanzung, hinzuzufügen. Des Weiteren kann durch eine XML-Schemadefinition (siehe Abbildung 2.3) die Struktur des Formats eindeutig festgelegt werden und XML-Dateien auf ihre Richtigkeit mit einem

Validierer überprüft werden. Zum Einsatz kam der als Erweiterung für Visual Studio frei erhältliche XML-Editor und -Validierer *Liquid XML Studio*[Technologies 08].

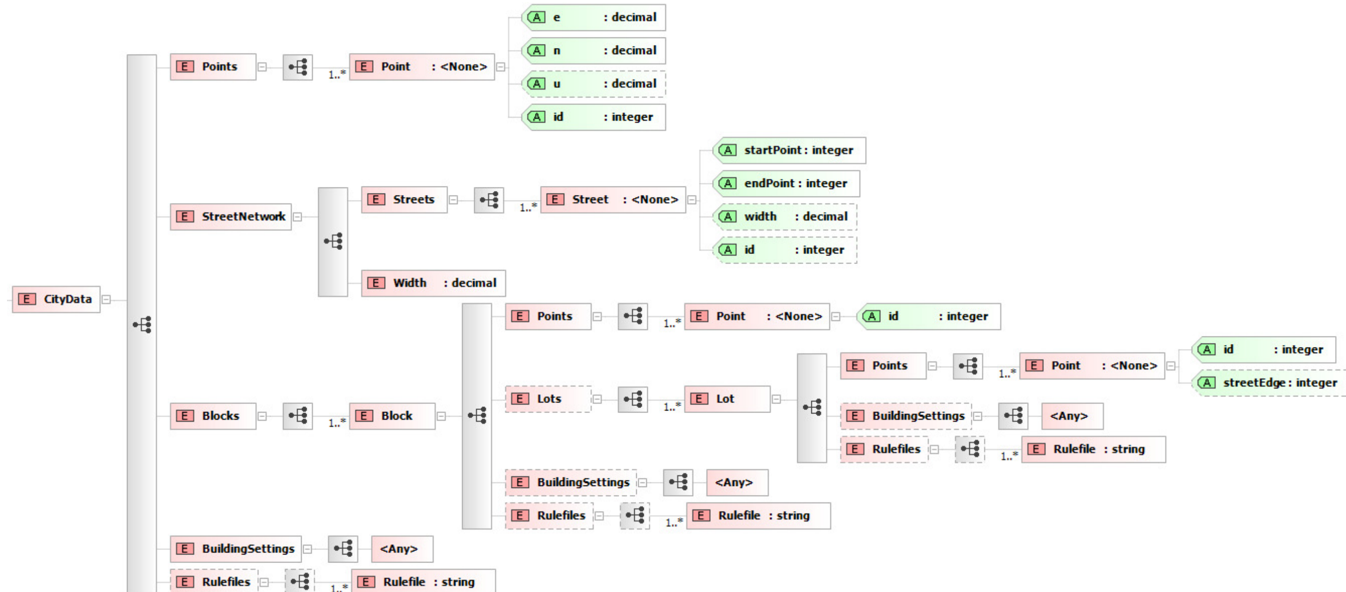


Abbildung 2.3: XML-Schemadefinition der Szenendatei

Eine Szenendatei beginnt mit dem *CityData*-Wurzelelement, das über das Attribut *xsi:noNamespaceSchemaLocation* auf die XML-Schema-Datei *CityData.xsd* verweist (siehe auch Abbildung 2.3). Als erstes Unterelement folgt das *Points*-Element, das die Punktmenge als Unterelemente beinhaltet. Ein Unterelement *Point* repräsentiert einen Punkt im Raum durch einen eindeutigen Identifizierer *id* und drei Koordinaten: e, n, u. Die Koordinaten werden als ENU-Koordinaten[1] und in Metern interpretiert. Als nächstes Unterelement des *CityData*-Elements folgt die Straßendefinition im Element *StreetNetwork*. Dieses hat ein Unterelement *Streets*, das die einzelnen Straßen beinhaltet und ein Element *Width*, das die Straßenbreite definiert. Eine Straße wird durch ein *Street*-Unterelement von *Streets* festgelegt. Ein *Street*-Element besitzt eine eindeutige Identifikationsnummer *id* sowie Referenzen auf Punkte-IDs als Start- und Endpunkte (*startPoint* und *endPoint*) als Attribute. Optional kann die global festgelegte Straßenbreite mit dem Attribut *width* in einer Straße überschrieben werden. Die Gebäudeblöcke folgen als nächstes Unterelement von *CityData* im *Blocks*-Element. Hierin werden einzelne Gebäudeblöcke in Form von *Block*-Elementen aufgezählt. Ein Block referenziert Punkte über ihre ID-Variablen als *Point*-Elemente mit einem *id*-Attribut in einem *Points*-Unterelement. Zu einem Block können optional Gebäudegrundrisse in Form von *Lot*-Elementen innerhalb eines *Lots*-Elements angegeben werden. Ebenfalls optional ist das Vorkommen von einem Variablen-Block *BuildingSettings*, in dem Variablen für diesen Gebäudeblock definiert werden können. Ein solcher Variablen-Block beinhaltet XML-Elemente, deren Name für den Variablennamen und deren innerer Text für den Variablenwert steht. Als Werte können Gleitkommazahlen oder Zeichenketten benutzt werden. Ein Grundriss beinhaltet analog zum Block referenzierte Punkte und einen optionalen Variablen-Block. Abschließend wird ein Variablen-Block als letztes Unterelement von *CityData* verwendet. Hierin werden Variablen deklariert, die dann global für die ganze Szene gelten, aber innerhalb von Variablen-Blöcken, die in Gebäudeblöcken oder Gebäudegrundrissen enthalten sein können,

[1] Local east, north, up (ENU) coordinates, siehe `<http://en.wikipedia.org/wiki/Geodetic_system>`

überschrieben werden können. Ein exemplarisches Beispiel einer Szenendatei ist in Listing 2.1 aufgeführt.

Listing 2.1: Szenendatei-Beispiel

```xml
<?xml version="1.0" encoding="utf-8" ?>
<CityData xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="CityData.xsd">

    <!-- Alle Punkte der Szene -->
    <Points>
        <!-- Ein Punkt -->
        <Point id="0" e="0"  n="0" />
        <Point id="1" e="30" n="30" />
        <Point id="2" e="50" n="0" />
        <Point id="3" e="20" n="-30" />
        ...
    </Points>

    <StreetNetwork>
        <!-- Alle Straßen der Szene -->
        <Streets>
            <!-- Eine Straße definiert durch Start- und Endpunkt in beliebiger Reihenfolge -->
            <Street id="0" startPoint="0" endPoint="1" />
            <Street id="1" startPoint="1" endPoint="2" width="8" />
            <Street id="2" startPoint="2" endPoint="3" />
            <Street id="3" startPoint="3" endPoint="0" />
        </Streets>
        <Width>5</Width> <!-- Globale Straßenbreite -->
    </StreetNetwork>

    <!-- Alle Gebäudeblöcke der Szene -->
    <Blocks>
        <!-- Ein Gebäudeblock, zu dem die Grundrisse generiert werden -->
        <Block>
            <!-- wird durch referenzierte Punkte definiert, die ein Polygon in dieser Reihenfolge bilden -->
            <Points>
                <Point id="3" />
                <Point id="2" />
                <Point id="1" />
                <Point id="0" />
            </Points>
            <!-- Variablen für diesen Block -->
            <BuildingSettings>
                <LODLevel>1</LODLevel>
            </BuildingSettings>
        </Block>

        <!-- Ein Gebäudeblock mit vorgegebenen Grundrissen -->
        <Block>
            <!-- wird durch referenzierte Punkte definiert, die ein Polygon in dieser Reihenfolge bilden -->
            <Points>
                <Point id="7" />
                <Point id="6" />
                <Point id="5" />
                <Point id="4" />
            </Points>
            <!-- Ein Gebäudegrundriss -->
            <Lots>
              <Lot>
                <!-- wird durch referenzierte Punkte definiert -->
                <Points>
                  <Point id="9"/>
                  <Point id="8"/>
                  <!-- Optional: Kante von Punkt 5 nach Punkt 4 liegt an einer Straße -->
                  <Point id="5" streetEdge="1"/>
                  <!-- Optional: Kante von Punkt 4 nach Punkt 9 liegt an einer Straße -->
                  <Point id="4" streetEdge="1"/>
                </Points>
                <!-- Variablen für diesen Grundriss -->
                <BuildingSettings>
                  <BuildingHeight>16</BuildingHeight>
                </BuildingSettings>
              </Lot>
            </Lots>
        </Block>
    </Blocks>

    <!-- Global gültige Variablen -->
    <BuildingSettings>
        <BuildingHeight>12</BuildingHeight>
        <BuildingHeightVar>2</BuildingHeightVar>
        ...
    </BuildingSettings>
    <!-- Globale Regeldatei(en) -->
    <Rulefiles>
      <Rulefile>SimpleBuilding.xml</Rulefile>
    </Rulefiles>

</CityData>
```

2.1.3 Implementierung der CityData-Klasse

Wie im UML-Diagramm in Abbildung 2.4 zu sehen, leiten sich die zur Szenenbeschreibung benötigten Klassen *Lot*, *Block* und *CityData* von der *CitySceneEntityBase*-Basisklasse ab. Diese speichert Verweise auf *Point*-Objekte und verfügt über ein *Settings*-Objekt sowie eine Liste von Regeldateinamen. Die *Point*-Klasse speichert die ID-Nummer in einer Integer-Variablen sowie die ENU-Koordinaten in einem OpenSG-Vektor. Die *Settings*-Klasse speichert Gleitkommavariablen, und Zeichenkettenvariablen jeweils in einer STL-Map. Variablen können über die Methoden *ReadFromXML()* und *WriteToXML()* aus einer XML-Datei gelesen und wieder gespeichert werden.

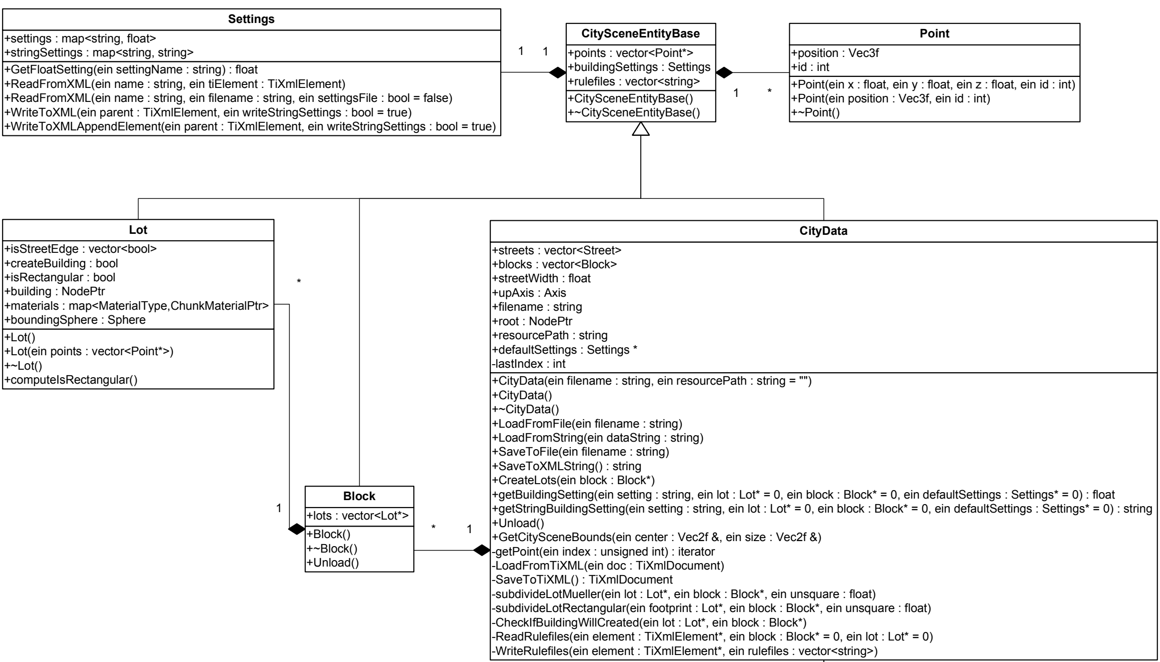


Abbildung 2.4: UML-Diagramm der CityData-Klasse und zugehörige Klassen

Die einen Gebäudegrundriss repräsentierende *Lot*-Klasse verfügt über (von der Basisklasse vererbte) Verweise auf *Point*-Objekte (wobei die Reihenfolge die Punkte eines Polygons bestimmt), über einen Bool-Vektor, der speichert, welche Kanten des Grundrisses an Straßen liegen (dies wird entweder aus der XML-Szenendatei eingelesen, falls angegeben, oder aufgrund der Lage des Grundrisses zu Straßen später berechnet), über ein vererbtes *Settings*-Objekt, eine Bool-Variable, ob der Grundriss rechteckig ist, über einen Zeiger auf einen OpenSG-Knoten, über den später das generierte Gebäudemodell zugänglich gemacht wird und über eine *BoundingSphere*, die die räumliche Ausdehnung des Grundrisses beschreibt. Die *Block*-Klasse, die einen Gebäudeblock repräsentiert, verfügt über Datenfelder für Verweise auf *Lot*-Objekte. Außerdem enthält sie ein vererbtes *Settings*-Objekt und vererbte Verweise auf *Point*-Objekte.

Die CityData-Klasse stellt eine Stadtszene dar. Sie setzt sich aus *Block*-Objekten und deren zugehörigen *Lot*-Objekten zusammen. Eine Szenendatei wird mithilfe der *CityData*-Klasse über die Methoden *LoadFromFile()* oder *LoadFromString()* eingelesen und in deren interne Datenstrukturen überführt. Die Umwandlung von XML in C++-Klassen und zurück wird mithilfe der frei verfügbaren *TinyXML*-Bibliothek[TinyXML 08] realisiert. Verfügt ein Gebäudeblock über keine Grundrisse, so können mit der Methode *CreateLots()* diese algorith-

misch erzeugt werden. Dabei wurden 2 Algorithmen implementiert, die einen Gebäudeblock als Polygon betrachten und dieses in kleinere Polygone zerlegen (diese werden weiter unten beschrieben). Da die Klasse ebenfalls von *CitySceneEntityBase* abgeleitet ist, verfügt sie ebenso über ein *Settings*-Objekt. Die Methoden *getBuildingSetting()* und *getStringBuildingSetting()* geben den Wert einer Gleitkomma- oder einer Zeichenkettenvariablen zurück. Man übergibt der Methode den Variablennamen und optional Verweise auf ein *Lot*-, ein *Block*- und ein *Settings*-Objekt. Dieses *Settings*-Objekt verweist auf Variablen, die in der Datei *DefaultSettings.xml* definiert wurden und die im *Resources/*-Ordner liegt. Ist eine Variable mehrfach in den verschiedenen Objekten deklariert, so wird folgende Vorrangsreihenfolge benutzt: *Lot*, *Block*, *CityData*, *Settings*. Somit kann man Variablen, die global in einem *CityData*-Objekt definiert wurden, lokal in einem *Lot*-Objekt überschreiben.

Die *Street*-Klasse (siehe UML-Diagramm in Abbildung 2.5) repräsentiert eine Straße und speichert zwei Verweise auf *Point*-Objekte, die Straßenbreite in einer Float-Variablen sowie ebenfalls eine ID-Nummer. Die Erzeugung von Straßengeometrien ist bisher nicht implementiert. Die Klasse *StreetGenerator* erzeugt in der Methode *CreateStreetNetworkSceneGraph()* aus *Street*-Objekten lediglich einfarbige Rechtecke und gibt diese als OpenSG-*NodePtr* zurück.

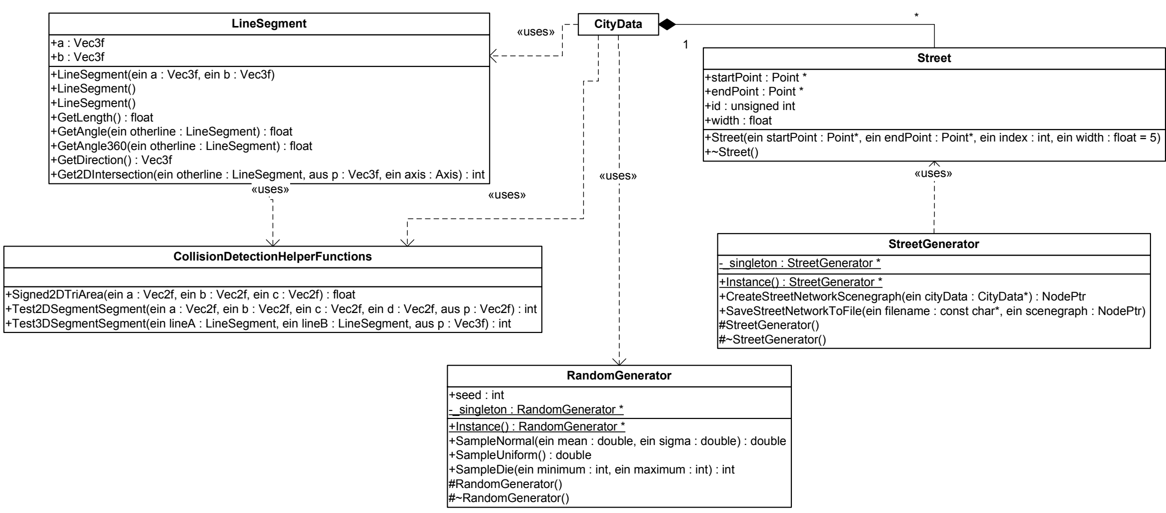


Abbildung 2.5: UML-Diagramm der zur CityData-Klasse gehörigen Hilsfklassen

Blockunterteilungs-Algorithmus nach Müller Dieser Algorithmus wurde aus [Müller 01] übernommen und in eigener Form implementiert (in der Methode *subdivideLotMueller()*). Er wird rekursiv auf einem Gebäudeblock ausgeführt. Folgende Schritte werden dabei durchgeführt:

1. Nur wenn der Block eine Kante zu einer Straße hin besitzt, wird weiter unterteilt. Somit werden Grundrisse, die im Inneren eines Gebäudeblocks (sozusagen im „Innenhof") liegen nicht weiter unterteilt.

2. Die Fläche des Blockpolygons wird berechnet. Ist die Fläche größer als die in einem dem Gebäudeblock zugehörige oder global definierte Variable *MaxLotArea*, so wird dieses Polygon weiter unterteilt, ansonsten ist die Unterteilung an dieser Stelle beendet.

3. Die beiden längsten Kantenpolygone werden ausgewählt. Die Kante mit mehr recht-

winkligen Winkeln zu den Nachbarkanten oder die längere (als 2. Wahlkriterium) wird ausgewählt, um in deren ungefährer Mitte (gaussverteilt) eine rechtwinklige Trenngerade zu legen, mit der das Polygon in 2 Polygone aufgeteilt wird. Diese beiden kleineren Polygone werden erneut rekursiv unterteilt.

Die erzeugten Gebäudegrundrisse müssen 3 Bedingungen erfüllen, um als bebaubar markiert zu werden:

- Die Polygonfläche muss größer sein als die *MinLotArea*-Variable.
- Die Winkel zwischen den Polygonkanten müssen alle größer als der in der *MinLotAngle*-Variablen festgelegte Wert sein.
- Die Variable *HousingDensity* gibt an, mit welcher Wahrscheinlichkeit ein geeigneter Grundriss auch als bebaubar markiert wird. Der Wertebereich für die Variable ist $[0, 1]$.

Dieser Algorithmus erzeugt jedoch bei nicht-rechtwinkligen Gebäudeblöcken auch Grundrisse, die nicht rechtwinklig sind. Je weniger rechtwinklige Ecken das Blockpolygon besitzt, desto weniger kantenparallele Grundrisse werden erzeugt (siehe auch Abbildung 2.6). Dies entspricht eher weniger der in Städten anzutreffenden Bebauung von Gebäudeblöcken, da diese meist parallel zur Straße gebaut werden.

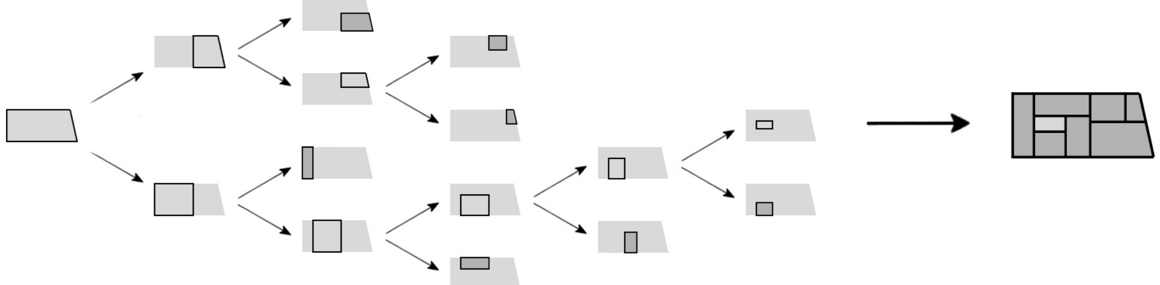


Abbildung 2.6: Blockunterteilungsalgorithmus nach Müller aus [Müller 01]

Blockunterteilungs-Algorithmus zur straßenparallelen Unterteilung Dieser Algorithmus (implementiert in der Methode *subdivideLotRectangular()*) erzeugt Gebäudegrundrisse, die parallel zu den Kanten des Gebäudeblocks sind. Dies wird *Blockrandbebauung* genannt. Dazu wird das Polygon nach innen verkleinert. Der Zwischenraum zwischen dem Blockpolygon und dessen verkleinerter Variante wird dann in Gebäudegrundrisse unterteilt. Folgende Vorbedingungen sind zu erfüllen, ansonsten wird nicht unterteilt:

1. Liegt das Blockpolygon nicht in einer horizontalen Ebene, so wird das Polygon nicht weiter unterteilt, d.h. alle u-Koordinaten der Punkte des Polygons müssen den gleichen Wert haben.
2. Es wird geprüft, ob die Punkte des Polygons gegen den Uhrzeigersinn angeordnet sind. Ist dies nicht der Fall, so wird eine Warnung ausgegeben und die Punktreihenfolge umgekehrt.
3. Es wird geprüft, ob das Polygon mindestens eine Kante an einer Straße hat. Ansonsten wird nicht weiter unterteilt.

4. Das Polygon wird um den in der Variablen *MinBuildingDepth* gespeicherten Wert verkleinert. Ist dies nicht möglich, z.B. wenn sich dadurch Seiten des verkleinerten Polygons überlappen, so wird nicht weiter unterteilt.

Nun folgt der Unterteilungsalgorithmus:

1. Ausgehend vom ersten Punkt des Blockpolygons wählt man einen neuen Zwischenpunkt auf der Kante zum nächsten Punkt des Polygons, der einen Abstand zum ersten Punkt hat, der sich normalverteilt im Bereich der festgelegten Variablen [*MinBuildingLength*, *MaxBuildingLength*] befindet.

2. Wenn die Strecke[2] zwischen dem neuen Zwischenpunkt und dem nächsten Polygonpunkt größer ist als *MinBuildingLength*, dann fahre beim nächsten Schritt fort, ansonsten gehe zu Schritt 8.

3. Erzeuge eine Gerade beginnend beim Zwischenpunkt und senkrecht zu ihm, die in Richtung des verkleinerten Polygons läuft. Falls diese Gerade die entsprechende Kante des verkleinerten Polygons schneidet, gehe zum nächsten Schritt, ansonsten gehe zu Schritt 8.

4. Sind die Strecken zwischen dem Schnittpunkt und dem nächsten Punkt des verkleinerten Polygons und zwischen dem Schnittpunkt und dem vorhergehenden Punkt des verkleinerten Polygons größer als die *MinBuildingInnerLength*-Variable, dann fahre beim nächsten Schritt fort, ansonsten gehe zu Schritt 9.

5. Es wird ein neuer Gebäudegrundriss aus eventuell bereits zuvor gespeicherten Punkten, dem aktuellen Polygonpunkt, seinem zugehörigen Punkt des verkleinerten Polygons sowie dem neu erzeugten Zwischenpunkt und dem Schnittpunkt erstellt. Gehe zu Schritt 9.

6. Wenn bereits die letzte Kante des Polygons den Algorithmus durchlaufen hat, so wird der erzeugte Gebäudegrundriss mit dem zuerst erzeugten Gebäudegrundriss verbunden, so dass auch an dem ersten Punkt des Polygons ein Eckgebäudegrundriss entsteht. Ansonsten gehe zur nächsten Kante des Polygons und speichere den Zwischenpunkt und den Schnittpunkt als Punkte des nächsten Grundrisses zwischen.

7. Beginne den Algorithmus von vorne.

8. Der Zwischenpunkt wird als neuer Ausgangspunkt gesetzt. Beginne den Algorithmus von vorn.

9. Der Zwischenpunkt und der Schnittpunkt bilden die neuen Ausgangspunkte. Beginne den Algorithmus von vorne.

Abschließend werden ebenso die 3 Bedingungen (siehe Seite 19) geprüft. Die Zwischenergebnisse des Algorithmus bei der Unterteilung eines Gebäudeblockes sind in Abbildung 2.7 dargestellt.

[2] Eine Strecke wird durch die Hilfsklasse *LineSegment* repräsentiert. Diese Verfügt über Methoden, um den Winkel zwischen zwei Strecken zu errechnen (*GetAngle()*) und zur Berechnung von Schnittpunkten zweier Strecken (*Get2DIntersection*). Hierfür wird außerdem die Hilfsklasse *CollisionDetectionHelperClasses* verwendet. Die Schnittpunktberechnung wurde aus [Ericson 05] übernommen.

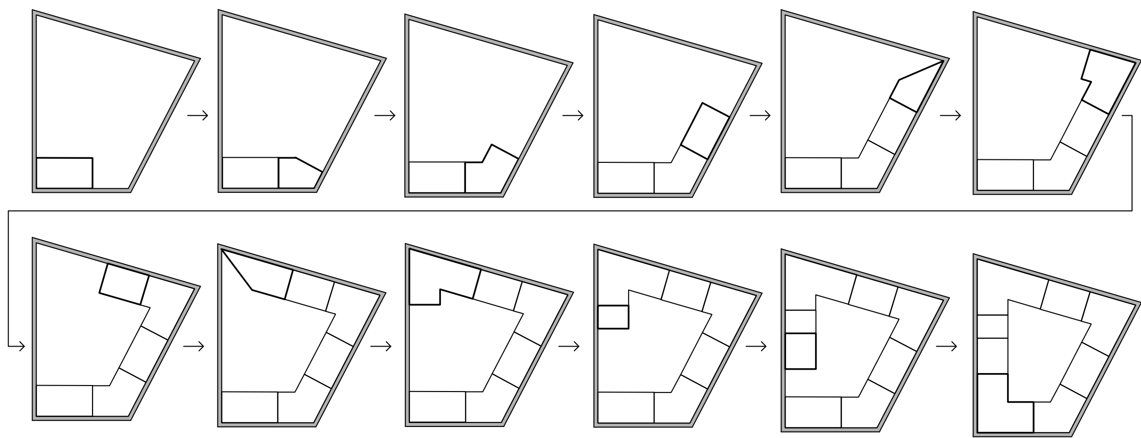


Abbildung 2.7: Ergebnis der straßenparallelen Gebäudeblock-Zerlegung

2.2 Regelsystem - RuleProcessor-Klasse

In diesem Kapitel wird das Regelsystem zur Erzeugung von Fassadenstrukturen vorgestellt. Das Regelsystem basiert auf einer Grammatik, die an der von Müller et. al. in [Müller 06b] Vorgestellten angelehnt ist. Die Unterschiede werden weiter unten erläutert.

2.2.1 Formale Grammatiken

Es ist eine *Algebra* (U, f_i) gegeben. Eine Grammatik $G = (N, T, P, S)$ besteht dann aus *Nichtterminalsymbolen* $N \subseteq U$, *Terminalsymbolen* $T \subseteq U$ (wobei $N \cap T = \emptyset$ gilt: es gibt somit kein Symbol, dass sowohl Terminal- als auch Nichtterminalsymbol ist), *Startsymbolen* $S \subseteq N$ und *Produktionsregeln* $P \subseteq U \times U$. Die Menge aller terminalen und nichtterminalen Symbole wird *Vokabular* $V = N \cup N$ einer Grammatik G genannt. Somit beschreiben formale Grammatiken *formale Sprachen*.

Nichtterminalsymbole sind Symbole, die zur Erzeugung der durch die Grammatik beschriebenen formalen Sprache verwendet werden, dabei jedoch nicht in den Wörtern der Sprache vorkommen. Die Wörter der formalen Sprache werden aus Terminalsymbolen zusammengesetzt. Eine Produktionsregel besteht aus der Prämisse R und der Konklusion Q und wird oft als $R \rightarrow Q$ notiert. R muss mindestens ein Nichtterminalsymbol enthalten ($R \in (N \cup N)^* N (N \cup T)^*$) während Q beliebig aus Terminal- und Nichtterminalsymbolen bestehen kann ($Q \in (N \cup T)^*$). Ein Startsymbol ist ein Nichtterminalsymbol von dem ausgehend die Wörter der durch die Grammatik definierten Sprache erzeugt werden. Die *Chomsky-Hierarchie* unterteilt die Grammatiken in Kategorien mit bestimmten Eigenschaften.

2.2.2 Regelsystem-Grammatik

Die für das Regelsystem verwendete Grammatik ist eine kontextfreie Grammatik, d.h. Symbole (in diesem Fall String-Objekte aus U) werden mithilfe einer Ersetzungsregel durch andere Symbole ersetzt, wobei auf der linken Seite immer ein Nichtterminalsymbol steht. Es handelt sich außerdem um eine sequentielle Grammatik, d.h. es wird ein Symbol nach dem anderen ersetzt und nicht wie bei parallelen Grammatiken, wie z.B. bei L-Systemen, immer alle Symbole in einem Schritt.

Als Symbole kommen so genannte *Shapes* zum Einsatz. Ein *Shape* besteht aus einem Namen, der als Symbol in der Grammatik fungiert. Als Namen können beliebige Zeichenketten eingesetzt werden. Die Abgrenzung der Terminalsymbole geschieht durch Anhängen der Dateiendung *.wrl* (somit stehen diese dann für eine 3D-Geometrie, siehe Kapitel 2.4.4). Die räumliche Anordnung der Shapes wird als Rechteck gespeichert und als *Scope* bezeichnet. Shapes haben eine Reihe weiterer Eigenschaften, die später zur Geometrieerzeugung genutzt werden. So wird die Transformation des Symbols relativ zum Startsymbol gespeichert und optional Texturnamen, UV-Koordinaten-Skalierungen sowie Größenskalierungen und Verschiebungsvektoren definiert. Diese Eigenschaften werden beim Ersetzungsprozess „vererbt“. Des Weiteren werden Symbole als aktiv oder inaktiv gekennzeichnet. Inaktive Symbole spielen im Ersetzungsprozess keine weitere Rolle mehr.

Dabei stehen Shapes für konkrete Symbole in einem Wort der Grammatik, während die nachfolgend mit Vorgänger- und als Nachfolgersymbole bezeichneten Begriffe für abstrakte Sym-

boleigenschaften innerhalb der Regeln stehen, die bei Anwendung der Ableitungsregeln auf konkrete Shapes übertragen werden.

Eine Ersetzungsregel ist definiert durch ein Vorgängersymbol (das zu ersetzende Nichtterminalsymbol). Es darf somit nur eine Regel mit einem entsprechenden Vorgängersymbol geben. Eine Regel besteht aus einer oder mehreren Ableitungsregeln. Eine der möglichen Ableitungsregeln wird mit einer bestimmten Wahrscheinlichkeit zufällig ausgewählt. Eine Ableitungsregel besteht aus einem Regeltyp und einem oder mehreren Nachfolgersymbolen, mit denen das Vorgängersymbol ersetzt wird. Das Vorgängersymbol wird jedoch nicht komplett entfernt sondern nur als inaktiv markiert.

Der Regeltyp *Substitution* ersetzt das Vorgängersymbol einfach durch ein oder mehrere Nachfolgersymbole ohne dabei die räumliche Anordnung zu ändern.

Der Regeltyp *Split* teilt das aktuelle Rechteck entlang einer Achse. Die Achse r (für *right*) steht für die horizontale, von links nach rechts verlaufende Achse. Die Achse u (für *up*) steht für die vertikale, von unten nach oben verlaufende Achse. Jedes Nachfolgersymbol definiert dabei eine Größe in horizontaler oder vertikaler Richtung, je nachdem, welche Achse gewählt wurde. Die Größe kann absolut in Metern oder relativ angegeben werden. Bei absoluten Größenangaben ist darauf zu achten, dass die Summe aller absoluten Größen in einem Split die Größe des aktuellen Scopes auf dieser Achse nicht überschreitet (ist dies doch der Fall, so wird in der Implementierung eine Warnmeldung ausgegeben und die überlappenden Shapes weiter bearbeitet). Die neu erzeugten Shapes erhalten nun auch der Unterteilung entsprechende Scopes.

Der Regeltyp *Repeat* unterteilt das aktuelle Volumen entlang einer Achse in gleichförmigen Abständen, definiert durch ein Nachfolgersymbol und einer Größe g. Die Größe wird immer absolut angegeben. Die Anzahl der Wiederholungen wird berechnet, indem die Scopeausdehnung in der gewählten Achse durch die Größe g geteilt und das Ergebnis abgerundet wird. Die Scopeausdehnung in der gewählten Achse geteilt durch die Anzahl der Wiederholungen ergibt die eigentliche Wiederholungsgröße. Die neu erzeugten Shapes erhalten nun auch der Unterteilung entsprechende Scopes. Für eine schematische Darstellung der Regeltypen siehe Abbildung 2.8.

Die Unterschiede zur Grammatik von Müller et al. bestehen in einigen Vereinfachungen. Zusätzlich zu den oben genannten Eigenschaften haben die Volumen bei Müller eine Ausrichtung (einen Richtungsvektor), während diese im vorgestellten Regelsystem immer rechtwinklig zum Startsymbol ausgerichtet sind. Durch den Richtungsvektor ist es möglich, Shapes zusätzlich zur Verschiebung und Skalierung auch zu rotieren. Müller definiert Prioritäten für die Ableitungsregeln. Somit kann Einfluss auf die Reihenfolge der Anwendung der Ableitungsregeln genommen werden. Dies würde es beispielsweise ermöglichen, eine Ableitungstiefe festzulegen, die bestimmt, wann die Ableitung gestoppt wird. Müller nutzt dies für ein Level-of-Detail-System, indem Regeln mit hoher Priorität die grundlegende Form eines Gebäudes erzeugen (niedrigste Detailstufe) und weitere Prioritätsstufen immer mehr Details hinzufügen (bis zur höchsten Detailstufe). Regeln können bei Müller außerdem verschachtelt werden, was die Benutzung erleichtert und die Regeldefinitionen verkürzt. Shapes können bei Müller durch eine *Snapping*-Funktionalität gleichmäßig angeordnet werden. Beim vorgestellten System müssen die Regeln und die Shape-Größen vorausschauend erstellt werden, um Shapes gleichmäßig auszurichten. Müller implementiert ein CSG[3]-artiges *Mass Modeling*-System, das

[3]CSG - Constructive Solid Geometry, dt.: Festkörpergeometrie

aus einfachen Grundkörpern, wie z.B. Quadern, durch boolsche Operationen komplexere Körper zusammensetzt, um so die Gebäudegrundgeometrie vielfältig erzeugen zu können. Durch die Grammatik-Operation *Component Split* erzeugt er aus diesen Geometrien die Fassaden-Start-Shapes. Beim vorgestellten Grammatik-System können die Start-Shapes nicht mit der Grammatik selbst erstellt werden, sondern müssen vorgegeben werden.

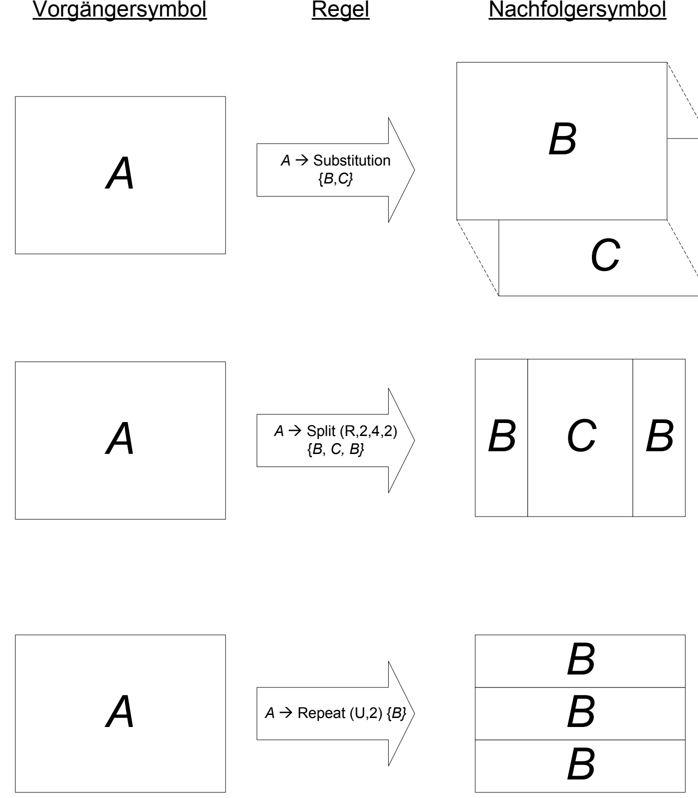


Abbildung 2.8: Schematische Darstellung der Regeltypen

Nachfolgersymbole können in einer Ableitungsregel weitere Eigenschaften haben. Der Größe des Symbols kann eine Variable zugewiesen werden. Dadurch wird die Größe mit dem Wert der Variablen multipliziert. Die Größe muss dabei absolut angegeben werden. Es kann ein Skalierungs- sowie ein Translationsvektor zugewiesen werden. Diese transformieren den Scope. Es kann außerdem ein Texturname sowie ein zweidimensionaler Texturkoordinaten-Skalierungsvektor angegeben werden.

2.2.3 XML-Format zur Regeldefinition

Nachfolgend wird ein XML-Datenformat zur Speicherung und Verarbeitung von Ersetzungsregeln vorgestellt.

Eine Regeldatei beginnt mit dem *Rulefile*-Wurzelelement. Dieses hat ein Namensattribut, um mehrere Regeldateien später unterscheiden zu können und verweist außerdem über ein Attribut auf die XML-Schema-Datei *Rules.xsd* (siehe auch 2.9). Eine Ersetzungsregel ist durch ein *Rule*-Element definiert. Dieses enthält ein Element für das Vorgängersymbol (*Predecessor*-Element) sowie ein *Rules*-Element in dem die einzelnen Ableitungsregeln enthalten sind.

Eine Ableitungsregel befindet sich in einem *Probability*-Element, dessen Attribut *value* eine Wahrscheinlichkeit definiert, mit der diese Ableitungsregel ausgewählt wird. Hierbei ist zu beachten, dass die Summe aller Wahrscheinlichkeiten innerhalb einer Ersetzungsregel 1 ergibt. Als erstes Kindelement folgt ggf. ein *Split*- oder ein *Repeat*-Element. Anderenfalls handelt es sich um ein Substitutionsregel. *Split*- und *Repeat*-Elemente haben ein *Axis*-Attribut mit den möglichen Werten *r* für *right* oder *u* für *up*. Ein *Repeat*-Element verfügt des weiteren über ein *size*-Attribut. Anschließend folgen ein oder mehrere *Symbol*-Elemente als weitere Kinder. Optional kann eine Regeldatei ein *BuildingSettings*-Element beinhalten, das in den Regeln genutzte Variablen definiert.

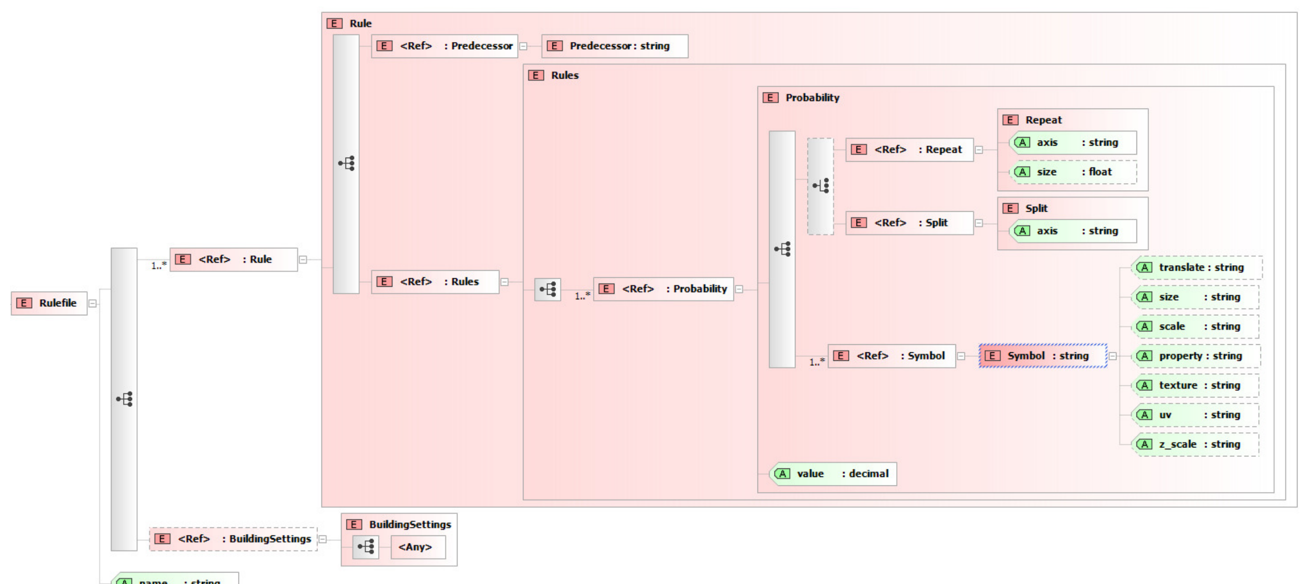


Abbildung 2.9: XML-Schemadefinition der Regeldatei

Nachfolgend ein exemplarisches Beispiel einer Regeldatei mit drei Regeln:

Listing 2.2: Regeldatei-Beispiel

```
<?xml version="1.0" encoding="utf-8"?>
<Rulefile name="SimpleBuildingVar" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:noNamespaceSchemaLocation="Rules.xsd">

  <!-- Eine Ersetzungsregel -->
  <Rule>
    <!-- Das Vorgängersymbol -->
    <Predecessor>FassadeStart</Predecessor>
    <Rules>
      <!-- Eine Substitution-Ableitungsregel -->
      <Probability  value="0.5">
        <!-- Das Nachfolgersymbol -->
        <Symbol>Fassade1</Symbol>
      </Probability>
      <!-- Eine weitere Substitutions-Ableitungsregel -->
      <Probability  value="0.5">
        <Symbol>Fassade2</Symbol>
      </Probability>
    </Rules>
  </Rule>

  <!-- Eine weitere Ersetzungsregel -->
  <Rule>
    <!-- Das Vorgängersymbol -->
    <Predecessor>Fassade1</Predecessor>
    <Rules>
      <!-- Eine Split-Ableitungsregel -->
      <Probability value="1.0">
        <Split axis="u"/>
        <Symbol size="4" texture="Bodenetage.jpg">
          Bodenetage.wrl
        </Symbol>
        <Symbol size="1r">WeitereEtagen</Symbol>
      </Probability>
    </Rules>
  </Rule>

  <!-- Eine weitere Ersetzungsregel -->
  <Rule>
    <!-- Das Vorgängersymbol -->
    <Predecessor>Fassade2</Predecessor>
    <Rules>
      <!-- Eine Repeat-Ableitungsregel -->
      <Probability value="1.0">
        <Repeat axis="r" size="1"/>
        <Symbol property="FassadenelementBreite">
          Fassadenelement
        </Symbol>
      </Probability>
    </Rules>
  </Rule>

</Rulefile
```

2.2.4 Implementierung der RuleProcessor-Klasse

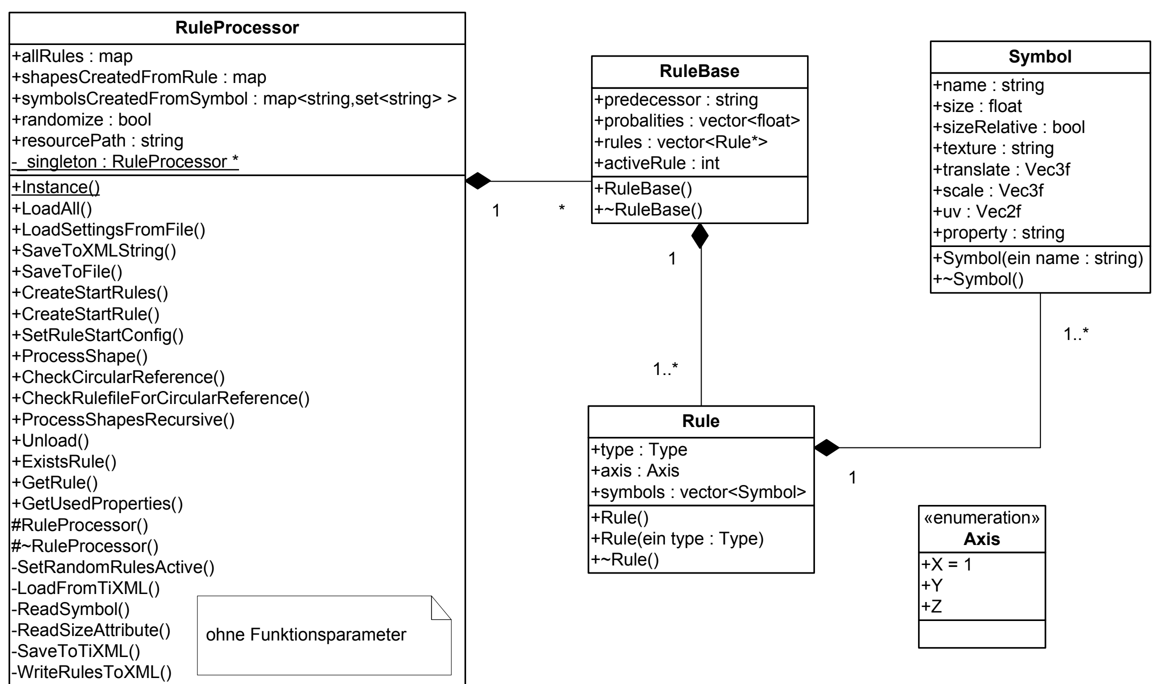


Abbildung 2.10: UML-Diagramm der Ruleprocessor-Klasse und zugehörigen Klassen

Die Struktur des *RuleProcessors* und der ihm zugehörigen Klassen sind im UML-Diagramm in Abbildung 2.10 zu sehen.

Die *RuleBase*-Klasse repräsentiert eine Ersetzungsregel. Sie speichert das Vorgängersymbol in einem String (`string predecessor`), die Ableitungsregeln (`vector<Rule*> rules`) und die zugehörigen Wahrscheinlichkeiten (`vector<float> probalities`) in STL-Vektoren.

Die *Rule*-Klasse stellt eine konkrete Ableitungsregel dar. Sie hat einen Typ (`Type type` mit den möglichen Werten `Substitution, Split, Repeat`), speichert eventuell eine Achse (`Axis axis`) und ein oder mehrere Symbole in einem Vektor (`vector<Symbol> symbols`).

Die *Symbol*-Klasse definiert ein Symbol als ein Nachfolgersymbol in einer Ableitungsregel. Sie wird durch Datenfelder für den Namen des Symbols (`string name`), die Größe (`float size`), ob die Größe relativ interpretiert werden soll (`bool sizeRelative`), den Namen einer Variablen (`string property`), mit der die Größe später multipliziert wird, einen Texturnamen (`string texture`), Vektoren für die Skalierung und Translation (`Vec3f translate, scale`) und für die Skalierung der Texturkoordinaten(`Vec2f uv`) definiert. Außerdem speichert die Klasse eine Variable vom Typ `ZScaleType`, die bestimmt, wie die 3D-Geometrie später auf der Z-Achse skaliert wird.

Die *Shape*-Klasse definiert ein Shape, also ein Symbol der Grammatik. Shapes können Nichtterminalsymbole sein, die durch Ableitung in Terminalsymbole überführt werden. Die Shape-Klasse hat ähnliche Datenfelder wie die Symbol-Klasse. Zusätzlich wird der Scope in einem OSG-*BoxVolume* (`BoxVolume scope`) und die Transformation relativ zum Start-Shape in einem OpenSG-*TransformPtr* (`TransformPtr transform`) gespeichert. Ein Verweis auf den Eltern-Shape wird in `Shape* parent` und Verweise auf die Kind-Shapes werden in `vector<Shape*>`

`children` gespeichert.

Die *RuleProcessor*-Klasse ist das Herzstück des Regelsystems. Der RuleProcessor implementiert Methoden zum Einlesen und Abspeichern von Regeln in Form einer XML-Regeldatei. Der RuleProcessor speichert alle Regeln in der STL-Map `map<string, vector<RuleBase*>> allRules`. Hierbei wird über einen Namensstring ein Satz an Regeln referenziert, d.h. der RuleProcessor kann über mehrere *Regel-Sets* verfügen. Somit ist es später möglich, für jeden Gebäudetyp ein Regel-Set aus einer Regeldatei zu laden. Der Methode *ProcessShapesRecursive()* wird ein Shape sowie ein Regel-Set übergeben und diese ruft dann rekursiv solange die Methode *ProcessShape()* auf, bis das Start-Shape inklusive aller von ihm abgeleiteten Shapes durch Regeln abgeleitet und ggf. durch Terminalsymbole ersetzt wurden.

Die *ProcessShape()*-Methode leitet nun ein Shape ab. Zu Beginn wird eine passende Ableitungsregel gesucht, also diejenige, bei der der Vorgängersymbolname dem Shape-Namen entspricht. Wurde ein Regel-Set angegeben, so wird auch nur in diesem gesucht. Ansonsten wird in allen Regel-Sets gesucht und die erste passende Ableitungsregel verwendet. Anschließend wird eine der möglichen Ersetzungsregeln innerhalb der ausgewählten Ableitungsregel entsprechend den angegebenen Wahrscheinlichkeiten zufällig ausgewählt. Alternativ kann auch eine zuvor als aktiv markierte Ersetzungsregel direkt gewählt werden. Danach wird die ausgewählte Regel auf den Shape angewandt. Dadurch werden neue Shapes erzeugt und der bearbeitete Shape wird als inaktiv markiert.

Exemplarisch wird nun die Implementierung der Split-Regel Schritt für Schritt erläutert. Abbildung 2.11 verdeutlicht einige verwendete Variablen. Die Anzahl an neuen Symbolen wird in `n` gespeichert:

```
int n = selectedRule->symbols.size();
```

Die Scopeausdehnung und der Scope-Mittelpunkt werden in `oldSize` und `oldCenter` gespeichert.

```
Vec3f oldSize;
shape->scope.getSize(oldSize);
Pnt3f oldCenter;
shape->scope.getCenter(oldCenter);
```

Es werden Variablen für die Summe aller absoluten sowie aller relativen Größen der Symbole angelegt. Außerdem wird eine Multiplikator-Variable angelegt.

```
float sumAbsSizes = 0.0f;
float sumRelSizes = 0.0f;
float mult = 1.0f;
```

Nun wird über die Symbole in der ausgewählten Regel iteriert:

```
for(vector<Symbol>::const_iterator it = selectedRule->symbols.begin();
  it != selectedRule->symbols.end(); ++it)
{
```

Für jedes Symbol wird nun geprüft, ob der Größe eine Variable (`property`) zugewiesen wurde. In diesem Fall wird der Wert der Variablen in der Multiplikationsvariablen gespeichert.

```
  if (settings->GetFloatSetting(it->property))
    mult = settings->GetFloatSetting(it->property);
```

Nun werden die einzelnen relativen oder absoluten Größen auf die entsprechenden Variablen aufsummiert.

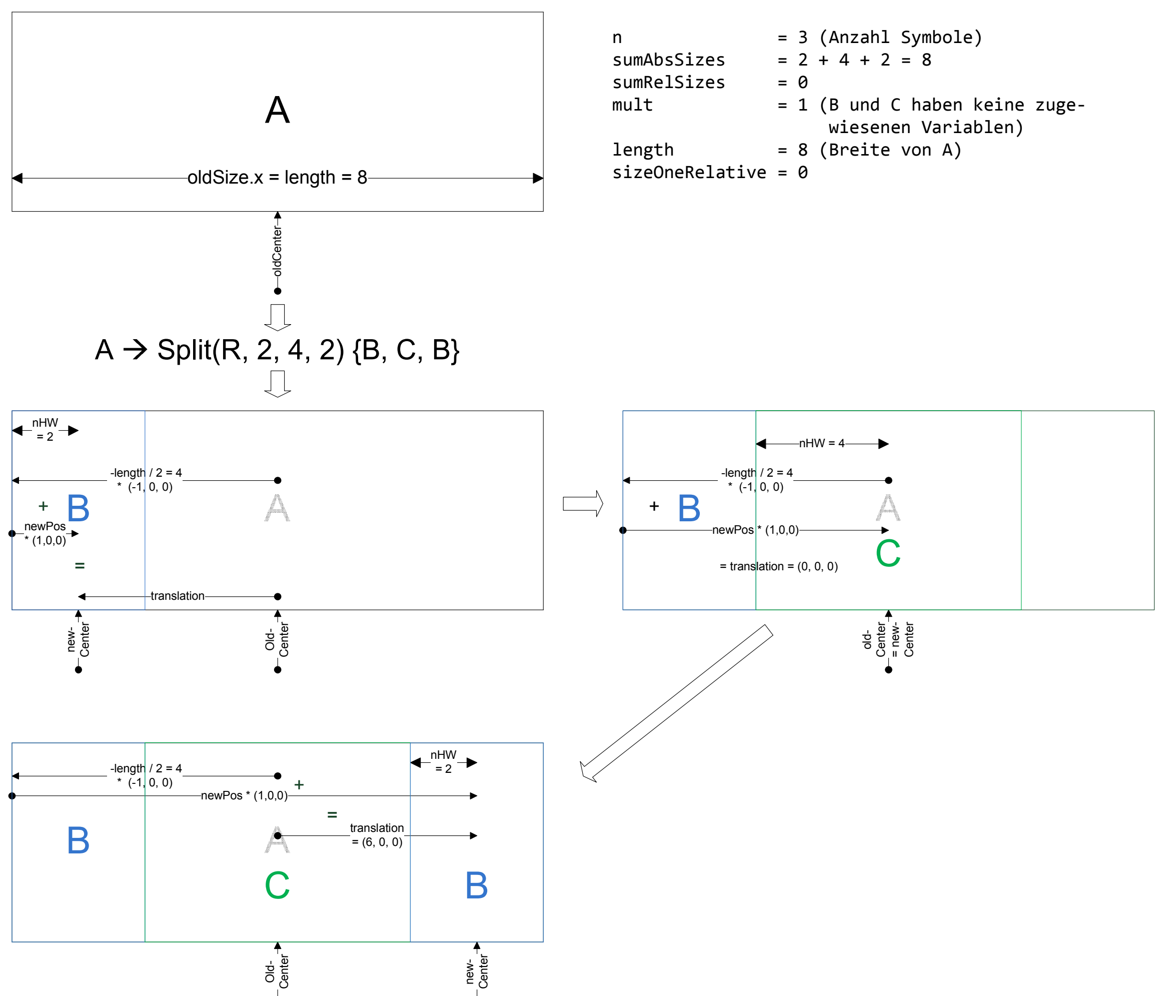


Abbildung 2.11: Exemplarischer Ablauf einer Split-Regel

```
    if(it->sizeRelative) sumRelSizes += it->size;
    else sumAbsSizes += it->size * mult;
}
```

Die Breite oder Höhe, je nach Achse, des aktuellen Shapes wird in `length` gespeichert.

```
float length;
if (selectedRule->axis == X) length = oldSize.x();
else length = oldSize.y();
```

Es wird die absolute Größe einer relativen Größeneinheit berechnet.

```
float sizeOneRelative = (length - sumAbsSizes) / sumRelSizes;
```

Nun wird der aktuelle Shape auf der gewählten Achse aufgeteilt. Dazu werden Variablen benötigt, die die neue Position des neuen Shapes relativ zum Shape-Mittelpunkt des aktuellen Shapes sowie die neue halbe Breite des Shapes speichern (`newPos` und `newWidth`). Anschließend wird über die Symbole iteriert:

```
float newPos = 0.0f;
```

```
float newHalfWidth = 0.0f;

  for (vector<Symbol>::const_iterator it = selectedRule->symbols.begin();
    it != selectedRule->symbols.end(); ++it)
  {
```

Die Multiplikationsvariable wird ggf. wieder gesetzt.

```
  if (settings->GetFloatSetting(it->property))
    mult = settings->GetFloatSetting(it->property);
```

Es wird ein neuer Shape erzeugt (`newShape`) und diesem der Name des aktuellen Symbols zugewiesen. Falls eine Textur oder ein Texturkoordinaten-Skalierungsvektor im aktuellen Symbol definiert worden sind, so werden diese dem neuen Shape zugewiesen, ansonsten werden sie vom aktuellen Shape kopiert.

```
  Shape* newShape = new Shape(it->name);

  if (it->texture.size() > 0) newShape->texture = it->texture;
  else newShape->texture = shape->texture;

  if (it->uv != Vec2f(1,1)) newShape->uv = it->uv;
  else newShape->uv = shape->uv;

  if(it->zScaleType != ZSCALE_NOT_SET) newShape->zScaleType = it->zScaleType;
  else newShape->zScaleType = shape->zScaleType;
```

Für den neuen Shape wird ein OpenSG-*Transform*-Objekt zur Speicherung der Transformation relativ zum Wurzelshape sowie ein OpenSG-*BoxVolume*-Objekt als Scope angelegt.

```
  newShape->transform = Transform::create();
  BoxVolume newScope;
```

Die Position des neuen Shapes ergibt sich aus der Position des vorhergehenden Shapes (die sich in der Mitte des Shapes befindet) plus der halben Breite des vorhergehenden Shapes. Die Position ist 0, wenn gerade der erste neue Shape bearbeitet wird.

```
  newPos += newHalfWidth;
```

Nun wird die halbe Breite des neuen Shapes berechnet. Dazu wird die Größe des aktuellen Symbols entweder mit der relativen Größeneinheit oder mit dem Größenmultiplikator mutlipliziert und anschließend durch 2 geteilt.

```
  if (it->sizeRelative) newHalfWidth = (it->size * sizeOneRelative) / 2;
  else newHalfWidth = (it->size * mult) / 2;
```

Um den neuen Mittelpunkt des Shapes zu erhalten, wird auf die neue Position noch die neue halbe Breite addiert.

```
  newPos += newWidth;
```

Anschließend werden einige Variablen zur Berechnung der Transformationen angelegt. Die Translationsmatrix `translate` repräsentiert eine Verschiebung um den Translationsvektor des aktuellen Symbols und die Skalierungsmatrix `scale` steht für eine Skalierung entsprechend des Skalierungsvektors des aktuellen Symbols. In der Matrix `old` wird die Transformationsmatrix des alten Shapes gespeichert.

```
Vec3f translation, newCenter, newSize;
Matrix m, old, translate, scale;

translate.setTranslate(it->translate);
scale.setScale(it->scale);
old = shape->transform->getMatrix();
```

Jetzt wird der Translationsvektor relativ zum vorhergehenden Shape berechnet. Nachfolgend wird der Ablauf für einen horizontalen Split erläutert. Der Translationsvektor „beginnt" am linken Ende des aktuellen Shapes (`(-length/2.0f), 0, 0)`), dazu wird die neue Position addiert (`(newPos)* Vec3f(1, 0, 0)`) und der Translationsvektor des aktuellen Symbols addiert. Der neue Mittelpunkt des Scopes berechnet sich aus dem alten Mittelpunkt plus dem eben berechneten Translationsvektor. Schließlich wird aus dem Translationsvektor die Translationsmatrix `m` erzeugt.

```
if (selectedRule->axis == X)
{
  translation =  Vec3f((-length/2.0f), 0, 0) + ((newPos) * Vec3f(1, 0, 0))
     + it->translate;
  newCenter = oldCenter + translation;
  newSize = Vec3f(newHalfWidth * 2.0f, oldSize.y(), oldSize.z());

  m.setTranslate(translation);
}
```

Der neue Scope wird auf den eben berechneten Mittelpunkt und Größe gesetzt.

```
newScope.setBoundsByCenterAndSize(newCenter, newSize);
```

Nun werden die Transformationsmatrizen miteinander multipliziert, um die endgültige Transformation zu erhalten. Diese Transformation wird in dem OpenSG-*Transform*-Objekt gespeichert.

```
m.multLeft(old);
m.mult(scale);

beginEditCP(newShape->transform);
newShape->transform->setMatrix(m);
endEditCP(newShape->transform);
```

Der neue Scope wird dem aktuellen Shape zugewiesen. Außerdem wird der alte Shape dem Neuen als Elternshape zugewiesen und umgekehrt.

```
newScope.setBoundsByCenterAndSize(newCenter, newSize);
newShape->scope = newScope;

newShape->parent = shape;
shape->children.push_back(newShape);
```

Der neue Shape wird einem Shape-Vektor hinzugefügt, der am Ende der Methode zurückgegeben wird.

```
newShapes.push_back(newShape);
```

Schließlich wird der neue Shape einer STL-Map hinzugefügt, die speichert, von welcher Regel welche Shapes erzeugt wurden.

```
shapesCreatedFromRule[selectedRule].push_back(newShape);
```

2.3 Ressourcenverwaltung - CityResourceManager-Klasse

Die *ResourceManager*-Klasse ist für das Laden und Bereitstellen der 3D-Modelle und der Materialien, die das Erscheinen der 3D-Modelle bestimmen, zuständig. Dafür greift er auf die Ordner *models/*, *shader/* und *textures/* im Unterordner *Resources/* zurück, in denen sich die 3D-Modelle, Shaderprogramme und Texturen als Dateien befinden.

2.3.1 Ressourcen

2.3.1.1 3D-Modelle

3D-Modelle bilden die Grundlage für die später generierten Häuser. Hierbei kann es sich sowohl um kleine Bausteine für Häuserfassaden, wie z.B. Fenster und Türen, als auch um komplexere Modelle, z.B. ein detailliertes Model eines kompletten Balkons mit Tür und Fenstern, handeln. Die den 3D-Modellen zugrunde liegende Geometrie kann mit gängigen 3D-Modellierungswerkzeugen, wie z.B. dem kommerziellen *Maxon Cinema 4D*[Maxon 08] oder dem frei verfügbaren *Blender*[Blender.org 08], erstellt werden.

2.3.1.2 Texturen

Texturen sind Bilddaten, die in geeigneter Art und Weise[4], auf die Oberfläche eines 3D-Modells aufgebracht werden. Sie geben dem 3D-Modell somit ein grundlegendes Aussehen und Material, wie z.B. Stein, Holz oder Marmor. Außerdem kommen *Cubemaps* zum Einsatz, die sich jeweils aus sechs Einzeltexturen zusammensetzen und eine räumlich Umgebung abbilden. Dies wird für spiegelnde Oberflächen (siehe auch Kapitel 2.3.1.3) und für die Darstellung einer Szenenumgebung genutzt. Ein Workflow zur Erstellung von Cubemaps und deren Aufbereitung zur Nutzung in OpenSG wird im nachfolgenden Abschnitt vorgestellt.

Erzeugung von Cubemaps Für die passende Umgebung einer Stadtszene sollen Cubemaps eingesetzt werden. Die im Internet frei verfügbaren Cubemaps sind allerdings meist recht unrealistisch von der Farbgebung und Wolkenform, so dass diese eher in Science-Fiction-Umgebungen passen würden. Deswegen wurde nach einer Möglichkeit gesucht, die Cubemaps selber anzufertigen. Das Programm *Vue 6 Infinite PLE*[e-on software 08] von *E-on software* ist ein Landschaftsgenerator und -renderer, der für den nicht kommerziellen Einsatz kostenlos ist. Eine weitere Variante ist *Vue 6 xStream PLE*, die das Programm in gängige 3D-Modellierungssoftware wie z.B. 3d Studio Max und Cinema 4D, einbindet. Vue bietet vielfältige Möglichkeiten zur Erstellung von realistischen Umgebungen. Dabei können Form und Farbe der Wolken detailliert eingestellt werden. Der Himmel wird entsprechend der Tageszeit- und Dunsteinstellungen realistisch eingefärbt.

Die Umgebung kann über *Hauptmenü→File→Export Sky* als Cubemap exportiert werden. Die Exporteinstellungen sind Abbildung 2.12 zu entnehmen. Hierbei ist es wichtig, den Punkt *Supporting geometry* auf *Cube* zu stellen und als Auflösung in X-Richtung das dreifache der Auflösung der späteren sechs Einzelbilder anzugeben. Dies liegt darin begründet, dass Vue die

[4] Wie eine Textur auf die Oberfläche eines 3D-Modells gebracht wird, ist durch Texturkoordinaten geregelt, die meist mit dem 3D-Modellierungswerkzeug erzeugt werden.

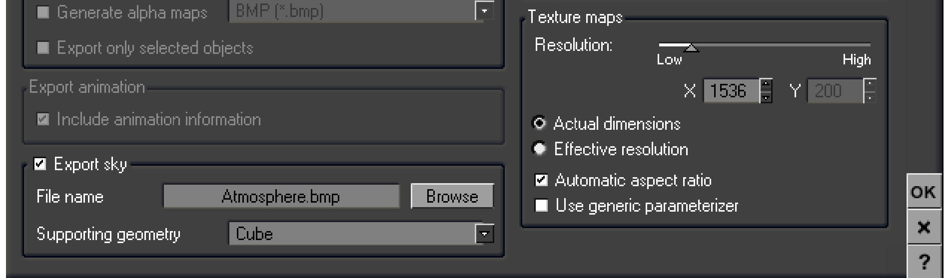
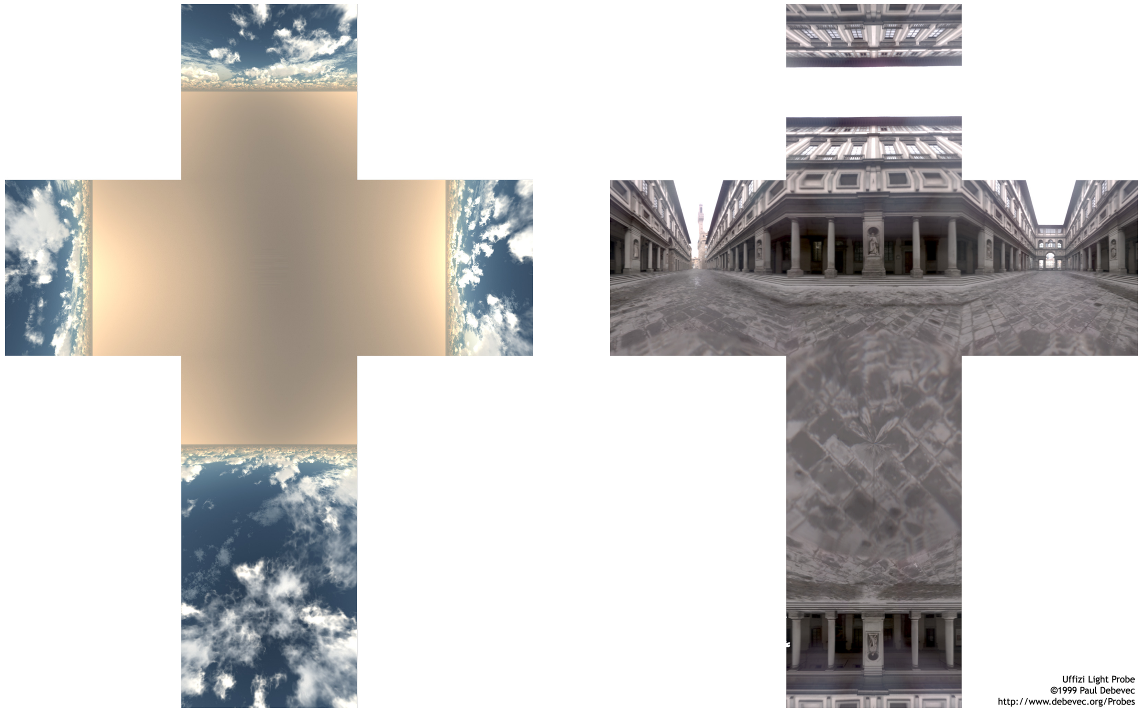


Abbildung 2.12: Vue Infinite Optionen zum Export von Cubemaps

Cubemap in einer kreuzförmigen Anordnung in eine einzelne Datei exportiert (siehe Abbildung 2.13a). Will man z.B. Cubemap-Bilder der Auflösung 512×512 haben, so wählt man $3 \times 512 = 1536$ als Auflösung in X-Richtung in Vue. Ist die Checkbox *Automatic aspect ratio* angeklickt, so wird die Auflösung in Y-Richtung automatisch angepasst.



(a) Cubemap einer Himmelsumgebung, die aus Vue Infinite exportiert wurde

(b) Cubemap einer Himmelsumgebung in HDRShop-Anordnung

Abbildung 2.13: Vertikale, kreuzförmige Cubemap-Anordnungen

Nun muss die Cubemap in ihre sechs Einzelbilder zerlegt werden. Dafür kann das frei verfügbare Programm *ATI CubeMapGen*[ATI 09] verwendet werden, das Cubemaps in verschiedene Formaten einlesen und exportieren kann. Man lädt die in Vue erstellte Cubemap über den Button *Load Cube Cross* (siehe Abbildung 2.14). Unter *Export Image Layout* stellt man *OpenGL Cube* ein. Klickt man die Checkbox *Skybox* an und stellt man *RenderMode* auf *Reflect (Per Pixel)*, so erhält man im 3D-Fenster eine Vorschau auf die Himmelsumgebung und eine diese spiegelnde Kugel. Die Cubemap wird in die sechs Einzelbilder über den Button *Save CubeMap to Images* exportiert.

Da CubeMapGen die Cubemap in einer Anordnung wie in Abbildung 2.13b erwartet (bei der der Boden unterhalb des Kreuzschnittpunktes liegt), diese aber anders angeordnet ist (siehe Abbildung 2.13a, der Boden liegt im Kreuzschnittpunkt), müssen die Einzelbilder anschließend transformiert werden. Tabelle 2.1 fasst die Transformationen zusammen. Die Transformationen können z.B. mit dem frei erhältlichen Bildbearbeitungsprogramm *Gimp*[GIMP 09] durchgeführt werden.

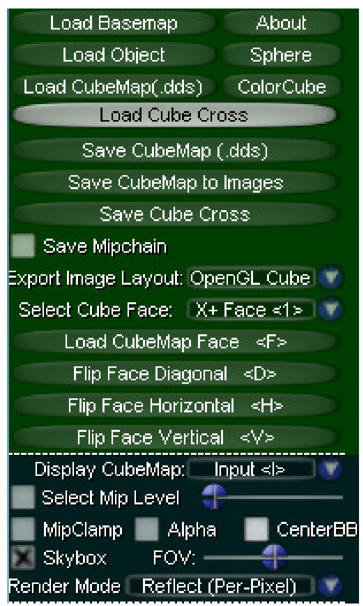


Abbildung 2.14: ATI CubeMapGen-Optionen

Dateiname aus CubeMapGen	**Transformationen**	**Würfelseite im ENU-Koordinatensystem**
c00.	$\hookrightarrow$ und $\leftrightarrow$	Osten
c01.	$\hookleftarrow$ und $\leftrightarrow$	Westen
c02.	$\updownarrow$	Norden
c03.	$\leftrightarrow$	Süden
c04.	$\updownarrow$	Unten
c05.	$\leftrightarrow$	Oben

$\hookleftarrow$ ... Rotation um 90° im Uhrzeigersinn

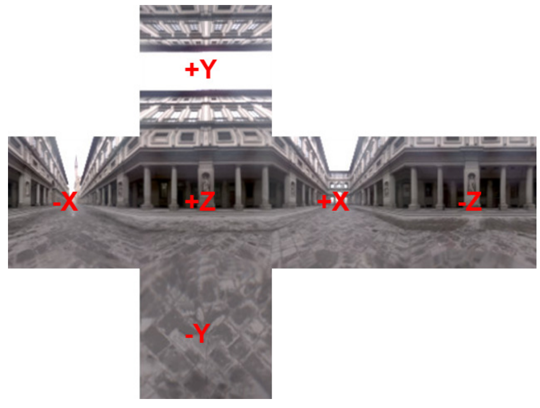
$\hookrightarrow$ ... Rotation um 90° gegen Uhrzeigersinn

$\leftrightarrow$ ... Spiegelung in horizontaler Richtung

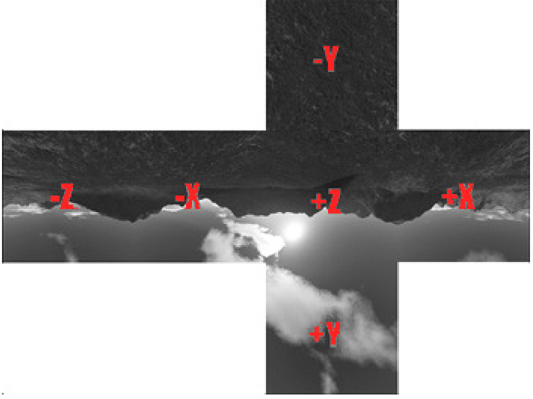
$\updownarrow$ ... Spiegelung in vertikaler Richtung

Tabelle 2.1: Übersicht der durchzuführenden Transformationen der aus CubeMapGen exportierten Bilder und deren Entsprechung im ENU-Koordinatensystem

Leider gibt es keinen einheitlichen Standard zur Anordnung von Cubemaps. Zusätzlich zu den Varianten aus Abbildung 2.13a gibt es z.B. noch eine horizontale, kreuzförmige Anordnung wie in Abbildung 2.15a dargestellt wird, sowie die OpenGL-Variante, bei der die horizontalen Bilder auf dem Kopf stehen (siehe Abbildung 2.15b).



(a) Cubemap in horizontaler, kreuzförmiger Anordnung



(b) Cubemap in horizontaler, kreuzförmiger OpenGL-Anordnung (aus [Wright 07])

Abbildung 2.15: Horizontale, kreuzförmige Cubemap-Anordnungen

2.3.1.3 Beleuchtungsmodelle mittels OpenGL Shading Language

Beleuchtungsmodelle bilden die physikalischen Eigenschaften von Licht in vereinfachter Form ab, um die Beleuchtung von 3D-Modellen in Echtzeitanwendungen zu realisieren. Die Beleuchtung eines Punktes wird im vorliegenden Beleuchtungssystem, das auf dem *Phong*-Beleuchtungsmodell basiert, durch drei Lichtarten definiert: ambientes Licht A, diffuses Licht D und spiegelndes Licht S (siehe Abbildung 2.16).

Die Farbe I eines Pixels wird durch folgende Gleichung definiert:

$$I = A + att * (D + S)$$

Der att-Term steht für die Abnahme des Lichts bei zunehmender Entfernung des Objektes zur Lichtquelle. Dieser wird im angewandten Lichtmodell ignoriert, da eine Sonnenbeleuchtung simuliert werden soll, die keine Abnahme der Lichtintensität vorsieht.

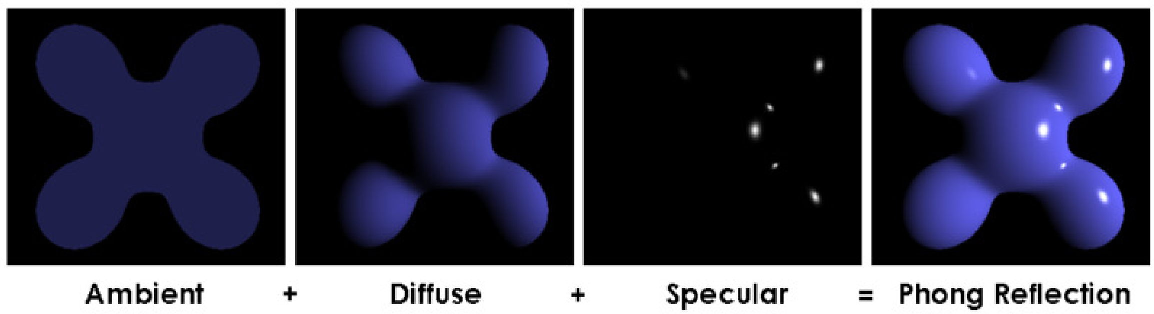


Abbildung 2.16: Visualisierung des Phong-Beleuchtungsmodells: Die Lichtfarbe ist Weiß, die ambiente und diffuse Farbe ist jeweils Blau und die spiegelnde Farbe ist Weiß. Quelle: [Wikipedia 09]

Eine Lichtquelle wird als Punktlichtquelle angesehen, die Licht in einer gleichmäßigen Intensität in alle Richtungen aussendet und über eine Position im Raum, und Lichtfarben für ambientes, diffuses und spiegelndes Licht definiert wird.

Das ambiente Licht A ergibt sich aus dem Produkt der ambienten Lichtfarbe Al und der ambienten Materialfarbe Am:

$$A = Al * Am$$

Das diffuse Licht D ist abhängig vom Einfallswinkel des Lichts auf die Objekt-Oberfläche, der diffusen Licht- Dl sowie Materialfarbe Dm:

$$D = (max\{\vec{l} \cdot \vec{n}, 0\}) * Dl * Dm$$

wobei $\vec{l}$ ein Einheitsvektor ist, der von der Oberfläche in Richtung der Lichtquelle zeigt und $\vec{n}$ die normalisierte Oberflächennormale ist, also ein Vektor, der senkrecht auf der Oberfläche steht.

Die Berechnung des spiegelndes Lichts ist etwas komplizierter:

$$S = (max\{\vec{s} \cdot \vec{n}, 0\})^{shininess} * Sl * Sm$$

Der Vektor $\vec{s}$ ist die Summe des normalisierten Vektors $\vec{l}$, der von der Vertexposition zur Lichtquelle zeigt und dem normalisierten Vektor $\vec{e}$, der von der Vertexposition zum Betrachter zeigt. Der Vektor $\vec{s}$ muss ebenfalls normalisiert werden. Das Punktprodukt von $\vec{s}$ mit $\vec{n}$ ist mit dem Exponent $shininess$ versehen. Je höher der Exponent, desto schmaler ist der Bereich des spiegelnden Lichts. Die berechnete Stärke des spiegelnden Lichts wird noch mit der spiegelnden Licht- Sl und Materialfarbe Sm multipliziert.

Die Bezeichnungen und Gleichungen wurden aus [Shreiner 08] übernommen.

Es wurden verschiedene Beleuchtungsmodelle in Form von GLSL-Shaderprogrammen für eine realistische Darstellung von Häuserfassaden bei Sonnenlicht implementiert, die nachfolgend in aufsteigender Komplexität aufgezählt werden:

- Per-Vertex Beleuchtung
- Per-Vertex Beleuchtung mit Umgebungslicht
- Per-Vertex Beleuchtung mit Umgebungslicht und Reflexion
- Per-Pixel Beleuchtung mit Umgebungslicht, Normal- und Specular-Mapping

Exemplarisch wird der Programmcode von zwei Shaderprogramme erklärt. Die anderen Shaderprogramme sind jeweils Vereinfachungen dieser beiden Shader.

Per-Vertex Shader mit Umgebungslicht und Reflexion Dieser Shader setzt ein Beleuchtungsmodell um, dass die Beleuchtung durch eine Punktlichtquelle auf Vertex-Basis berechnet und dabei eine vereinfachte Art von Umgebungslicht miteinbezieht. Bei einem Punktlicht breitet sich das Licht gleichmäßig in alle Richtungen von einem Punkt im Raum aus. Die Berechnung erfolgt nur für jedes Vertex und ist daher weniger rechenintensiv als Per-Pixel-Berechnungen. Da dieses Beleuchtungsmodell nur für die flachen Oberflächen von einfachen Fassaden (siehe Kapitel 2.4.1), den Dächern und für Fenster eingesetzt wird, ist die Per-Vertex-Berechnung in diesem Fall auch ausreichend. Die Punktlichtberechnung entspricht dem oben beschriebenen Modell und wurde aus [Rost 06] übernommen und um Environment-Lighting und Reflection-Mapping ergänzt, was im folgenden beschrieben wird. Der Vertex-Shader beginnt mit Variablendeklarationen für so genannte *varying*-Variablen. Diese werden nach der Berechnung im Vertex-Shader interpoliert an den Pixel-Shader weitergegeben. Außerdem kommt eine *uniform*-Variable zum Einsatz, die von der OpenSG-Anwendung automatisch zur Verfügung gestellt wird. Diese beinhaltet eine Matrix, die eine Transformation vom lokalen Objektkoordinatenraum in den Weltkoordinaten der Szene darstellt.

```
varying vec3 color;
varying vec3 reflectDir;
varying vec3 normalWorldCoordinates;
varying vec3 normal;
uniform mat4 OSGWorldMatrix;
```

Zu Begin wird die Vertexposition in Bildschirmkoordinaten transformiert.

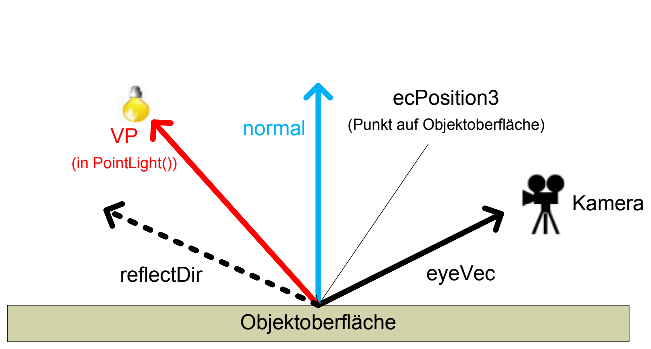
```
gl_Position = ftransform();
```

Die Normale des Vertex (`gl_Normal`) wird durch eine Multiplikation mit der Matrix `WorldMatrix3` in Weltkoordinaten umgerechnet, anschließend normalisiert und an den Pixel-Shader in der Variable `normalWorldCoordinates` weitergeleitet. Die Matrix `WorldMatrix3` ist der homogene Teil der `OSGWorldMatrix` und wird durch die Hilfsfunktion *GetLinearPart()* errechnet.

```
mat3 WorldMatrix3 = GetLinearPart(OSGWorldMatrix);
normalWorldCoordinates = normalize( WorldMatrix3 * gl_Normal);
```

Die Vertex-Position wird mithilfe der `gl_ModelViewMatrix` in das Kamerakoordinatensystem (wird auch als *Eye-Space* bezeichnet) transformiert und von homogenen Koordinaten in einen dreidimensionalen Vektor (`vec3 ecPosition3`) umgewandelt. Des Weiteren wird die Sichtrichtung (`eyeVec`) in Kamerakoordinaten durch Normalisieren und Negieren des erzeugten Vektors `ecPosition3` gebildet (siehe Abbildung 2.17 links):

```
vec4 ecPosition = gl_ModelViewMatrix * gl_Vertex;
vec3 ecPosition3 = (vec3(ecPosition)) / ecPosition.w;
vec3 eyeVec = -normalize(ecPosition3);
```



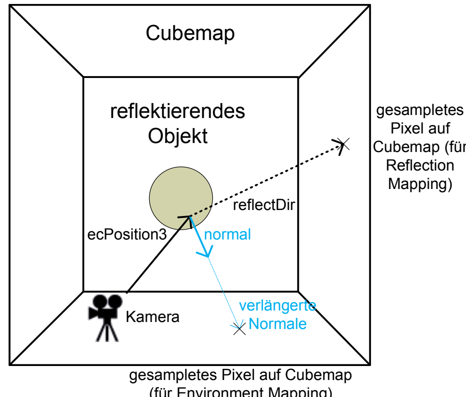


Abbildung 2.17: Links: Vektoren zur Licht- und Reflexionsberechnung, Rechts: Vektoren zur Berechnung des Umgebungslichtes und der Reflexion

Die Oberflächennormale des Vertex wird mithilfe der OpenGL-Matrix `gl_NormalMatrix` ebenfalls in Kamerakoordinaten transformiert und anschließend normalisiert.

```
normal = normalize(gl_NormalMatrix * gl_Normal);
```

Die Texturkoordinaten werden an den Pixelshader weitergeleitet:

```
gl_TexCoord[0] = gl_MultiTexCoord0;
```

Nun wird die aus [Rost 06] übernommene Lichtberechnung durchgeführt. Die dafür benötigten Variablen für ambientes, diffuses und spiegelndes Licht werden vorher definiert. Die bereits berechneten Vektoren für Sichtrichtung, die Vertexposition und die Vertexnormale in Kamerakoordinaten sowie die erzeugten Variablen für die Lichtarten werden der Lichtberechnung übergeben.

```
vec4 amb = vec4(0.0);
vec4 diff = vec4(0.0);
vec4 spec = vec4(0.0);

// aus [Rost 06] :
PointLight(0, eyeVec, ecPosition3, normal, amb, diff, spec);
```

Die in `PointLight()` berechneten Lichtfarben werden nun mit den in den Materialeigenschaften definierten Farben multipliziert und aufaddiert und das Ergebnis in der Variable `color` an den Pixelshader weitergeleitet.

```
color = vec3(amb * gl_FrontMaterial.ambient * 2.0 + diff * gl_FrontMaterial.
    diffuse + spec * gl_FrontMaterial.specular);
```

Schließlich wird noch der Reflexionsvektor am Vertex in Kamerakoordinaten berechnet und in Weltkoordinaten transformiert. Dazu wird die GLSL-Funktion *reflect()* genutzt, der die Vertexposition und die Vertexnormale in Kamerakoordinaten übergeben werden (siehe Abbildung 2.17 rechts):

```
reflectDir = reflect(ecPosition3, normal);
reflectDir = WorldMatrix3 * reflectDir;
```

Der Fragment-Shader (Pixel-Shader) erhält die vom Vertex-Shader berechnete Variablen:

```
varying vec3 color;
varying vec3 reflectDir;
varying vec3 normal;
varying vec3 normalWorldCoordinates;
```

Weitere Variablen sind der Reflexionsgrad (`uniform float reflectivity`) sowie Texturen für Oberflächenfarbe, Oberflächenreflexion und Umgebungstexturen als *Cubemaps*, also sechs Texturen, die einen Würfel bilden und somit eine komplette räumliche Umgebung abdecken.

```
uniform float reflectivity;
uniform sampler2D Tex0;
uniform sampler2D ReflectionMap;
uniform samplerCube DiffuseEnvMap;
uniform samplerCube ReflectionEnvMap;
```

Im Fragment-Shader wird zuerst die Umgebungsfarbe berechnet. Dazu wird mittels der GLSL-Funktion *textureCube()* ein Pixel der Umgebungstextur durch einen dreidimensionalen Vektor bestimmt (siehe Abbildung 2.17 rechts).

```
vec3 envColor  = vec3(textureCube(DiffuseEnvMap,  normalWorldCoordinates));
```

Die Stärke der Reflexion wird durch das Produkt des Grauwerts der Reflexionstextur mit der Variable *reflectivity* gebildet:

```
float reflection = (texture2D(ReflectionMap, gl_TexCoord[0].xy)).x;
float reflectionFactor = reflectivity * reflection
```

Reflektiert die Oberfläche, so wird die im Vertex-Shader berechnete Beleuchtungsfarbe abgeschwächt, um eine Übersättigung der Oberflächenfarbe nach dem Addieren der Reflexionsfarbe zu vermeiden.

```
vec3 weightedColor = color * (1.0 - reflectionFactor);
```

Die Reflexionsfarbe wird ähnlich der Umgebungsfarbe, jedoch mit dem Reflexionsvektor bestimmt und mit dem Reflexionsfaktor multipliziert (siehe Abbildung 2.17 rechts).

```
vec3 reflectionColor = vec3(textureCube(ReflectionEnvMap, normalize(
    reflectDir))) * reflectionFactor;
```

Schlussendlich wird die gewichtete Beleuchtungsfarbe *weightedColor* mit der Farbtextur multipliziert und zum Ergebnis jeweils die Umgebungsfarbe, als auch die Reflexionsfarbe addiert, wobei die Umgebungsfarbe nur zu einem Zehntel in die Addition eingeht.

```
gl_FragColor = vec4(weightedColor, 1) * texture2D(Tex0, gl_TexCoord[0].xy) +
    vec4((envColor * 0.1), 1) + vec4(reflectionColor,1);
```

Per-Pixel Shader mit Umgebungslicht, Normal- und Specular-Mapping Dieser Shader beleuchtet eine Oberfläche pixelgenau und mit einer simulierten Oberflächenstruktur durch Normal- und Specular-Maps. Normal-Mapping fügt der Oberfläche Details hinzu, indem es die Oberflächennormale mithilfe einer Normalentextur ändert. Der RGB-Farbwert eines Pixels einer Normal-Map wird als dreidimensionaler XYZ-Vektor interpretiert, der die Normale an einem Punkt auf der Oberfläche eines 3D-Modells beschreibt (der Blauwert repräsentiert die senkrecht auf der Oberfläche stehende Z-Achse, was die meist bläuliche Farbgebung von Normal-Maps erklärt).

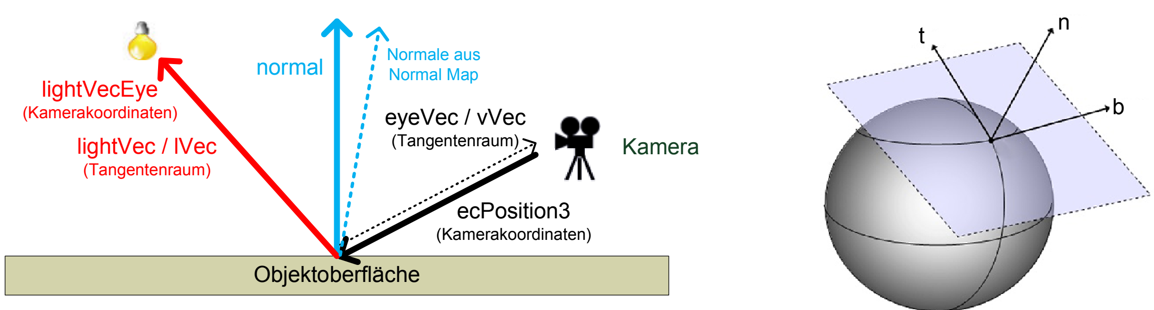


Abbildung 2.18: Links: Vektoren zur Lichtberechnung, Rechts: Basisvektoren des Tangentenraums

Dieser Normalenvektor bezieht sich auf das lokale Koordinatensystem eines Punktes auf der Modelloberfläche, in der der Punkt im Koordinatenursprung liegt und die Oberflächennormale an jedem Punkt $(0, 0, 1)$ ist. Um die Beleuchtung an diesem Punkt zu berechnen, wird die Richtung aus der das Licht kommt, sowie die Richtung zum Betrachter ebenfalls in dieses lokale Koordinatensystem überführt. Um eine entsprechende Transformationsmatrix zu erhalten, muss zu jedem Vertex des Modells ein Tangentenvektor und ein Binormalenvektor definiert sein. Diese drei Orthonormalvektoren bilden das lokale Koordinatensystem des Vertex, welches auch als Tangentenraum bezeichnet wird (siehe Abbildung 2.18 rechts). Der Tangentenvektor steht senkrecht zur Oberflächennormale und wird mithilfe der Texturkoordinaten in der OpenSG-Funktion *calcVertexTangents()* berechnet. Die Funktion berechnet ebenso den Binormalenvektor, der senkrecht zur Oberflächennormale und zum Tangentenvektor steht. Der Vertexshader ist also für die Transformation des Lichtvektors und der Sichtrichtung in den lokalen Tangentenraum verantwortlich. Folgende Variablen werden im Vertexshader definiert, um an den Pixelshader weitergeleitet zu werden:

```
varying vec3 lightVec;
varying vec3 eyeVec;
varying vec2 texCoord;
varying vec3 normalWorldCoordinates;
```

Außerdem wird wieder die *uniform*-Variable *OSGWorldCoordinates* benötigt, die OpenSG zur Verfügung stellt.

```
uniform mat4 OSGWorldMatrix;
```

Zu Begin wird die Vertexposition in den Bildraum transformiert und die Texturkoordinaten weitergeleitet.

```
gl_Position = ftransform();
texCoord = gl_MultiTexCoord0.xy;
```

Die Oberflächennormale wird in den Weltkoordinatenraum transformiert und normalisiert.

```
mat3 WorldMatrix3 = GetLinearPart(OSGWorldMatrix);
normalWorldCoordinates = normalize( WorldMatrix3 * gl_Normal);
```

Die Vektoren für den Tangentenraum werden, wie oben beschrieben, gebildet, wobei die Tangente durch das Texturkoordinatenfeld 1, und die Binormale durch das Texturkoordinatenfeld 2 des 3D-Objektes bereitgestellt werden.

```
vec3 t = normalize(gl_NormalMatrix * gl_MultiTexCoord1.xyz);
vec3 b = normalize(gl_NormalMatrix * gl_MultiTexCoord2.xyz);
vec3 n = normalize(gl_NormalMatrix * gl_Normal);
```

Alternativ könnte die Binormale $\vec{b}$ auch durch das Kreuzprodukt der Oberflächennormalen $\vec{n}$ und des Tangentenvektors $\vec{t}$ gebildet werden: `vec3 b = cross(n, t)`. Die Vertexposition und der Lichtvektor werden in den Kamerakoordinatenraum transformiert:

```
vec4 ecPosition = gl_ModelViewMatrix * gl_Vertex;
vec3 ecPosition3 = (vec3(ecPosition)) / ecPosition.w;

vec3 lightVecEye = gl_LightSource[0].position.xyz - ecPosition3;
```

Die transformierten Vektoren werden weiter in den Tangentenraum transformiert und an den Pixelshader weitergeleitet. Die Sichtrichtung im Objektraum ergibt sich durch Negieren der Vertexposition im Kamerakoordinatenraum und anschließende Transformation in den Objektraum (siehe auch Abbildung 2.18 links).

```
lightVec.x = dot(lightVecEye, t);
lightVec.y = dot(lightVecEye, b);
lightVec.z = dot(lightVecEye, n);

ecPosition3 = -ecPosition3;
eyeVec.x = dot(ecPosition3, t);
eyeVec.y = dot(ecPosition3, b);
eyeVec.z = dot(ecPosition3, n);
```

Im Pixelshader werden die aus dem Vertexshader übernommenen Variablen sowie die Texturen definiert:

```
varying vec3 lightVec;
varying vec3 eyeVec;
varying vec2 texCoord;
varying vec3 normalWorldCoordinates;
uniform sampler2D colorMap;
uniform sampler2D normalMap;
uniform sampler2D specularMap;
uniform samplerCube DiffuseEnvMap;
```

Die übergebenen Licht- und Sichtrichtungsvektoren werden normalisiert und in `lVec` und `vVec` gespeichert.

```
vec3 lVec = normalize(lightVec);
vec3 vVec = normalize(eyeVec);
```

Die Farbe der Oberflächentextur wird in `baseColor` gespeichert.

```
vec4 baseColor = texture2D(colorMap, texCoord);
```

Der Normalenvektor wird aus der Normalentextur gelesen. Der Wertebereich einer Farbkomponente der Textur ist $[0, 1]$ und wird in den Bereich $[-1, 1]$ durch $x * 2.0 - 1.0$ transformiert, um einen dreidimensionalen Vektor aus den drei Farbwerten zu erhalten.

```
vec3 normalFromMap = normalize(texture2D(normalMap, texCoord).xyz*2.0 - 1.0);
```

Die Menge an diffusem Licht (`float diffuse`) wird durch das Punktprodukt aus dem Licht- und dem Normalenvektor der Textur errechnet. Die diffuse Farbe des Materials wird mit der diffusen Lichtfarbe und der errechneten Menge multipliziert:

```
float diffuse = max( dot(lVec, normalFromMap), 0.0 );
vec4 vDiffuse = gl_LightSource[0].diffuse*gl_FrontMaterial.diffuse*diffuse;
```

Der spiegelnde Lichtanteil wird ebenfalls durch den Normalenvektor beeinflusst:

```
float specular = pow(clamp(dot(reflect(-lVec, normalFromMap), vVec), 0.0,
    1.0), gl_FrontMaterial.shininess );
```

Der Grauwert der Specular-Map wird ausgelesen.

```
float specularFromMap = (texture2D(specularMap, gl_TexCoord[0].xy)).x;
```

Die endgültige spiegelnde Farbe ergibt sich aus dem Produkt der spiegelnden Farbe des Materials und der Lichtquelle, sowie den beiden eben berechneten Faktoren:

```
vec4 vSpecular = gl_LightSource[0].specular * gl_FrontMaterial.specular *
    specular * specularFromMap;
```

Das Umgebungslicht wird mithilfe der Oberflächennormalen in Weltkoordinaten und der Umgebungstextur berechnet.

```
vec3 envColor  = vec3(textureCube(DiffuseEnvMap, normalWorldCoordinates));
```

Die endgültige Pixelfarbe ergibt sich wie folgt:

```
gl_FragColor = (vAmbient * 1.5 + vec4(envColor, 1) * 0.5 + vDiffuse * 0.5) *
    baseColor + vSpecular;
```

Die Umgebungsfarbe wird zur ambiente Farbe addiert, zu der die von der Lichtrichtung abhängige diffuse Farbe abgeschwächt dazu addiert wird. Diese Farbe wird mit der Oberflächentextur multipliziert und schließlich die spiegelnde Farbe hinzu addiert.

2.3.1.4 Materialien

Materialien setzen sich aus einem Beleuchtungsmodell, ein oder mehreren Texturen sowie Eigenschaften wie Glanzlichtfarbe und -Intensität zusammen. Die 3D-Modelle wurden aus Cinema4D bewusst ohne Materialdefinition exportiert, um später bei der Gebäudeerzeugung zufällig ein passendes Material zuweisen zu können, um somit Varianten eines Modells mit unterschiedlichen Oberflächentexturen zu schaffen.

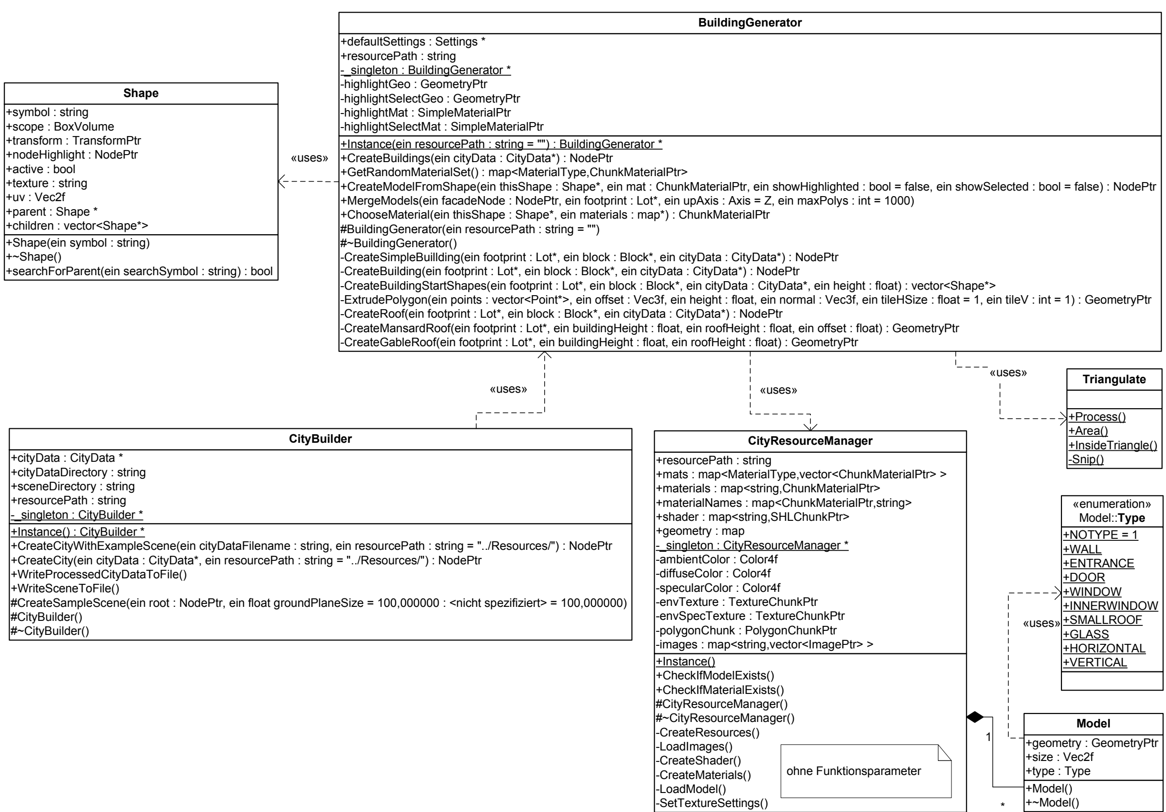


Abbildung 2.19: UML-Diagramm der BuildingGenerator- und der CityResourceManager-Klasse

2.3.2 Implementierung der CityResourceManager-Klasse

Die *CityResourceManager*-Klasse verwaltet die Ressourcen in STL-Map-Containern wie im UML-Diagramm in Abbildung 2.19 zu sehen. Die Methode *CreateResources()* lädt alle benötigten Ressourcen, indem sie weitere spezielle Methoden aufruft.

Die Cubemaps, die als Umgebungstexturen genutzt werden, werden in der Methode *CreateEnvironmentMaps()* geladen. Die sechs im Verzeichnis *Resourcen/textures/skybox* liegenden Texturen werden als OpenSG-*CubeTextureChunk* geladen und später für den OpenSG-*SkyBackground* genutzt (siehe Kapitel 3.1.3). Dabei wird eine vertikale Cubemap-Anordnung, wie in Abbildung 2.13b dargestellt, genutzt. Die gleiche Cubemap sollte in einem GLSL-Shader für reflektierende Materialien als Reflexions-Cubemap genutzt werden. Dies ist jedoch durch eine Inkonsistenz in OpenSG nicht möglich. Wird der *CubeTextureChunk* in einem Shader genutzt, stehen dadurch die vier horizontalen Teilbilder der Cubemap auf dem Kopf, wenn man die Cubemap mit der GLSL-Funktion *textureCUBE* samplelt. Der Shader erwartet die Cubemap in einer vertikalen Cubemap-Anordnung wie in Abbildung 2.15b dargestellt, was dem OpenGL-Standard entspricht. Deshalb werden separate Cubemaps für die GLSL-Shader geladen (*Resources/textures/DiffuseCubeMap.dds* und *Resources/textures/SpecularCubeMap.dds*).

Ein 3D-Modell wird durch die Methode *LoadModel()* geladen und durch die Hilfsklasse *Model* repräsentiert. Die Klasse besteht aus einem Verweis auf ein OpenSG-*Geometry*-Objekt, das

die eigentlichen Geometriedaten (z.B. die Dreiecke aus denen das Model zusammengesetzt wird und Texturkoordinaten) enthält, sowie einem Modeltyp (siehe Tabelle 2.2).

Modeltyp	**Beschreibung**	**Verzeichnis**
NOTYPE	Kein spezieller Typ	*misc/*
WALL	Mauerwerk	*wall/*
DOORS	Türen	*doors/*
WINDOW	Fensterumrahmungen	*windows/*
SMALLROOF	Zierdächer	*smallRoofs/*
GLASS	Fensterglas	*glass/*
HORIZONTAL	Modelle, die horizontal gestreckt werden	*horizontal/*
HORIZONTAL_NDH	wie oben, jedoch wird keine Skalierung in der Tiefe vorgenommen	*horizontal/noDepthScale/*
VERTICAL	Modelle, die vertikal gestreckt werden	*horizontal/*
VERTICAL_NDH	wie oben, jedoch wird keine Skalierung in der Tiefe vorgenommen	*vertical/noDepthScale/*
TEXTURED	Modelle, die aus mehreren Geometrien mit Materialdefinitionen bestehen können	*textured/*

Tabelle 2.2: Übersicht Modeltypen

Modelle werden aufgrund ihres Typs beim Zusammensetzen von Häusern (siehe Kapitel 2.4.4) unterschiedlich weiterverarbeitet. Sie können im VRML2[5]-Format, das z.B. von Cinema 4D exportiert wird, geladen werden. Dabei exportiert Cinema 4D ein Modell in der in Abbildung 2.20 dargestellten Hierarchie. Die *LoadModel()*-Methode pickt aber nur das Geometry-Objekt heraus. Besteht ein Modell aus mehreren Geometry-Objekten, so werden diese zu einem Geometry-Objekt zusammengefügt (über die OpenSG-Methode *Geometry::merge()*). Ist ein Modell vom Typ *TEXTURED*, so werden zusätzlich die Materialnamen aus der Datei geladen und gespeichert. Über die OpenSG-Methode *calcVertexTangents()* werden weiterhin beim Laden aus den Texturkoordinaten des Modells die für das Normal-Mapping erforderlichen (siehe Kapitel 2.3.1.3) Tangenten- und Binormalen-Vektoren berechnet. Schließlich wird das Modell in einer STL-Map des ResourceManagers gespeichert.

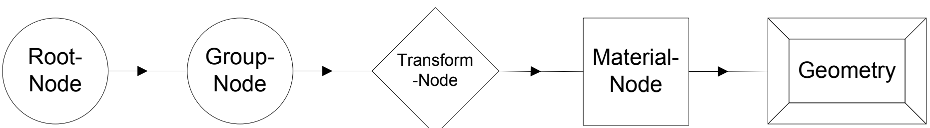


Abbildung 2.20: VRML2-Export-Struktur aus Cinema4D

Texturen werden durch die Methode *LoadImage()* geladen. Die Methode *CreateImages()* lädt die Texturen aus den Unterverzeichnissen des *textures/*-Verzeichnisses. Die Unterverzeichnisse ordnen den Texturen ähnlich wie beim Laden von 3D-Modellen eine Bedeutung zu (siehe Tabelle 2.3).

[5]VRML - Virtual Reality Markup Language

Modelverzeichnis	Beschreibung
facade/	Bilder von Fassadenkacheln für einfache Modelle
roof/	Dachtexturen
materials/	Texturen für Türen
wall/	Texturen für das Mauerwerk
window/	Fenstertexturen
reflection/	Texturen für reflektierende Oberflächen
misc/	Verschiedene Texturen
miscNormal/	Verschiedene Texturen mit Normal-Mapping

Tabelle 2.3: Übersicht Texturenverzeichnisse

Außerdem befinden sich in einigen Texturordnern weitere Unterordner für Normal- und Specular- oder Reflection-Maps, die für die in 2.3.1.3 beschriebenen Beleuchtungsmodelle benötigt werden. Die Texturen werden als OpenSG-*Image*-Objekte in einer STL-Map mit dem Texturnamen als Index als auch in mehreren STL-Vektoren im *CityResourceManager*-Objekt zum schnellen Zugriff gespeichert.

Die Methode *CreateMaterials()* erzeugt auf Grundlage der bereits geladenen Texturen entsprechende Materialien. Die Tabelle 2.4 zeigt die verschiedenen Materialkategorien in der ersten Spalte. In der zweiten Spalte wird das zugrunde liegende Beleuchtungsmodell in abgekürzter Form genannt. PPSUNS steht dabei z.B. für Per-Pixel Shader mit Umgebungslicht, Normal- und Specular-Mapping. Die dritte Spalte nennt das Texturenverzeichnis. Die nächsten drei Spalten geben an, ob zusätzlich zur diffusen Farbtextur eine Normalen-, Glanzlicht- (Specular) oder Reflexionstextur vom Beleuchtungsmodell verwendet werden. Die Spalten sieben und acht geben den Glanzlichtexponenten (siehe 2.3.1.3) sowie die Glanzlichtintensität im Bereich $[0, 1]$ an. Die letzte Spalte zählt auf, welche Modelle später im Gebäudegenerator das entsprechende Material zugewiesen bekommen.

Als ambiente Farbe wird ein dunkles Grau (RGB-Wert $(0.4, 0.4, 0.4)$), als diffuse Farbe ein helles Grau (RGB-Wert $(0.65, 0.65, 0.65)$) und als spiegelnde Farbe ein helles Gelb (RGB-Wert $(1.0, 1.0, 0.85)$) in den Materialeigenschaften verwendet.

Material	Shader	Diffuse	N	S	R	Shininess	Specular	Modell[a]
MAT_FACADE	PVSU	*facades/*	-	-	-	128	1.0	generiert
MAT_ROOF	PVS	*roof/*	-	-	-	32	1.0	generiert
MAT_MATERIAL	PPSUNS	*materials/*	x	x	-	32	0.2	alle, außer die unten genannten
MAT_DOOR	PPSUNS	*door/*	x	x	-	32	0.5	*door/*
MAT_WALL	PPSUNS	*wall/*	x	x	-	10	0.3	*wall/*
MAT_WINDOW	PVSUR	*window/*	-	-	x	32	0.5[b]	*glass/*
MAT_REFLECTION	PVSUR	*reflection/*	-	-	x	32	0.5[c]	manuell

[a] Zuweisung erfolgt im BuildingGenerator (siehe Kapitel 2.4.4)
[b] Jedoch wird Weiß als spiegelnde Farbe verwendet
[c] Jedoch wird Weiß als spiegelnde Farbe verwendet

Tabelle 2.4: Übersicht Materialzusammensetzungen

Ein Material ist als OpenSG-*ChunkMaterial* implementiert. Dieses setzt sich aus mehreren so genannten *Chunks* zusammen. Der *MaterialChunk* speichert Materialeigenschaften wie Farben und Glanzlichtintensität. Ein *TextureChunk* steht für eine Textur, ein *SHLChunk* für einen

Shader und der *PolygonChunk* speichert Einstellungen (genauer: OpenGL-*Renderstates*) zum Rendering. Je nach Shader werden weitere *TextureChunks* für die Normalen-, die Glanzlicht- und die Reflexionstextur verwendet. Die Abbildung 2.21 verdeutlicht den Aufbau.

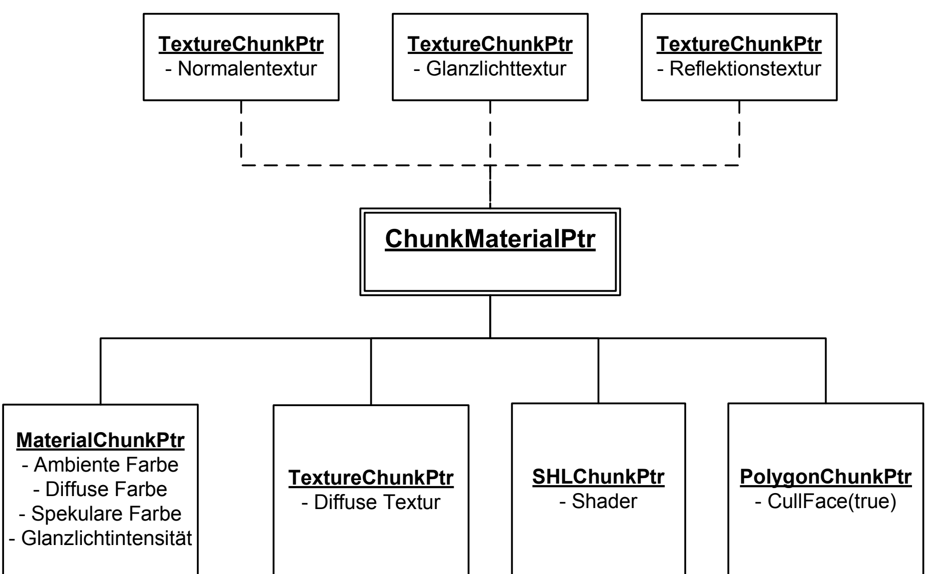


Abbildung 2.21: Bestandteile eines Materials implementiert als ChunkMaterial

2.4 Gebäudegenerator - BuildingGenerator-Klasse

2.4.1 Einfache Gebäudemodelle mit Fassadentexturen

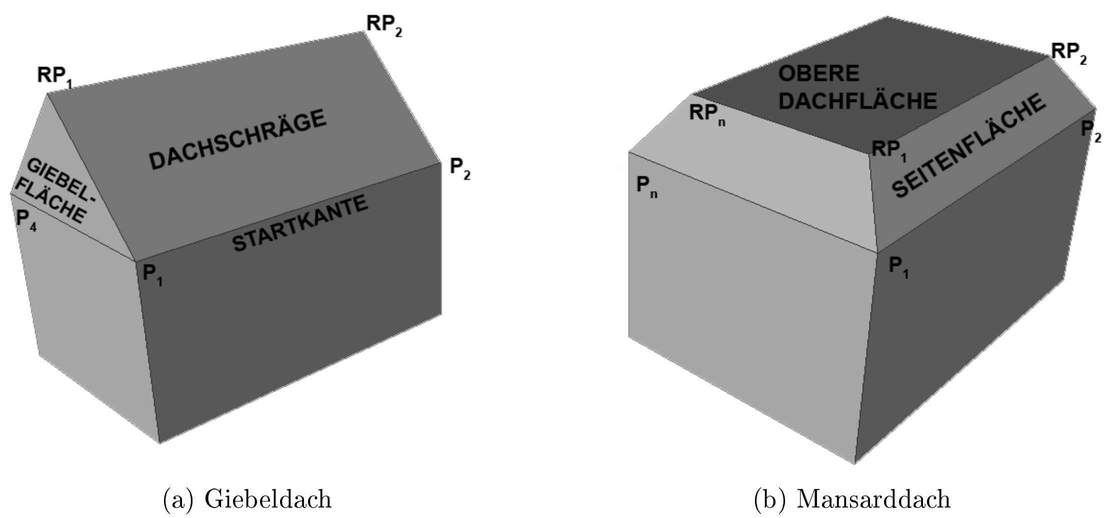
Der Gebäudegenerator (*BuildingGenerator*-Klasse) erzeugt ausgehend von Gebäudegrundrissen 3D-Modelle von Gebäuden mit Dächern. Er kann einfache Gebäude, deren Fassaden nur aus flachen Flächen bestehen, die mit einer Fassadentextur versehen sind, erstellen. Diese einfachen Gebäude sollten als erste Stufe eines *Level-of-Detail*[6]-Systems zur entfernten Darstellung großer Gebäudemengen verwendet werden.

2.4.2 Prozedurale Gebäudemodelle

Bei näherer Betrachtung sollten möglichst detaillierte Gebäude, die aus echten 3D-Modellbausteinen, wie Fenster, Türen, Verzierungen usw., bestehen, einen realistischen Eindruck der Gebäudeumgebung erzeugen. Solche 3D-Modelle werden vom Gebäudegenerator im Zusammenspiel mit einer Szenendefinition und einem Regelset erzeugt.

2.4.3 Dächergenerieung

Auf die Gebäude werden einfache Dächergeometrien gesetzt. Diese sollen sich nur aus Flächenpolygonen und Ziegeltexturen zusammensetzen. Für rechteckige Grundrisse werden Giebeldächer (siehe Abbildung 2.22 links) erzeugt, wobei der Giebel auf den beiden kürzeren Seiten liegt. Für alle weiteren Grundrissformen werden flache Mansarddächer (siehe Abbildung 2.22 rechts) eingesetzt, da diese aufgrund ihrer Form auf jeden Grundriss passen.



(a) Giebeldach (b) Mansarddach

Abbildung 2.22: Schematische Darstellung eines Giebeldaches und eines flachen Mansarddaches

[6]abgekürzt: LOD, Variationen eines 3D-Modells werden in Abhängigkeit zur Entfernung des Betrachters anzeigt.

2.4.4 Implementierung

Zur Erläuterung der Implementation des BuildingGenerators werden die Abläufe innerhalb der Methoden der Klasse nur beschrieben. Auf eine detaillierte Quellcodeauflistung wurde aufgrund des Umfangs der Methoden verzichtet.

Die Methode ***CreateBuildings()*** erzeugt anhand der Gebäudegrundrisse eines *CityData*-Objekts Häusermodelle. Für jedes *Lot*-Objekt werden folgende Schritte durchgeführt:

- Eine Transformation, die eine Verschiebung vom Koordinatenursprung zum Mittelpunkt des Gebäudegrundrisses darstellt, wird erzeugt und in einem OpenSG-*TransformPtr* gespeichert.
- Ein OpenSG-*DistanceLODPtr* wird für das LOD-System erstellt. Die Distanz, ab der zwischen zwei LOD-Stufen umgeschaltet wird, wird aus einem Variablenblock gelesen (Voreinstellung: 300 m).
- Über die Methode *GetRandomMaterialSet()* wird eine zufällige Materialauswahl getroffen. Für jeden Materialtyp wird ein entsprechendes Material zufällig ausgewählt und in der STL-Map *materials* des *Lot*-Objekts gespeichert.
- Diese zufällig erzeugten Materialien können in einem Variablenblock überschrieben werden. Folgende Variablennamen stehen dabei zur Verfügung: *TextureWall*, *TextureMaterial*, *TextureWindow* und *TextureDoor*.
- Für das Gebäude werden Variablen für die Gebäudehöhe (*BuildingHeight*, Voreinstellung 15 m), die Etagenhöhe (*FloorHeight*, 3 m) und für die Dachhöhe (*RoofHeight*, 4 m) ausgelesen. Für jede Höhe ist eine entsprechende Variable definiert, die die maximale Abweichung von der Höhe angibt (z.B. *BuildingHeightVar*). Mit dieser maximalen Abweichung werden nun zufällig die endgültigen Höhen für dieses Gebäude in *ActualBuildingHeight*, *ActualFloorHeight* und *ActualRoofHeight* gespeichert.
- Ein einfaches Gebäudemodell wird mit der Methode *CreateSimpleBuilding()* erzeugt (siehe unten).
- Ein prozedurales Gebäudemodell wird mit der Methode *CreateBuilding()* erzeugt (siehe unten).
- Die erzeugten Modelle werden als Kindelemente an einen OpenSG-*NodePtr* angehangen, der den *DistanceLODPtr* als *Core*[7] enthält.
- Der *NodePtr* wird als Kindelement an einen Transformationsknoten angehängt, der die anfangs erzeugte Transformation beinhaltet.
- Schließlich wird der Transformationsknoten einem Gruppenknoten hinzugefügt, der nach der Erzeugung aller Gebäude von der Methode zurückgegeben wird.

Einfache Gebäudemodelle Die Methode ***CreateSimpleBuilding()*** der *BuildingGenerator*-Klasse erzeugt ein einfaches Gebäudemodell. Als Fassadentexturen werden Fassadenfotos eines wiederholbaren Bereiches einer Etage (siehe Bild 2.23) verwendet. Diese werden dann je nach Gebäudehöhe und Etagenhöhe auf der Fassadenoberfläche wiederholt.

[7]In OpenSG wird die Funktionsweise eines Knotens im Szenengraphen durch den so genannten *Core*

Abbildung 2.23: Zwei Fassadentexturen, die je nach Größe der Fassade und der Etagenhöhe oft gekachelt werden

- Der Methode wird der Gebäudegrundriss, der zugehörige Gebäudeblock und eine Referenz auf das Szenenobjekt übergeben. Die Gebäudehöhe und die Etagenhöhe werden aus dem aktuell gültigen Variablenblock gelesen. Die Anzahl der Etagen ergibt sich aus $\lfloor \textit{Gebäudehöhe}/\textit{Etagenhöhe} \rfloor$.
- Die Hilfsmethode *ExtrudePolygon()* erzeugt aus dem Gebäudegrundriss-Polygon die einzelnen Fassadenflächen mit der ausgelesenen Höhe und der Etagenhöhe entsprechenden Texturkoordinaten für die Texturwiederholung. Abbildung 2.24 zeigt die Generierung der Texturkoordinaten für eine Fassade innerhalb der *ExtrudePolygon()*-Methode.
- Ist in der Szenendatei die Z-Achse als Hochachse angegeben, so muss das Modell noch durch die Methode *TransformGeoZUp()* transformiert werden, denn es wurde im normalen OpenSG-Koordinatensystem erzeugt (in dem die Y-Achse die Hochachse ist).
  - Die *TransformGeoZUp()*-Methode transformiert ein OpenSG-Geometry-Objekt, so dass die Z-Achse die Hochachse des Modells wird. Die Matrix, mit der die Vertexposition, die Normalen, die Tangenten- und die Binormalenvektoren des Geometry-Objekts multipliziert werden müssen ist Folgende:

    $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
- Abschließend wird dem Geometry-Objekt ein Fassadenmaterial zugewiesen und es wird ein OpenSG-*Node* erstellt, der die Geometrie als Core beinhaltet und von der Funktion zurückgegeben wird.

Prozedurale Gebäude Die Methode ***CreateBuilding()*** erzeugt auf Grundlage einer Regeldatei ein Gebäude prozedural.

---

bestimmt. Es gibt z.B. *Cores* für Transformationen, Geometrien, Gruppen usw.

Abbildung 2.24: Beispiel zur Texturkoordinatengenerierung der einfachen Fassaden mit zweifacher Kachelung in vertikaler und vierfacher Kachelung in horizontaler Richtung

- Anfangs wird die Gebäudehöhe ausgelesen und Start-Shapes für jede Fassade mithilfe der Methode *CreateBuildingStartShapes()* erzeugt. Diese erstellt für jede Kante des Gebäudegrundrisses einen Shape. Liegt die Kante nicht an einer Straße und auch nicht an der Hofseite des Gebäudes, so bekommt der Shape den Namen *SideFacade*. Stellt die Kante eine Hofseite dar, so wird der Shape *BackFacade* genannt. Die längste Kante, die an einer Straße liegt, bekommt den Start-Shape-Namen *FrontFacade*. Shapes, die eine geringere Breite als 3 m haben, bekommen den Namen *SmallFacade*. Alle anderen Start-Shapes heißen *Facade*.

- Nun wird zufällig eine Regeldatei ausgewählt, wenn mehrere Dateien für dieses Gebäude definiert wurden. Die Vorrangsreihenfolge ist auch hier wieder *Lot*-, *Block*-, *CityData*-Objekt und schließlich die *DefaultSettings.xml*-Datei.

- Ein Gruppenknoten (OpenSG-*NodePtr* mit einem OpenSG-*GroupPtr* als Core), der alle Fassadenknoten beinhalten wird, wird erzeugt.

- Nun wird über die einzelnen Fassaden-Start-Shapes iteriert.

  - Der *RuleProcessor* erzeugt ausgehend vom Start-Shape mithilfe der ausgewählten Regeldatei die Fassaden-Shapes.

  - Ein *NodePtr*, an den dann die Fassadengeometrien später angehängt werden, wird erzeugt.

- Nun wird über jeden erzeugten Shape iteriert.
  * Falls es sich bei dem Shape um ein Nichtterminalsymbol handelt, also beinhaltet sein Name nicht die Endung *.wrl*, so wird mit diesem Shape nicht weiter verfahren und zum Nächsten übergegangen.
  * Es wird entweder ein Material durch die Methode *ChooseMaterial()* zufällig ausgewählt oder falls direkt auf dem Shape ein Material definiert ist, so wird dieses benutzt. Ansonsten wird je nachdem, welchen Modeltyp das durch den Shape definierte Modell hat, ein Material entsprechend der zuvor erstellten Materialauswahl des Gebäudes zugewiesen (siehe Tabelle 2.4 auf Seite 44).
  * Nun wird die 3D-Geometrie mit der Methode *CreateModelFromShape()* (siehe unten) erstellt und in einem *NodePtr* gespeichert. Dieser wird an den Fassadenknoten als Kindelement angehängt.
- Ist die *MergeGeometries*-Variable auf 1 gesetzt, werden die erzeugten vielen kleinen Geometrien nach Material sortiert und zu wenigen großen Geometrien zusammengefügt. Die erzeugten Geometrien bestehen dabei aus maximal soviel Dreiecken, wie in der Variable *MaxPolygons* definiert ist. In der *MergeModels()*-Methode werden die erzeugten Geometrien außerdem ggf. transformiert, je nachdem wie die Variable *UpAxis* gesetzt ist.
- Nun wird der Fassadenknoten an den Gruppenknoten angehangen, die erzeugten Shapes gelöscht und schließlich mit der nächsten Fassade fortgefahren.

- Am Ende der Methode werden ebenfalls die Start-Shapes gelöscht und die fertige Gebäudegeomtrie als *NodePtr* zurückgegeben.

Nachfolgend wird die Methode ***CreateModelFromShape()*** erläutert, die aus einem Shape mit einem Terminalsymbol eine an den Shape angepasste Geometrie erzeugt.

- Zu Beginn wird geprüft, ob die zu erzeugende Geometrie vom CityResourceManager geladen wurde.
- Ist dies der Fall, so wird als nächstes geprüft, ob das Model nicht vom Typ *TEXTURED* ist, woraufhin ein Verweis auf die Geometrie durch einem OpenSG-*GeometryPtr* gespeichert wird.
- Ein Materialknoten wird erstellt und das ausgewählte Material gesetzt.
- Die Texturkoordinaten (UV-Koordinaten) von Geometrien einiger Modeltypen werden mithilfe der Shape-Größe angepasst. So werden z.B. die Texturkoordinaten der Wandgeometrien so angepasst, dass obwohl eine Wand aus mehreren nebeneinander liegenden Geometrien besteht, die Textur nahtlos aufgebracht wird. Außerdem können Texturkoordinaten durch einen zweidimensionalen Vektor, der bei einem Symbol in einer Regel angegeben wurde, skaliert werden. Zuerst wird dieser Vektor ausgelesen. Ein Wert größer als 1 hat eine Vergrößerung der aufgebrachten Textur zur Folge, wodurch diese gröber wirkt.
- Nun wird ein *NodePtr* erzeugt, der später die neue Geometrie als Core enthalten wird.
- Nachfolgend wird nur die Texturkoordinatenskalierung für Wandtexturen erklärt.
  - Es werden dazu neue Texturkoordinaten erzeugt.

  - Die räumliche Position (in Form der vier Eckpunkte) des Shapes wird ausgelesen.
  - Nun werden für die vier Eckpunkte einer Wandgeometrie die neuen Texturkoordinaten erzeugt. Dazu werden die Positionen der Shape-Eckpunkte mit der Texturkoordinatenskalierung multipliziert.
  - Würde man nun die neuen Koordinaten einfach dem vom ResourceManager geladenen Geometrie-Objekt zuweisen, so würden die neuen Texturkoordinaten bei jeder Instanz der Geometrie vorhanden sein. Deswegen wird die Geometrie erst kopiert und der Kopie dann die neuen Texturkoordinaten zugewiesen. Schließlich wird mit dem vorher erzeugten *GeometryPtr* auf diese neue Geometrie verwiesen.
- Bei Geometrien vom Typ *Model::HORIZONTAL* und *Model::HORIZONTAL_NDH* wird die y-Texturkoordinate entsprechend der Breite des Shapes angepasst. Bei Geometrien vom Typ *Model::VERTICAL* und *Model::VERTICAL_NDH* wird die x-Texturkoordinate entsprechend der Höhe des Shapes angepasst. Wird keine Texturkoordinatenveränderung vorgenommen, so verweist der *GeometryPtr* auf die vom ResourceManager geladene Geometrie.
- Nun wird die Transformationsmatrix der Geometrie berechnet. Diese erhält man durch die Shape-Methode *GetTransformMatrix()*
- Mithilfe der Transformationsmatrix wird ein OpenSG-*TransformPtr* erzeugt. Dieser wird als Core an einen OpenSG-*NodePtr* angehängt, der als Transformationsknoten die Geometrie als Kindelement erhält.
- Der Transformationsknoten wird als Kindelement an einen Materialknoten angehängt. Der Materialknoten wird von der Methode schließlich zurückgegeben.

Die Methode ***Shape::GetTransformMatrix()*** berechnet die Transformationsmatrix eines Shapes.

- Die Skalierungsfaktoren der X- und der Y-Achse ergeben sich aus der Größe des Shapes (Scope) geteilt durch die Größe der Geometrie.
- Falls die Skalierungstyp-Variable des Shapes auf *ZSCALE_NOT_SET* gesetzt ist, wird der Skalierungsfaktor der Z-Achse je nach Geometrietyp unterschiedlich berechnet.
  - Geometrien vom Typ *Model::HORIZONTAL* werden entsprechend der Y-Skalierung und Geometrien vom Typ *Model::VERTICAL* entsprechend der X-Skalierung in der Z-Achse skaliert.
  - Geometrien vom Typ *Model::HORIZONTAL_NDH* und *Model::VERTICAL_NDH* werden nicht auf der Z-Achse skaliert.
  - Alle anderen Geometrien werden entsprechend dem Mittelwert der beiden X- und Y-Skalierungsfaktoren auf der Z-Achse skaliert.
  - Ansonsten wird die Skalierung der Z-Achse entsprechend der *zScaleType*-Variable gesetzt.
- Aus den Skalierungsfaktoren wird eine Skalierungsmatrix erstellt. Diese wird mit der Shape-Transformationsmatrix multipliziert und das Ergebnis in die Shape-Transformationsmatrix zurückgeschrieben.

- Die Vorgehensweise für Modelle vom Typ *TEXTURED* ist ähnlich, nur dass in diesem Fall die Texturkoordinaten nicht skaliert werden und nicht das zufällig ausgewählte Material, sondern das aus der VRML-Datei ausgelesene Material zugewiesen wird.

Dächer Die ***CreateRoof()***-Methode erzeugt eine Dachgeometrie zu einem Gebäudegrundriss.

- Hierbei wird zunächst die Gebäude- und die Dachhöhe ausgelesen.
- Die Anzahl an Kanten des Grundrisses an Straßen wird gezählt.
- Ist diese Anzahl gleich 1, die Anzahl der Grundrisseckpunkte gleich 4 und der Grundriss rechteckig, so wird eine Giebeldach-Geometrie mit der Methode *CreateGableRoof()* (siehe unten) erstellt. Ansonsten wird mit der Methode *CreateMansardRoof()* (siehe unten) eine Mansarddach-Geomtrie erzeugt.
- Die Geometrie wird ggf. wieder in das Koordinatensystem konvertiert, in dem die Z-Achse die Hochachse ist (siehe Anhang A). Ein Material vom Typ *MAT_ROOF* wird zugewiesen und die Geometrie in einem Geometrieknoten als Core hinzugefügt. Dieser Knoten wird schließlich zurückgegeben.

Nachfolgend wird die ***CreateGableRoof()***-Methode beschrieben.

- Zu Beginn wird geprüft, ob der Grundriss aus genau vier Eckpunkten besteht. Wenn dies nicht der Fall ist, dann wird kein Dach erstellt.
- Als Startkante des Grundrisses zur Dacherzeugung wird die erste Grundrisskante verwendet, die zu einer Straße hin zeigt.
- Für die Dachgiebel kommen Dreiecke und für die Dachschrägen Vierecke als Primitive zum Einsatz (siehe Abbildung 2.22a).
- Dabei werden jeweils zwei Drei- und Vierecke verwendet, es werden also 14 Indizes benötigt $(2 * 3 + 2 * 4)$.
- Nun wird jeweils ein Verschiebungsvektor für die Gebäudehöhe und die Dachhöhe berechnet.
- Die vier Punkte der Grundfläche des Daches werden erstellt (siehe Abbildung 2.22a, Punkte P_1 bis P_4).
- Die beiden Punkte, die an der Spitze des Daches liegen und den Giebel abschließen werden zwischen den Punkten 1 und 4 und 3 und 2 erstellt und mit dem Dachhöhen-Verschiebungsvektor nach oben verschoben (siehe Abbildung 2.22a, Punkte RP_1 und RP_2).
- Die Punkte werden als Vertizes gespeichert. Die Indizes für die beiden Dachschrägen (2×4 Indizes) und die beiden Dachgiebel (2×3 Indizes) werden erstellt.
- Nun werden die Texturkoordinaten generiert. Dazu wird aufgrund der Höhe und Länge des Daches berechnet, wie oft die Textur auf der Geometrie gekachelt werden soll. Die Kachelung in Längsrichtung entspricht der Länge und die Kachelung in Höhenrichtung entspricht der Höhe (in Metern). Die Texturkoordinaten der Giebelflächen werden auf

(0, 0) gesetzt, um eine einfarbige Fläche zu erhalten (da in diesem Fall immer der selbe Punkt der Textur verwendet wird).

- Das eigentliche Geometrie-Objekt wird mithilfe der Vertizes, Indizes und Texturkoordinaten erstellt.
- Die Oberflächennormalen werden mit der OpenSG-Funktion *calcVertexNormals()* erzeugt und das Geometrie-Objekt von der Funktion zurückgegeben.

Nachfolgend wird die ***CreateMansardRoof()***-Methode beschrieben. Diese Dachart wird erzeugt, indem das Polygon der Dachgrundfläche verkleinert und nach oben verschoben wird. Anschließend werden die zugehörigen Flächen erzeugt.

- Zu Beginn wird die Anzahl der Punkte der Dachgrundfläche in n gespeichert. Dies entspricht der Anzahl der Punkte des Gebäudegrundrisses.
- Die Dachgrundfläche (bestehend aus den Punkten P_1 bis P_n, siehe Abbildung 2.22b) wird mit der Methode *scalePolygon2D()* um den Wert des Methodenparameters *offset* verkleinert. Je nach Grundflächengeometrie und Größe des Offset-Wertes kann es passieren, dass eine Skalierung der Grundfläche nicht möglich ist. In diesem Fall liefert die Funktion *scalePolygon2D()* weniger Punkte zurück, denn es würde ein *nicht-einfaches* Polygon entstehen (d.h. Kanten würden sich schneiden). Der Offset-Wert wird nun solange halbiert, bis die Grundflächensklalierung möglich ist.
- Die verkleinerte Dachgrundfläche (bestehend aus den Punkten RP_1 bis RP_n, siehe Abbildung 2.22b) bildet die obere Dachfläche. Diese wird nun durch die Hilfs-Methode *Triangulate::Process()* in Dreiecke zerlegt (trianguliert).
- Die Geometrie wird aus Vierecken für die Seitenflächen und Dreiecken für die obere Dachfläche bestehen. Dabei werden $4 \times n$ Indizes für die Vierecke und eine entsprechende Anzahl an Indizes für die durch die Triangulierung entstandenen Dreiecke benötigt.
- Die Verschiebungsvektoren für die Punkte der Dachgrundfläche und der oberen Dachfläche werden berechnet und jeweils vier Punkte für die Dachseitenflächen als Vertizes hinzugefügt. Für jede Fläche gilt folgende Punktreihenfolge: linker, unterer Punkt $\rightarrow$ rechter, unterer Punkt $\rightarrow$ rechter, oberer Punkt $\rightarrow$ linker, oberer Punkt.
- Die Indizes für die Dachseitenflächen werden erstellt.
- Die Normalen für die Seitenflächen werden berechnet, indem aus jeder Fläche ein OpenSG-*Plane*-Objekt gebildet wird und dessen Methode *getNormal()* die Normale der Fläche zurück liefert. Diese Normale wird dann für jedes der vier Vertizes der Seitenfläche genutzt.
- Die Texturkoordinaten werden für die Seitenflächen genauso generiert wie beim Giebeldach bei den Dachschrägen.
- Nun wird die obere Dachfläche erzeugt. Dazu werden die Punkte der in Dreiecke zerlegten oberen Dachfläche als Vertizes hinzugefügt. Für jedes dieser Dreiecke werden drei Indizes erzeugt und die Normale zeigt dabei nach oben (in positiver Y-Richtung). Die Texturkoordinaten werden so gesetzt, dass auf einer Fläche von $1\ m \times 1\ m$ die Textur einmal gekachelt wird.
- Das Geometrie-Objekt wird erzeugt und von der Funktion zurückgegeben.

Kapitel 3

Anwendungen

3.1 Facade Editor

3.1.1 Einführung

Der Fassadeneditor ermöglicht das grafische Erstellen und Editieren von Regeldateien mit sofortigem optischen Feedback. Ein 3D-Fenster zeigt ein Fassadenmodell, das aufgrund der erstellten Regeln in Echtzeit generiert wird. Die Auswirkungen von Regeländerungen werden durch Neugenerierung des Fassadenmodells sofort visualisiert.

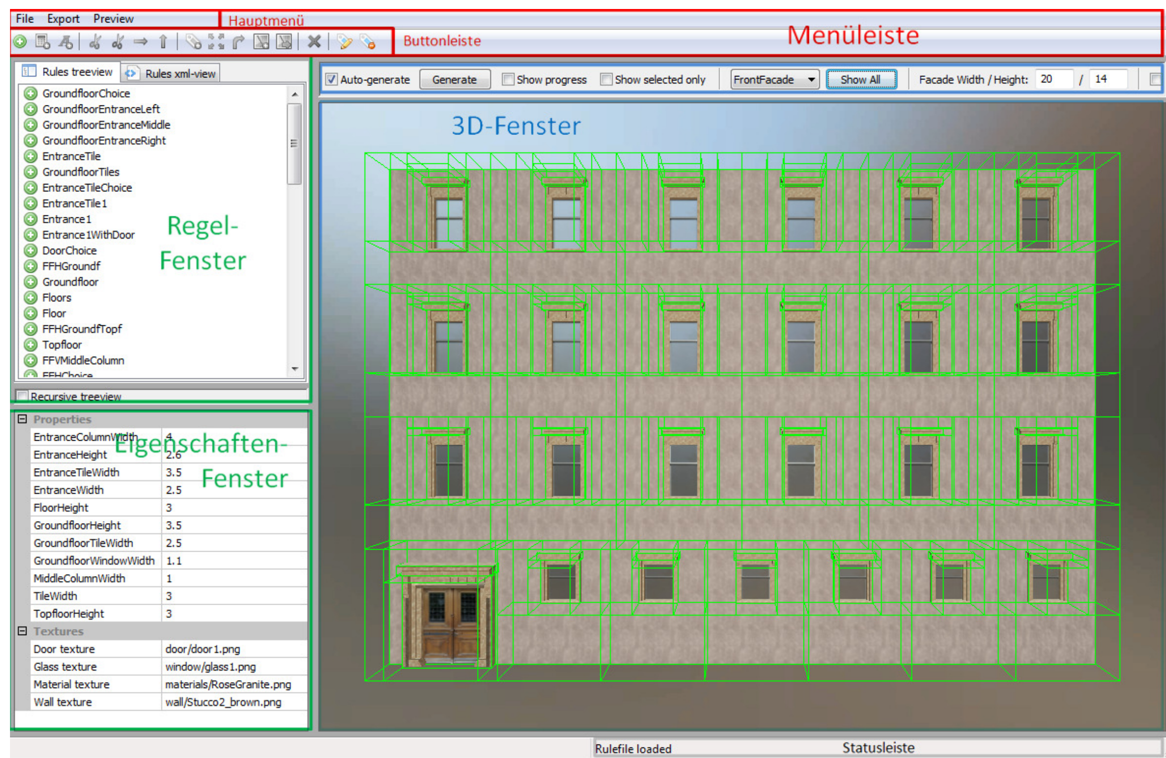


Abbildung 3.1: Das Facade Editor-Programmlayout

3.1.2 Bedienung

3.1.2.1 Die Oberfläche

Die Oberfläche des Fassadeneditors gliedert sich in drei Hauptbestandteile. Die Menüleiste mit dem Hauptmenü und einer Buttonleiste befindet sich am oberen Fensterrand. Am linken Fensterrand ist das Regel- und das Eigenschaftenfenster zu finden. Im Regelfenster werden die Regeln angezeigt und editiert. Am unteren Rand des Regelfensters befinden sich Optionen zur Darstellung der Regeln. Im Eigenschaftenfenster werden Eigenschaften angezeigt. Der restliche Teil des Programmfensters wird vom 3D-Fenster ausgefüllt. Hier wird das Fassadenmodell dargestellt und am oberen Rand des Fenster gibt es Einstellmöglichkeiten für die Darstellung. Am unteren rechten Rand der Anwendung befindet sich noch eine Statusleiste mit Hinweisen über die zuletzt durchgeführte Aktion. Die Bestandteile der Anwendung sind in Abbildung 3.1 verdeutlicht.

3.1.2.2 Hauptmenü

Das Hauptmenü enthält die zwei Untermenüs *File* und *Export*. Im *File*-Menü gibt es folgende Menüpunkte:

- *Create new rulefile*: Hier wird das Programm in den Ausgangszustand zurückversetzt, d.h. es werden alle bisher erstellten Regeln gelöscht.
- *Load rulefile*: Löscht bestehende Regeln und lädt eine Regeldatei.
- *Add rulefile*: Lädt Regeln aus einer Regeldatei und fügt sie den bestehenden Regeln hinzu. Existiert eine Regel bereits, so wird diese beim Laden nicht überschrieben.
- *Save rulefile*: Speichert alle erstellten Regeln in eine Regeldatei.
- *Save multiple rulefiles*: Speichert alle erstellten Regeln in den zugehörigen Regeldateien. Jede Regel gehört zu einer Regelgruppe. Im Editor erstellte Regeln gehören immer der Gruppe *StartRules* an. Wurden Regeln mit *Add rulefile* hinzugefügt und verändert, so werden diese in einer separaten Regeldatei mit dem entsprechenden Regelgruppennamen gespeichert.
- *Load properties from city data file*: Lädt Eigenschaften aus einer Szenendatei.
- *Save properties to file*: Speichert die erstellten Eigenschaften in einer XML-Datei.
- *Save properties into city data file*: Speichert die erstellten Regeln in eine bestehende Szenendatei.

Im *Export*-Menü gibt es folgende Menüpunkte:

- *Export*: Exportiert die in der 3D-Ansicht dargestellte Fassade als OpenSG-Datei in das Verzeichnis *scenes/*.
- *Export as*: Exportiert die in der 3D-Ansicht dargestellte Fassade in ein von OpenSG unterstütztes Format. Das Format wird durch die Dateiendung bestimmt. Möglich sind dabei die folgenden Endungen:
  - *.osb*: OpenSG-Binärformat

  - *.osg*: OpenSG-Standardformat
  - *.wrl*: VRML (Mit einfachen Materialien, allerdings müssen alle Texturen im selben Verzeichnis wie die .wrl-Datei liegen. Siehe auch Kapitel 3.3)

Das *Preview*-Menü verfügt über folgende Menüpunkte:

- *Preview sample scene*: Generiert auf Grundlage der erstellten Regeln eine Vorschau-Stadtszene. Die Stadtszene ist in der Datei *Resources/DefaultPreviewScene.xml* definiert.
- *Switch back facade view*: Verwirft die erstellte Vorschau-Stadtszene und wechselt die 3D-Ansicht wieder auf die aktuelle Fassade.

3.1.2.3 Werkzeugleiste

Die Werkzeugleiste bietet einen schnellen Zugriff auf die wichtigsten und am häufigsten genutzten Funktionen, um Regeln zu erstellen und zu bearbeiten. Im Folgenden werden die einzelnen Buttons beschrieben:

- Erstellt eine neue Regel.
  Im erscheinenden Dialog gibt man einen Symbolnamen ein.
- Erstellt eine neue Regelwahrscheinlichkeitsverzweigung.
  Die Wahrscheinlichkeiten bereits bestehender Verzweigungen werden so angepasst, dass die Wahrscheinlichkeiten in der Summe 1 ergeben.
- Erstellt ein neues Symbol.
  Das Symbol wird unter den bereits bestehenden Symbolen eingefügt.
- Definiert eine horizontale Unterteilungsregel.
- Definiert eine vertikale Unterteilungsregel.
- Definiert eine horizontale Wiederholungsregel.
- Definiert eine vertikale Wiederholungsregel.

  Anmerkung: Beim Ändern des Regeltyps werden die bisher erstellte Symbole dieser Regel gelöscht.
- Fügt einem Symbol eine Eigenschaftsvariable zu.
  Im erscheinenden Dialog wählt man eine Eigenschaftsvariable aus.
- Fügt einem Symbol eine Skalierung hinzu.
  Im erscheinenden Dialog gibt man einen Skalierungsvektor an, der folgende Form hat: *Breite Höhe Tiefe*, z.B. steht *2.0 10.0 1.0* für eine Verdopplung der Breite und einer Verzehnfachung der Höhe.
- Fügt einem Symbol einen Parameter hinzu, der die Skalierung in Tiefenrichtung festlegt.
  Im erscheinenden Dialog wählt man eine Z-Skalierung aus, wobei *None* keine Skalierung, *Right* eine Skalierung relativ zur Breite, *Up* eine Skalierung relativ zur Höhe und *Average* eine Skalierung relativ zum Mittelwert der Höhe und der Breite des Symbols bewirkt.

- Fügt einem Symbol eine Verschiebung hinzu.
Im erscheinenden Dialog gibt man einen Verschiebungsvektor an, der folgende Form hat: *Rechts Oben Vorn*, z.B. steht *0.0 0.5 0.1* für eine Verschiebung um 0.5 m nach oben und 0.1 m nach vorne (von der Fassade weg).

- Fügt einem Symbol eine Textur hinzu.
Es wird nicht nur eine Textur zugewiesen sondern eigentlich ein Material mit entsprechendem Shader, je nachdem in welchem Unterordner von *textures/* sich die Textur befindet (siehe Tabelle 2.4).

- Fügt einem Symbol eine Texturkoordinatenskalierung hinzu.
Im erscheinenden Dialog gibt man einen Skalierungsvektor an, der folgende Form hat: *Horizontal Vertikal*, z.B. steht *0.5 0.5* für eine halb so große Darstellung der Textur, sie wird doppelt so oft „gekachelt“.

- Löscht das gerade ausgewählte Objekt im Regelfenster.

- Erstellt eine neue Eigenschaftsvariable.
In den erscheinenden Dialogen wird erst die neue Eigenschaftsvariable benannt und dann ein numerischer Wert zugewiesen.

- Löscht die gerade ausgewählte Eigenschaftsvariable.

3.1.2.4 Regelfenster

Im Regelfenster werden alle erstellten Regeln angezeigt. Hinter einer Regel verbirgt sich eine Baumstruktur mit weiteren Eigenschaften der Regel. Eine Regel und auch weitere Elemente des Fensters können über einen Doppelklick „aufgeklappt“ werden. Ein weiterer Doppelklick „klappt“ sie wieder „zu“. Dieser Mechanismus kann auch über die kleinen *Plus*- und *Minus*-Zeichen links neben den Elementen ausgelöst werden.

Man kann Elemente per Mausklick auswählen. Je nach ausgewähltem Element stehen bestimmte Werkzeuge zur Verfügung (siehe 3.1.2.2). Ist ein Element ausgewählt und klickt man erneut auf die Beschriftung des Elements, so kann der jeweilige Name (z.B. bei Symbolen) oder die jeweilige Eigenschaft (z.B. die Veränderung eines Skalierungsvektors) direkt geändert werden.

Auf bestimmte Elemente kann mit den folgenden Auswirkungen rechtsgeklickt werden:

Wahrscheinlichkeiten Wählt einen Regelzweig zur Darstellung aus.

Symbole Öffnet einen Dialog zur Auswahl eines 3D-Modells oder eines bestehenden Symbols.

Texturen Öffnet einen Dialog zur Auswahl einer Textur.

Z-Skalierung Öffnet einen Dialog zur Auswahl der Z-Skalierung.

Ist ein Symbol ausgewählt, so gelangt man durch Drücken der Leertaste zur entsprechenden Regel. Die neue Regel wird „aufgeklappt“ und die alte Regel wird „zugeklappt“.

Ist ein Symbol ausgewählt und drückt man die Enter-Taste, so wird dieses Symbol als Startsymbol für die 3D-Ansicht genutzt, d.h. man sieht in der 3D-Ansicht nur noch die von diesem

Symbol abgeleiteten Geometrien. Selektiert man in der Menüleiste der 3D-Ansicht wieder eine Fassade, so wird die Ansicht wieder zurückgesetzt.

Des Weiteren kann die Ansicht der Regel durch Aktivieren der Option *Recursive treeview* in eine Baumansicht umgewandelt werden. In dieser Ansicht ist der Aufbau der Fassade sehr gut sichtbar, allerdings sind hier nicht alle Funktionen zum Erstellen und Ändern von Regeln implementiert. Man kann die Option jederzeit durch erneutes Anklicken der Checkbox deaktivieren.

Die Regeln werden alphabetisch sortiert, wenn die Option *Alphabetically* aktiviert ist.

Der Karteireiter *Rules xml-view* bietet eine Vorschau auf die durch die Regeln definierte Regel-Datei.

3.1.2.5 Eigenschaftenfenster

Im Eigenschaftenfenster werden über die entsprechenden Buttons in der Werkzeugleiste erstellte Eigenschaften tabellarisch angezeigt. Man kann den Wert einer Eigenschaft ändern, indem man in das Textfeld klickt und einen neuen Wert eingibt. Die gerade selektierte Eigenschaft kann über den *Eigenschaften löschen*-Button in der Werkzeugleiste gelöscht werden.

Außerdem werden die Basistexturen der Fassade angezeigt. Es werden Texturen für die Fassadenwand (*Wall texture*), für die Türen (*Door texture*), für Fenster (*Glass texture*) und für sonstige 3D-Geometrien (*Material texture*) zufällig festgelegt. Ein Rechtsklick auf den Texturnamen öffnet ein Dialogfenster zur Auswahl einer anderen Textur.

3.1.2.6 3D-Fenster

Am oberen Rand des 3D-Fensters befinden sich einige Einstellmöglichkeiten:

- Checkbox-*Auto-generate*: Wenn diese Option angekreuzt ist, so wird die 3D-Ansicht bei jeder Regeländerung aktualisiert.
- Button-*Generate*: Generiert die Fassade neu.
- Checkbox-*Show progress*: Zeigt den Prozess der Fassadengenerierung Schritt für Schritt. Vorsicht: Dieser Vorgang kann je nach Komplexität der Fassade und Leistungsfähigkeit des Computers einige Zeit dauern.
- Checkbox-*Show selected only*: Zeigt nur die Begrenzungsboxen des gerade ausgewählten Symbols. Ansonsten werden immer alle Begrenzungsboxen angezeigt.
- *Fassadenauswahl*: Ermöglicht die Auswahl der gerade aktiven Fassade oder des ganzen Gebäudes (Unterpunkt *Building*).
- Button-*Show All*: Setzt die Kamera so, dass die gesamte Fassade im Bild ist.
- *Facade Width / Height*: Hier wird die Breite und Höhe der Fassade eingestellt.
- Checkbox-*Merge Meshes*: Verbindet die der Fassade zugrunde liegenden 3D-Objekte. Dieser Vorgang kann je nach Komplexität der Fassade und Leistungsfähigkeit des Computers einige Zeit dauern, beschleunigt allerdings die 3D-Ansicht bei komplexen Fassaden.

Die Navigation in der 3D-Ansicht erfolgt mit der Maus und verhält sich wie mit einer Standard-OpenSG-Anwendung (da der *SimpleSceneManager* zum Einsatz kommt). Mit gedrückter linker Maustaste dreht man die Ansicht. Mit gedrückter rechter Maustaste zoomt man. Ein Klick auf ein Objekt setzt den Navigationsmittelpunkt an diese Stelle im Raum.

3.1.3 Implementierung

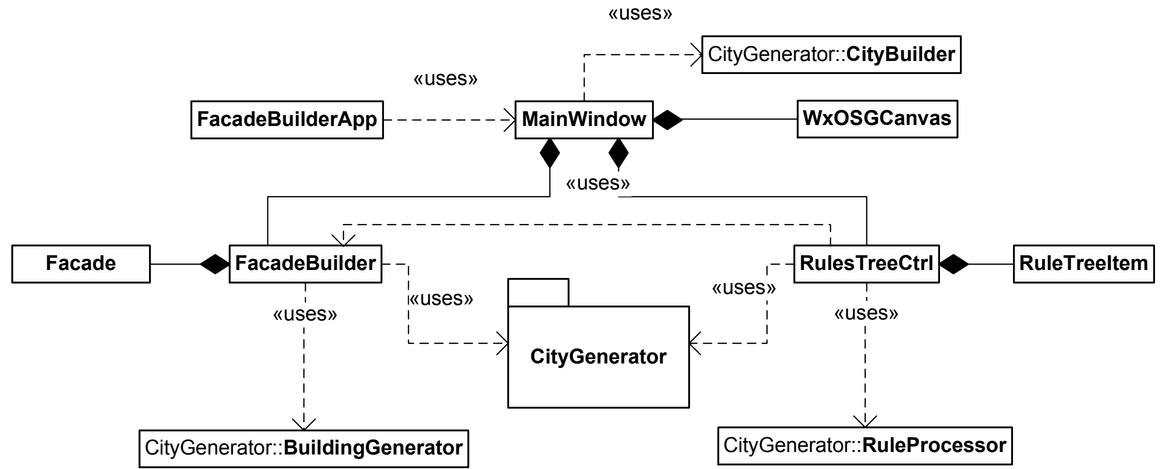


Abbildung 3.2: Übersicht aller FacadeBuilder-Klassen in einem UML-Diagramm

Der Fassadeneditor ist ein grafisches Werkzeug und benutzt die frei verfügbare GUI-Bibliothek *WxWidgets*[wxWidgets 08]. Die Hauptanwendung *FacadeBuilderApp* ist von der WxWidgets-Klasse *wxApp* abgeleitet und erstellt in der *OnInit()*-Methode nur eine Instanz der *MainWindow()*-Klasse. Die Oberfläche wurde mit dem Programm *DialogBlocks*[Anthemion 09] gestaltet. Ein DialogBlocks-Projekt besteht aus einer speziellen DialogBlocks-Projektdatei und einem normalen Visual Studio-Projekt. Dieses wurde in die bestehende Projektmappe aufgenommen und eine parallele Bearbeitung der Programmoberfläche in DialogBlocks und der Programmlogik in Visual Studio ist dadurch problemlos möglich. DialogBlocks kennzeichnet Quellcode-Abschnitte mit speziellen Kommentaren wie z.B. Folgendem:

```
////@begin MainWindow member variables
    wxStatusBar* statusBar;
    wxBoxSizer* leftVerticalBoxSizer;
////@end MainWindow member variables

// ab hier könnten eigene Variablen für die Klasse MainWindow folgen
// ...
```

Für das Hauptmenü, die Werkzeugleiste und sonstige Buttons und Auswahlfelder wurden Standard-GUI-Elemente verwendet, die hier nicht weiter erläutert werden. Für das Regelfenster wurde die Klasse *wxTreeCtrl* genutzt. Diese Klasse bietet eine baumartige, navigierbare Struktur an. Das Eigenschaftenfenster benutzt die Klasse *wxPropertyGrid*, die einer Eigenschaft (auf der linken Seite) einen Wert zuweist (auf der rechten Seite). *wxPropertyGrid* ist erst seit Version 2.9 von wxWidgets dessen Bestandteil.[1] Das 3D-Fenster wird durch die Klas-

[1] Für ältere wxWidgets-Versionen kann es unter folgendem Link heruntergeladen werden: <http://wxpropgrid.sourceforge.net>

se *WxOSGCanvas* realisiert. Diese ist von *wxGLCanvas* abgeleitet und erzeugt ein OpenSG-*PassiveWindow* und einen -*SimpleSceneManager*. Die *OnMouseEvent()*-Methode leitet Mauseingaben an den Szenenmanager weiter, um mit der Maus in der Szene navigieren zu können. Die alles umfassende *MainWindow*-Klasse verfügt hauptsächlich über Methoden, die durch ausgelöste Ereignisse (z.B. Mausklicks auf Buttons) aufgerufen werden (siehe UML-Diagramm in Abbildung 3.3). Diese sind am vorangestellten „On“ erkennbar, z.B. *OnMenuLoadRulefileClick()*. Die Methode *MyInit()* wird beim Starten des Programms aufgerufen und initialisiert den *FacadeBuilder*, den *CityBuilder* und die *RulesTreeCtrl*. Außerdem wird hier ein OpenSG-*SkyBackground* erstellt, der an einen OpenSG-*Viewport* gebunden wird, der wiederum die Standardkamera des *SimpleSceneManagers* zugewiesen bekommt:

```
SkyBackgroundPtr skyBackground = SkyBackground::create();
beginEditCP(skyBackground);
skyBackground->setFrontTexture(HelperFunctions::CreateTexture(string(
    resourcePath + "textures/skybox_south.png")));
skyBackground->setBackTexture(HelperFunctions::CreateTexture(string(
    resourcePath + "textures/skybox_north.png")));
skyBackground->setLeftTexture(HelperFunctions::CreateTexture(string(
    resourcePath + "textures/skybox_east.png")));
skyBackground->setRightTexture(HelperFunctions::CreateTexture(string(
    resourcePath + "textures/skybox_west.png")));
skyBackground->setTopTexture(HelperFunctions::CreateTexture(string(
    resourcePath + "textures/skybox_up.png")));
skyBackground->setBottomTexture(HelperFunctions::CreateTexture(string(
    resourcePath + "textures/skybox_down.png")));
endEditCP(skyBackground);

ViewportPtr viewport = simpleSceneManager->getWindow()->getPort(0);
beginEditCP(viewport);
viewport->setCamera(simpleSceneManager->getCamera());
viewport->setBackground(skyBackground);
endEditCP(viewport);
```

Die sechs Texturen werden zu einem Würfel zusammengesetzt, der die gesamte Szene umgibt und man somit die Illusion einer Umgebung erhält. Diese Technik wird auch als Skybox oder Skycube bezeichnet.

Die mit „OnTool“ und mit „OnTreectrlRules“ beginnenden Methoden leiten das Ereignis einfach an die *RulesTreeCtrl*-Klasse weiter.

Der bedeutendste Bestandteil der Anwendung ist die *RulesTreeCtrl*-Klasse, da diese die Verbindung zwischen Programmoberfläche (durch einen Verweis auf das *wxTreeItem*-Objekt) und Regelprozessor (durch einen Verweis auf das *RuleProcessor*-Objekt) schafft. Die Elemente der Baumstruktur sind von der Klasse *wxTreeItemData* abgeleitet und sind vom Typ *RuleTreeItem* (siehe UML-Diagramm in Abbildung 3.4). Ein *RuleTreeItem* verweist auf ein *RuleBase*- und ggf. auf ein *Rule*-Objekt und durch den Typ wird definiert, welchen Regelbestandteil es definiert. Ein Element der Baumstruktur hat außerdem eine Beschriftung. Diese definiert den eigentlichen Inhalt des referenzierten Regelbestandteils. Wird z.B. ein Nachfolgersymbol durch ein Baumelement repräsentiert, so steht die Beschriftung für den Namen des Symbols, ein Skalierungsvektor hat z.B. die Form 2.0 1.0 1.5, eine Textur z.B. *„wall/stucco1.png“*, usw.

Die Klasse *RulesTreeCtrl* besitzt des Weiteren einen Verweis auf die Werkzeugleiste, mit der der Baumansicht neue Elemente hinzugefügt werden können. Je nachdem, welches Element in der Baumansicht ausgewählt wurde, werden die entsprechenden Werkzeug-Buttons durch

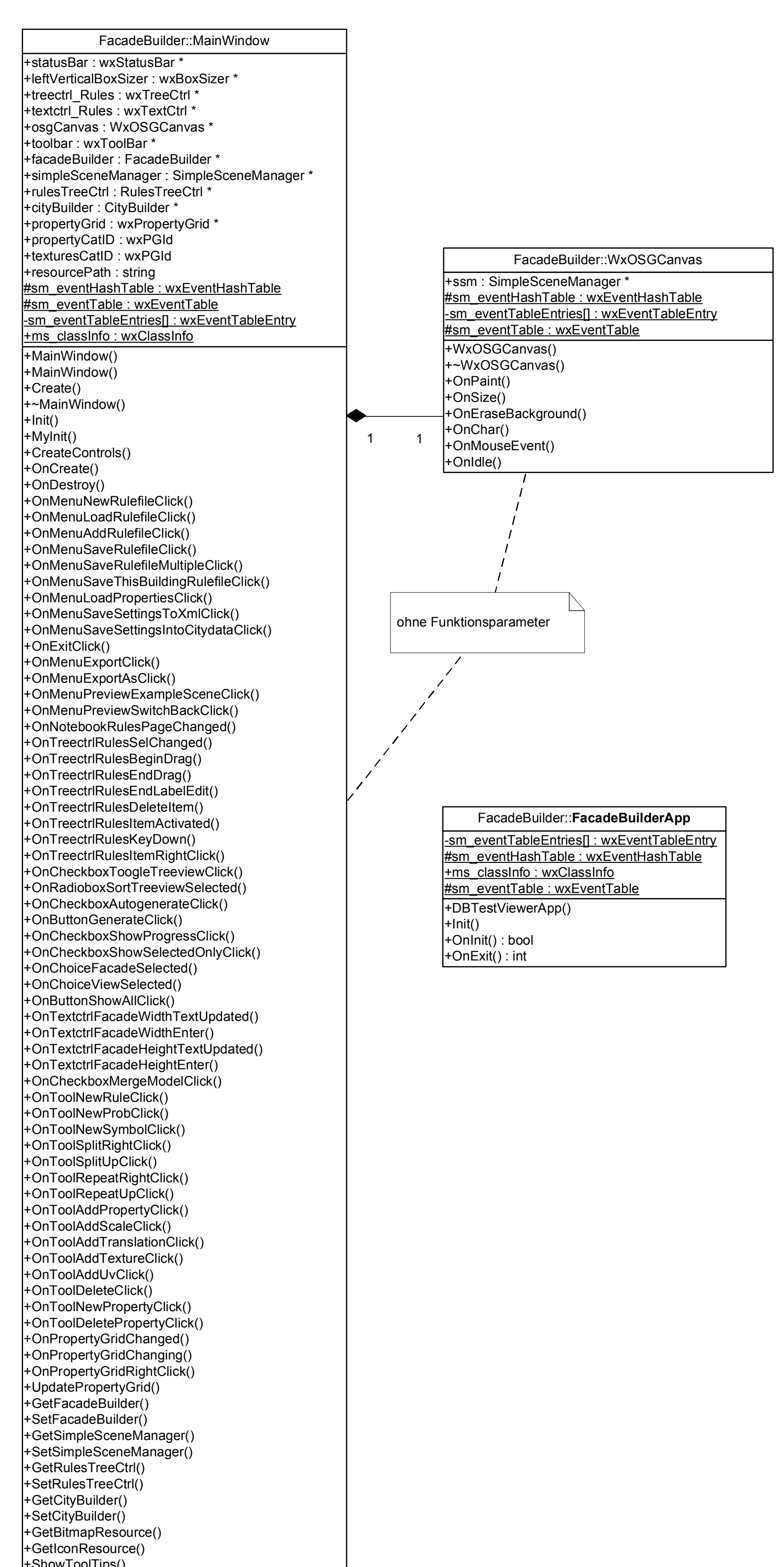


Abbildung 3.3: UML-Diagramm der MainWindow-Klasse und zugehörige Klassen

die Methode *SetToolbar()* aktiviert. Die Ereignisverarbeitung geschieht durch die mit „On“ beginnenden Methoden. Die Methoden führen meist zwei Aktionen aus. Erst führen sie die gewünschten Änderungen auf der Seite des Regelprozessors aus, d.h. es wird z.B. der Name eines Nachfolgersymbols innerhalb einer *Rule*-Struktur geändert. Anschließend wird die Änderung in die Baumansicht übernommen, indem z.B. die Beschriftung des Elements geändert wird. Der komplette Baum kann durch Aufrufen der Methode *RebuildTree()* neu erzeugt werden. Diese Methode ruft ihrerseits die Methode *AppendTreeItemRulebases()*, *AppendTreeItemRulebase()*, *AppendTreeItemRules()* und *AppendTreeItemRule()* auf, um mehrere Ableitungsregeln, eine Ableitungsregel, mehrere Ersetzungsregeln oder eine Ersetzungsregel anzuhängen.

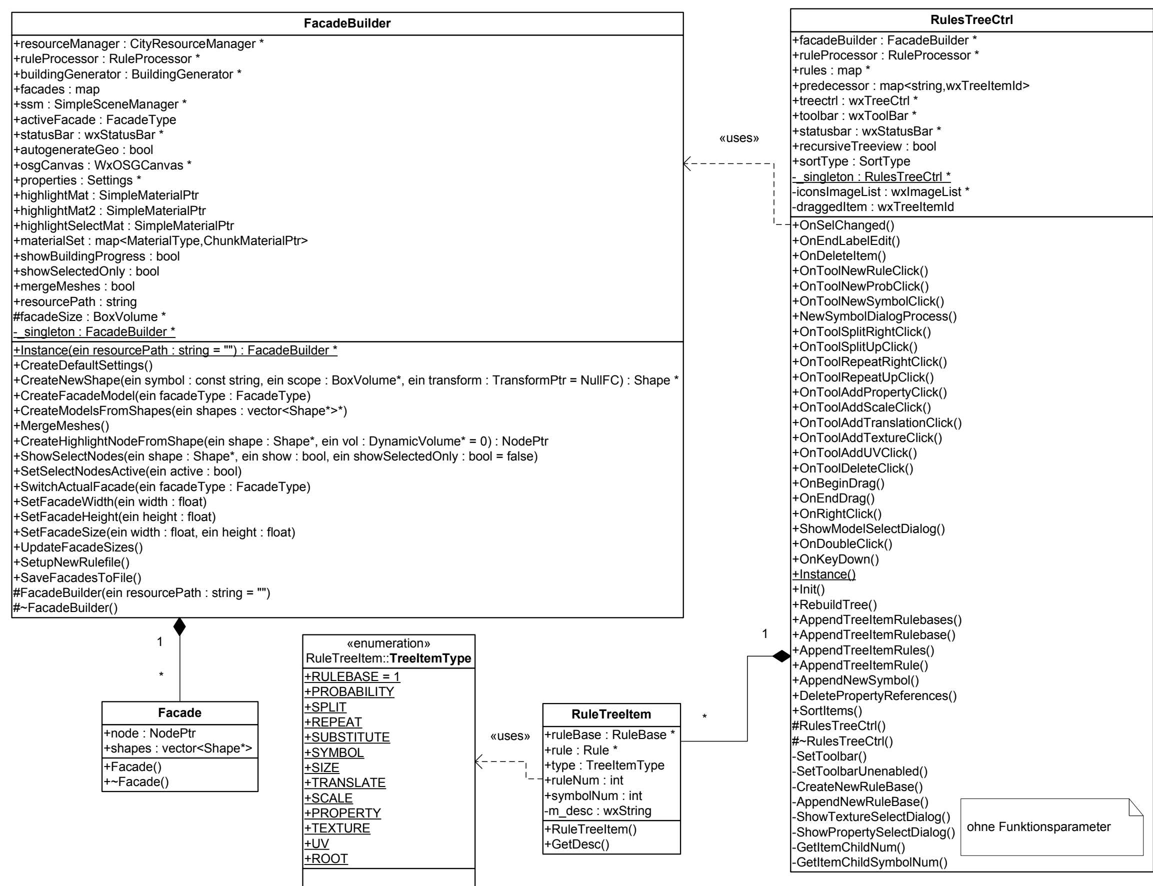


Abbildung 3.4: UML-Diagramm der FacadeBuilder-, RulesTreeCtrl-Klasse und zugehörige Klassen

3.2 CityGeneratorCL

3.2.1 Bedienung

Die Anwendung *CityGeneratorCL* ist eine Kommandozeilenanwendung, die aus einer XML-Szenendatei eine OpenSG-Datei erzeugt. Dazu übergibt man dem Programm den Namen der XML-Szenendatei als Parameter wie folgt:

```
CityGeneratorCL.exe Szenendatei.xml
```

Daraufhin wird eine OpenSG-Binärdatei (mit der Endung *.osb*) mit gleichem Dateinamen erstellt. Um das Ergebnis sofort betrachten zu können, übergibt man den optionalen Parameter *-v* wie folgt:

```
CityGeneratorCL.exe Szenendatei.xml -v
```

Nach der Erstellung der OpenSG-Datei wird diese in einer einfachen OpenSG-Anwendung geladen, in der die Szene betrachtet werden kann. Des weiteren kann die Szene in eine Anwendung geladen werden, die den 3DConnexion Space Navigator als Eingabegerät benutzt. Der Kommandozeilenaufruf sieht dann so aus:

```
CityGeneratorCL.exe Szenendatei.xml -vsn
```

Dabei muss sichergestellt sein, dass ein Space Navigator-Eingabegerät über einen VRPN-Server verfügbar ist, siehe dazu auch [Bilke 07].

Mit der Kommandozeilenoption `-export` kann die Szene als *.wrl*-Datei gespeichert werden, die mit gängigen 3D-Modellierungsprogrammen wieder eingelesen und weiterverarbeitet werden kann (siehe auch Kapitel 3.3):

```
CityGeneratorCL.exe Szenendatei.xml -export
```

3.2.2 Implementierung

Der CityGeneratorCL ruft Methoden der Klasse *CityBuilder* auf. Diese erzeugt aus einem *CityData*-Objekt, das entweder an die Klasse übergeben oder aus einer Datei geladen werden kann, das 3D-Modell der Stadtszene. Die Klasse nutzt dafür die Klassen *CityResourceManager*, *StreetGenerator*, *RuleProcessor* und *BuildingGenerator* (siehe UML-Diagramm in Abbildung 3.5).

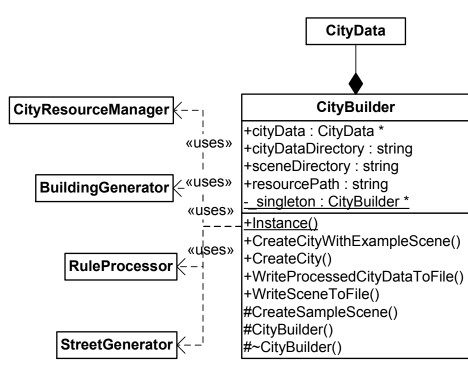


Abbildung 3.5: UML-Diagramm der CityBuilder-Klasse

Die beiden mit *CreateCityWithExampleScene*-beginnenden Methoden erzeugen das 3D-Modell einer Stadt. Hierbei wird erst der *Ressourcenmanager* initialisiert, d.h. alle Ressourcen werden geladen. Dann wird, falls kein *CityData*-Objekt, sondern ein Dateiname, übergeben wurde, das Objekt aus der Datei geladen. Der Regelprozessor wird initialisiert und die benötigten Regeldateien geladen. Falls der Methodenparameter `createStreets` mit `true` übergeben wurde, erzeugt der *StreetGenerator* die Straßengeometrie. Der *BuildingGenerator* wird initialisiert und erzeugt die Gebäudemodelle. Anschließend wird ein OpenSG-*NodePtr* als Wurzelknoten für die Szene erstellt. An diesen wird eine Startszene, bestehend aus einer Lichtquelle und einer Bodenebene, durch die Funktion *CreateSampleScene()* angehängt. Schließlich werden die Gebäudegeometrien sowie eventuell die Straßengeometrie dem Wurzelknoten hinzugefügt und der Wurzelknoten von der Methode zurückgegeben. Die Szene kann durch

WriteSceneToFile()-Methode im OpenSG-Format *.osb* in einer Datei gespeichert werden.

3.3 Exportmöglichkeiten der generierten Modelle

3.3.1 Einführung

Die erstellten OpenSG-Szenendateien lassen sich nur in OpenSG-Anwendungen weiter benutzen, dabei kommen allerdings nur selber programmierte Programme sowie der *Virtual Reality Editor-VRED* in Frage. Weitere Nutzungsmöglichkeiten für die erstellten Häusermodelle werden erst durch ein allgemein gültiges Austauschformat möglich. So können die Modelle in andere 3D-Anwendungen importiert und weiterverarbeitet werden. Als Austauschformat wurde das VRML2-Format mit der Dateiendung *.wrl* gewählt, da OpenSG bereits einen Exporter dafür implementiert.

Da der VRML2-Standard bereits 1997 verabschiedet wurde enthält er keine Möglichkeit zur Einbettung komplexer Materialdefinitionen und unterstützt auch keine Shader. Das Exportieren der erstellten Häusermodelle in das VRML2-Format hat somit zur Folge, dass sämtliche Materialien und die zugehörigen Texturen fehlen. Aus diesem Grund wurde der Resource-Manager modifiziert, um zu jedem komplexen Shader-Material ein entsprechendes einfaches Material zu generieren, dass sich in das VRML2-Format exportieren lässt. Außerdem ist zu beachten, dass die Modellgeometrien zusammengefasst wurden (mit der Einstellung *mergeGeometries* auf *1* in einem Variablenblock), ansonsten werden die Transformationen der einzelnen Gebäudeteile nicht korrekt exportiert. Die Einstellung *LODLevel* sollte auf *1* gesetzt sein, um nur die prozedural erstellten Fassadenmodelle zu exportieren. Wenn beide LOD-Stufen erstellt werden, so verdecken beim späteren Betrachten des Modells in einem 3D-Modellierungsprogramm die flachen Fassaden die prozedural erstellten Fassaden.

3.3.2 Implementierung

In der Methode *CreateMaterials()* des *CityResourceManagers* wird für jedes Material ein entsprechendes OpenSG-*SimpleTexturedMaterial* mit der diffusen Farbtextur erstellt:

```
SimpleTexturedMaterialPtr simpleMat=SimpleTexturedMaterial::create();
beginEditCP(simpleMat);
simpleMat->setAmbient(Color3f(ambientColor.red(),ambientColor.green(),
    ambientColor.blue()));
simpleMat->setDiffuse(Color3f(diffuseColor.red(),diffuseColor.green(),
    diffuseColor.blue()));
simpleMat->setSpecular(Color3f(specularColor.red(), specularColor.green(),
    specularColor.blue()) * specularIntensity);
simpleMat->setShininess(shininess);
simpleMat->setImage(images[dir][i]);
endEditCP(simpleMat);
```

Diese Materialien werden in den STL-Maps `map<string, SimpleTexturedMaterialPtr> simpleMats` und `map<SimpleTexturedMaterialPtr, string> simpleMatNames` gespeichert.

```
simpleMats[textureFilename] = simpleMat;
simpleMatNames[simpleMat] = textureFilename;
```

Der Methode *WriteSceneToFile(bool useSimpleMaterial)* des *CityBuilder* traversiert den Szenengraphen (mit der OpenSG-Funktion *traverse()*), wenn man ihr als Funktionsparameter

true übergibt und ruft dabei auf jedem Knoten des Graphen die Methode *SwitchMaterialType()* auf.

```
if (useSimpleMaterial)
  traverse(cityData->root, osgTypedFunctionFunctor1CPtrRef<Action::ResultE,
      NodePtr>(SwitchMaterialType));
```

Der *SwitchMaterialType()*-Methode wird ein *NodePtr* übergeben. Zuerst wird überprüft, ob der Knoten als *Core* einen *MaterialGroupPtr* besitzt.

```
MaterialGroupPtr matGroup = MaterialGroupPtr::dcast(node->getCore());
if (matGroup != NullFC)
{
```

Ist dies der Fall, so wird aus dem *MaterialGroupPtr* das Material extrahiert.

```
  CityResourceManager* crm = CityResourceManager::Instance();
  MaterialPtr mat = matGroup->getMaterial();
```

Nun wird versucht, das Material in einen *ChunkMaterialPtr* und in einen *SimpleTexturedMaterialPtr* umzuwandeln.

```
  ChunkMaterialPtr chunkMat = ChunkMaterialPtr::dcast(mat);
  SimpleTexturedMaterialPtr simpleMat =SimpleTexturedMaterialPtr::dcast(mat);
```

Handelt es sich um ein *ChunkMaterial*, so wird das Material des *Cores* des Knotens auf das entsprechende *SimpleTexturedMaterial* umgesetzt. Dazu wird der Name des *ChunkMaterials*, der der diffusen Farbtextur entspricht, genutzt, um das entsprechende *SimpleTexturedMaterial* aus der STL-Map des ResourceManagers zu erhalten.

```
  if (chunkMat != NullFC)
  {
    beginEditCP(matGroup);
    string matName = crm->materialNames[chunkMat];
    matGroup->setMaterial(crm->simpleMats[matName]);
    endEditCP(matGroup);
  }
```

Handelt es sich jedoch um ein *SimpleTexturedMaterial* so werden die vorhergehenden Schritte ähnlich ausgeführt, nur das aufgrund des Namens des *SimpleTexturedMaterials* das entsprechende *ChunkMaterial* ausgewählt und als Material des *Cores* des Knotens gesetzt wird.

```
  else if (simpleMat != NullFC)
  {
    beginEditCP(matGroup);
    matGroup->setMaterial(crm->materials[crm->simpleMatNames[simpleMat]]);
    endEditCP(matGroup);
  }
```

3.3.3 Ergebnis

Cinema 4D Die VRML-Datei wird in Cinema 4D über *Menü→ Öffnen* importiert. Die Geometrie wird korrekt geladen, allerdings müssen die Texturpfade aller Materialien angepasst werden. Die Pfade werden beispielsweise in folgender Form exportiert:

..\Resources\textures\materials\PolarGranite.png

Nun sind einige Fehler in Cinema 4D zu beachten:

- 64-Bit-Versionen von Cinema 4D können keine PNG-Bilddateien lesen. Die Ursache dafür scheint die *Quicktime*[2]-Runtime zu sein, die Cinema 4D für PNG-Dateien nutzt und die noch nicht 64-Bit-kompatibel ist. Da hauptsächlich PNG-Dateien verwendete wurden, muss die 32-Bit-Version von Cinema 4D genutzt werden.
- Cinema 4D unterstützt zwar relative Pfade zur Modelldatei, allerdings nicht die Zeichenfolge *../*, um im Ordnerbaum eine Ebene höher zu gelangen.

Ein möglicher Weg wäre z.B. die VRML-Datei in den Projekt-Wurzelordner zu kopieren und die Datei in Cinema 4D dort zu öffnen. In den Texturpfaden wird der Anfang *..* entfernt. Ein Texturpfad sieht dann z.B. so aus:

Resources\textures \materials \PolarGranite.png

Nun können die Häusermodelle weiter bearbeitet werden oder in verbesserten Darstellungsmethoden gerendert werden (siehe Abbildung 3.6 Oben).

3ds max 3ds max kann ebenso VRML2-Dateien importieren. Auch hier müssen die Texturpfade wieder angepasst werden. Dies geht jedoch recht einfach, indem man die Szene rendert. Daraufhin erscheint ein Fenster, indem darauf hingewiesen wird, das nicht alle Texturen gefunden werden konnten. Man klickt auf den rechten Button *Browse* woraufhin ein weiteres Fenster mit Texturpfaden erscheint. Nun fügt man den Pfad des Ordners *Projekt-Ordner\Resources* hinzu. Ab nun werden die Texturen der exportierten Modelle immer korrekt gefunden und das Modell kann in 3ds max weiterverarbeitet werden (siehe Abbildung 3.6 Mitte).

Blender Der Import in Blender wurde in Version 2.48 getestet. In älteren Versionen fehlt die VRML-Importfunktion. Der Import erfolgt über Hauptmenü→ *File*→ *Import*→ *X3D & VRML97*. Die Texturpfade werden korrekt geladen, allerdings wird bei allen Texturen die Texturkachelung auf einen Wert gleich der Auflösung der Textur gestellt. Bei einer Texturauflösung von 512×512 Pixeln wird die Textur 512mal gekachelt. Man kann dies manuell für jede Textur über *F6 Texture Buttons*→ *Preview*→ Button *Default Vars* korrigieren. Des Weiteren werden die Oberflächennormalen und einige Texturkoordinaten falsch dargestellt (siehe Abbildung 3.6 Unten).

[2] Qucktime ist eine Multimedia-Architektur von Apple. Zu weiteren Informationen siehe <http://www.apple.com/de/quicktime/>

Abbildung 3.6: In 3D-Modellierungsanwendungen exportierte Häusermodelle

Kapitel 4

Erstellung von Regeln und Ressourcen für Gründerzeithäusermodelle

4.1 Einführung in die Architektur der Gründerzeit

Als Gründerzeit wird eine wirtschaftliche Phase in Deutschland und Österreich zwischen den Anfängen der *Industrialisierung* Anfang der 1840er-Jahre bis zum Börsencrash 1873 bezeichnet. In dieser Zeit kam es zu einem starken wirtschaftlichen Aufschwung, begünstigt durch neue Entwicklungen, wie die Eisenbahn, das Telefon und Elektrizität. Unternehmensgründer aber auch das Bürgertum erlebten in dieser Zeit einen rasanten sozialen Aufstieg. Im Rahmen der Industrialisierung wurden immer mehr Arbeitskräfte in den Unternehmen benötigt, sodass große Teile der Landbevölkerung in die Städte zogen. Dadurch wuchs der Bedarf nach Wohnraum und es wurden ganze Stadtviertel neu geschaffen.

Abbildung 4.1: Typische Gründerzeitarchitektur im Leipziger Waldstraßenviertel, Rechts: Detailaufnahme des Eingangsbereiches

Das zusehends wohlhabende Bürgertum verlangte nach repräsentativen Villen und Stadt-

wohnungen. Meist private Wohnungsbaugesellschaften errichteten vier- bis sechsgeschossige Wohnhäuser in Blockrandbebauung (vorwiegend rechtwinklig angeordnete Häuserblöcke, deren Häuser parallel zur Straße liegen). In den dadurch entstehenden Hinterhöfen wurden meist Quartiere für Arbeiter errichtet.

Der Begriff Gründerzeitstil im Sinne von Architektur steht umgangssprachlich für eine Zeitspanne im *Historismus*. Dieser bezeichnet eine die vorherrschende Stilrichtung in der Architektur zwischen der Mitte des 19. Jahrhunderts bis Anfang des 20. Jahrhunderts. Beim Historismus griff man auf ältere Stilformen aus der klassischen Antike, Romanik, Gotik, Romantik, Barock und Renaissance zurück oder ahmte sie nach, teils auch in Stilmixen.

Die Wohnhäuser wurden jedoch nicht nur für wohlhabende Bürger geschaffen, sondern spiegeln auch in ihrem Aufbau die Klassenhierarchie wider. So war die erste Etage oder die Hochparterre mit ihren hohen Decken und reichen Stuckverzeirungen (siehe Abbildung 4.2 rechts unten) sowohl innen und außen den wohlhabenderen Bürgern vorbehalten. Nach oben hin nahm die Geschosshöhe sowie die soziale Stellung der Bewohner immer weiter ab. Der Hinterhof und die darauf errichteten einfachen Arbeiterquartiere wurden durch eine Durchfahrt mit der Straße an der vorderen Fassade des Gebäudes verbunden. Dadurch war der Haupteingang auch an der Hofseite zu finden. Die Fassaden sind meist symmetrisch aufgebaut.

Typische Gründerzeitstil-Häuser mit prachtvollen Stuckelementen und Verzierungen sind in Leipzig im Waldstraßenviertel zu finden. Die Abbildungen 4.1 und 4.2 zeigen einige Häuser im Waldstraßenviertel.

Abbildung 4.2: Weitere Gründerzeitstilhäuser im Leipziger Waldstraßenviertel, Rechts unten: Nahaufnahme von Stuckverzierungen

4.2 Erstellen von 3D-Geometrien in Cinema 4D

Die 3D-Geometrien, die als Bausteine für die Fassaden eingesetzt werden, wurden mit Cinema 4D Version 10 modelliert. Nachfolgend werden ein paar Hinweise zur Modellierung gegeben, die jedoch in ähnlicher Form auch für andere Modellierungssoftware von Relevanz sein können.

Grundsätzlich kann eine Geometrie frei modelliert werden. Am Ende des Modellierungsprozesses und vor dem Exportierten sollten folgende Bedingungen erfüllt sein:

- Die Geometrie muss als *Polygon-Objekt* vorliegen.
- Die Geometrie darf im Objektbrowser nicht anderen Objekten untergeordnet sein.
- Der lokale Koordinatenursprung des Polygon-Objektes muss sich in der horizontalen Mitte und am vertikalen unteren Ende des Objektes befinden (siehe Abbildung 4.3 links).
- Der lokale Koordinatenursprung wird um die Hochachse (Y-Achse) um 90° mit dem Uhrzeigersinn rotiert (um vom Cinema 4D- in das OpenSG-Koordinatensystem zu transformieren, siehe auch Anhang A).

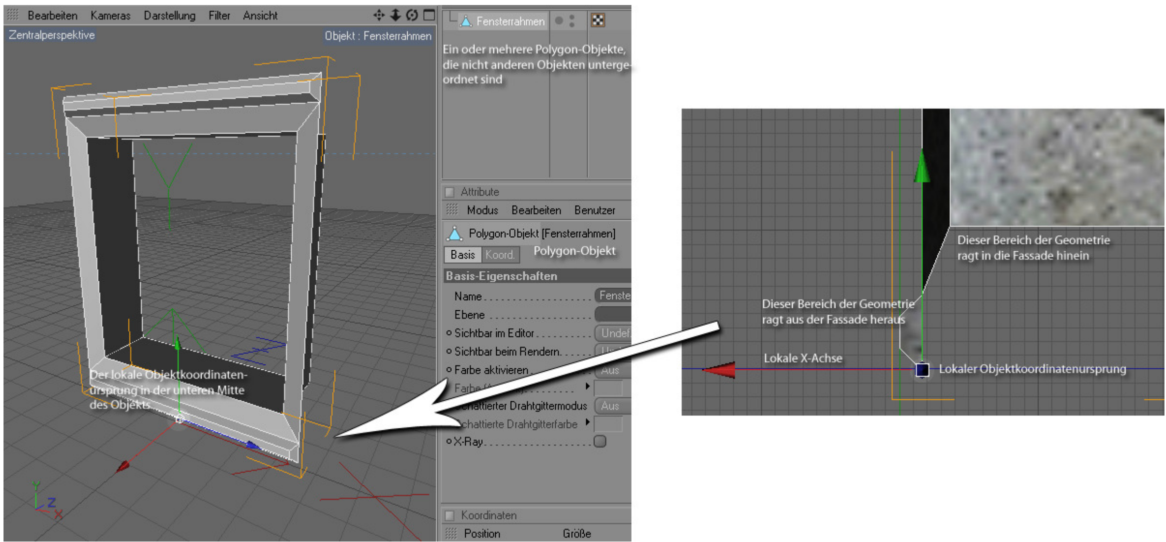


Abbildung 4.3: Ein Fensterrahmen-Modell in Cinema 4D mit seitlicher Detailansicht

Die Geometrie muss sich nicht komplett in einem Polygon-Objekt befinden, sondern kann auf mehrere Polygon-Objekten aufgeteilt sein. Diese werden beim Laden durch den Ressourcenmanager zu einer einzelnen Geometrie zusammengefügt. Die lokalen X-Koordinaten von Punkten der Geometrie (bezüglich des lokalen Modelkoordinatensystems) beeinflussen, welche Teile der Geometrie später aus der Gebäudefassade heraus- und hineinragen. Abbildung 4.3 rechts zeigt, dass Teile der Geometrie, die lokal auf der positiven X-Seite liegen, aus der Gebäudefassade herausragen, während der restliche Teil der Geometrie in die Gebäudefassade hineinragt.

Materialdefinitionen von Cinema 4D werden beim Einlesen nicht beachtet. Jedoch sollte trotzdem testweise ein Material in Cinema 4D zugewiesen werden, um die Ausrichtung der Texturkoordinaten überprüfen und ggf. anpassen zu können. Die Texturkoordinaten können über das Layout *BP UV Edit* angepasst werden (siehe Abbildung 4.4). Man wählt das Polygon-Objekt aus, wechselt in den Modus *UV-Polygone bearbeiten*, wählt das Material aus und passt das UV-Mapping manuell oder über das Fenster *UV Mapping* (Einstellungen wie in Abbildung 4.4) an.

Sind die Texturkoordinaten festgelegt, so kann die Materialdefinition wieder vom Polygon-Objekt gelöscht werden und das Modell über *Hauptmenü→ Datei→ Exportieren→ VRML2* exportiert werden. Unter *Hauptmenü→ Bearbeiten→ Programm-Voreinstellungen→ Import/Export→ VRML 2 Export* sollte die Option *Faktor* auf 100 gesetzt werden, wenn in Zen-

timetern cm modelliert wurde. Wenn die exportierten Modelle in einem Unterordner von *Resources/models* gespeichert wurden, so werden sie beim nächsten Programmstart vom Ressourcenmanager geladen.

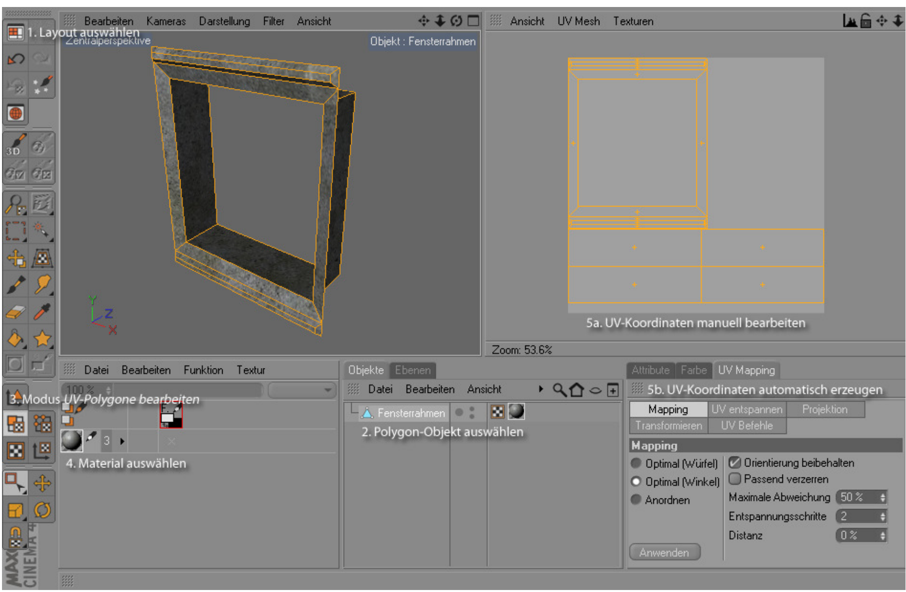


Abbildung 4.4: Texturkoordinatengenerierung des Fensterrahmen-Modells in Cinema 4D

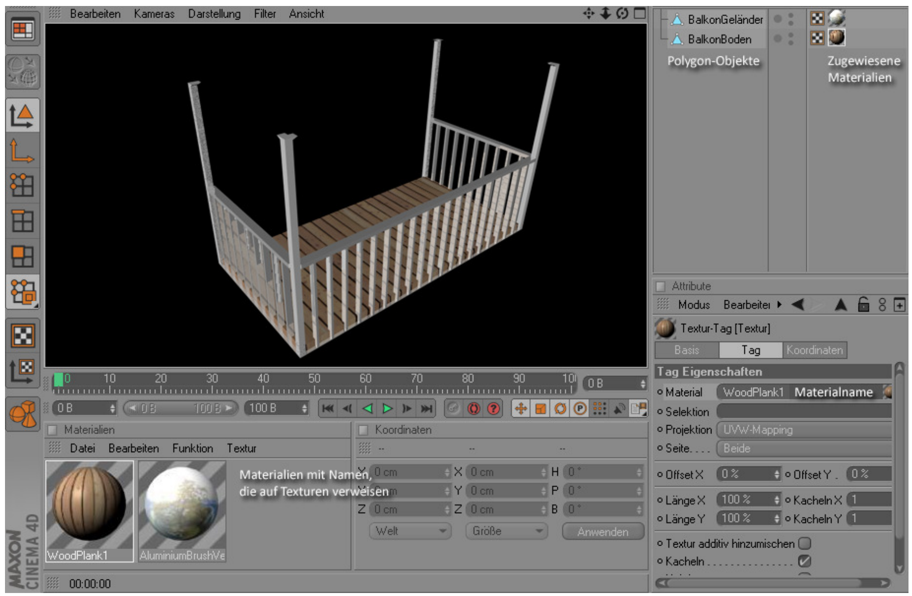


Abbildung 4.5: 3D-Geometrien mit zugewiesenen Materialien in Cinema 4D

Es können auch Geometrien, die aus mehreren Objekten bestehen, die auf verschiedene Materialien verweisen, erstellt werden und vom CityGenerator eingelesen werden. Man erstellt, wie oben beschrieben, Polygon-Objekte und weist ihnen Materialien zu. Die Namen der Materialien beziehen sich dabei auf Texturen, die vom Ressourcenmanager geladen worden sind. So wird dem Objekt in Abbildung 4.5, dem das Holzmaterial *WoodPlank1* zugewiesen ist, vom Ressourcenmanager das OpenSG-Material zugewiesen, dessen Texturname *WoodPlank1* beinhaltet. Alle weiteren Einstellungen der Materialdefinition, die in Cinema 4D gemacht

werden, werden vom Ressourcenmanager ignoriert. Die Geometrien müssen in das Verzeichnis *Resources/models/textured* exportiert werden.

Abbildung 4.6 auf Seite 73 zeigt alle erstellten Modelle in einer Übersicht.

4.3 Erstellen von Normalen- und Specular-Texturen

Die für die Shader benötigten Normalen- und Specular-Texturen wurden mit dem Programm *Shader Map Pro*[Systems 09] erstellt. Dabei handelt es sich um ein grafisches Werkzeug, das aus Oberflächentexturen Normalen-, Specular- und Höhentexturen erzeugt. In einem Vorschaufenster werden die erstellten Texturen mit einem entsprechenden Shader auf einem 3D-Objekt dargestellt (siehe Abbildung 4.7). Eine Kommandozeilenversion des Programms ist kostenlos erhältlich.

4.4 Erarbeiten eines Regelsets

Die nachfolgend vorgestellten Regeln sind in der Datei *Resources/rules/FacadeStructure.xml* im Projektordner zu finden.

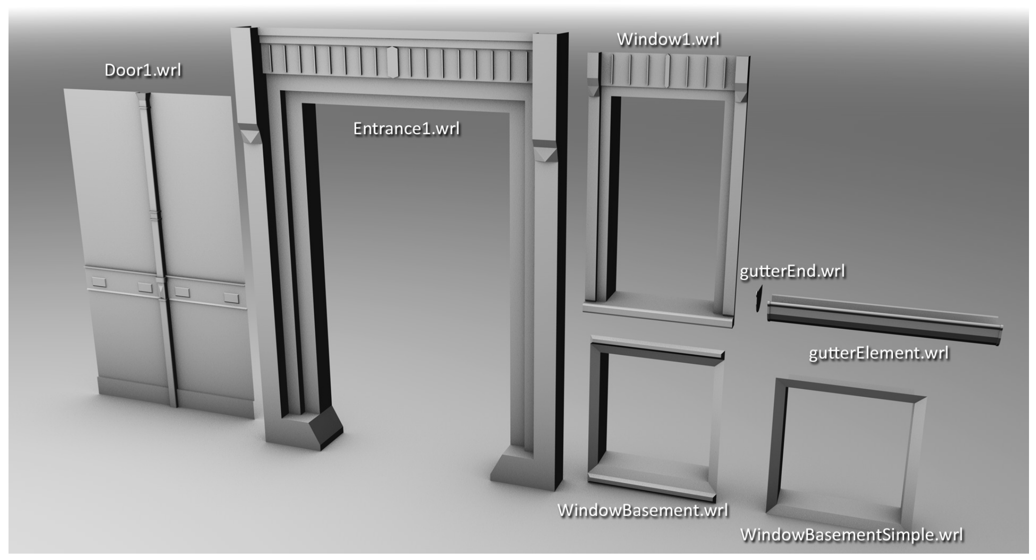
4.4.1 Regeln zur Fassadenstrukturierung

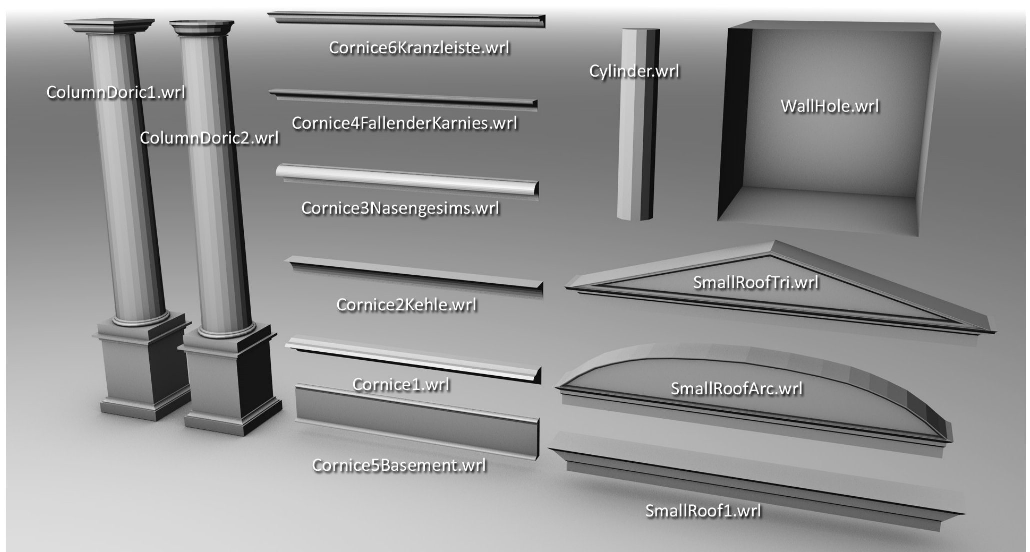
Fassaden können mit der vorgestellten Regelgrammatik beliebig in eine unregelmäßige Gitterstruktur unterteilt werden. Jedoch bietet sich für die meisten Fassadentypen eine regelmäßige, vertikale Erstunterteilung in Stockwerke an. Das Erdgeschoss sollte einen anderen Symbolnamen als die restlichen Stockwerke bekommen, um diesem z.B. später die Eingangstür zuweisen zu können. Außerdem ist es sinnvoll, die Höhe des Erdgeschosses über eine Variable steuern zu können. Die Stockwerke werden durch eine Wiederholung realisiert. Optional kann noch ein separates Symbol für das oberste Stockwerk eingefügt werden.

```
<Rule>
  <Predecessor>FrontFacade</Predecessor>
  <Rules>
      <Probability value="1.0">
          <Split axis="u" />
          <Symbol size="1" property="GroundfloorHeight">Groundfloor</Symbol>
          <Symbol size="1r">Floors</Symbol>
          (<Symbol size="1" property="TopFloorHeight">TopFloor</Symbol>)
      </Probability>
  </Rules>
</Rule>
```

Die Stockwerke werden vertikal wiederholt und dann horizontal, wiederholend in einzelne Kacheln unterteilt, in die später die Fenster eingesetzt werden (siehe Abbildung 4.8a).

```
<Rule>
    <Predecessor>Floors</Predecessor>
    <Rules>
        <Probability value="1.0">
            <Repeat size="1" axis="u" />
            <Symbol property="FloorHeight">Floor</Symbol>
```





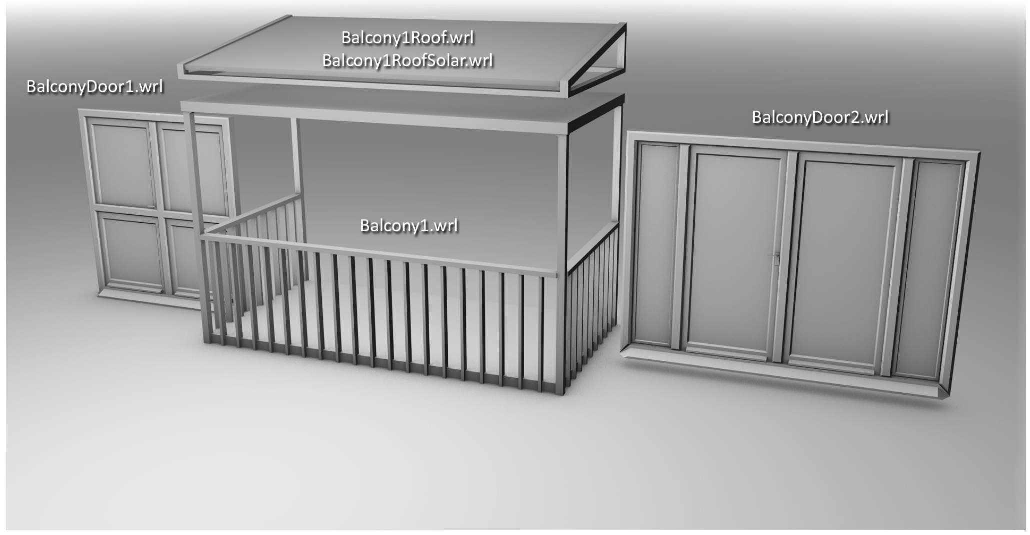


Abbildung 4.6: Alle mit Cinema 4D erstellten Geometrien

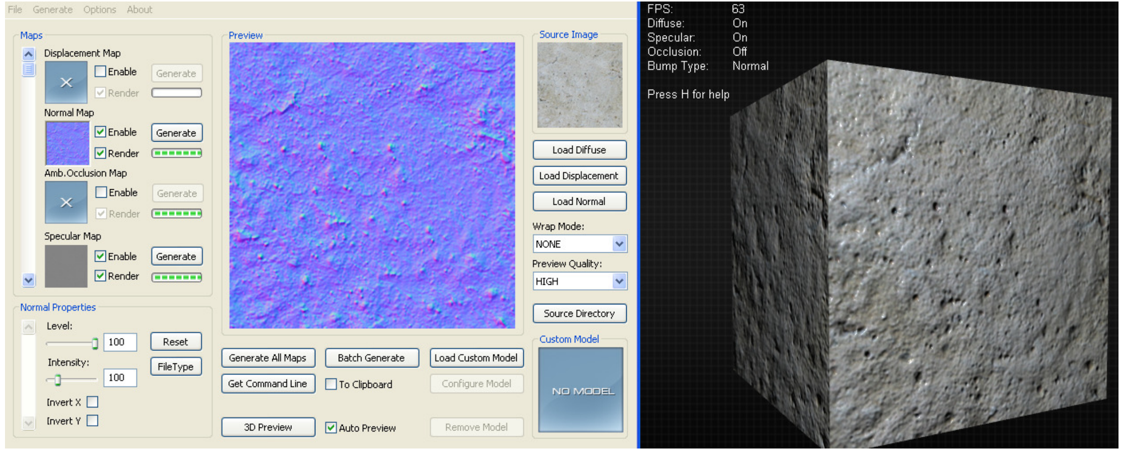
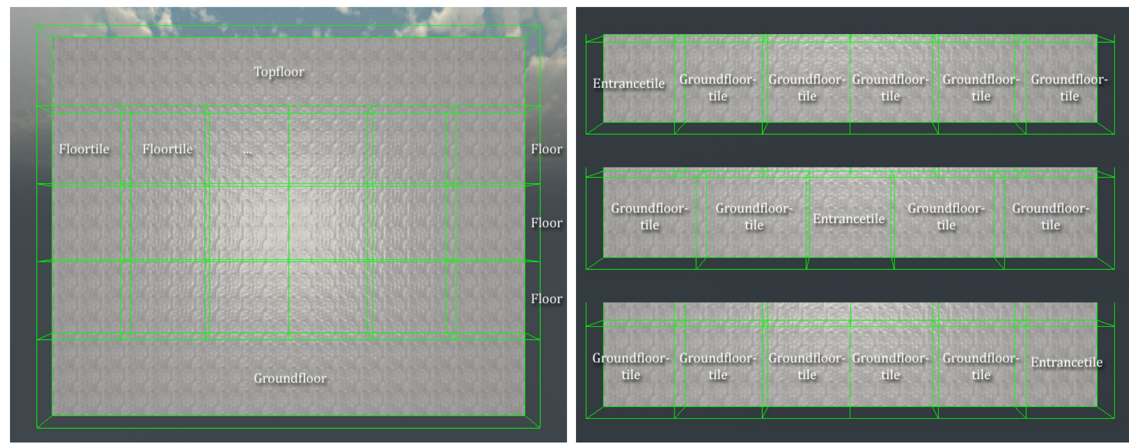


Abbildung 4.7: Shadermap-Oberfläche mit Vorschaufenster



(a) Grundstrukturierung der Fassade durch vertikale Unterteilung in Stockwerke

(b) Verschiedene Möglichkeiten der Anordnung des Erdgeschosses

Abbildung 4.8: Möglichkeiten zur vertikalen Unterteilung einer Fassade

```
        </Probability>
    </Rules>
</Rule>
<Rule>
    <Predecessor>Floor</Predecessor>
    <Rules>
        <Probability value="1.0">
            <Repeat size="1" axis="r" />
            <Symbol property="TileWidth">FloorTiles</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Nun wird das Erdgeschoss weiter unterteilt. Das Erdgeschoss kann auch variiert werden, indem separate Unterteilungen für Erdgeschosse mit linker, mittiger und rechter Anordnung des Eingangs erstellt wurden (siehe Abbildung 4.8b). Es wird also ein neues Symbol für den Eingangsbereich und ein Symbol für die wiederholten Erdgeschoss-Kacheln durch eine Split-Regel eingefügt.

```
<Rule>
    <Predecessor>Groundfloor</Predecessor>
    <Rules>
        <Probability value="0.333333">
            <Split axis="r" />
            <Symbol size="1" property="EntranceTileWidth">EntranceTile</
                Symbol>
            <Symbol size="1r">GroundfloorTiles</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Split axis="r" />
            <Symbol size="1r">GroundfloorTiles</Symbol>
            <Symbol size="1" property="EntranceTileWidth">EntranceTile</
                Symbol>
            <Symbol size="1r">GroundfloorTiles</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Split axis="r" />
            <Symbol size="1r">GroundfloorTiles</Symbol>
            <Symbol size="1" property="EntranceTileWidth">EntranceTile</
                Symbol>
        </Probability>
    </Rules>
</Rule>
```

Das oberste Stockwerk kann ähnlich unterteilt werden.

Eine andere Möglichkeit zur Fassadenstrukturierung wird durch horizontale Teilung erreicht. Dies ermöglicht es, die Betonung auf vertikale Strukturen zu richten, wie z.B. einem Treppenhaus. Hierbei kann man ähnlich variieren wie bei dem bereits vorgestellten Erdgeschoss. Die vertikale Struktur, die z.B. auch den Gebäudeeingang beinhaltet, kann auf der linken Seite, mittig oder auf der rechten Seite der Fassade platziert werden. Außerdem ist es möglich, weitere vertikale Strukturen einzubauen. Abbildung 4.9 zeigt die Varianten. Die restliche Fassade kann weiter vertikal in Stockwerke unterteilt werden, wie bereits beschrieben wurde.

```
<Rule>
    <Predecessor>FrontFacade</Predecessor>
    <Rules>
        <Probability value="0.25">
            <Split axis="r" />
            <Symbol size="1" property="EntranceColumnWidth">EntranceColumn</
                Symbol>
            <Symbol size="1r">Facade</Symbol>
        </Probability>
        <Probability value="0.25">
            <Split axis="r" />
            <Symbol size="1r">Facade</Symbol>
            <Symbol size="1r">Facade</Symbol>
            <Symbol size="1r">EntranceColumn</Symbol>
        </Probability>
        <Probability value="0.25">
            <Split axis="r" />
            <Symbol size="1r">Facade</Symbol>
            <Symbol size="1" property="EntranceColumnWidth">EntranceColumn</
                Symbol>
        </Probability>
        <Probability value="0.25">
            <Split axis="r" />
```

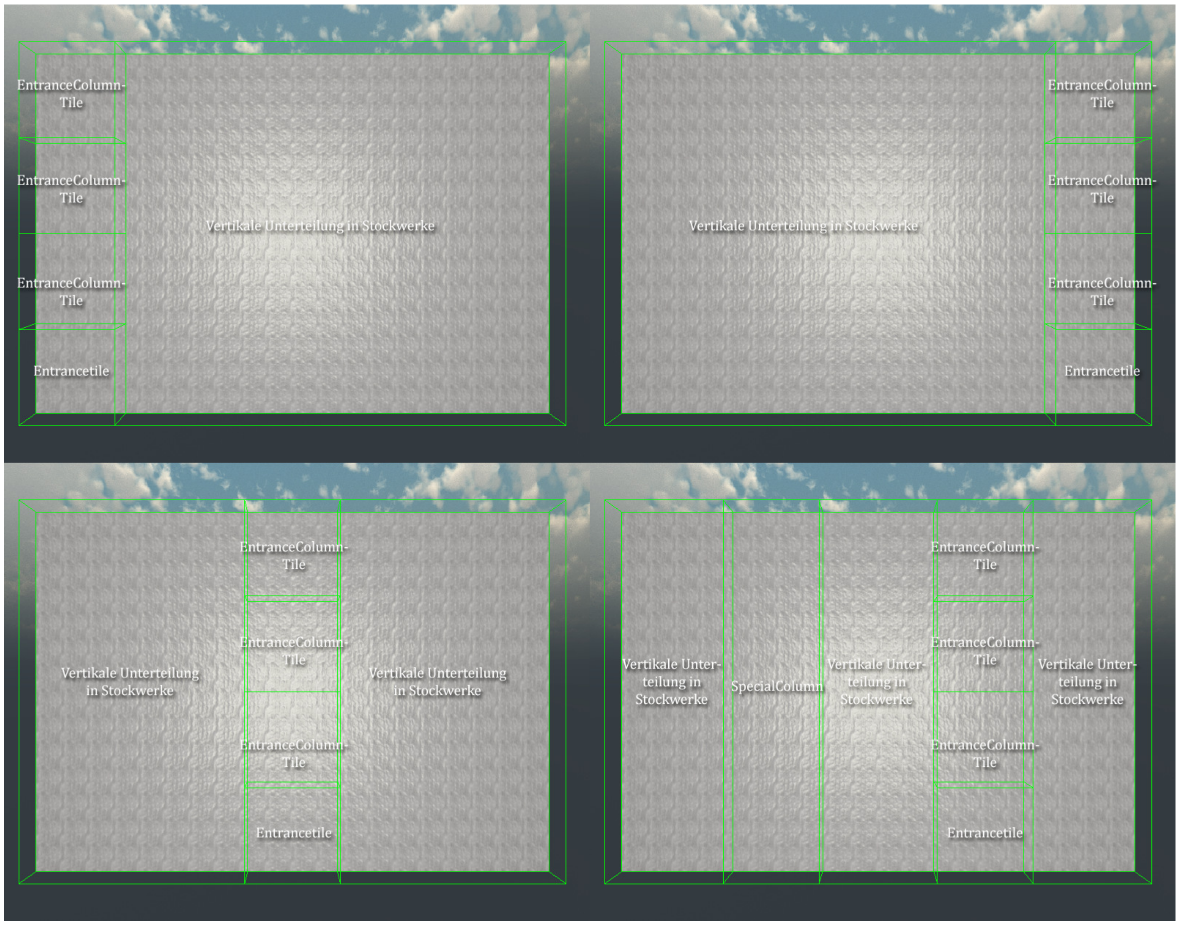


Abbildung 4.9: Verschiedene Möglichkeiten zur horizontalen Anordnung der Fassade

```
            <Symbol size="1r">Facade</Symbol>
            <Symbol size="1" property="EntranceColumnWidth">SpecialColumn</
                Symbol>
            <Symbol size="1r">Facade</Symbol>
            <Symbol size="1" property="EntranceColumnWidth">EntranceColumn</
                Symbol>
            <Symbol size="1r">Facade</Symbol>
        </Probability>
    </Rules>
</Rule>
```

4.4.2 Regeln für Fassadenkacheln

Im Folgenden werden Regeln für Fassadenkacheln des Erdgeschosses, normale Fassadenkacheln und Kacheln für die oberste Etage vorgestellt.

Eine Kachel des Erdgeschosses kann entweder mit oder ohne einem abschließenden Gesims gestaltet werden. Dazu werden zwei Regel-Verzweigungen angelegt. Die erste Verzweigung führt einen vertikalen Split durch und ersetzt das Erdgeschoss-Kachelsymbol mit einem Symbol, das die untere Hälfte der Kachel beschreibt (*GroundfloorBasementWallChoice*), und einem Symbol, das später das Fenster erzeugt (*GroundfloorWindowTileChoice*).

```
<Rule>
    <Predecessor>GroundfloorTile</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Split axis="u" />
            <Symbol size="1r">GroundfloorBasementWallChoice</Symbol>
            <Symbol size="1r">GroundfloorWindowTileChoice</Symbol>
        </Probability>
        <Probability value="0.500000">
            <Split axis="u" />
            <Symbol size="1r">GroundfloorBasementWindowTile</Symbol>
            <Symbol size="1r">GroundfloorWindowTileChoice</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Regel für *GroundfloorBasementWallChoice* ersetzt das Symbol entweder durch eine leere Wand oder setzt ein Gesims-Element ein, das die Wand zum Boden hin abschließt.

```
<Rule>
    <Predecessor>GroundfloorBasementWallChoice</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Symbol>Wall0.wrl</Symbol>
        </Probability>
        <Probability value="0.500000">
            <Split axis="u" />
            <Symbol size="1">Cornice5Basement.wrl</Symbol>
            <Symbol size="1r">Wall0.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Es ist oft von Vorteil, Fassadenobjekte durch eigene Regeln zu erzeugen, denn in diesen Regeln wird je Gebäude immer gleich verzweigt. Das bedeutet, dass wenn man z.B. die eben genannte Regel sowohl für die Vorderfassade als auch für die Hinterfassade benutzt, man auch eine konsistente Verzweigung erhält. So würde z.B. das Boden-Gesims-Element (*Cornice5Basement.wrl*) auf jeder Gebäudeseite entweder vorhanden oder nicht vorhanden sein.

Die andere Verzweigung der Regel *GroundfloorTile* sieht für die untere Hälfte das Symbol *GroundfloorBasementWindowTile* vor, das ein Kellerfenster erzeugt. Die zugehörige Regel unterteilt das Symbol durch drei neue Symbole in einem horizontalen Split. Dabei wird das Symbol *BasementWindow* links und rechts vom bereits besprochenem Symbol *Groundfloor-BasementWallChoice* umrandet.

```
<Rule>
    <Predecessor>GroundfloorBasementWindowTile</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="r" />
            <Symbol size="1r">GroundfloorBasementWallChoice</Symbol>
            <Symbol size="1r">GroundfloorBasementWindow</Symbol>
            <Symbol size="1r">GroundfloorBasementWallChoice</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Regel für das Symbol *GroundfloorBasementWindow* führt einen vertikalen Split durch und ersetzt das Symbol durch das neue Symbol *BasementWindowChoice* und dem Terminalsymbol *Wall0.wrl*.

```
 <Rule>
    <Predecessor>GroundfloorBasementWindow</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="u" />
            <Symbol size="1r">GroundfloorBasementWindowChoice</Symbol>
            <Symbol size="1r">Wall0.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Regel für *GroundfloorBasementWindowChoice* ist eine Substitutionsregel und setzt entweder das Fenstermodell *WindowBasement.wrl* oder *WindowBasementSimple.wrl* und jeweils das Symbol *BasementGlassChoice* ein.

```
<Rule>
    <Predecessor>GroundfloorBasementWindowChoice</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Symbol>WindowBasement.wrl</Symbol>
            <Symbol>BasementGlassChoice</Symbol>
        </Probability>
        <Probability value="0.500000">
            <Symbol>WindowBasementSimple.wrl</Symbol>
            <Symbol>BasementGlassChoice</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Regel für *BasementGlassChoice* setzt entweder eine Fenstertextur durch die *Glass0.wrl*-Geometrie ein oder erzeugt zusätzlich 3 Gitterstäbe vor dem Fenster. Den Gitterstäben wird ein Reflexionsmaterial zugewiesen und sie werden durch Translations- und Skalierungskommandos in der Position und Größe an das Fenster angepasst.

```
<Rule>
    <Predecessor>BasementGlassChoice</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Symbol translate="0␣0.025␣-0.2" scale="0.85␣0.925␣1">Glass0.wrl<
                /Symbol>
        </Probability>
        <Probability value="0.500000">
            <Symbol translate="0␣0.025␣-0.2" scale="0.85␣0.925␣1">Glass0.wrl<
                /Symbol>
            <Symbol translate="-0.25␣0.05␣-0.1" scale="0.05␣0.9␣0.05" texture
                ="reflection\AluminiumBrushVert.png">Cylinder.wrl</Symbol>
            <Symbol translate="0␣0.05␣-0.1" scale="0.05␣0.9␣0.05" texture="
                reflection\AluminiumBrushVert.png">Cylinder.wrl</Symbol>
            <Symbol translate="0.25␣0.05␣-0.1" scale="0.05␣0.9␣0.05" texture=
                "reflection\AluminiumBrushVert.png">Cylinder.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Nachfolgend werden die Regeln zum Einsetzen des Fensters erläutert. In der Regel für das Symbol *GroundfloorTile* wurde das Symbol *GroundfloorWindowTileChoice* eingesetzt. Die Ableitungsregel für dieses Symbol teilt es in horizontaler Richtung und platziert das Symbol *WindowPlaceXChoice* mittig zwischen zwei Wänden. Später kann hier eine weitere Verzweigung eingefügt werden.

```
<Rule>
    <Predecessor>GroundfloorWindowTileChoice</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Split axis="r" />
            <Symbol size="1r">Wall0.wrl</Symbol>
            <Symbol size="1" property="GroundfloorWindowWidth">
                WindowPlaceXChoice</Symbol>
            <Symbol size="1r">Wall0.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Regel für *WindowPlaceXChoice* unterteilt in vertikaler Richtung und setzt entweder das Symbol *WindowChoice* für normale Fenster oder *WindowWithColumnPlaceX* für Fenster mit Säulenverzierungen ein, gefolgt von dem Symbol *SmallRoofChoice* für ein abschließendes, kleines Zierdach und einem kurzen Wandabschnitt.

```
<Rule>
    <Predecessor>WindowPlaceXChoice</Predecessor>
    <Rules>
        <Probability value="0.75">
            <Split axis="u" />
            <Symbol size="1r">WindowChoice</Symbol>
            <Symbol size="0.2" scale="1.1␣1␣1">SmallRoofChoice</Symbol>
            <Symbol size="0.3">Wall0.wrl</Symbol>
        </Probability>
        <Probability value="0.25">
          <Split axis="u" />
          <Symbol size="1r">WindowWithColumnPlaceX</Symbol>
          <Symbol size="0.2" scale="1.1␣1␣1">SmallRoofChoice</Symbol>
          <Symbol size="0.3">Wall0.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Regel für das Symbol *WindowChoice* setzt durch eine Substitution einfach den umgebenden Fensterrahmen aus der Geometrie *Window1.wrl* und eine Fenstertextur durch die Geometrie *Glass0.wrl* ein.

```
<Rule>
    <Predecessor>WindowChoice</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Symbol>Window1.wrl</Symbol>
            <Symbol translate="0␣0␣-0.1" scale="0.8␣0.85␣1">Glass0.wrl</
                Symbol>
        </Probability>
    </Rules>
</Rule>
```

WindowWithColumnPlaceX wird in horizontaler Richtung geteilt und platziert links und rechts das Symbol *PillarChoice* und mittig das eben besprochene Symbol *WindowChoice*. *PillarChoice* wird durch eine Wand und die Geometrie *ColumnDoric1.wrl* ersetzt.

```
<Rule>
  <Predecessor>WindowWithColumnPlaceX</Predecessor>
  <Rules>
    <Probability value="1.000000">
      <Split axis="r" />
      <Symbol size="0.2">PillarChoice</Symbol>
      <Symbol size="1r">WindowChoice</Symbol>
      <Symbol size="0.2">PillarChoice</Symbol>
    </Probability>
  </Rules>
</Rule>
<Rule>
    <Predecessor>PillarChoice</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol>ColumnDoric1.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

SmallRoofChoice verzweigt so, dass eine von drei Dachgeometrien eingesetzt werden, die den Abschluss des Fensters bilden.

```
<Rule>
    <Predecessor>SmallRoofChoice</Predecessor>
    <Rules>
        <Probability value="0.333333">
            <Symbol>SmallRoof1.wrl</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Symbol>SmallRoofArc.wrl</Symbol>
            <Symbol scale="0.9091␣1␣1">Wall0.wrl</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Symbol>SmallRoofTri.wrl</Symbol>
            <Symbol scale="0.9091␣1␣1">Wall0.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Abbildung 4.10 zeigt einige Variationsmöglichkeiten der vorgestellten Regeln für die Erdgeschosskacheln.

Die Regeln für die Kacheln der normalen Stockwerke sind ähnlich. Das Symbol *FloorTileChoice* wird in vertikaler Richtung in die Symbole *FloorTileBottomWallChoice* und *FloorTileWindowXPlacement* unterteilt

```
<Rule>
    <Predecessor>FloorTileChoice</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="u" />
            <Symbol size="1r">FloorTileBottomWallChoice</Symbol>
            <Symbol size="2r">FloorTileWindowXPlacement</Symbol>
```

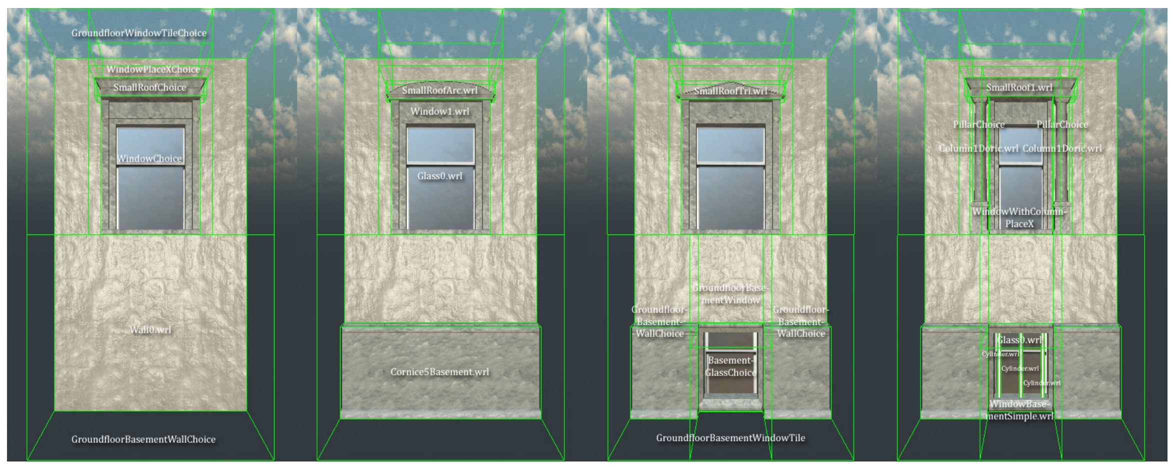


Abbildung 4.10: Verschiedene Möglichkeiten zur Anordnung der Erdgeschosskacheln

```
        </Probability>
    </Rules>
</Rule>
```

Die Regel für *FloorTileBottomWallChoice* hat drei Verzweigungen und setzt entweder eine Wand, ein Gesimselement oder ein kleineres Gesimselement mit darüber liegender Wand ein.

```
<Rule>
    <Predecessor>FloorTileBottomWallChoice</Predecessor>
    <Rules>
        <Probability value="0.333333">
            <Symbol>Wall0.wrl</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Split axis="u" />
            <Symbol size="0.34" scale="1␣1␣0.1">CorniceChoice</Symbol>
            <Symbol size="1r">Wall0.wrl</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Split axis="u" />
            <Symbol size="1r" scale="1␣1␣0.1">CorniceChoice</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Regel für das Symbol *CorniceChoice* setzt eine von fünf verschiedenen Gesims-Geoemtrien ein.

```
<Rule>
    <Predecessor>CorniceChoice</Predecessor>
    <Rules>
        <Probability value="0.200000">
            <Symbol>Cornice1.wrl</Symbol>
        </Probability>
        <Probability value="0.200000">
            <Symbol>Cornice2Chamfer.wrl</Symbol>
        </Probability>
        <Probability value="0.200000">
            <Symbol>Cornice3Nasengesims.wrl</Symbol>
        </Probability>
```

```
        <Probability value="0.200000">
            <Symbol>Cornice4FallenderKarnies.wrl</Symbol>
        </Probability>
        <Probability value="0.200000">
            <Symbol>Cornice6Kranzleiste.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die *FloorTileWindowXPlacement*-Regel platziert das bereits vorgestellte Symbol *Groundfloor-WindowTileChoice* mittig zwischen zwei Wände.

```
<Rule>
    <Predecessor>FloorTileWindowXPlacement</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="r" />
            <Symbol size="1r">Wall0.wrl</Symbol>
            <Symbol size="2">GroundfloorWindowTileChoice</Symbol>
            <Symbol size="1r">Wall0.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Alternativ könnte man an dieser Stelle weitere Verzweigungen einfügen, um die Fenster der normalen Stockwerke anders als die Fenster des Erdgeschosses zu gestalten.

Die Kachel des Hauseingangs (*EntranceTile*) bietet zwei Verzweigungen. Die Verzweigungen platzieren die Symbole *Entrance1* und *Entrance2* mittig zwischen den bereits beschrieben Symbolen *GroundfloorBasementWallChoice*.

```
<Rule>
    <Predecessor>EntranceTile</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Split axis="r" />
            <Symbol size="1r">GroundfloorBasementWallChoice</Symbol>
            <Symbol size="1" property="EntranceWidth">Entrance1</Symbol>
            <Symbol size="1r">GroundfloorBasementWallChoice</Symbol>
        </Probability>
        <Probability value="0.500000">
            <Split axis="r" />
            <Symbol size="1r">GroundfloorBasementWallChoice</Symbol>
            <Symbol size="1" property="EntranceWidth">Entrance2</Symbol>
            <Symbol size="1r">GroundfloorBasementWallChoice</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Regel für das Symbol *Entrance1* erzeugt durch einen vertikalen Split einen Eingangsbereich mit Tür (*Entrance1WithDoor*), auf den ein kleines Zierdach aufgesetzt ist.

```
<Rule>
    <Predecessor>Entrance1</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="u" />
            <Symbol size="1" property="EntranceHeight">Entrance1WithDoor</
                Symbol>
```

```
            <Symbol size="0.1" scale="1.07␣1␣1">SmallRoof1.wrl</Symbol>
            <Symbol size="1r">Wall0.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
<Rule>
    <Predecessor>Entrance1WithDoor</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Symbol>Entrance1.wrl</Symbol>
            <Symbol>DoorChoice</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Regel für das Symbol *Entrance2* bietet drei Verzweigungen. Die erste Verzweigung ist ähnlich der eben vorgestellten Eingangskachel. Nur dass hierbei zwei Ziersäulen rechts und links des Eingangs hinzugefügt werden.

```
<Rule>
    <Predecessor>Entrance2</Predecessor>
    <Rules>
        <Probability value="0.333333">
            <Split axis="u" />
            <Symbol size="1" property="EntranceHeight">
                Entrance2WithDoorAndColumns</Symbol>
            <Symbol size="0.1" scale="1.1␣1␣1.1">SmallRoof1.wrl</Symbol>
            <Symbol size="1r">Wall0.wrl</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Split axis="u" />
            <Symbol size="1" property="EntranceHeight">
                Entrance2WithDoorAndColumns</Symbol>
            <Symbol size="0.2" scale="1.1␣1␣1.55">
                EntranceTile2SmallRoofChoice</Symbol>
            <Symbol size="1r">Wall0.wrl</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Split axis="u" />
            <Symbol size="1" property="EntranceHeight">
                Entrance2WithDoorAndColumns</Symbol>
            <Symbol size="0.1" scale="1.1␣1␣1.25">SmallRoof1.wrl</Symbol>
            <Symbol size="1r">EntranceTile2SmallRoofChoice</Symbol>
        </Probability>
    </Rules>
</Rule>
<Rule>
    <Predecessor>Entrance2WithDoorAndColumns</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="r" />
            <Symbol size="0.4">PillarChoice</Symbol>
            <Symbol size="1r">Entrance1WithDoor</Symbol>
            <Symbol size="0.4">PillarChoice</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die zweite Verzweigung erzeugt einen Eingang mit Säulen und setzt eins von zwei möglichen

Zierdächern auf (Regel *Entrance2TileSmallRoofChoice*) und füllt den restlichen Raum mit einer Wand aus.

```
<Rule>
    <Predecessor>EntranceTile2SmallRoofChoice</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol>SmallRoofTri.wrl</Symbol>
        </Probability>
        <Probability value="0.500000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol>SmallRoofArc.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die dritte Verzweigung setzt dem Eingangsbereich aus der ersten Verzweigung ein weiteres Zierdach auf. Abbildung 4.11 zeigt einige Variationen der Regeln für die Eingangsbereichkacheln.

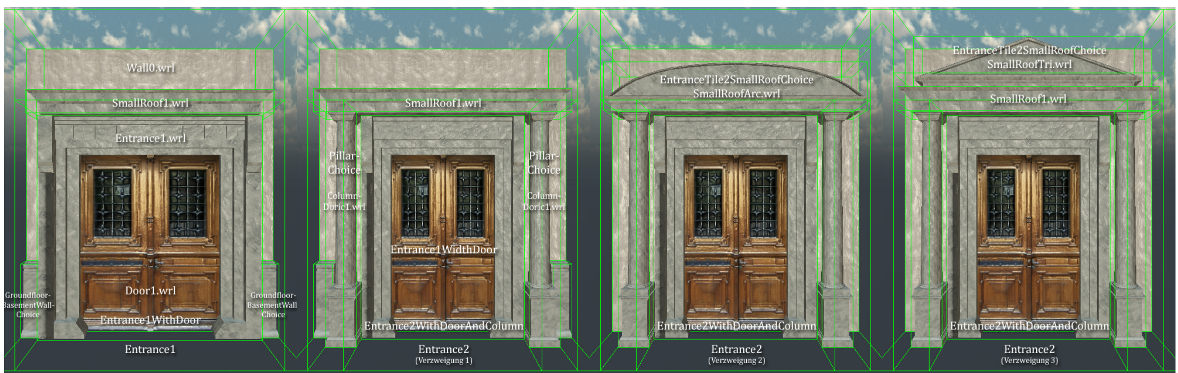


Abbildung 4.11: Verschiedene Möglichkeiten zur Anordnung der Eingangsbereichkacheln

4.4.3 Regeln für Verzierungen

Stuckstreifen oder Gesimse können in die Fassadenkacheln, die durch einen vertikalen Split erstellt werden (z.B. *GroundfloorTile* einfach durch ein weiteres Nachfolgersymbol eingefügt werden:

```
<Rule>
  <Predecessor>GroundfloorTile</Predecessor>
  <Rules>
      <Probability value="0.500000">
          <Split axis="u" />
          <Symbol size="1r">GroundfloorBasementWallChoice</Symbol>
          <Symbol size="1r">GroundfloorWindowTileChoice</Symbol>
          \emph{<Symbol size="0.4">GroundfloorTopWallChoice</Symbol>}
      </Probability>
      ...
```

Das neue Wandsymbol kann nun vertikal in ein Ornamentsymbol (*OutpickingOrnamentWall*), das von zwei Stuckstreifenmodellen oben und unten umschlossen wird, weiter unterteilt wer-

den. Alternativ kann auch nur das Stuckstreifensymbol flach (durch einen Skalierungsoperator) eingesetzt werden oder es wird nur eine leere Wand eingefügt.

```
<Rule>
    <Predecessor>GroundfloorTopWallChoice</Predecessor>
    <Rules>
        <Probability value="0.333333">
            <Split axis="u" />
            <Symbol size="1r" scale="1␣1␣3.5" z_scale="none">
               Cornice4FallenderKarnies.wrl</Symbol>
            <Symbol size="1r" scale="1␣1␣0.17">OutpickingOrnamentWall</Symbol
               >
            <Symbol size="1r" scale="1␣1␣3.5" z_scale="none">
               Cornice4FallenderKarniesOben.wrl</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Symbol scale="1␣1␣0.0001">OutpickingOrnamentWall</Symbol>
        </Probability>
        <Probability value="0.333333">
            <Symbol>Wall0.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

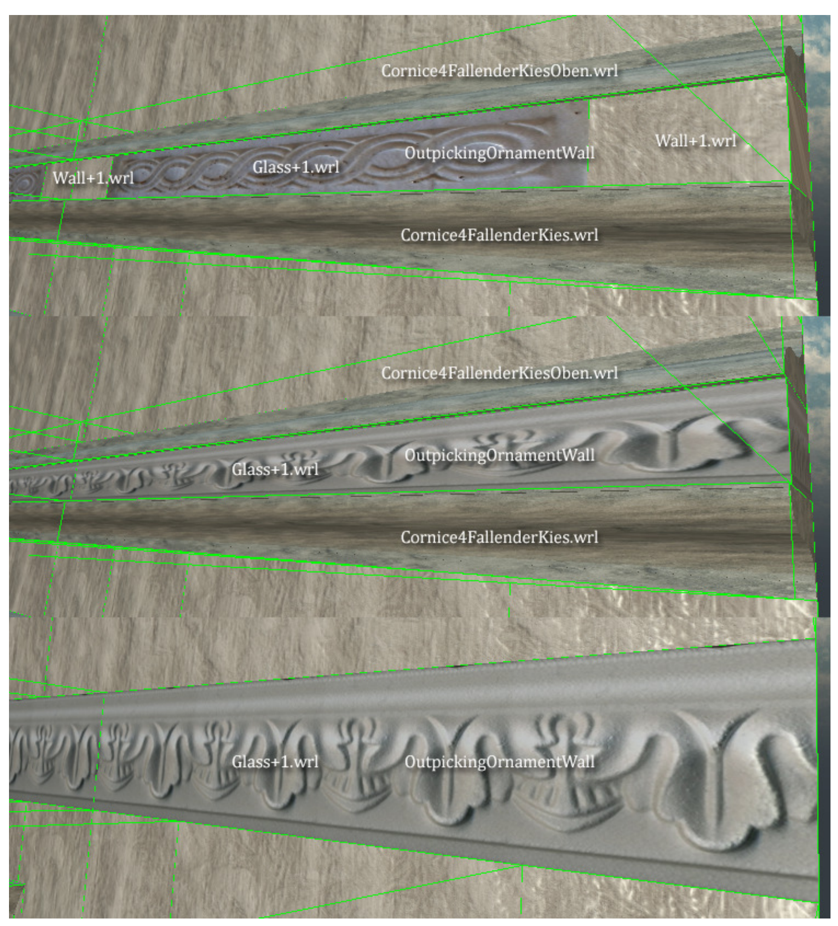


Abbildung 4.12: Verschiedene Möglichkeiten zur Anordnung von Ornament-Elementen

Das Ornamentsymbol wird horizontal unterteilt, wobei links und rechts eine aus der Fassade herausragende Wand (Geometrie *Wall+1.wrl*) und in der Mitte die Geometrie *Glass+1.wrl* eingesetzt wird. Bei dieser Geometrie wird die Textur genau einmal auf jede Seite projiziert, also nicht auf die Größe der Geometrie angepasst, wie dies bei der *Wall+1.wrl*-Geometrie der Fall ist. Durch einen UV-Skalierungs-Operator kann man somit bestimmen, wie oft die Textur auf der Geometrie gekachelt werden soll. Dies wird bei der zweiten Verzweigung der Regel genutzt, bei der nur die *Glass+1.wrl*-Geometrie verwendet wird auf die eine Textur mehrmals gekachelt wird (in diesem Fall $1/0.25 = 4$ mal):

```
<Rule>
    <Predecessor>OutpickingOrnamentWall</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Split axis="r" />
            <Symbol size="1r" z_scale="none">Wall+1.wrl</Symbol>
            <Symbol size="6r" texture="miscNormal\OrnamentBorder0081_S.jpg"
                z_scale="none">Glass+1.wrl</Symbol>
            <Symbol size="1r" z_scale="none">Wall+1.wrl</Symbol>
        </Probability>
        <Probability value="0.500000">
            <Symbol texture="miscNormal\OrnamentBorder0093_2_S.jpg" uv="0.25␣
                1" z_scale="none">Glass+1.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Durch den Operator *z_scale=none* wird verhindert, dass die Geoemtrien in Z-Richtung skaliert werden. Obwohl die Ornamente nur durch Texturen dargestellt werden, ergibt sich durch die verwendeten Normalentexturen ein räumlicher Eindruck der Ornamentstruktur. Abbildung 4.12 zeigt die erzeugten Modelle.

Viele Hoffassaden sind mit **Balkonen** versehen. Diese sind meist in einer vertikalen Fassadenspalte angeordnet und wiederholen sich auf jeder Etage. Das vertikale Symbol, das alle Balkone enthält wird *BackFacadeBalconyColumn* genannt. Die Spalte wird durch einen vertikalen Split in die Symbole *BackFacadeGroundfloorBalconyTile*, *BackFacadeBalconyTiles* und *BackFacadeBalconyTopfloorTile* aufgeteilt.

```
<Rule>
    <Predecessor>BackFacadeBalconyColumn</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="u" />
            <Symbol size="1" property="GroundfloorHeight">
                BackFacadeGroundfloorBalconyTile</Symbol>
            <Symbol size="1r">BackFacadeBalconyTiles</Symbol>
            <Symbol size="1" property="TopfloorHeight">
                BackFacadeBalconyTopfloorTile</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Kachel für das Erdgeschoss wird nochmal vertikal in den Balkonboden (die Balkonstützen) und in das eigentlichen Balkonsymbol unterteilt. Alternativ könnte an dieser Stelle auch eine Terrasse oder Ähnliches eingesetzt werden.

```
<Rule>
```

```
    <Predecessor>BackFacadeGroundfloorBalconyTile</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="u" />
            <Symbol size="0.45">BackFacadeGroundfloorBalconyGroundTile</
                Symbol>
            <Symbol size="1r">BackFacadeBalconyTile</Symbol>
        </Probability>
    </Rules>
</Rule>

<Rule>
    <Predecessor>BackFacadeGroundfloorBalconyGroundTile</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol z_scale="right">Balcony1Ground.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Eine Balkon-Kachel setzt sich durch eine Substitutions-Regel aus der Fassadenwand und dem eigentlichen Balkonmodell zusammen. Die Fassadenwand kann dabei z.B. mit einer Balkontür oder auch mit einer nach hinten versetzten Wand gestaltet werden (siehe Abbildung 4.13). Wichtig ist, dass die Balkongeometrien einheitlich skaliert werden, was durch den Z-Skalierungs-Operator gleich *right* erreicht wird.

```
<Rule>
    <Predecessor>BackFacadeBalconyTile</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Symbol>BackFacadeBalconyWallTile</Symbol>
            <Symbol>BackFacadeBalcony</Symbol>
        </Probability>
    </Rules>
</Rule>

<Rule>
    <Predecessor>BackFacadeBalconyWallTile</Predecessor>
    <Rules>
        <Probability value="0.250000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol scale="0.5␣0.9␣1">BalconyDoor.wrl</Symbol>
        </Probability>
        <Probability value="0.250000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol scale="0.95␣0.9␣1">BalconyDoor2.wrl</Symbol>
        </Probability>
        <Probability value="0.250000">
            <Symbol scale="1␣1␣0.5" z_scale="none">WallHole.wrl</Symbol>
            <Symbol translate="0␣0␣-0.5" scale="0.5␣0.9␣1">BalconyDoor.wrl</
                Symbol>
        </Probability>
        <Probability value="0.250000">
            <Symbol scale="1␣1␣0.5" z_scale="none">WallHole.wrl</Symbol>
            <Symbol translate="0␣0␣-0.5" scale="0.95␣0.9␣1">BalconyDoor2.wrl<
                /Symbol>
        </Probability>
```

```
    </Rules>
</Rule>

<Rule>
    <Predecessor>BackFacadeBalcony</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Symbol z_scale="right">Balcony1.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Balkon-Kachel für das oberste Stockwerk setzt sich aus der normalen Balkonkachel sowie einem Symbol mit zwei möglichen Ableitungen für ein Balkondach zusammen. Abbildung 4.13 zeigt die Ergebnisse dieser Regeldefinitionen.

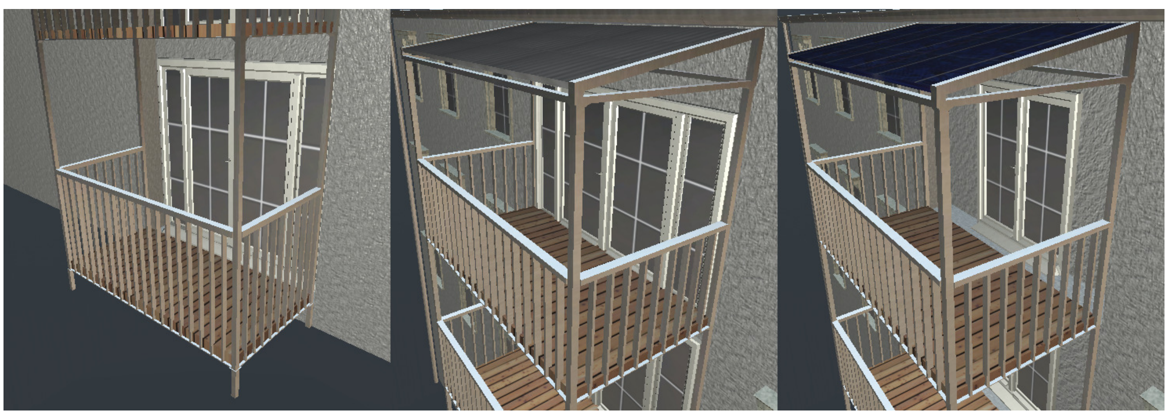


Abbildung 4.13: Verschiedene Möglichkeiten für Balkonmodelle

```
 <Rule>
    <Predecessor>BackFacadeBalconyTopfloorTile</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="u" />
            <Symbol size="1r">BackFacadeBalconyTile</Symbol>
            <Symbol size="0.5">BackFacadeBalconyRoofTile</Symbol>
        </Probability>
    </Rules>
</Rule>

<Rule>
    <Predecessor>BackFacadeBalconyRoofTile</Predecessor>
    <Rules>
        <Probability value="0.800000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol z_scale="right">Balcony1Roof.wrl</Symbol>
        </Probability>
        <Probability value="0.200000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol z_scale="right">Balcony1RoofSolar.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Ein weiteres Fassadendetail ist die **Abflussrinne** vom Gebäudedach und das zugehörige Fallrohr. Die Regeln zur Erzeugung des Regenabflusses werden direkt an das Startsymbol für die hofseitige Fassade angehängt. Dabei wird zwischen einer Fassade ohne Abfluss (*BackFacade-Content*) und mit Abfluss (BackFacadeGutterChoice) verzweigt.

```
<Rule>
    <Predecessor>BackFacade</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Symbol>BackFacadeContent</Symbol>
        </Probability>
        <Probability value="0.500000">
            <Symbol>BackFacadeGutterChoice</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Die Fassade mit Abfluss wird in vertikaler Richtung unterteilt, wobei oben in einem schmalen Streifen die Abflussrinne unter dem Dach eingesetzt wird und im unteren Teil das Symbol *GutterHorizontal* eingesetzt wird, das in einem weiteren Schritt das Symbol für das Fallrohr entweder links oder rechts an der Fassade einsetzt.

```
<Rule>
    <Predecessor>BackFacadeGutterChoice</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="u" />
            <Symbol size="1r">BackFacadeGutterBottom</Symbol>
            <Symbol size="0.2">GutterHorizontal</Symbol>
        </Probability>
    </Rules>
</Rule>

<Rule>
    <Predecessor>BackFacadeGutterBottom</Predecessor>
    <Rules>
        <Probability value="0.500000">
            <Split axis="r" />
            <Symbol size="0.2">GutterVertical</Symbol>
            <Symbol size="1r">BackFacadeContent</Symbol>
        </Probability>
        <Probability value="0.500000">
            <Split axis="r" />
            <Symbol size="1r">BackFacadeContent</Symbol>
            <Symbol size="0.2">GutterVertical</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Das Symbol *GutterHorizontal* wird durch die Fassadenwand und das Symbol *GutterHorizontalRepeat* ersetzt, das wiederum in horizontaler Richtung in ein Dachrinnenendstück, dem Symbol *GutterElements* und einem weiteren Dachrinnenendstück unterteilt wird. Außerdem wird eine Textur zugewiesen, die auf ein reflektierendes Material verweist. Schließlich werden einzelne Dachrinnenelemente durch eine Repeat-Regel in horizontaler Richtung eingesetzt.

```
<Rule>
    <Predecessor>GutterHorizontal</Predecessor>
    <Rules>
```

```
        <Probability value="1.000000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol>GutterHorizontalRepeat</Symbol>
        </Probability>
    </Rules>
</Rule>

<Rule>
    <Predecessor>GutterHorizontalRepeat</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Split axis="r" />
            <Symbol size="0.01" scale="1␣1␣0.725" texture="reflection\
                AluminiumBrushVert.png">gutterEnd.wrl</Symbol>
            <Symbol size="1r" texture="reflection\AluminiumBrushVert.png">
                GutterElements</Symbol>
            <Symbol size="0.01" scale="1␣1␣0.725" texture="reflection\
                AluminiumBrushVert.png">gutterEnd.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>

<Rule>
    <Predecessor>GutterElements</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Repeat size="1" axis="r" />
            <Symbol>gutterElement.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

Das Fallrohr wird einfach durch eine Zylinder-Geometrie mit einem reflektierenden Material erzeugt. Abbildung 4.14 zeigt die Dachrinne mit Fallrohr.

```
<Rule>
    <Predecessor>GutterVertical</Predecessor>
    <Rules>
        <Probability value="1.000000">
            <Symbol>Wall0.wrl</Symbol>
            <Symbol translate="0␣0␣0.15" scale="0.8␣1.003␣0.8" texture="
                reflection\AluminiumBrushVert.png">Cylinder.wrl</Symbol>
        </Probability>
    </Rules>
</Rule>
```

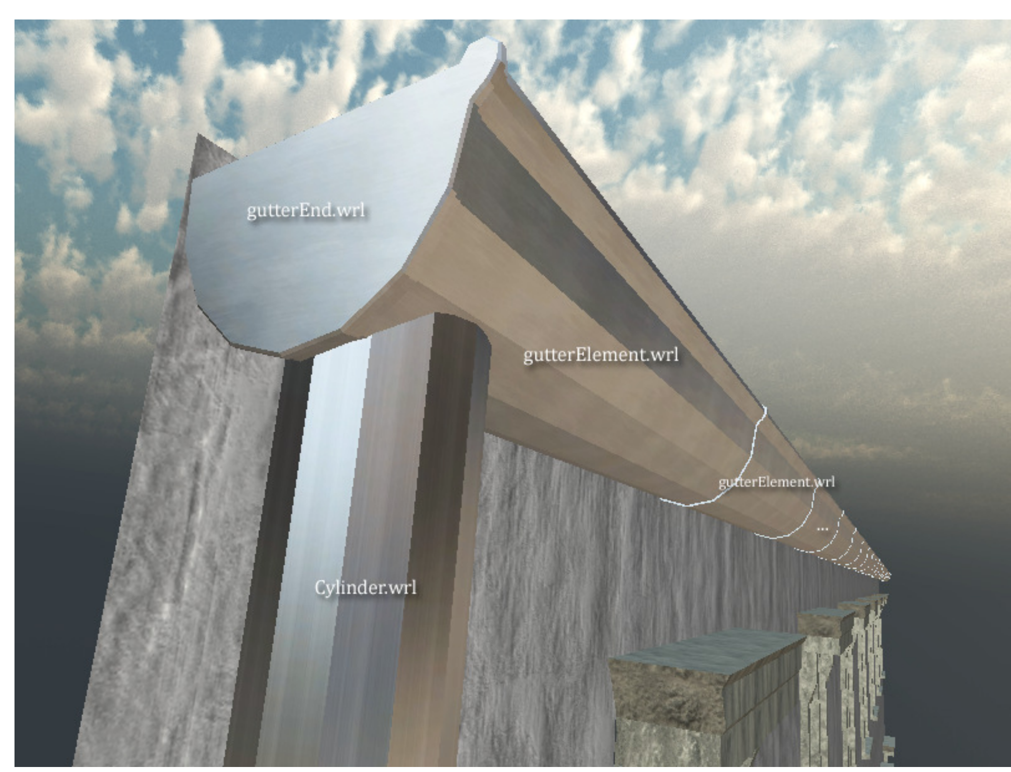


Abbildung 4.14: Die Dachrinne mit Fallrohr

Kapitel 5

Beispiel eines prozedural generierten Stadtgebiets

Mithilfe der im Kapitel 4.4 erarbeiteten Regeln und der modellierten Fassadenbausteinen (siehe Kapitel 4.2 auf Seite 69) wurde ein kleines Stadtgebiet mit vier Gebäudeblöcken rund um die Etkar-André-, Virchow- und Coppi-Straße in Leipzig-Gohlis erstellt (siehe Abbildung 5.1). In diesem Gebiet sind die Häuser weniger detailliert und verziert gestaltet als z.B. im Waldstraßenviertel. Da nur einige wenige Fassadenbausteinen im Rahmen der Arbeit modelliert werden konnten, wurde ein Stadtgebiet mit einfacheren Fassaden gewählt. Um Gebäude ähnlich denen im Waldstraßenviertel als 3D-Modelle zu erstellen sind auch deutlich zahlreichere und komplexere Fassadenbausteine erforderlich. Die Szenendefinition wurde aus *OpenStreetMaps* (siehe Kapitel 6.2 auf Seite 95) exportiert. Die auf Abbildung 5.3 oben links zu sehenden ersten beiden Gebäude auf der rechten Straßenseite wurden als Vorlage für zwei neue Regeldateien ausgewählt, da diese typisch für den Architekturstil dieses Gebietes sind. Die Regeldateien sollen genau diese beiden Gebäude als 3D-Modelle erzeugen.

Abbildung 5.1: Links: Luftbildaufnahme des Gebietes um die Etkar-André-Straße, Quelle: [Google 09], Rechts: OpenStreetMap-Daten des Gebietes mit den gelb markierten, erstellten Gebäuden

Das Ergebnis ist in Abbildung 5.3 auf Seite 94 zu sehen. Abgesehen von ein paar kleinen Details, wie z.B. der Reliefs über den Fenstern, kommen die prozedural erzeugten Fassadenmodelle den echten Fassaden schon sehr nahe (siehe Abbildung 5.3 Mitte). Und das, obwohl die Fassadenbausteine gar nicht für genau diese Häuser modelliert wurden. Die Häuserfassaden wirken in der Echtzeitvisualisierung (siehe Abbildung 5.3 Unten) nicht ganz so realistisch, da die Beleuchtungsmodelle weniger komplex als in einem Modellierungsprogramm wie Cinema 4D sind. Im Kapitel 7 auf Seite 102 wird eine Möglichkeit zur Verbesserung der Grafik zur Echtzeitvisualisierung kurz vorgestellt. Die generierten OpenSG-Szenen sind im im Projektordner in *Resources/scenes/* und die generierten Cinema 4D-Szenen sind im Projektordner zu finden.

Mit den in Kapitel 4.4 vorgestellten, einfacheren Regeln wurde das Modell in Abbildung 5.2 erstellt. Des Weiteren wurde ein Stadtmodell von Gebäudeblöcken rund um die Eisenbahnstraße in Leipzig angefertigt. Dies wird in Kapitel 6.2 auf Seite 95 beschrieben.

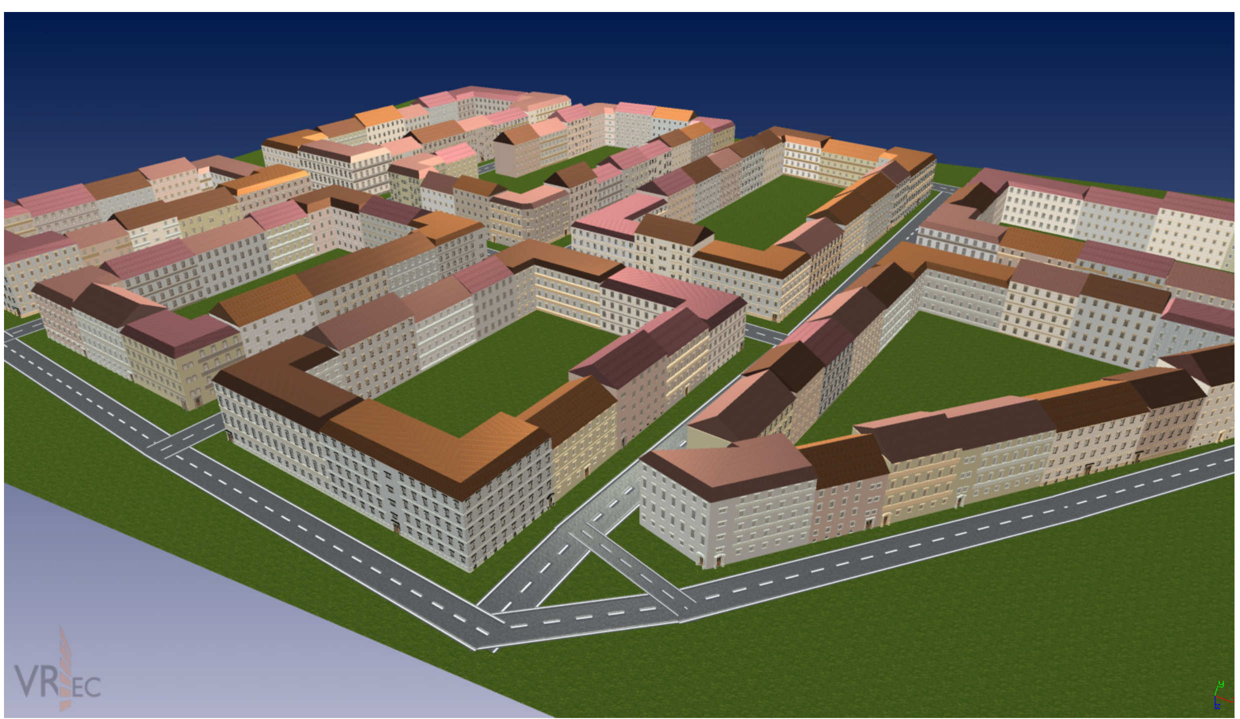


Abbildung 5.2: Prozedural erzeugtes Modell einiger Häuserblöcke um die Etkar-André-Straße

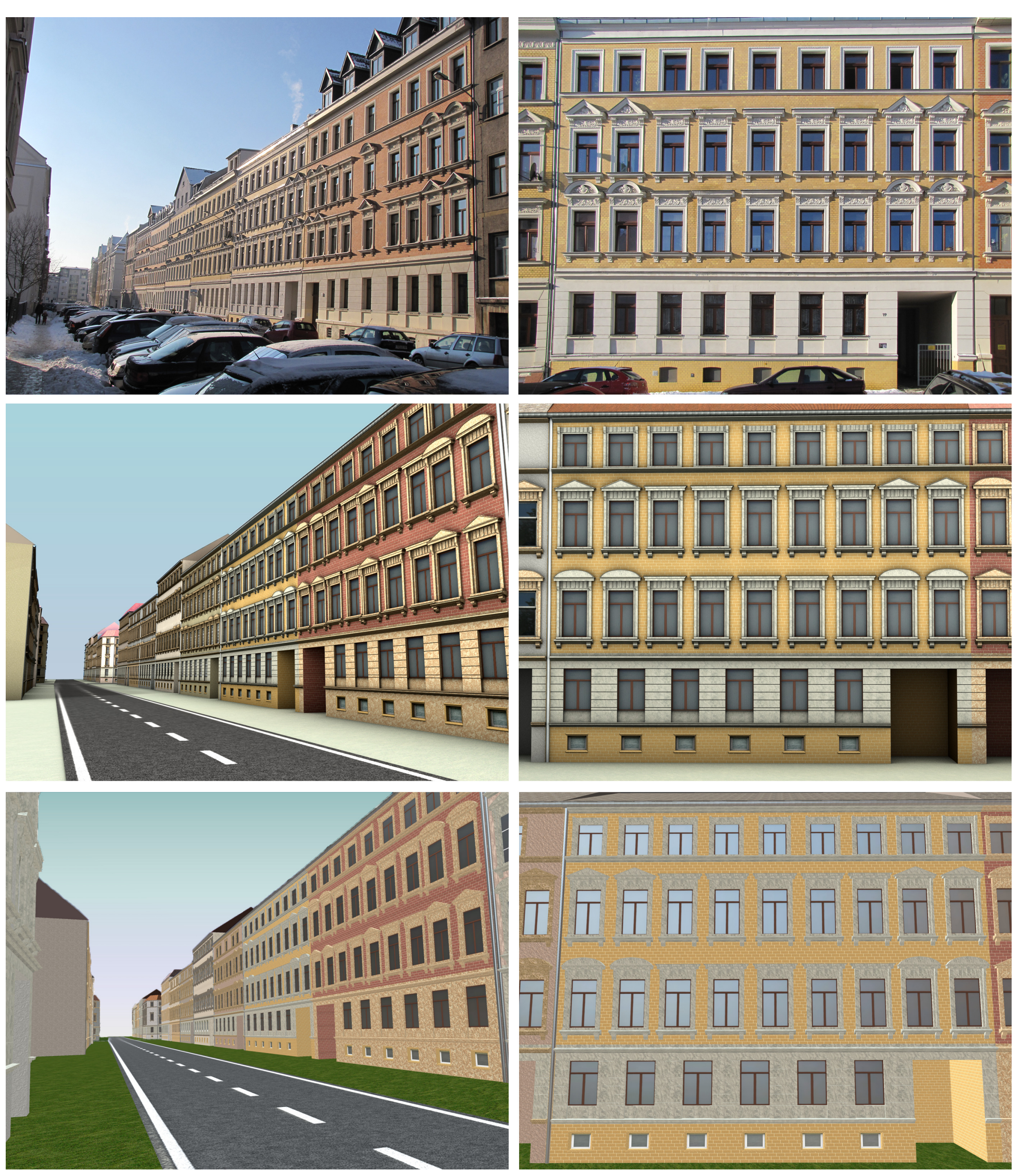


Abbildung 5.3: Oben: Foto mit Blick auf die Häuser der Etkar-André-Straße und eine zugehörige Fassade, Mitte: Prozedural erzeugtes Modell, das in Cinema 4D gerendert wurde, Unten: Dasselbe Modell in VRED als Echtzeitvisualisierung

Kapitel 6

Vergleich zwischen manueller und prozeduraler Modellierung

6.1 Vorstellung des City3D-Projektes

Im Rahmen des City3D-Projekts wurde bereits eine räumlich begrenzte Stadtszene von den ehemaligen UFZ-Mitarbeitern Michael Vieweg und Miguel Fonsecca in OpenSG visualisiert. Es handelt sich hierbei um mehrere Gebäudeblöcke an der Straßenkreuzung Eisenbahn-, Rosa-Luxemburg- und Lutherstraße in Leipzig. Abbildung 6.1 zeigt eine Luftaufnahme des Gebietes.

Hierbei wurde eine manuelle Modellierungsmethode gewählt. Die Gebäudegrundrisse sowie die Lage der Straßen wurden aus einem Geoinformationssystem entnommen. Die Fassaden der echten Gebäude wurde fotografiert. Aufgrund des Standortss der Fotoaufnahmen auf der Straße kommt es zu einer räumlichen Perspektive und Verzerrung der Fassaden. Diese Verzerrung musste in einem Bildbearbeitungsprogramm korrigiert werden, um die Fotos als Fassadentexturen verwenden zu können. Durch die Neuberechnung und Neuinterpolation des Fotos kommt es dabei zu einer Qualitätsminderung. Außerdem mussten eventuelle auf den Fotos vorhandene Hindernisse wie z.B. Straßenlaternen, Autos und Straßenbahnoberleitungen aus den Fotos entfernt werden. Die Gebäudemodelle wurden als einfache Geometrien in Cinema 4D erzeugt und mit den bearbeiteten Fotos als Texturen versehen. Die Straßengeometrie wurde ebenso von Hand erstellt und mit passenden Texturen versehen. Anschließend wurde die Szene in das VRML-Format exportiert und in VRED eingelesen. In VRED wurden die Häuserknoten jeweils über OpenSG-*Switch*-Knoten mit weiteren Texturen versehen. Damit wurde ermöglicht, in einer OpenSG-Anwendung die Häuser auszuwählen und deren Texturen zu wechseln, um einen Alterungsprozess in der Stadtszene visualisieren zu können.

6.2 Die City3D-Stadtszene mit dem CityGenerator erstellen

Um die Stadtszene mit dem CityGenerator erstellen zu können, wird eine XML-Szenendefinition benötigt, die in dem in Kapitel 2.1.2 beschriebenem Format vorliegt. Diese XML-Szenendefinition kann von Hand geschrieben werden, indem man sich Kartenmaterial oder Luftbildaufnahmen des entsprechenden Gebietes zur Hand nimmt, diese in ein Koordinatenraster einteilt, Gebäudeblock- und Gebäudeeckpunkte anlegt und dann jedes Gebäude

Abbildung 6.1: Der vom City3D-Projekt visualisierte Stadtbereich ist hell hervorgehoben, Quelle: 95

in die XML-Datei mit den entsprechenden Koordinaten einträgt. Dieses Vorgehen kann jedoch zeitaufwendig und vor allem fehlerbehaftet sein. Aus diesen Gründen wurde nach einem weiteren Weg gesucht, die Szenendefinition zu erhalten.

6.2.1 Importieren von Daten aus OpenStreetMap

OpenStreetMap[OSM 09b] (Abk.: OSM) stellt geographische Daten, wie z.B. Straßenkarten, zur freien Verfügung und stellt somit eine Art Open-Source-Variante von Google Maps[Inc. 09] dar. OSM basiert ebenfalls auf einem einfachen XML-Format. Die CityData-Klasse wurde dahingehend geändert, daß sie auch Daten in diesem Format einlesen kann. Das OSM-Format besteht aus Punkten mit Raumkoordinaten im geodätischen Koordinatensystem, die über IDs von so genannten *Way*- und *Area*-Elementen referenziert werden. Des Weiteren kann jedes Element Schlüssel-Wert-Paare in Form von untergeordneten Tag-Elementen beinhalten. Way-Elemente verbinden mindestens zwei Punkte. Area-Elemente sind eigentlich nur Way-Elemente deren Punkte einen geschlossen Zyklus ergeben, d.h. der erste Punkte ist gleich dem letzten Punkt. Area-Elemente sind somit nur Pseudoelemente und repräsentieren Flächen.

Diese Elemente müssen nun auf die Datenstrukturen Point, Street, Block und Lot der CityData-Klasse abgebildet werden.

OSM-Punktelemente können nach Umwandlung der geodätischen Koordinaten in lokale ENU-

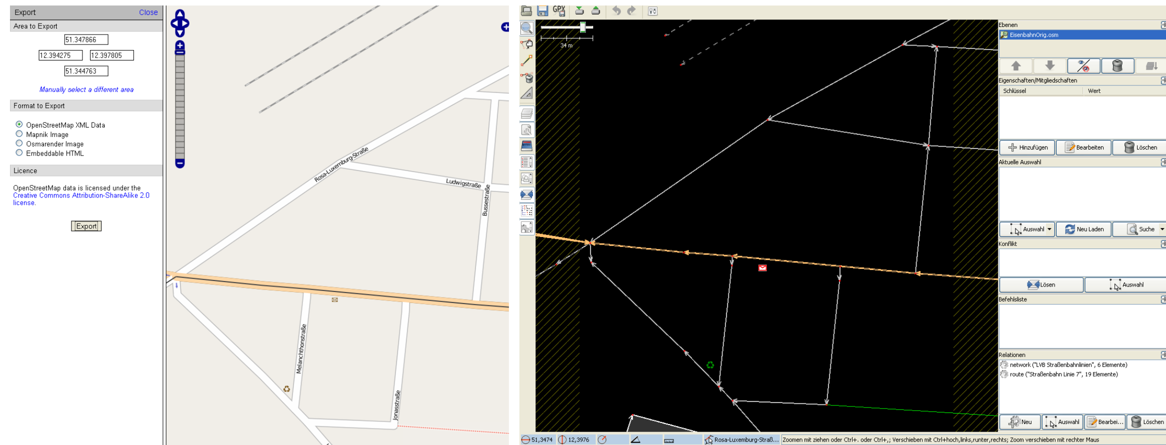


Abbildung 6.2: Links: Daten aus OpenStreetMap-Website exportieren, Rechts: JOSM-Fenster mit den geladenen OSM-Daten der Eisenbahnstraße

Koordinaten in Point-Elemente umgewandelt werden. Geodätische Koordinaten beschreiben einen Punkt auf der Erdoberfläche durch einen Breiten- (engl.: *latitude*) und einen Höhengrad (engl.: *longitude*). Die Umwandlung der Koordinaten wurde in der Hilfsklasse *CoordConverter* implementiert. Dazu wurde auf den frei verfügbaren Quellcode des Programms *GeoTrans*[Agency 09] zurückgegriffen. Mit GeoTrans können Koordinaten zwischen den verschiedensten Koordinatensystemen umgewandelt werden. Vor der Umwandlung muss der Koordinatenursprung des lokalen ENU-Koordinatensystems einem Punkt in geodätischen Koordinaten zugeordnet werden. Dazu wird immer der erste in der OSM-Datei vorkommende Punkt verwendet.

Anschließend werden alle Way-Elemente ausgelesen. Die enthaltenen Tag-Elemente werden ebenso ausgelesen. Wenn ein Way-Element das Tag *highway* enthält, so wird dieses in Street-Objekte überführt. Dabei werden $Punktanzahl - 1$ Street-Objekte generiert, indem immer zwei Punkte ein Street-Objekt bilden. Ist ein *width*-Tag vorhanden, so wird die Straßenbreite entsprechend dem Tag gesetzt.

Enthält ein Way-Element das Tag *landuse* mit dem Wert *residential*, so wird daraus ein Block-Objekt erzeugt. Die Tag-Elemente werden dabei in Einträge in der Membervariable *buildingSettings* des Block-Objekts überführt. Somit können für einzelne Gebäudeblöcke individuelle Eigenschaften bereits im OSM-Format festgelegt werden, auf die der Ruleprocessor und der BuildingGenerator dann Zugriff hat (z.B. könnte man das Tag *BuildingHeight* hinzufügen, um den BuildingGenerator anzuweisen, Gebäude mit dieser Höhe zu generieren).

Enthält ein Way-Element jedoch das Tag *building* mit dem Wert *yes* so wird ein Lot-Objekt erzeugt. Auch hier werden wieder die Tags in die Lot-Membervariable *buildingSettings* überführt. Schließlich werden die erzeugten Objekten im CityData-Objekt gespeichert.

6.2.2 Generieren der OpenStreetMap-Daten

Die benötigten OSM-Daten können direkt aus der OpenStreetMap-Website exportiert werden. Dazu passt man die Darstellung der Karte so an, dass der gewünschte Ausschnitt im Browser sichtbar ist und wählt anschließend unter *Export* die Option *OpenStreetMap XML Data* (siehe

Abbildung 6.2 links). Die Daten werden in einer XML-Datei gespeichert. Diese kann mit dem kostenlos verfügbaren OSM-Editor JOSM[OSM 09a] weiter bearbeitet werden. Dieser stellt die Elemente grafisch dar und stellt Werkzeuge zur Bearbeitung zur Verfügung. Man kann Elemente löschen oder Neue zeichnen (siehe Abbildung 6.2 rechts).

Abbildung 6.3: Links: Die bereinigten OSM-Daten der Eisenbahnstraße und Luftbildaufnahmen im Hintergrund, Mitte: hinzugefügte Gebäudeblöcke, Rechts: hinzugefügte Gebäudegrundrisse

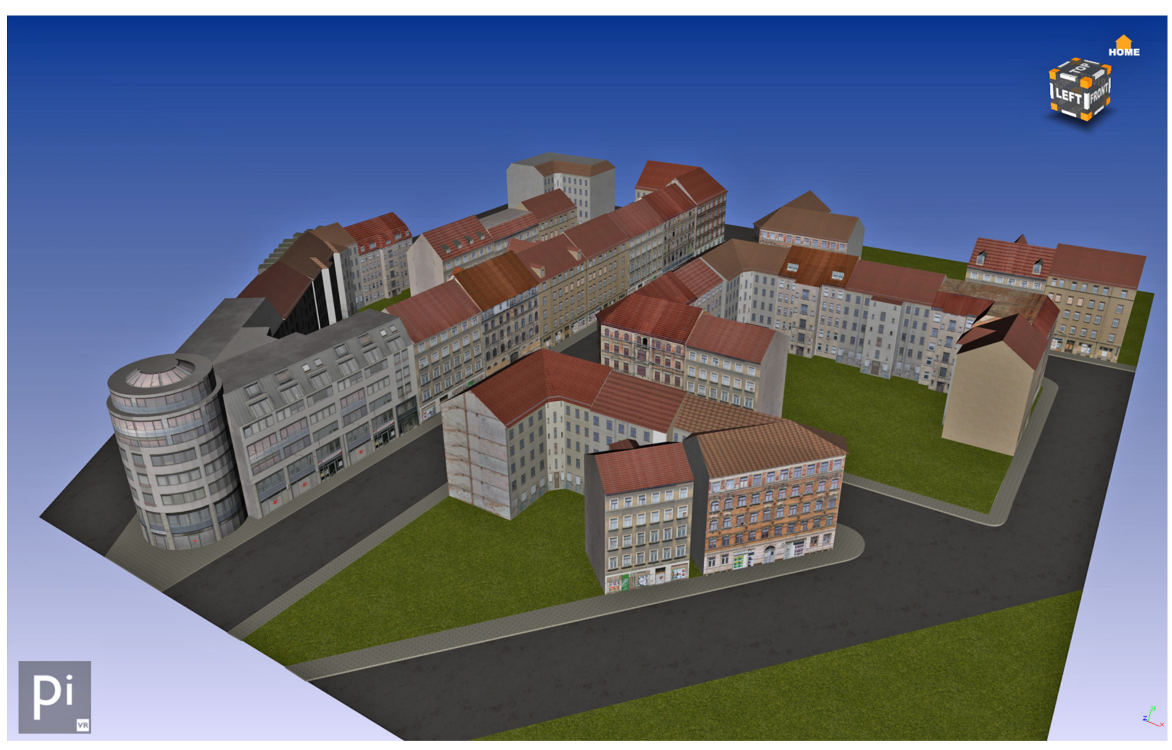
Eine hilfreiche Funktion ist die Möglichkeit, Luftbildaufnahmen zu laden und als Hintergrund im Editierfenster zu verwenden. Das *Yahoo-WMS*-Plugin bietet diese Möglichkeit. Die Auflösung der Luftaufnahmen ist zwar recht gering, sie reicht jedoch aus, um Korrekturen und Anpassungen an den OSM-Objekten vornehmen zu können. So können z.B. nicht benötigt Objekte entfernt und der Straßenverlauf angepasst werden. Abbildung 6.3 links zeigt die bereinigten OSM-Daten.

Dem OSM-Datensatz müssen nun noch Gebäudeblöcke und ggf. Gebäudegrundrisse hinzugefügt werden. Um Gebäudeblöcke zu erstellen, legt man im JOSM-Editor eine geschlossene Punktlinie an, markiert diese und weist ihr das Tag *landuse* mit dem Wert *residential* zu. Für die Stadtszene werden drei Blöcke wie in Abbildung 6.3 mitte dargestellt, angelegt. Man könnte die OSM-Datei nun abspeichern und mithilfe des CityGeneratorCL-Programms daraus das 3D-Modell erzeugen. Für ein realitätsnaheres Ergebnis sollten jedoch noch die Gebäudegrundrisse in JOSM mithilfe der Luftbildaufnahmen nach gezeichnet werden. Diese müssen abschließend ausgewählt werden und das Tag *building* mit dem Wert *yes* zugewiesen werden (alternativ über *Menü→Vorlagen→Gebäude→Gebäude*). Hiernach ergibt sich die Grundrissanordnung wie in Abbildung 6.3 rechts dargestellt.

6.3 Vergleiche

Beide Modellieransätze haben ihre Vor- und Nachteile, meistens sind die Vorteile der einen Methode die Nachteile der anderen Methode. Die Methode der Wahl ist entscheidend von der Größe der zu visualisierenden Stadtszene abhängig. Während die manuelle Modellierung für kleine Stadtgebiete noch handhabbar ist, aus entfernterer Betrachtung auch recht realistisch wirkt und die Möglichkeit bietet, individuelle Gebäudegeometrien zu realisieren, ist sie jedoch für größere Gebiete viel zu zeitaufwendig. Außerdem bietet sie zu wenig Details aus näherer Betrachtung.

Die vorgestellte prozedurale Modellierung bietet, insofern die Regeldefinitionen, 3D-Geometrien der einzelnen Fassadenelemente und Texturen vorliegen, eine sehr schnelle Ge-



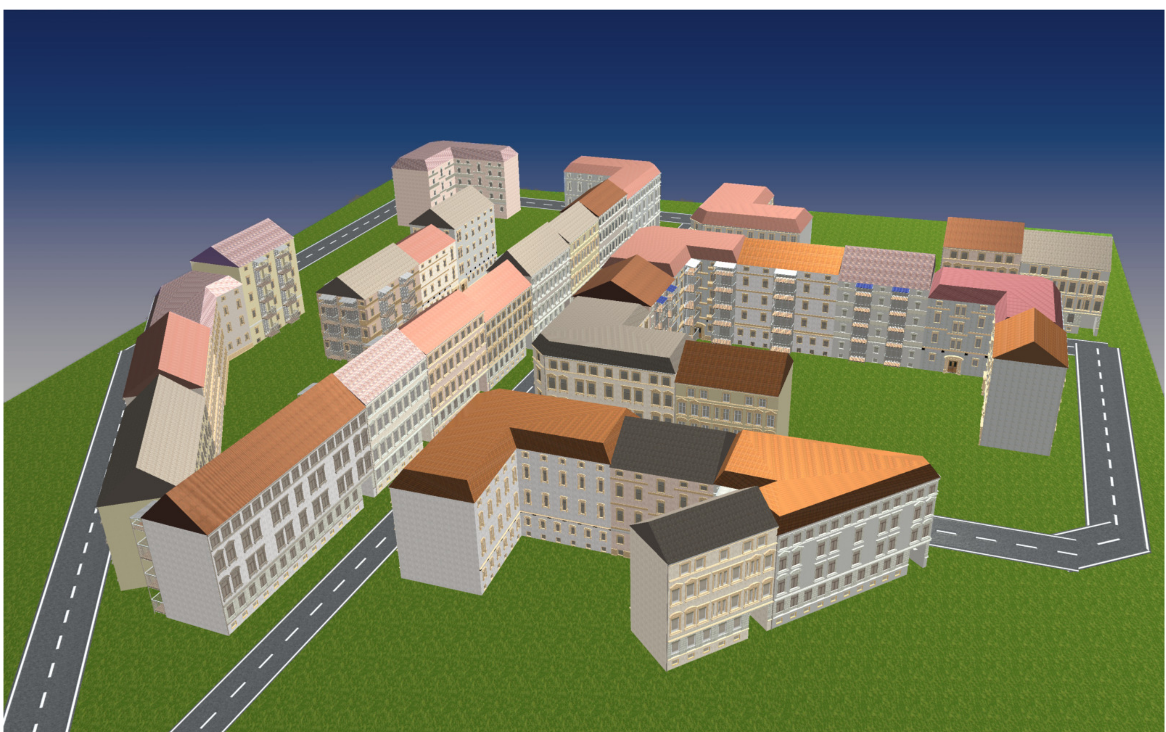


Abbildung 6.4: Die City3D-Stadtszene in VRED, Oben: manuell modelliert, Unten: prozedural erstellt

Abbildung 6.5: Detailansicht eines Straßenzuges im City3D-Projekt, Oben: manuell modelliert, Unten: prozedural erstellt

nerierung und je nach dem, wie komplex die Regelfedinitionen und die Anzahl und Güte der verwendeten 3D-Geometrien ausfallen, eine detaillierte Darstellung auch bei näherer Betrachtung. Die vorgestellte Stadtszene ist je nach Rechnergeschwindigkeit in höchstens zwei Minuten generiert und kann durch die Verwendung zufallsgesteuerter Regelverzweigungen und Variablen in kürzester Zeit variiert werden. Auch die Performance der Echtzeit-Visualisierung wird durch das LOD-System zu interaktiven Frameraten sichergestellt. Jedoch erreichen die Fassadentexturen der niedrigen LOD-Stufe nicht die Qualität der Fototexturen des manuellen Ansatzes. Des Weiteren können keine Gebäude mit nicht-polygonförmigen Grundriss und mit etagenweise sich verändernden Grundrissen erstellt werden. Die generierten Gebäudemodelle kommen ihren echten Vorbildern vom Eindruck her nahe, allerdings entsprechen sie jedoch nicht in jedem Detail den Original-Häusern. Schließlich wird auch für die Erstellung der Regeldefinitionen und der 3D-Geometrien einige Zeit benötigt. Sind diese allerdings erst einmal erstellt, so lassen sie sich beliebig weiter verwenden. Die Abbildungen 6.4 und 6.5 zeigen die Ergebnisse der beiden Methoden im Vergleich.

Das beste Ergebnis kann wahrscheinlich nur durch Kombination der beiden Ansätze erzielt werden. So könnte der Großteil der Stadtszene prozedural erzeugt werden und einige wenige markante Gebäude von Hand modelliert werden und diese zu einer Szene vereinigt werden. So kann man sich die Vorteile der beiden Ansätze zugleich zunutze machen.

Kapitel 7

Ausblick

Im Rahmen der Arbeit wurde ein Software-System erarbeitet, das virtuelle Modelle von Gebäudefassaden und Stadtgebieten mithilfe einer prozeduralen Methode ausgehend von Regeln zur Fassadengestaltung und einfachen Gebäude-Bausteinen generiert. Die Stadtszene kann dabei per Hand oder ausgehend von OpenStreetMaps definiert werden. Durch die Implementierung des gesamten Systems in OpenSG und durch ein einfaches LOD-Systems ist die Lauffähigkeit am Visualisierungszentrum des UFZ sichergestellt. Die erzeugten Stadtmodelle können durch den VRML-Export in Modellierungsanwendungen weiterverarbeitet werden. Insgesamt ergibt sich der in Abbildung 7.1 dargestellte Workflow.

7.1 Problemfelder

Der CityGeneratorCL stößt bei einem Speicherbedarf von ca. 1.8 GByte an seine Grenzen und speichert dabei nur die bis dahin erzeugten 3D-Modelle. Abhilfe könnte hier geschaffen werden, indem sowohl die CityGenerator- als auch die OpenSG-Bibliothek mit einem 64Bit-Compiler übersetzt werden oder einen Mechanismus zu implementieren, der kurz vorm Ausreizen des zur Verfügung stehenden Speichers die bis dahin erstellte Szene in einer OpenSG-Binärdatei speichert, die erzeugten Geometrien aus dem Speicher löscht und mit der Stadtmodellgenenerierung fortfährt. Man würde dadurch mehrere Dateien für eine Szene erhalten, die sich aber z.B. mit VRED kombinieren ließen.

Der CityGenerator erstellt Fassadenmodelle nur auf Grundlage eines polygonförmigen Gebäudegrundriss. Die Fassaden selbst sind daher gerade und flach. Herausstehende oder hereinragende Strukturen sind nur umständlich zu realisieren. Interessant wäre daher ein CSG[1]-System einzusetzen, das aus einfachen Grundkörpern, wie z.B. Quadern, durch boolsche Operationen komplexere Körper zusammensetzt.

7.2 Möglichkeiten zur Weiterentwicklung

Es bieten sich eine Reihe an Möglichkeiten zur Weiterentwicklung der Anwendungen und des Software-Systems an.

[1]CSG - Constructive Solid Geometry, dt.: Festkörpergeometrie

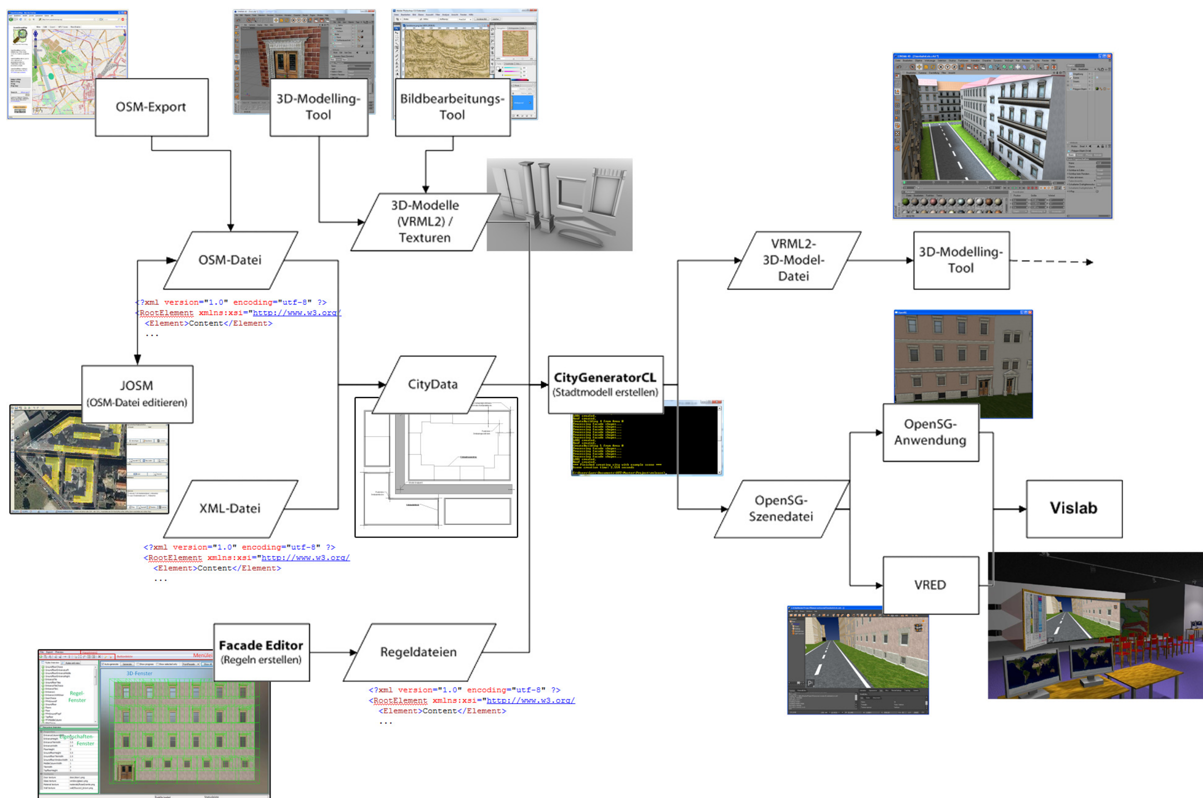


Abbildung 7.1: CityGenerator Workflow-Übersicht

Verbesserung der Benutzerschnittstelle zum Facade Editor

Der Facade Editor könnte komfortabler bedient werden, wenn z.B. Regeln im Regelfenster per Drag and Drop kopiert oder umgruppiert werden könnten. Außerdem könnten Regeln gruppiert dargestellt werden, um eine bessere Übersicht zu erhalten. Bisher dient das 3D-Fenster nur zur Betrachtung der von den erstellten Regeln erzeugten Fassade. Eine intuitive Anpassung der Regelparameter direkt in der 3D-Ansicht würde der Bedienbarkeit sehr zu Gute kommen.

Anbindung an GIS und Google Earth

Durch eine direkte Anbindung an ein Geoinformationssystem über einen Importfilter oder Ähnliches könnte man auf bereits vorhandene Kartendaten von Städten zurückgreifen, in denen teilweise auch die Gebäudegrundrisse eingezeichnet sind. Da diese Daten durch Vermessungen entstanden sind, sind sie sehr genau. Eine weitere Möglichkeit echte Stadtdaten verwenden zu können, wäre die Anbindung an Google Earth ebenfalls durch einen Importfilter, um die Straßendaten zu erhalten. Noch nützlicher wäre allerdings eine Exportfunktion, um erstellte Stadtmodelle nach Google Earth exportieren zu können.

Grafischer Editor für Szenendefinition

Es könnte ein grafischer Editor für die Erstellung und Modifikation der Szenendefinition entwickelt werden oder als Alternative dazu könnten weitere Informationen aus dem

OpenStreetMaps-Format, wie z.B. Punkte mit der Eigenschaft *natural = tree* für Bäume oder *model = model.wrl* für beliebige Modelle, ausgelesen werden, um den JOSM-Editor zur Erstellung der Szenendefinition zu verwenden.

Straßengeometrie-Generierung
Die bisherige Straßengenerierung erzeugt nur flache Rechtecke mit einer Straßentextur. Eine echte Straßengeometrie-Erzeugung mit Straßenkreuzungen, Bordsteinkanten und Gehwegen würde den Realitätsgrad des Stadtmodells deutlich erhöhen.

Bessere Texturengenerierung
Die verwendeten Texturen für die Fassadenmaterialien sind zwar kachelbar, lassen sich also nahtlos wiederholen, trotzdem sind bei entfernterer Betrachtungen periodische Muster erkennbar. Es gibt Methoden, z.B. die im Artikel *Tile Based Texture Mapping* aus [Phar 05] Vorgestellte, die nichtperiodische Texturen durch einen Shader erzeugen.

Regelverzweigung nach Häuservariante
Um den Anforderungen von städtischer Visualisierung und insbesondere der Visualisierung von Stadtentwicklung gerecht zu werden, wäre es z.B. sinnvoll, mehrere Varianten der einzelnen Häuser zu erzeugen. Dabei könnte die erste Variante bspw. ein normales Haus und die zweite Variante ein verlassenes, unsaniertes Haus (z.B. durch andere Texturen) repräsentieren. Dies könnte als weitere Regelverzweigung oder separate Regeldateien realisiert werden.

Bessere Texturen für einfache Häusermodelle
Die Texturen der einfachen Häusermodelle (der niedrigen LOD-Stufe) werden nur je nach Größe der Fassade enstsprechend oft gekachelt. Sie entsprechen in Form und Farbgebung jedoch nicht der prozedural erzeugten Häuser-LOD-Stufe, so dass das Umschalten zwischen den LOD-Stufen deutlich sichtbar ist. Ein deutlich besseres Ergebnis würde die Erzeugung von Fassadentexturen aus Abbildungen der erstellten prozeduralen Fassadenmodellen liefern.

Verbesserung der Grafik
Shader-Programme ermöglichen weiterführende Beleuchtungs- und Renderingtechniken, die mit der traditionellen Rendering-Pipeline nicht realisierbar sind. Ein Beispiel dafür sind so genannte Post-Processing-Shader, die auf dem fertig gerenderten Bild arbeiten. Eine vereinfachte Form von *Ambient Occlusion* kann als Shader implementiert werden, was als *Screen Space Ambient Occlusion* bezeichnet wird. Eine vereinfachte und nicht korrekte Version dieses Shaders wurde testweise in den SSMViewer nach [GameDev.Net 08] implementiert (siehe Abbildung 7.2). Dieser Shader benötigt den *Tiefenpuffer* (engl.: *depth-* oder *z-buffer*) der gerenderten Szene und berechnet eine Ambient Occlusion-Textur (siehe Abbildung 7.2 unten), die das normale Render-Ergebnis (siehe Abbildung 7.2 oben) an bestimmten Stellen, wie z.B. Kanten und Einschließungen, abdunkelt. Dadurch werden Geometriedetails durch Schattierungen besser sichtbar und das Ergebnis wirkt allgemeiner plastischer und weniger künstlich (siehe Abbildung 7.2 mitte). Die implementierte Methode ist jedoch nicht zur Verwendung geeignet, da sie blickwinkelabhängig zur Oberflächennormalen der Geometrien ist. Eine bessere Methode muss implementiert werden, die zusätzlich zum Tiefenpuffer auch die Oberflächennormalen im Bildraum als Textur benötigt.

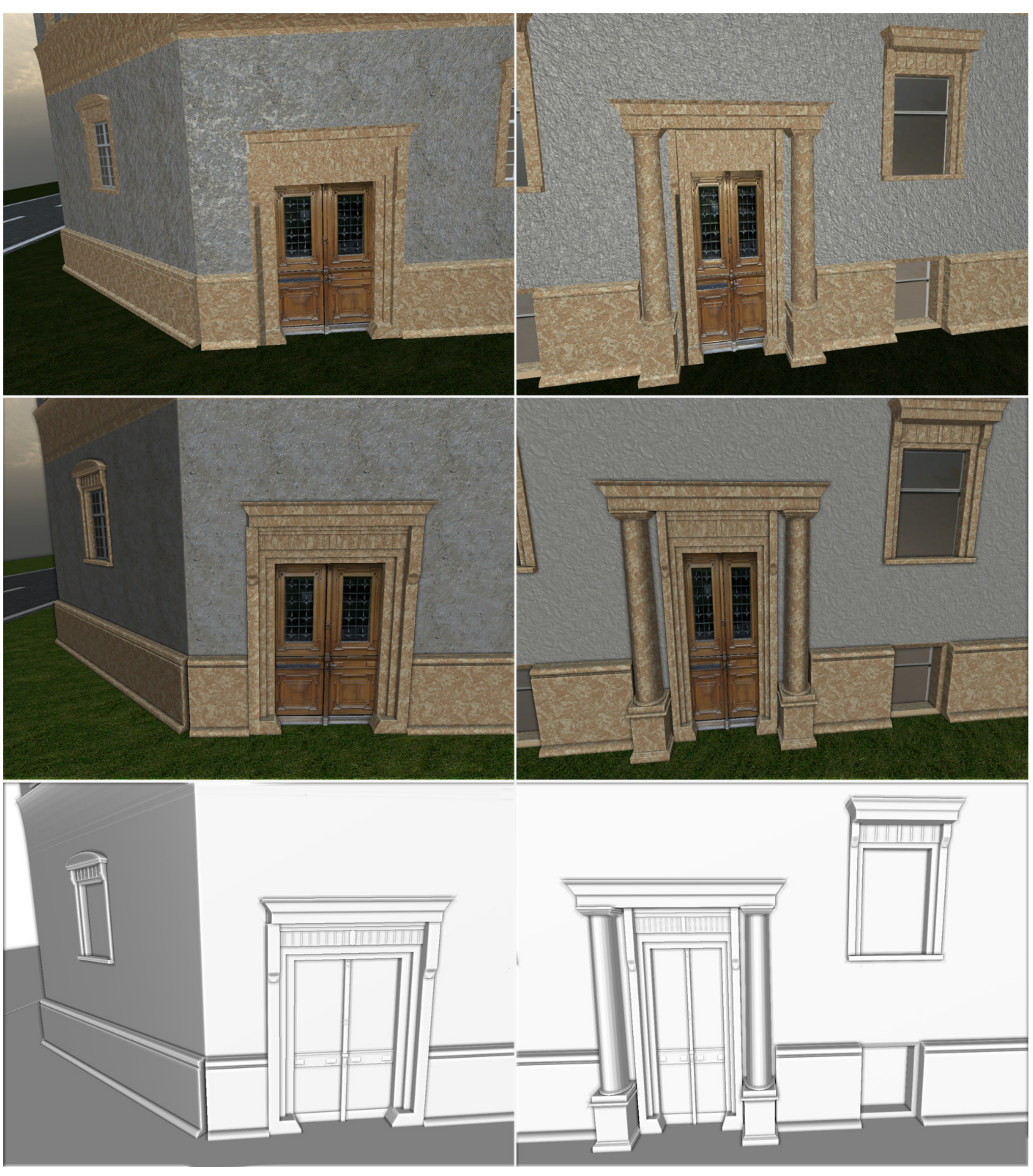
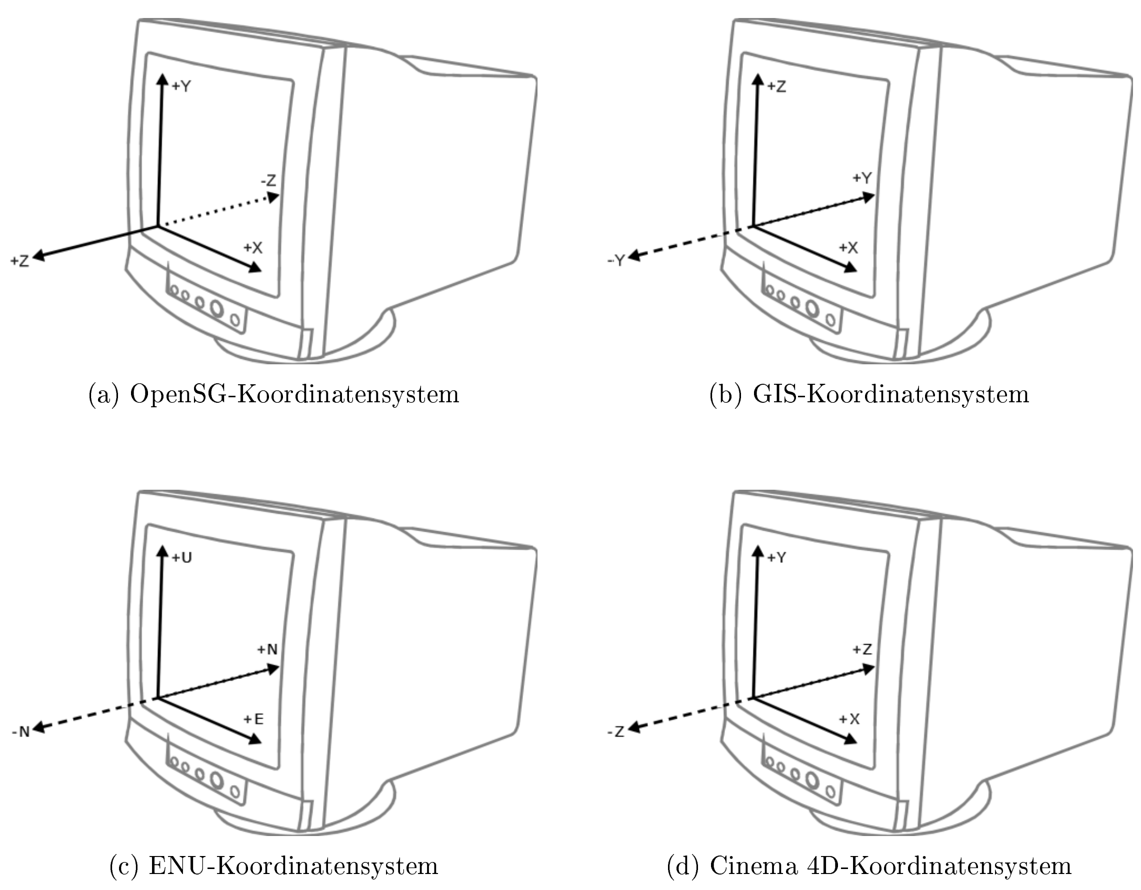


Abbildung 7.2: Oben: Normales Rendering, Mitte: mit Screen Space Ambient Occlusion-Rendering, unten: Ambient Occlusion-Textur

Anhang A

Koordinatensysteme



(a) OpenSG-Koordinatensystem (b) GIS-Koordinatensystem

(c) ENU-Koordinatensystem (d) Cinema 4D-Koordinatensystem

Abbildung A.1: Koordinatensysteme

Anhang B

Kompilieren des Visual Studio Projekts und Begleit-CD

Zum Kompilieren des Visual Studio-Projektes wird *Visual Studio 2005* oder das frei erhältliche *Visual Studio 2005 Express Edition*[Microsoft 08] benötigt. Als zusätzliche Bibliotheken müssen *WxWidgets 2.8.7*[wxWidgets 08], *Boost 1.35*[Boost 08] und *OpenSG 1.8*[OpenSG 08] installiert werden. Die Installationsverzeichnisse werden im Projekt über die anzulegenden Umgebungsvariablen „WXWIN“, „BOOST“ und „OPENSG“ angesprochen, d.h. diese Umgebungsvariablen müssen angelegt werden und auf die jeweiligen Bibliotheksverzeichnisse auf der Festplatte verweisen. Optional wird noch *VRPN 7.15*[VRPN 08] benötigt, wenn man die mit dem CityGeneratorCL erstellten Modelle mit Hilfe eines Eingabegeräts von *3DConnexion* (z.B. *Space Navigator* oder *Space Pilot*) betrachten möchte (Anwendung *SpaceNavigatorTest-SSM*). In diesem Fall ist eine weitere Umgebungsvariable „VRPN“ anzulegen, die ebenso auf das zugehörige Verzeichnis verweist. Außerdem wird der XML-Parser *TinyXML*[TinyXML 08] verwendet, der jedoch direkt in die Projektmappe als zugehöriges Projekt aufgenommen wurde. Die für die Ausführung der Programme CityGeneratorCL und FacadeEditor benötigten Dateien (z.B. Texturen und Geometrien) liegen im Projektunterordner *Resources/*.

Die der Masterarbeit beiliegende CD beinhaltet das komplette CityGenerator-Projekt sowie die Masterarbeit als PDF-Datei.

Literaturverzeichnis

[Agency 09] National Geospatial-Intelligence Agency. Geotrans. WWW: <http://earth-info.nga.mil/GandG/geotrans/>, 2009.01.10.

[Anthemion 09] Anthemion. Dialogblocks dialog editor for wxwidgets. WWW: <http://www.dialogblocks.com/>, 10.01.2009.

[ATI 09] ATI. Cubemapgen. WWW: <http://developer.amd.com/gpu/cubemapgen>, 10.01.2009.

[Bilke 07] Lars Bilke. Einbindung und Nutzung von 3DConnexion 6dof-Eingabegeräten für auf OpenSG-basierende Anwendungen in der 3D-Visualisierung. Ein Master-Projekt am Helmholtz-Zentrum für Umweltforschung GmbH - UFZ. 2008.10.06.

[Birch 02] P.J. Birch, S.P. Browne, V.J. Jennings, A.M. Day, D.B. Arnold. Rapid procedural-modeling of architectural structures. Proceedings of ACM SIGGRAPH, 2002.

[Blender.org 08] Blender.org. WWW: <http://www.blender.org/>, 19.12.2008.

[Boost 08] Boost. C++ libraries. WWW: <http://www.boost.org/users/download/>, 18.12.2008.

[Chen 08] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, Eugene Zhang. Interactive procedural street modeling. Proceedings of ACM SIGGRAPH, 2008.

[e-on software 08] e-on software. Solutions for natural 3d environments. WWW: <http://www.e-onsoftware.com/>, 19.12.2008.

[Ebert 98] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley. Texturing And Modelling - A Procedural Approach. Academic Press, San Diego, USA, second edition Auflage, 1998.

[Ericson 05] Christer Ericson. Real-Time Collision Detection. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann Publishers, 2005.

[Finkenzeller 06] Dieter Finkenzeller, Alfred Schmitt. Representation of complex façades using typed graphs. Proceedings of Virtual Concept, November 2006.

[GameDev.Net 08] GameDev.Net. Ssao - gamedev.net discussion forums. <http://www.gamedev.net/community/forums/topic.asp?topic_id=463075>, 12.12.2008.

[GIMP 09] GIMP. The gnu image manipulation program. WWW: <http://www.gimp.org>, 10.01.2009.

[Gockel 08] T. Gockel. Form der wissenschaftlichen Ausarbeitung. Springer Verlag, Heidelberg, 2008. Begleitende Materialien unter <http://www.formbuch.de>.

[Google 09] Google. Maps. WWW: <http://www.maps.google.com>, 12.01.2009.

[Greuter 03] Stefan Greuter, Jeremy Parker, Nigel Stewart, Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. GRAPHITE Proceedings, 2003.

[Inc. 08] Procedural Inc. 3d modeling software for urban environments. WWW: <http://www.procedural.com/>, 18.12.2008.

[Inc. 09] Google Inc. Google maps. WWW: <http://maps.google.de/>, 10.01.2009.

[Kelly 06] George Kelly, Hugh McCabe. A survey of procedural techniques for city generation. ITB Journal, 14, 2006.

[Legakis 01] Justin Legakis, Steven Gortler Julie Dorsey. Feature-based cellular texturing for architectural models. Proceedings of ACM SIGGRAPH, August 2001.

[Lipp 08] Markus Lipp, Peter Wonka, Michael Wimmer. Interactive visual editing of grammars for procedural architecture. Proceedings of ACM SIGGRAPH, 2008.

[Maxon 08] Maxon. The makers of Cinema 4D und BodyPaint 3D. WWW: <http://www.maxon.net>, 19.12.2008.

[Microsoft 08] Microsoft. Visual Studio 2005 Express Editions. WWW. <http://www.microsoft.com/express/2005/>, 18.12.2008.

[Müller 01] Pascal Müller. Design und Implementation einer Preprocessing Pipeline zur Visualisierung prozedural erzeugter Stadtmodelle. Diplomarbeit, ETH Zürich, 2001.

[Müller 06a] Pascal Müller, Tijl Vereenooghe, Peter Wonka, Iken Paap, Luc Van Gool. Procedural 3d reconstruction of puuc buildings in xkipché. Eurographics Symposium on Virtual Reality, Archaeology and Cultural Heritage, 2006.

[Müller 06b] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, Luc Van Gool. Procedural modeling of buildings. Proceedings of ACM SIGGRAPH, 2006.

[Müller 07] Pascal Müller, Gang Zeng, Peter Wonka, Luc Van Gool. Image-based procedural modeling of facades. Proceedings of ACM SIGGRAPH, 2007.

[OpenSG 08] OpenSG. WWW: <http://opensg.vrsource.org/trac>, 18.12.2008.

[OSM 09a] OSM. Java openstreetmap editor. WWW: <http://josm.openstreetmap.de/>, 10.01.2009.

[OSM 09b] OSM. Open street map. WWW: <http://www.openstreetmap.org/>, 10.01.2009.

[Parish 01] Yoav I. H. Parish, Pascal Müller. Procedural modeling of cities. Proceedings of ACM SIGGRAPH, August 2001.

[Phar 05] Matt Phar, Randima Fernando. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, 2005. Online verfügbar: <http://http.developer.nvidia.com/GPUGems2/gpugems2_

part01.html>.

[Prusinkiewicz 96] Przemyslaw Prusinkiewicz, Aristid Lindenmayer. The Algorithmic Beauty Of Plants. Springer Verlage, New York, USA, 1996.

[Rost 06] Randi J. Rost. OpenGL(R) Shading Language (2nd Edition). Addison-Wesley Professional, January 2006.

[Shreiner 08] Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis. OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2.1 (6th Edition). Addison-Wesley Professional, 2008.

[Systems 09] Rendering Systems. Shader map. WWW: <http://www.shadermap.renderingsystems.com/>, 10.01.2009.

[Technologies 08] Liquid Technologies. Liquid XML Studio. WWW: <http://www.liquid-technologies.com/>, 18.12.2008.

[TinyXML 08] TinyXML. WWW: <http://www.grinninglizard.com/tinyxml/>, 19.12.2008.

[VRPN 08] VRPN. Virtual Reality Peripheral Network. WWW: <http://www.cs.unc.edu/Research/vrpn/>, 18.12.2008.

[Wikipedia 09] Wikipedia. Phong shading. WWW: <http://en.wikipedia.org/wiki/Phong_shading>, 13.01.2009.

[Wonka 03] Peter Wonka, Michael Wimmer, François Sillion, William Ribarsky. Instant architecture. Proceedings of ACM SIGGRAPH, Seiten 669 – 677, Juli 2003.

[Worley 96] Steven Worley. A cellular texture basis function. Proceedings of ACM SIGGRAPH, 1996.

[Wright 07] Richard S. Wright, Benjamin Lipchak, Nicholas Haemel. OpenGL SuperBible: Comprehensive Tutorial and Reference. Addison-Wesley Professional, 4. Auflage, 2007.

[wxWidgets 08] wxWidgets. WWW: <http://www.wxwidgets.org/downloads/>, 18.12.2008.