Der Semantic Building Modeler - Ein System zur prozeduralen Erzeugung von 3D-Gebäudemodellen

Inauguraldissertation
zur Erlangung des Doktorgrades
der Philosophischen Fakultät
der Universität zu Köln
im Fach Informationsverarbeitung

vorgelegt von
Patrick Gunia

Köln, den 22.07.2013

Datum der Defensio am 06.11.2013

1. Referent: Prof. Dr. Manfred Thaller
2. Referent: Prof. Dr. Reinhard Förtsch
3. Referent: Prof. Dr. Ulrich Lang

1 Inhaltsverzeichnis

2 Einleitung

Die Erzeugung von virtuellen Städten innerhalb von Computersystemen ist ein Aufgabenbereich, der in verschiedenen Bereichen eine wichtige Rolle spielt. Bereits seit Längerem befassen sich die Film- und Computerspielindustrie mit dieser Thematik. In der Filmproduktion nimmt die Bedeutung computergenerierter Bilder seit vielen Jahren stetig zu, der Einsatz moderner 3D-Modellierungstechnologien ermöglicht die Erschaffung realistischer Welten und deren Darstellung innerhalb beliebiger Filmszenarien. Solche computergenerierten Bilder sind im Vergleich zu alternativen Verfahren, beispielsweise dem maßstabsgetreuen Nachbau von Städten in Miniaturmodellen, vergleichsweise preiswert und bieten darüber hinaus noch weitere Vorteile, bsw. eine schnellere und kostengünstigere Adaptierbarkeit des erstellten Modells oder die einfache Erstellung beliebiger Kamerafahrten und Sichtperspektiven in der Szene selber. Bei Computerspielen ist die Bedeutung solcher 3D-Modelle noch größer, da bei deren Produktion keine Alternative zu computergenerierten Modellen zur Verfügung steht. Die Welt, die der Spieler durchläuft, muss vollständig im Computer erzeugt worden sein, damit auch ein solcher sie darstellen kann. Zusammenfassend ist die Unterhaltungsindustrie, speziell in Form von Film und Computerspiel, schon seit vielen Jahren ein wichtiger Einsatzbereich von computergenerierten Modellen, seien es nun Lebewesen, Landschaften, Städte oder ganze Welten.

Neben solchen Anwendungen, die auf die Unterhaltung der Nutzer ausgerichtet sind, gewinnen weitere Anwendungsbereiche zunehmend an Bedeutung. Innerhalb eines pädagogischen Umfeldes kann man sich eine Verwendung von 3D-Modellen von Städten vorstellen, um Schülern innerhalb des Geschichtsunterrichts ein realistisches Bild von den Lebensumständen bsw. der Menschen im alten Rom oder im Mittelalter zu vermitteln. Die Möglichkeit, eine virtuelle Stadt aus einer beliebigen Epoche am Computer zu durchwandern, könnte hier das Verständnis geschichtlicher, wirtschaftlicher und politischer Entwicklungen erleichtern, da es ein besseres Nachempfinden des täglichen Lebens der jeweiligen Epoche gestattet. Anwendungen dieser Art sind nicht nur auf den Schulunterricht oder das universitäre Umfeld beschränkt, sondern könnten auch bei Ausstellungen und in Museen Besuchern einen neuen Zugang zum Erleben von Geschichte vermitteln. Hier ist bereits eine Reihe von Projekten zu nennen, die sich damit befassen, historische Städte im Computer neu zu erschaffen.

Das *Rome Reborn*-Projekt [Ro10] ist eines der bekanntesten Projekte in diesem Bereich. Es befasst sich mit der Modellierung sowohl der antiken Stadt Rom als auch ihrer Stadtentwicklung beginnend in der späten Bronzezeit bis ins frühe Mittelalter. Das *Cyberwalk*-Projekt ist ein von der Europäischen Union gefördertes Forschungsprojekt, welches als Zielsetzung die „Entwicklung einer vollkommen neuartigen virtuellen Laufumgebung“ [Fa10] verfolgt, „die es der Versuchsperson ermöglicht, sich aktiv und ungehindert in verschiedene Richtungen durch virtuelle Welten zu bewegen“. Als Testszenario wurde ein Stadtmodell des historischen Pompei erzeugt, welches mit der entwickelten Hardware, der sogenannten *omni-directional treadmill* erkundet werden kann.

Wie bereits diese Beispiele erahnen lassen, handelt es sich auch bei der Nachbildung historischer und moderner Städte um einen Bereich mit intensiver Forschung was die Entwicklung von Soft- und Hardware angeht.

Auf die unterschiedlichen Technologien, die zur Erzeugung von Gebäudemodellen bis hin zur Erzeugung vollständiger virtueller Welten zur Verfügung stehen, wird im Abschnitt „Verwandte Arbeiten - Technologien zur Erstellung von 3D-Gebäudemodellen“ detailliert eingegangen. Für einen ersten Einstieg in die Thematik soll darum eine grobe Einteilung der vorhandenen Ansätze in prozedurale und nicht-prozedurale Technologien genügen.

Der Unterschied zwischen diesen beiden Technologiegruppen sei am Beispiel von 3D-Gebäudemodellen erläutert. Der klassische Ansatz basiert auf dem Einsatz von 3D-Modellierungswerkzeugen, in denen der Gebäudedesigner sämtliche Bestandteile des Gebäudes beginnend mit dem Grundriss bis hin zu Fenster-, Tür- oder Gesimsemodellen manuell erzeugt und anschließend zusammenfügt, um ein fertiges Gebäude zu erstellen. Jeder Bearbeitungsschritt muss durch den Nutzer festgelegt und ausgeführt werden. Der Computer stellt durch das Modellierungsprogramm nur das Werkzeug bereit, mit dem die Gebäude gestaltet werden

Prozedurale Ansätze dagegen basieren auf komplexen Algorithmen, mittels derer ein Computersystem in der Lage ist, Modelle selbstständig zu generieren, im Idealfall ohne dass ein Eingreifen des Nutzers erforderlich ist. Der Vorteil solcher Verfahren liegt auf der Hand. Wenn ein Computer in der Lage ist, eine Vielzahl von Gebäuden basierend auf nutzerdefinierten Parametern automatisch zu erstellen, spart dies speziell im Kontext der Generierung umfangreicher virtueller Welten sehr viel Zeit und somit auch Geld. Mit der

wachsenden Größe und Komplexität virtueller Welten sowohl in Filmen wie auch in Computerspielen gewinnt dieser Faktor zunehmend an Bedeutung.

Das Forschungsfeld der prozeduralen Inhaltsgenerierung ist dabei sehr umfangreich. Speziell im Zusammenhang mit Computerspielen kann man verschiedene Forschungsschwerpunkte voneinander abgrenzen. Ein Beispiel ist hierbei die Fragestellung, ob die virtuellen Welten *on-the-fly* oder offline erstellt werden. On-the-fly-Technologien erzeugen Gebäude erst kurz bevor sie für den Nutzer sichtbar werden. Bei offline erstellten Welten ähnelt der Produktionsprozess dagegen dem auch heute noch weit verbreiteten Modellieren der Weltkomponenten durch einen 3D-Designer, nur werden diese nicht mehr durch einen menschlichen Nutzer manuell erzeugt, sondern prozedural durch den Computer.

Verfahren zur prozeduralen Inhaltserzeugung sind dabei nicht nur auf die Berechnung von Gebäude- und Stadtmodellen beschränkt, sondern eignen sich für eine Vielzahl unterschiedlicher Bereiche. Gute Beispiele sind die Generierung von Vegetation unter Verwendung theoretischer Formalismen oder die geometrieabhängige, automatische Erzeugung von Texturen. Innerhalb der vorliegenden Arbeit werden verschiedene prozedurale Technologien vor- und anschließend einander gegenübergestellt. Unabhängig von den zugrundliegenden Algorithmen sehen viele Entwickler die „Zukunft der Spieleentwicklung in der Automatisierung, in der prozeduralen Generierung von Inhalten“ [Ch12].

Gegenstand der vorliegenden Arbeit ist die Konzeption und Umsetzung des *Semantic Building Modelers*, eines Systems, dessen Aufgabe in der automatischen Offline-Produktion von Gebäudemodellen besteht. Im Unterschied zu anderen bereits vorhandenen Softwarewerkezeugen liegt der Fokus des Semantic Building Modelers darauf, speziell unerfahrenen Nutzern durch die Verwendung semantischer Beschreibungsparameter den Einstieg in die prozedurale Gebäudeerzeugung zu erleichtern. Die semantische Zwischenebene versteckt die geometrische und algorithmische Komplexität vor dem Nutzer und soll dadurch die Einstiegshürden möglichst niedrig halten. Darüber hinaus zeichnet sich der Semantic Building Modeler durch seine Fähigkeit aus, automatisch Variationen in die berechneten Gebäude zu integrieren, so dass auch bei gleicher Beschreibung die erzeugten Gebäude unterschiedlich sind. Die hierfür entwickelten Algorithmen, die Struktur des Systems und die Möglichkeiten, die es dem Nutzer bietet, sind Gegenstand des zweiten Teils dieser Arbeit. Der erste Teil „Basistechnologien“ gibt zunächst einen Überblick über verschiedene zentrale Konzepte und Technologien, die für die Entwicklung des Semantic

Building Modelers eine Rolle spielen, deren Einsatzzweck allerdings nicht auf die prozedurale Inhaltsgenerierung beschränkt ist, sondern die auch in verschiedenen anderen Bereichen der Informatik relevant sind. Anschließend wird im Abschnitt „Basistechnologien für die prozedurale Inhaltsgenerierung“ der Fokus auf die unterschiedlichen Möglichkeiten zur Erzeugung von 3D-Modellen im Allgemeinen und von 3D-Gebäudemodellen im Speziellen gelegt, bevor das vorliegende System ausführlich vorgestellt wird.

3 Basistechnologien

Der folgende Abschnitt befasst sich mit einer Reihe von Basistechnologien, die unabhängig von ihrer Bedeutung für das vorliegende System auch abseits der prozeduralen Inhaltsgenerierung von Belang sind. Dazu gehören unter anderem unterschiedliche Formen der Repräsentation von 3D-Modellen in der Computergraphik, Formalismen wie L-Systeme und die darauf aufbauenden Shape-Grammatiken sowie grundlegende Eigenschaften und Unterscheidungsmerkmale moderner Programmiersprachen. Diese allgemeinen Themenbereiche werden zum Teil ausführlich vorgestellt, da das Verständnis der nachfolgenden Arbeiten ein Verständnis der zugrunde liegenden theoretischen Grundlagen erfordert. Nach der Erläuterung dieser Grundlagen werden die drei großen Technologien zur Gebäudeerzeugung zunächst allgemein vorgestellt und anschließend anhand ausgewählter Vertreter diskutiert. Abschließend werden die unterschiedlichen Systeme anhand verschiedener Kriterien miteinander verglichen und auch dem in dieser Arbeit entwickelten Semantic Building Modeler gegenübergestellt.

3.1 **Repräsentationsformen für 3D-Modelle**

In der Computergrafik existieren eine Reihe unterschiedlicher Techniken, mit denen 3D-Modelle gespeichert und dargestellt werden. Zunächst wird ein kurzer Überblick über die verschiedenen Repräsentationsarten gegeben. Hierbei handelt es sich um das Punkt-, das Volumen- und das Oberflächenmodell. Anschließend werden zwei verschiedene Arten von Oberflächenmodellen erläutert, bevor abschließend Datenstrukturen für Polygonnetze thematisiert werden. Polygonnetze sind die am weitesten verbreitete Form der Oberflächenmodelle und werden auch in dieser Arbeit für die Repräsentation der berechneten 3D-Gebäudemodelle eingesetzt.

3.1.1 **Repräsentationsmodelle**

3.1.1.1 **Punktmodelle**

Punktmodelle sind die einfachste Form der Repräsentation von 3D-Modellen. Bei diesen wird die Oberfläche eines Objektes durch eine Punktwolke dargestellt, die keinerlei Informationen über die topologische Beziehung der einzelnen Punkte zueinander verwaltet. Kanten oder Oberflächenelemente müssen zunächst aus einer solchen Wolke durch

geeignete Verfahren extrahiert werden, um Aufschluss über die tatsächliche Form des repräsentierten Objektes zu erhalten [Fi08].

3.1.1.2 **Volumenmodelle**

Volumenmodelle repräsentieren im Gegensatz zu den nachfolgend erläuterten Oberflächenmodellen nicht nur die Oberfläche eines Objektes, sondern den gesamten eingenommenen Raum. Die drei großen Vertreter der Volumenmodelle sind die *Constructive Solid Geometry* (CSG), das *Voxelmodell* und räumliche Unterteilungsschemata wie *Octrees* [BM05]. Kernbestandteil aller drei Formen ist die Verwendung dreidimensionaler Primitive. Bei der CSG können dies beliebige Volumenkörper wie beispielsweise Quader, Zylinder oder auch komplexere Grundkörper sein. Diese kombiniert man anschließend über boolesche Operatoren wie Vereinigung oder Schnitt und kann dadurch komplexe Objekte auf der Basis einfacher, kombinierter Grundkörper erzeugen.

Beim Voxelmodell besteht das primitive Grundobjekt aus einem Voxel (*volume element*), dem dreidimensionalen Pendant zum Pixel (*picture element*). Diese Voxel sind alle gleich groß und werden innerhalb eines dreidimensionalen Rasters angeordnet. In der einfachsten Variante speichert jedes Voxel nur, ob es Teil des Volumens ist, das durch den repräsentierten Körper eingenommen wird. Erweiterte Varianten können darüber hinaus analog zu Pixeln Farbwerte oder ähnliches kodieren. Offensichtlich hängt die Qualität der Repräsentation des Objektes durch das Voxelmodell von der Größe der Voxel ab. Je kleiner diese sind, desto exakter kann die Form des Objekts abgebildet werden, desto größer wird aber automatisch auch der Speicherbedarf der Repräsentation [BM05].

Octrees stellen eine Erweiterung des einfachen Voxelmodells dar. Ziel der Octrees ist eine adaptive Unterteilung des Objektraums derart, dass nicht ein einziges Primitiv fester Ausdehnung verwendet wird, sondern stattdessen die Größe variiert werden kann [Wa02]. Dadurch soll ein großer Nachteil des Voxelmodells, der in der Ineffizienz der Speicherplatznutzung besteht, behoben werden. Während nämlich im einfach Voxelmodell der gesamte zur Verfügung stehende Objektraum gleichmäßig in Voxel unterteilt wird, erfolgt eine solche Unterteilung bei Octrees als Funktion der Belegung des Objektraums durch ein Objekt. Sind große Bereiche des Raumes unbelegt, ist es nicht erforderlich, diesen durch eine Vielzahl von Voxeln zu repräsentieren. Um diesen Ansatz umzusetzen, wird darum bei Octrees der Objektraum rekursiv unterteilt, wodurch eine hierarchische

Raumstruktur entsteht, deren Wurzel den gesamten zur Verfügung stehenden Raum beschreibt. Innerhalb dieses Baumes besitzt jeder Knoten genau acht oder keinen einzigen Nachfolger. Im letzteren Fall handelt es sich um Blätter. Die Unterteilung erfolgt rekursiv. So lange ein durch einen Knoten beschriebener Raum durch das Objekt belegt ist, wird dieser Knoten weiter unterteilt. Ist nach der Unterteilung ein Knoten leer, terminiert die Rekursion für diesen Weg durch den Baum. Dadurch richtet sich die Auflösung des Baumes nach der Struktur des Objektes, das repräsentiert wird. Dies reduziert einerseits den Speicherverbrauch und erhöht andererseits die Detailgenauigkeit der Repräsentation, da durch die variable Größe der primitiven Grundobjekte auch feine Strukturen abgebildet werden können [BM05].

3.1.1.3 **Oberflächenmodelle**

Bei Oberflächenmodellen (engl. *Boundary Representation, B-Rep*) werden 3D-Objekte über ihre Oberflächen repräsentiert und im Gegensatz zu Volumenmodellen nicht durch den Bereich, den sie innerhalb des Objektraumes einnehmen [OM04]. Dabei stehen Punkte, Kanten und Oberflächenelemente zur Verfügung, um beliebige Objekte zu modellieren. Je nach Oberflächenmodell existieren Festlegungen bezüglich der Struktur der verwendeten Grundkomponenten, beim Polygonnetzmodell müssen beispielsweise sämtliche Oberflächenelemente planar sein, während dies bei der Verwendung bikubisch parametrischer Patches nicht der Fall ist. Da Oberflächenmodelle die am weitesten verbreitete Form der Repräsentation von 3D-Modellen sind und auch in dieser Arbeit die Wahl auf Polygonnetze fiel, werden nachfolgend zunächst die beiden wichtigsten Vertreter dieser Gruppe, die *Polygonnetzdarstellungen* und *bikubische parametrische Patches* erläutert. Anschließend werden verschiedene Datenstrukturen diskutiert, die zur Speicherung von Polygonnetzstrukturen eingesetzt werden, da das implementierte System eine Mischung aus verschiedenen Datenstrukturen einsetzt, um Geometrie und Topologie der erzeugten Modelle zu verwalten.

3.1.1.3.1 Polygonnetzdarstellungen

Die am weitesten verbreitete und geläufigste Repräsentationsform für 3D-Modelle ist die Polygon-Netzdarstellung. Bei diesem Verfahren nähert man Objekte durch eine Menge planarer Polygonflächen an. Die Genauigkeit dieser Annäherung kann beliebig variiert

werden, indem die Polygongröße angepasst wird. Handelt es sich um Objekte, deren Form durch planare Polygone exakt darstellbar ist, ist auch die Polygonnetzdarstellung exakt. Als Beispiel kann man einen Quader betrachten, dessen sechs Seitenflächen durch sechs Polygone repräsentiert werden können. Möchte man dagegen eine Kugel oder allgemein gewölbte Oberflächen durch planare Polygone beschreiben, ist dies nur näherungsweise möglich ist [OM04]. Mit wachsender Polygonanzahl verbessert sich die Qualität der Näherung, die Kugel wirkt runder. Reduziert man dagegen die Polygonanzahl und vergrößert somit die Fläche der einzelnen Polygone, wird die Kugel zunehmend „eckig“.

Die weite Verbreitung der polygonalen Objektdarstellung hat verschiedene Gründe. Zum einen handelt es sich um ein Verfahren, bei dem es möglich ist, beliebig komplexe Objekte aus einer Menge von Komponenten aufzubauen, die selber nur eine geringe Komplexität besitzen. Dabei sind die Polygone die einfachen Bausteine, die für die Objektmodellierung eingesetzt werden [Wa02]. Bei komplexen Szenen in Polygondarstellung wird die Anzahl der verwendeten Polygone sehr schnell sehr groß. Dies ist einer der Hauptnachteile dieser Repräsentationsform, da mit der Anzahl der darzustellenden Primitive zwangsläufig auch der Zeitaufwand steigt, der für Berechnungsverfahren erforderlich ist, die auf diesen Primitiven arbeiten. Die große Verbreitung dieses Ansatzes hängt damit zusammen, dass es eine Reihe effizienter Algorithmen gibt, die das Rendering komplexer Polygonstrukturen übernehmen. Dazu gehören unter anderem Beleuchtungsverfahren wie das *Gouraud-Shading*, die darauf ausgelegt sind, die stückweise lineare Darstellung im gerenderten Bild nicht als solche erscheinen zulassen. Dadurch wirken auch angenäherte Darstellungen eigentlich kontinuierlicher Oberflächen (wie beispielsweise bei einer Kugel) in der Ausgabe nicht eckig, sondern rund. Außerdem sind moderne Grafikkarten auf das Rendern solcher Polygondarstellungen hin optimiert und führen Berechnungen auf den Polygonen direkt in darauf spezialisierter Hardware durch, was enorme Geschwindigkeitsvorteile erbringt.

Neben der potentiell großen Anzahl an zu rendernden Primitiven weisen Polygonnetze während der Modellierung der Objekte einen weiteren Nachteil auf, da sie keine einfachen Formveränderungen zulassen. Modelliert man ein Objekt durch eine Menge planarer Polygone, kann man nachträglich nicht ohne Weiteres Änderungen an den Positionen der Netzpunkte vornehmen, ohne die Ausgangsstruktur zu zerstören. Dadurch kann es passieren, dass das Objekt als Folge der Transformation nicht mehr durch die gleichen Polygone darstellbar ist. Verdeutlichen lässt sich dies bereits am Beispiel eines Quaders. Verdreht man

diesen, so ist der entstehende Körper nicht mehr länger durch sechs Rechtecke repräsentierbar [Wa02].

Im Zusammenhang mit der Komplexität von 3D-Modellen bezüglich der Anzahl der verwendeten Grundelemente gibt es eine Reihe von Ansätzen, die versuchen, den Detailgrad eines Objektes an dessen Projektionsgröße im finalen Bild anzupassen. Da mit steigender Polygonanzahl die Renderingzeit wächst, versucht man, die Anzahl der zu rendernden Polygone so klein wie möglich zu halten. Erscheint ein Objekt beispielsweise in großer Distanz zum Betrachter, so ist die Projektionsgröße des Objekts typischerweise sehr klein und belegt nur wenige Pixel. In einem solchen Fall kann demnach der Detailgrad des Objekts deutlich geringer sein, ohne dass der Nutzer einen sichtbaren Unterschied wahrnimmt. Ist man in der Lage, den Detailgrad eines Objektes als Funktion seiner Projektionsgröße dynamisch zu modifizieren, so kann man die Renderperformance deutlich erhöhen, ohne die visuelle Qualität des Bildes zu reduzieren. Ein Ansatz in diesem Kontext ist die Verwendung unterschiedlicher *Level of Details* (LoD) [BB06].

Diese Idee findet auch im Bereich der Texturierung von 3D-Modellen Verwendung. Auch dort muss die Auflösung einer Textur nicht konstant sein, sondern kann sich nach der Entfernung des Nutzers vom texturierten Objekt und somit nach dessen Projektionsgröße richten. Die Grundidee ist dabei die Erstellung von Objekten / Texturen mit unterschiedlichen Auflösungen. Während des Renderings wird nun jeweils ein LoD als Funktion der Projektionsgröße ausgewählt, wodurch Renderressourcen eingespart werden können.

Aufgrund der enormen Leistungsressourcen moderner Grafikkarten setzen Game Engines in diesem Zusammenhang auch immer stärker auf Algorithmen, die in der Lage sind, den Detailgrad der Objekte während der Laufzeit zu modifizieren. Hier spielen Verfahren, die eine intelligente und dynamische Unterteilung komplexer Polygone ermöglichen, eine wichtige Rolle. Die Unterteilung von Polygonen wird als *Tesselation* bezeichnet. *Triangulation* ist eine Sonderform der Tesselation, bei der die Oberflächen in Dreiecke unterteilt werden. Eine dynamische Tesselation während der Laufzeit ist ein vergleichsweise teures Verfahren, ermöglicht aber eine fast kontinuierliche Anpassung des Detailgrads an die Projektionsgröße und besitzt darüber hinaus den Vorteil, dass a priori nicht mehrere Objekte unterschiedlicher Auflösung erstellt werden müssen. Die Rechenleistung der Grafikkarten nimmt dabei dem Modellierer diese Arbeit ab [Gr09].

3.1.1.3.2 Bikubische parametrische Patches

Bikubische parametrische Patches sind in ihrer Grundstruktur mit Polygonnetzen vergleichbar, allerdings sind die Basisprimitive keine planaren Polygone, sondern gekrümmte Vierecke. Ihre Position und Form im dreidimensionalen Raum ist durch mathematische Formeln festgelegt, wodurch solche Patches an jedem Punkt im Raum eindeutig definiert sind. Dies ist bei planaren Polygonen nur an deren Eckpunkten der Fall. Durch eine Modifikation der Parameter in den mathematischen Gleichungen kann man direkten Einfluss auf die Form der Patches nehmen und diesen modifizieren [Wa02]. Aufgrund ihrer mathematischen Definition und der daraus resultierenden Exaktheit ihrer Darstellung werden Patchnetze im Bereich des *Computer-Aided Designs* (CAD) eingesetzt. Dort werden die im Computer modellierten 3D-Objekte als Bauplan verwendet. So ist es beispielsweise möglich, durch den Einsatz von *Computerized Numerical Contr*ol-Fräsen (CNC), die CAD-Objekte als Eingabe erhalten, Bauteile oder Ähnliches zu erstellen. Offensichtlich spielt in solchen Bereichen die Exaktheit der Darstellung die entscheidende Rolle, so dass angenäherte Darstellungsformen, wie sie durch Polygonnetze realisiert werden können, nicht ausreichend sind. Für solche Anwendungszwecke sind Patchnetze sehr gut geeignet, für das Echtzeitrendering eignen sie sich dagegen nicht, da es sehr aufwendig ist, sie zu visualisieren.

Hierfür existieren zwei Gruppen von Ansätzen. Die erste Gruppe rendert die Patches direkt aus ihrer Darstellung heraus, die zweite wandelt die Patches zunächst in eine Polygondarstellung um, die anschließend mittels der für Polygonnetze zur Verfügung stehenden Algorithmen gerendert wird. Verfahren der ersten Gruppe sind typischerweise sehr rechenintensiv und komplex, weshalb der zweite Ansatz der Beliebtere ist [Wa02].

Die Tesselation von Patches ist vergleichsweise einfach. Konzeptuell unterteilt man die Patches gleichmäßig in planare Polygone. Anschließend testet man, wie groß die Abweichung ist, die durch die Annäherung eines potentiell gekrümmten Vierecks durch ein planares Polygon entsteht. Liegt diese oberhalb eines Grenzwertes, unterteilt man das planare Polygon so lange rekursiv weiter, bis die Qualität der Annäherung der Vorgabe entspricht.

Offensichtlich eignen sich bikubische planare Patchnetze sehr gut für die Modellierung komplexer Objekte, die dann später für die Weiterverwendung in Polygonnetze

umgewandelt werden können. Dabei ist es möglich, die Exaktheit der Polygonnetznäherung durch Parameter zu steuern und dadurch den Tesselationsgrad als Funktion der Oberflächenkrümmung zu modellieren. Aus diesem Grund sind bikubische Patches auch im Bereich der 3D-Modellierung beliebt.

Neben der schlechten Renderperformance weisen Patches allerdings einen weiteren Nachteil auf, der aber erst dann zum Tragen kommt, wenn einzelne Patches zu Patchnetzen zusammengesetzt werden. Da die Parameter der mathematischen Gleichungen, die die Patches beschreiben, nur für jeweils einen Patch gültig sind, muss darauf geachtet werden, dass an den gemeinsamen Kanten adjazenter Patches keine Brüche entstehen, sobald einer der Nachbarpatches verändert wird. Dadurch bleibt die Stetigkeit der durch die Patches beschriebenen Oberfläche gewahrt [Wa02].

Vergleicht man Polygon- mit Patchnetzen, so stellen Patchnetze eine Modellierungsform dar, bei der es möglich ist, mit vergleichsweise wenigen Elementen komplexe Objekte zu modellieren. Speziell bei gekrümmten Oberflächen bieten Patchdarstellungen deutliche Vorteile gegenüber der nur näherungsweisen Polygonrepräsentation. Dafür sind Patches aufgrund ihrer mathematischen Definition komplexer als planare Polygone. Gebäude zeichnen sich im Gegensatz zu Pflanzen oder Lebewesen im Allgemeinen dadurch aus, dass die meisten verwendeten Oberflächen planar sind. Gekrümmte Oberflächen stellen dagegen Ausnahmen dar, zu finden sind solche Strukturen beispielsweise an Gesimsen oder Säulen, die dem Gebäude als Komponenten hinzugefügt werden. Darüber hinaus sind die verwendeten mathematischen Berechnungen deutlich einfacher, wenn sie auf planare Polygone und nicht auf bikubische parametrische Patches angewendet werden. Aus den genannten Gründen arbeitet das vorliegende System mit Polygonnetzdarstellungen anstelle von bikubischen parametrischen Patches.

3.1.1.4 **Datenstrukturen zur Repräsentation von Polygonnetzen**

Für die Repräsentation von Polygonnetzmodellen existiert eine Reihe von Datenstrukturen, die sich in ihrer Komplexität unterscheiden. Bevor nachfolgend verschiedene Ansätze vorgestellt werden, sollen zunächst kurz die Begriffe der *Geometrie* und der *Topologie* im Kontext von Polygonnetzen definiert werden, da diese für die Unterscheidung der einzelnen Strukturen von Bedeutung sind. Die Topologie beschreibt die Struktur des einzelnen Polygons, beispielsweise eines Dreiecks. Diese bleibt bei allen Starrkörpertransformationen

erhalten, ein Dreieck bleibt somit in seiner Struktur ein Dreieck, auch wenn es verschoben oder rotiert wird. Als Geometrie bezeichnet man die Festlegung der räumlichen Lage des Polygons [BB06]. Bei einem Dreieck erfolgt dies durch die Angabe der Eckpunktkoordinaten.

Die verschiedenen Datenstrukturen unterscheiden sich unter anderem darin, ob und wie sie die Topologie speichern. Davon hängt ab, wie effizient sie in der Beantwortung topologischer Fragestellungen sind. Beispiele für solche Fragestellungen sind die Bestimmung aller Elemente, die sich eine bestimmte Kante teilen oder das Auffinden aller Kanten, die an einem bestimmten Eckpunkt enden. Je mehr topologische Informationen eine Datenstruktur speichert, desto schneller wird sie solche Fragestellungen beantworten können, desto komplexer wird aber auch ihre interne Organisation und Verwaltung.

Die einfachste Form der Polygonrepräsentation besteht in der Angabe einer *Punktliste*. Verwendet man eine *implizite Speicherung*, so definiert man, dass zwischen zwei aufeinanderfolgenden Punkten eine Kante existiert, ebenso vom letzten auf den ersten Punkt der Liste. Ohne diese Festlegung speichert man die Kanten *explizit*, in diesem Fall ist die Abfolge der Punkte beliebig. Offensichtlich besitzt diese Struktur eine Reihe von Nachteilen. Zunächst vermischt sie Topologie und Geometrie, da die Topologie erst aus der Geometrie hervorgeht und nicht explizit verwaltet wird. Speichert man mehrere adjazente Polygone auf diese Art, so teilen sich diese verschiedene Punkte. Dies kann allerdings in der Datenstruktur nicht explizit ausgedrückt werden. Somit müssen auch gemeinsame Punkte in jedem Polygon erneut gespeichert werden, was speziell bei komplexen Objekten zu einer problematischen Verschwendung von Speicher führt. Außerdem erfordern topologische Fragen aufwendige Suchprozesse, beispielsweise um gemeinsame Ecken oder Kanten zu finden, da diese nicht explizit gespeichert werden.

Eine Milderung des Ressourcenproblems erreicht man durch die Verwendung einer *Eckenliste*. Dabei werden in den Polygonen nicht mehr die Punkte gespeichert, sondern nur noch Verweise auf diese [BB06]. Strukturen dieser Art werden von den beiden großen *Application Programming Interfaces* (API) *OpenGL* und *DirectX* im Bereich der 3D-Grafikprogrammierung bereitgestellt. Für die Speicherung der eigentlichen Punktkoordinaten stehen *Vertex Buffer* (*DirectX*) [Gr09] bzw. *Vertex Arrays* (*OpenGL*) [BAH09] zur Verfügung. Die Referenzen werden meist als einfache Indices auf die Vertexlisten realisiert, beispielsweise durch 16 Bit breite Integer-Werte. Da Grafikkarten ihre Stärken besonders darin besitzen, Dreiecke zu verarbeiten, gibt es für diese Art von

Polygonen Erweiterungen in den Datenstrukturen, die darauf abzielen, den Speicherbedarf zur Verwaltung der Polygondaten noch weiter zu reduzieren. Für ein beliebiges Polygon, das vorab durch einen Tesselationsalgorithmus trianguliert wurde, gibt man bei *Triangle Strips* oder *Triangle Fans* sämtliche Punkte der entstehenden Dreiecke in einer vorgegebenen Reihenfolge an [Sh08]. Durch die Festlegung, dass die Angabe der Vertexkoordinaten als *Strip* oder *Fan* erfolgt, kann die Grafikkarte bei der Verarbeitung der Strukturen wieder einzelne Dreiecke aus den Übergabekoordinaten erzeugen und verarbeiten. Durch die kompaktere Eingabe der Koordinaten wird Speicher gespart, was speziell in Anwendungsbereichen mit limitierten Ressourcen wie beispielsweise Spielekonsolen von großer Bedeutung ist, da hier der *Memory Footprint*[1] so klein wie möglich sein soll. Abbildung 1 zeigt ein Beispiel für einen Triangle Strip. Man erkennt, wie durch die implizite Ordnung der Stripstruktur die Duplikation von Punkten in der Punktliste vermieden und somit Speicher gespart wird. Indizierte Strukturen dieser Art bieten gegenüber den einfachen Punktlisten zwei Vorteile. Zum einen ist dies die bereits erwähnte Speicherersparnis durch die Vermeidung von Punktduplikationen. Zum anderen realisiert dieser Ansatz die Trennung von Geometrie und Topologie, da diese getrennt voneinander gespeichert werden. Dadurch sind Änderungen der Geometrie möglich, ohne dabei automatisch auch die Topologie zu modifizieren. Trotzdem sind auch diese Strukturen für topologische Anfragen ungeeignet, da deren Beantwortung aufwendige Suchprozesse erfordert.

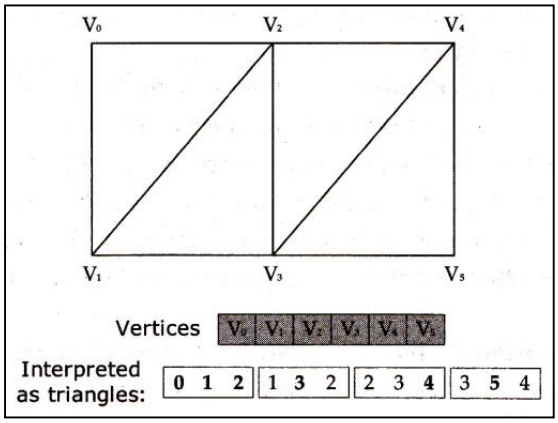

Abbildung 1: Beispiel einer Triangle Strip Struktur [Gr09]

Dies ist eine generelle Schwäche punktbasierter Umrissmodelle, da sie aufgrund ihrer Struktur für die Beantwortung topologischer Fragestellungen schlecht geeignet sind.

[1] Der Memory Footprint bezeichnet die Menge an Hauptspeicher, der durch eine Anwendung während ihrer Laufzeit verwendet oder referenziert wird.

Allerdings sind solche Anfragen in vielen Anwendungsbereichen auch nicht relevant. Möchte man beispielsweise in einer 3D-Engine ein komplexes 3D-Modell rendern, so spielen die Nachbarschaftsbeziehungen zwischen einzelnen Polygonen eine untergeordnete Rolle. Speziell im Kontext der prozeduralen Modellierung ist es aber häufig wichtig, Wissen über die topologische Struktur eines Objekts zu besitzen und solche Anfragen effizient beantworten zu können.

Kantenbasierte Umrissmodelle eignen sich besser für diese Aufgabe, da sie die topologische Struktur auf der Ebene von Kanten und nicht auf der Ebene von Punkten abbilden. Die *Winged-Edge-Datenstruktur* nach Baumgart [Ba72] ist der Vorreiter der kantenbasierten Umrissmodelle, die in der Folge weiterentwickelt wurden. Exemplarisch sei nur die Baumgartsche Struktur vorgestellt, um ein Verständnis des dahinterstehenden Ansatzes zu erlangen. Im Zentrum der Winged-Edge-Struktur steht die Kante, für die eine Reihe von Informationen gespeichert wird. Dazu gehören:

- die Vertices, die zu dieser Kante gehören
- das linke und rechte Polygon dieser Kante
- die Vorgänger- und Nachfolgerkante, wenn man das linke Polygon durchläuft
- die Vorgänger- und Nachfolgerkante, wenn man das rechte Polygon durchläuft

All diese Informationen werden in einer Kantendatenstruktur verwaltet und bei Modifikationen des Objekts aktualisiert. Wichtig dabei ist die Festlegung der Orientierung der jeweiligen Kante, um entscheiden zu können, wo links bzw. rechts und was Vorgänger bzw. Nachfolger einer Kante ist. Meist verwendet man eine Orientierung, bei der die Kanten eines Polygons im Uhrzeigersinn durchlaufen werden, wenn man das Polygon von außerhalb des Polyeders betrachtet, dessen Oberfläche es teilweise definiert. Eine solche Festlegung ist auch für die zwei weiteren Datenstrukturen relevant, die Teil der Winged-Edge-Struktur sind. Dazu gehören eine Vertex- und eine Polygontabelle. Für jedes Vertex und jedes Polygon speichert man jeweils eine anliegende Kante und kann dadurch sehr schnell beantworten, welche Kanten, Vertices oder Polygone adjazent zueinander sind oder sich gemeinsame Elemente teilen [Ba72]. Diese Basisstruktur funktioniert nur, wenn die Polygone keine Löcher enthalten. Ein weit verbreiteter Ansatz zur Integration von Strukturen mit Löchern ist die Verwendung einer anderen Abfolge für die Lochdefinition. Verwendet man für die Polygone eine Abfolge im Uhrzeigersinn, so werden die Löcher

entgegen dem Uhrzeigersinn definiert und können dadurch als solche in die Winged-Edge-Datenstruktur integriert werden.

Das hier vorgestellte System verwendet einen hybriden Ansatz mit einer hierarchischen Anordnung der Strukturelemente unter Verwendung von Vertex- und Indexbuffern sowie einer Kantenverwaltungsstruktur, die topologische Anfragen nach allen Elementen ermöglicht, die sich eine gegebene Kante teilen. Eine detailliertere Darstellung der Strukturverwaltung wird an späterer Stelle gegeben.

3.2 **Die eXtensible Markup Language (XML)**

3.2.1 **Entwicklung von XML**

Die *eXtensible Markup Language* (XML) ist eine Metasprache, deren erste Konzeption im Jahr 1996 veröffentlicht und im Februar 1998 durch das *World Wide Web Consortium* (W3C) standardisiert wurde [Sh02]. Die Ursprünge der Sprache existieren bereits deutlich länger. Zu nennen sind hier die *Standard Generalized Markup Language* (SGML) und die *Hypertext Markup Language* (HTML). Die Entwicklung des XML-Standards hatte zum Ziel, die Schwächen von SGML und HTML zu überwinden und deren jeweilige Stärken in einer neuen Metasprache zusammenzufassen. XML wird innerhalb des Semantic Building Modelers sowohl für die Konfiguration des Systems selber, als auch für die Beschreibung von Gebäude- und Stadtstrukturen verwendet. Dabei wird die Validität der nutzergenerierten Konfigurationsstrukturen durch die Verwendung von *XML Schema* geprüft. Aufgrund der großen Bedeutung der XML-Technologien für diese Arbeit, wird nachfolgend kurz auf die Ursprünge von XML eingegangen, bevor die zentralen Konzepte erläutert werden.

3.2.2 **Die Standard Generalized Markup Language (SGML)**

SGML ist wie XML eine Metasprache, also eine Sprache, mit deren Sprachmitteln man in der Lage ist, die Struktur und Form anderer Sprachen zu beschreiben. Ein Hauptmotiv für die Entwicklung von SGML war die Notwendigkeit, „die Datenspeicherung unabhängig von Softwarepaketen oder Softwareherstellern“ [NH99] durchführen zu können. Es existierte ein großer Bedarf nach einer standardisierten, nicht-proprietären Lösung, die es ermöglichte, Dokumente zu beschreiben, ohne dass die jeweilige Beschreibung selber abhängig von einer bestimmten Software oder Plattform war. Dadurch sollte unter anderem der Dokumentenaustausch zwischen verschiedenen Systemen und Plattformen erleichtert

werden. Weiterhin erhoffte man sich durch die Verwendung eines solchen Beschreibungsstandards die Lebensdauer von Dokumenten unabhängig von den Entwicklungszyklen eines konkreten Softwareprodukts zu gestalten. Konformität zum SGML-Standard sollte garantieren, dass Dokumente beliebig zwischen unterschiedlichen SGML-konformen Systemen transferiert und dargestellt werden können. Dass dies bei proprietären Systemen häufig nicht der Fall ist, stellt man spätestens dann fest, wenn sich Dateiformate zwischen aufeinanderfolgenden Versionen der Systeme ändern. Im besten Fall sind die Systeme abwärtskompatibel und in der Lage, Dokumente im älteren Dateiformat zu öffnen, der umgekehrte Fall stellt dagegen eine absolute Ausnahme dar.

Bevor nachfolgend auf SGML eingegangen wird, soll kurz definiert werden, was man unter einem *Markup* versteht. Vereinfacht gesagt, „werden einer Information“ durch ein Markup „Zeichen hinzugefügt, die genutzt werden, um diese Information auf eine ganz bestimmte Weise zu verarbeiten“ [NH99]. In dieser Definition zeigt sich bereits, dass der Begriff des Markups sehr weit gefasst ist und sich unterschiedliche Markupsprachen in der Komplexität ihres Markups stark unterscheiden können. Betrachtet man beispielsweise eine Datei, deren Daten im *Comma-Separated Value* (CSV)-Format vorliegen, so enthält diese mindestens ein vordefiniertes Markup, nämlich das Komma. Dieses trennt die unterschiedlichen Datenbereiche voneinander und zeigt dadurch der verarbeitenden Anwendung das Ende eines Blocks und den Beginn des jeweils Nachfolgenden an. Aufgrund der genannten Definition gehören beispielsweise auch Steuerzeichen in Textdateiformaten wie dem *Rich Text Format* (RTF) zur Gruppe der Markups, da auch diese Informationen darüber enthalten, wie die Information, auf die sich das Steuerzeichen bezieht, verarbeitet, also beispielsweise dargestellt wird. Solche Formen des Markups bezeichnet man *prozedurales Markup*, da sie der Verarbeitungssteuerung einer Anwendung dienen [NH99]. Markup-Sprachen dieser Art sind häufig proprietär. So stellt es eher eine Ausnahme dar, falls die Software eines Herstellers in der Lage ist, das prozedurale Markup eines Konkurrenten zu verarbeiten und korrekt darzustellen. Dies hängt damit zusammen, dass proprietäre Markups typischerweise nicht offen sind, meist existiert weder eine Beschreibung noch eine offene Dokumentation des Standards.

Ein Kernziel bei der Entwicklung von SGML war es nun, diesem Problem entgegenzutreten. SGML selber sollte völlig unabhängig von jedweder Anwendung sein. Jeder SGML-Code, der in einer bestimmten Anwendung erstellt wird, soll direkt portabel in beliebige andere SGML-Anwendungen sein.

Im Kontext von XML bezeichnet man die Definition einer Sprache als *Anwendung*, diese Bezeichnung soll auch im Kontext von SGML genutzt werden, um Markup-Sprachen zu bezeichnen, die unter Verwendung der SGML-Konstrukte beschrieben wurden. Eine SGML-Anwendung setzt sich zusammen aus einer SGML-Deklaration und einer SGML-Dokumenttypdefinition (SGML-DTD). Die SGML-Deklaration legt dabei fest, welche Zeichen innerhalb der Sprache als Markup aufzufassen sind. Während in XML-Anwendungen Tags immer durch „<“ geöffnet und durch „>“ geschlossen werden, kann der Nutzer in SGML auch beliebige andere Zeichen für dieses Markup deklarieren.

Die SGML-DTD kann eine Reihe weiterer Regeln für die erzeugte Markup-Sprache deklarieren. Dazu gehören beispielsweise *Minimalisierungsregeln*, deren Verwendung es ermöglicht, Markups aus dem Kontext abzuleiten [NH99]. Der SGML-Parser, der ein Dokument verarbeitet, das in einer SGML deklarierten Markup-Sprache verfasst ist, kann anhand des Kontexts und der in der DTD deklarierten Regeln beispielsweise den Anfang und das Ende von Elementen implizit annehmen, obwohl diese nicht explizit deklariert sind. Die Möglichkeit, solche Minimalisierungsregeln zu deklarieren, trägt zu der Komplexität von SGML bei, da sowohl die Implementation eines vollwertigen SGML-Parsers deutlich aufwendiger wird, aber auch das Verfassen von Dokumenten in einer SGML-Markup-Sprache unter Verwendung solcher Minimalisierungsregeln zu einem schwierigen Problem werden kann.

Die große Komplexität von SGML ist einer der wichtigsten Gründe, warum die Sprache heutzutage kaum mehr genutzt wird. Die Komplexität hängt eng mit ihrer großen Leistungsfähigkeit zusammen. SGML ist ausdrucksstark und in der Lage, komplexe Sprachen formal zu beschreiben, speziell für den Einsatz im *World Wide Web* (WWW) allerdings zu umfangreich und kompliziert. Die Komplexität bezieht sich dabei nicht nur auf die reine Verwendung der SGML zur Modellierung von SGML-Anwendungen, sondern auch in Bezug auf die Umsetzung aller in der SGML vorhandenen Funktionalitäten und Sprachfeatures in Softwarekomponenten, die zur Darstellung von SGML-basierten Sprachen eingesetzt werden. Die vorab erwähnten Minimalisierungsregeln sind hierfür ein gutes Beispiel. Zwar resultiert aus dem großen Sprachumfang eine große Mächtigkeit, allerdings führt dies unter Umständen zu einer deutlich aufwendigeren und dadurch zeitintensiveren Verarbeitung. Im Kontext des World Wide Web fällt dies noch schwerer ins Gewicht, da neben der erforderlichen Zeit zum Download der Daten von einem beliebigen Server auch noch eine potentiell hohe Verarbeitungszeit durch den Browser erforderlich ist. Dies ist

einer der Gründe, warum der Funktionsumfang von XML im Vergleich zu SGML deutlich kleiner ist.

3.2.2.1 **Semantisches vs. darstellungsorientiertes Markup**

Möchte man unterschiedliche Formen des Markup miteinander vergleichen, so bietet sich die Unterscheidung zwischen *semantischem* und *darstellungsorientiertem Markup* an.

Semantisches Markup sagt nichts darüber aus, wie die Information, die durch das Markup ausgezeichnet wird, dargestellt werden soll. Als gutes Beispiel einer semantischen SGML-Anwendung kann man die erste Version der *Text Encoding Initiative* (TEI)-DTD betrachten. Die Motivation zur Entwicklung einer SGML-basierten TEI-DTD ähnelte stark der Motivation, die zur Entwicklung von SGML selber führte. Bevor die Text Encoding Initiative 1987 gegründet wurde, wollten auch wissenschaftliche Projekte und Bibliotheken von den Möglichkeiten profitieren, die der Fortschritt der digitalen Technologien mit sich brachte. Das Ziel der Initiative bestand darum in der Entwicklung von Technologien und Richtlinien für einen möglichst reibungslosen Austausch von Materialien. Man erhoffte sich die Lösung eines Kernproblems, das darin bestand, dass zwar eine Vielzahl unterschiedlicher Systeme existierte und auch eingesetzt wurde. Neben der teils schlechten Qualität der verwendeten Softwaresysteme waren diese allerdings größtenteils proprietär und zueinander inkompatibel. Die Entwicklung eines allgemein akzeptierten Standards zur Auszeichnung von Dokumenten sollte diese Probleme lösen. Die erste Version der TEI-DTD basierte auf SGML und wurde im Juni 1990 veröffentlicht [Te07b]. Sie erlaubte unter anderem die Auszeichnung der Textstruktur mit einer beliebigen Dokumentationstiefe [Te07a] und stellt eine Form des semantischen Markups dar.

Demgegenüber ist HTML eine SGML-Anwendung, die deutlich stärker darstellungs- als semantisch orientiert ist. Der folgende Abschnitt befasst sich mit der Entstehung und den Zielsetzungen von HTML.

3.2.3 **Die Hypertext Markup Language (HTML)**

HTML ist wie TEI (in den Versionen P1-P3) eine SGML-Anwendung und im Gegensatz zu XML keine eigene Metasprache. Die Entwicklung von HTML geht auf Tim Berners-Lee zurück, der in den 1980er Jahren am Forschungszentrum CERN in der Schweiz Technologien entwickelte, die den Dokumentenaustausch über das TCP/IP (*Transmission*

Control Protocol/Internet Protocol) ermöglichen sollten. Ergebnis dieser Arbeit waren drei Technologien, die auch heute noch das Fundament des World Wide Webs darstellen. Dazu gehört ein Ansatz, über den beliebige Internetdokumente eindeutig und universell adressiert werden können, der *Uniform Ressource Locator* (URL). Weiterhin entwickelte Berners-Lee das *Hypertext Transfer Protocol* (HTTP), das im TCP/IP-Protokoll-Stack direkt auf TCP aufsetzt und hauptsächlich dazu dient, Webseiten von entfernten Servern zu laden, um sie anschließend in einem Browser darzustellen. Die letzte zentrale Technologie von Berners-Lee war die erwähnte Hypertext Markup Language, die für die Beschreibung von Dokumenten im Internet gedacht war [Sh02]. Allerdings wurde HTML im Laufe seiner Entwicklung immer mehr in ein Werkzeug umgeformt, das der reinen Präsentation von Webseiten diente. Die Entwicklung entfernte sich immer mehr vom semantischen Markup und fokussierte stattdessen zunehmend die darstellungsorientierte Auszeichnung. Trotzdem ist HTML auch als rein darstellungsorientierte Markup-Sprache nur bedingt einsetzbar und für manche Darstellungsformen ungeeignet. Als Beispiele kann man fehlende oder nur eingeschränkte Möglichkeiten zur Steuerung von Leerräumen in Texten nennen oder auch die nicht vorhandenen Sprachmittel zur Formatierung mehrspaltiger Texte, wie sie beispielsweise in Zeitungen üblich sind.

Auf der anderen Seite besitzt HTML zwar Elemente, die in der Lage sind, die Struktur von Dokumenten zu beschreiben, beispielsweise Tags zur Auszeichnung von Überschriften oder Abschnitten, die ausgezeichnete Struktur wird allerdings durch die Browser nicht auf Gültigkeit geprüft. So ist es problemlos möglich, innerhalb eines beliebigen Abschnitts Überschriften unterschiedlicher Ebenen zu deklarieren, obwohl dies faktisch nicht korrekt ist. Erschwerend kommt hinzu, dass beispielsweise die genannten Tags zur Auszeichnung von Überschriften eine Vermischung von Form und Inhalt darstellen. Mit der inhaltlichen Auszeichnung als Überschrift einer bestimmten Gliederungsebene existiert automatisch auch eine bestimmte Formatierung des ausgezeichneten Textes [Vo07]. Hier ist das Problem allerdings weniger bei HTML als viel mehr bei der Implementation der Browser zu suchen. Prinzipiell ist es möglich, HTML über dessen DTD zu validieren, dies wird von den Browsern allerdings nicht getan, da diese derart entwickelt werden, „dass sie fast alles akzeptieren, was auch nur im entferntesten wie HTML aussieht“ [NH99]. Somit ist man zwar in der Lage, strukturelle Informationen der Dokumente über HTML-Tags auszuzeichnen, diese Auszeichnung wird auf Seiten der Browser allerdings nur für die Formatierung verwendet und nicht semantisch ausgewertet. Dadurch werden die eigentlich

semantischen bzw. strukturellen Markups, die HTML enthält, durch den Browser zu einem darstellungsorientierten Markup uminterpretiert.

Ein grundsätzlicher Nachteil von HTML resultiert aus der Tatsache, dass HTML eine SGML-Anwendung und keine eigene Metasprache ist. Der Nutzer von HTML ist darum nicht in der Lage, selber neue Tags zu definieren, die von den Browsern dargestellt werden. Selbst wenn man wollte, könnte man HTML also nicht um Tags erweitern, die eine semantische Auszeichnung der eigenen Dokumente gestatten. Diese Inflexibilität ist im Vergleich zu Metasprachen wie XML ein großes Manko. Verschärft wird diese Problematik wiederum auf Seiten der Browserhersteller [Vo07]. Denn obwohl der Standard eigentlich ein fixiertes Tagset innerhalb seiner DTD festlegt, wurden immer wieder neue Tags hinzugefügt, die allerdings den eigentlichen Standard untergraben und Browserinkompatibilitäten erzeugen. Solche Browserinkompatibilitäten sind ein großes Problem für Webentwickler, da sie dazu führen, dass Webseiten auf einem Browser korrekt, auf dem anderen im schlimmsten Fall gar nicht dargestellt werden. Dadurch ist es im Bereich der Webentwicklung notwendig, browserspezifischen Code zu entwerfen, um die Darstellbarkeit der eigenen Webseite auf unterschiedlichen Browsern garantieren zu können.

3.2.4 **Die eXtensible Markup Language (XML)**

Nachdem im vorherigen Abschnitt auf die beiden Ursprünge von XML eingegangen wurde, soll nun die Sprache selber erörtert werden. Wie vorab erwähnt, ist XML wie SGML eine Metasprache. Sie besteht aus einer Menge von Regeln, durch deren Anwendung man eine Menge semantischer Tags definieren kann, mittels derer man ein Dokument in unterschiedliche Abschnitte gliedert und diesen eine semantische Bedeutung zuweist. XML definiert die Syntax, mittels derer man anwendungsspezifische, semantische Markup-Sprachen entwickeln kann [Ha99]. Dabei zeichnet sich XML durch seine große Einfachheit aus. Es ist möglich, XML als reinen ASCII-Text (*American Standard Code for Information Interchange*) mit jedem beliebigen Texteditor zu verfassen. ASCII-Text selber ist dabei sehr robust gegenüber Datenkorruption, da selbst der Verlust großer Bytefolgen nur Teile des Gesamttextes unlesbar machen, der Rest des Textes kann dagegen weiterhin verarbeitet werden. Bei proprietären Formaten, die binär gespeichert werden, ist dies typischerweise nicht der Fall. Hier kann bereits das Kippen weniger Bits dazu führen, dass die Datei nicht mehr lesbar ist.

Von einer höheren Ebene betrachtet, ist XML aufgrund der semantischen Textauszeichnung selbstbeschreibend, vorausgesetzt, die Tagbezeichner wurden aussagekräftig gewählt. Diese Selbstbeschreibungseigenschaft ist von entscheidender Bedeutung für die Langzeitspeicherung von XML-Dokumenten. Durch die Kombination eines einfachen Dateiformats mit einem semantischen, aussagekräftigen Tagset ist die Wahrscheinlichkeit groß, dass XML-Dokumente auch noch langfristig gelesen und zumindest in Teilen verstanden werden können [Ha99].

XML wurde als Metasprache mit Blick auf seine Eignung für den Einsatz im World Wide Web entwickelt. Dieser Fokus manifestiert sich in einer Reihe von Eigenschaften des Sprachstandards und zeigt sich speziell im Vergleich mit der deutlich umfangreicheren Metasprache SGML. Wie im Zusammenhang mit SGML bereits thematisiert, leidet diese Sprache unter ihrer großen Komplexität und ist darum für den Einsatz im Internet ungeeignet. Eine Vielzahl der teils komplexen Funktionen des SGML-Sprachumfangs wurde darum von Anfang an aus dem XML-Standard weggelassen, andere für das Web benötigte Funktionalität wurde dagegen hinzugefügt [NH99].

Auch im direkten Vergleich mit HTML erkennt man, dass die Entwickler des XML-Standards versucht haben, aus den Schwächen von HTML zu lernen und diese in XML zu vermeiden. Aufgrund seines eingeschränkten Tagsets ist HTML zu spezifisch und dadurch nicht allgemein einsetzbar. HTML gestattet es nicht, eigene Tags zu definieren, um Dokumente domänenspezifisch auszuzeichnen. XML dagegen erbt von SGML dessen generische Natur und ist dadurch flexibel und in völlig unterschiedlichen Anwendungskontexten einsetzbar [NH99].

Auch bezüglich der Internationalisierung ist XML den beiden vorab genannten Standards überlegen. HTML und SGML basieren beide auf der ASCII-Zeichenkodierung, welche zwar bezüglich der Langlebigkeit den Vorteil hat, robuster zu sein, allerdings kodiert ASCII jedes Zeichen mit 7 Bit, was dazu führt, dass nur 128 Zeichen durch den Standard dargestellt werden können. Offensichtlich reichen diese 128 Zeichen im internationalen Gebrauch nicht aus, da ein international einsetzbarer Standard in der Lage sein muss, sämtliche Schriftzeichen unterschiedlichster Alphabete korrekt abzubilden. Darum wurde für XML eine andere Zeichenkodierung gewählt, die diese Zielsetzung realisieren kann. Die *Unicode*-Zeichenkodierung ist aufgrund ihrer Struktur in der Lage, jedem Zeichen in jeder beliebigen Sprache einen eindeutigen Code zuzuweisen. Diese Kodierung hängt dabei weder von der verwendeten Software, dem Betriebssystem oder dem Gerät ab, auf dem das jeweilige

Zeichen ausgegeben werden soll [Ko06]. Dadurch ist der Unicode-Zeichensatz eine zentrale Säule sowohl der Plattformunabhängigkeit von XML als auch seiner internationalen Einsetzbarkeit.

Nachdem im vorliegenden und den vorherigen Abschnitten auf die besonderen Eigenschaften von XML und die Entwicklungslogik basierend auf den Vorgängern SGML und HTML eingegangen wurde, befasst sich der folgende Abschnitt mit den Möglichkeiten, die Gültigkeit und Wohlgeformtheit von XML-Dokumenten zu überprüfen.

3.2.4.1 **Wohlgeformtheit von XML-Dokumenten**

XML ermöglicht dem Anwender, eigene Tagsets zu definieren und diese in seinen Dokumenten einzusetzen. Im Gegensatz zu HTML definiert XML fast keine eigenen Tags. Stattdessen wird eine Menge von Regeln vorgegeben, die sowohl bei der Definition von Tagsets als auch bei deren Einsatz in XML-Dokumenten eingehalten werden müssen. Hält sich ein Dokument an diese Regeln, so ist es *wohlgeformt*. Das Kriterium der Wohlgeformtheit ist die minimale Anforderung, die XML-Parser an XML-Dokumente stellen, um diese zu verarbeiten und darzustellen. Ist dieses Kriterium verletzt, gibt der Parser eine Fehlermeldung aus. Grundlage der Entscheidung, ob ein Dokument wohlgeformt ist oder nicht, ist die XML zugrunde liegende Grammatik. Diese besteht aus 81 Produktionsregeln und ist in der *Erweiterten-Backus-Naur*-Form (EBNF) notiert, einem Formalismus zur Beschreibung von Grammatiken [Vo07]. Diese Regeln definieren unter anderem die Struktur von XML-Dokumenten. Dazu gehört die Festlegung, dass innerhalb eines XML-Dokuments die Daten in einer Baumstruktur vorliegen müssen. Bis auf den Wurzelknoten dieses Baumes besitzt jedes Element genau ein Elternelement aber beliebig viele Kindelemente. Das Wurzelelement muss dabei sämtliche anderen Elemente enthalten [Vo07]. Innerhalb eines Elements sind verschiedene Inhalte gestattet, so kann ein XML-Element weitere Kindelemente oder Zeicheninhalt enthalten. Auch eine Mischung aus Zeicheninhalt und Kindelementen ist durch die XML-Grammatik erlaubt, sollte aber vermieden werden. Zentral für ein wohlgeformtes Dokument ist außerdem die Berücksichtigung einer gültigen Schachtelung der Elemente. Elemente sind genau dann gültig geschachtelt, wenn keine Überlappung zwischen Elementen existiert [NH99].

Sofern man sich beim Verfassen eines XML-Dokuments an die festgelegten Regeln hält, handelt es sich um ein wohlgeformtes Dokument, das den Syntaxregeln des XML-Standards

entspricht. Die Wohlgeformtheit alleine ist allerdings nur ein erster Schritt und eine notwendige Bedingung, damit XML-Dokumente überhaupt verarbeitet werden können. In einem zweiten Schritt ist es möglich, die Gültigkeit eines XML-Dokuments zu überprüfen. Grundlage einer solchen Validierung ist das Vorhandensein eines Schemas, das das Informationsmodell beschreibt, welches dem jeweiligen Dokument zugrunde liegt. Die Tatsache, dass ein solches Schema für die Validierung von Dokumenten herangezogen wird, impliziert, dass auch für die Schemaformulierung ein Formalismus herangezogen werden muss, der durch Maschinen verarbeitet werden kann. Die am weitesten verbreiteten Mechanismen zur Schemadefinition in XML sind Dokumenttypdefinitionen (DTD), die auch schon im Bereich von SGML und HTML aufgetaucht sind, und XML Schema-Dokumente. Der Semantic Building Modeler verwendet XML Schema-Dokumente für die Beschreibung einer gültigen Dokumentstruktur, da XML Schema gegenüber DTDs eine Reihe wichtiger Vorteile bietet. Aufgrund der Entscheidung für XML-Schema wird auf die Darstellung der DTD-Mechanismen an dieser Stelle verzichtet und auf die gängige Literatur verwiesen, beispielsweise [NH99], [Vo07] oder [Sh02].

3.2.4.2 **Gültigkeit von XML-Dokumenten**

3.2.4.2.1 XML Schema

Eine wichtige Triebfeder für die Entwicklung von XML Schema waren verschiedene Schwachstellen des DTD-Konzepts, die durch den neuen Standard überwunden werden sollten. Das erste große Manko von DTDs besteht im Fehlen eines umfangreichen Datentypkonzepts. Bis auf die undifferenzierte Angabe, dass ein Element Zeicheninhalt enthalten darf, ist keine weitere Festlegung möglich. Speziell für Anwendungen, die XML-Dokumente für die Konfiguration ihrer Verarbeitungslogik einsetzen, ist dies ein großes Problem, da die Validierung der Inhalte innerhalb des Programmcodes vorgenommen werden muss. So erwartet der Semantic Building Modeler bei der Angabe der Gebäudehöhe eine positive Zahl. Ob der durch den Nutzer angegebene Wert tatsächlich dieser Forderung entspricht, ist unter Verwendung von DTDs nicht überprüfbar [Vo07].

Solche Schwierigkeiten werden durch XML Schema gelöst, weshalb es sich für komplexe Anwendungen deutlich besser eignet. Neben dem umfangreicheren Datentypkonzept erlaubt XML Schema die Kombination beliebig vieler Schema-Dokumente und die Entwicklung komplexer Datenstrukturen vergleichbar den Möglichkeiten, die auch objektorientierte Programmiersprachen bieten [SWW11].

Die Entwicklung von XML Schema begann offiziell mit der Formulierung einer Reihe von Anforderungen, die im Februar 1999 veröffentlich wurden. Eine zentrale Forderung bestand in der Umsetzung eines Namensraum-Konzeptes, welches auch in zahlreichen Programmiersprachen erfolgreich eingesetzt wird, um Mehrdeutigkeiten zu vermeiden. Auch eine möglichst große Kompatibilität zu Programmiersprachen wie *Java* oder Abfragesprachen wie der *Standard Query Language* (SQL) wurde explizit gefordert und sollte durch die Umsetzung innerhalb dieser Sprachen weit verbreiteter Datentypen realisiert werden. Aufgrund des großen Erfolges objektorientierter Programmiersprachen sollte der neue Standard einen Vererbungsmechanismus vorsehen, der die Konstruktion komplexer Datentypen ermöglichen sollte [Vo07]. Basierend auf diesen Anforderungen entwickelte das W3C eine Empfehlung für XML Schema, die im Mai 2001 veröffentlicht wurde. Zwischen der Veröffentlichung der XML- und der XML Schema-Empfehlung vergingen demnach insgesamt vier Jahre [Sh02].

3.2.4.2.1.1 Struktur eines XML Schema-Dokuments

XML Schema-Dokumente sind gültige XML-Dokumente und müssen darum auch konform zur XML-Grammatik sein. Aus diesem Grund beginnt jedes XML Schema-Dokument mit dem Prolog, der unter anderem die verwendete XML-Version und den Zeichensatz angibt. Darauf folgt die Deklaration des Wurzelelements `<xs:schema>`. Die Verwendung dieses Elements zeigt dem XML-Parser, dass es sich beim vorliegenden Dokument um die Deklaration eines Inhaltsmodells in XML Schema handelt. Das Präfix `xs:` gibt an, dass das Element `schema` zum Namespace *http://www.w3.org/2001/XMLSchema* gehört, der mit `xs` abgekürzt wird [SWW11]. Auf das Namespace-Konzept wird im Abschnitt „Namensräume in XML Schema“ detaillierter eingegangen.

3.2.4.2.1.2 Deklaration von Elementen

Nach der Angabe des Wurzelelements können unter Verwendung des Elementnamens `<element>` neue Elemente deklariert werden. Dies entspricht konzeptuell der Nutzung von `<!ELEMENT` in DTDs, allerdings handelt es sich bei `<element>` um ein gültiges XML-Markup. Deklariert man ein leeres `<element>`-Tag, so muss neben dem obligatorischen `name`-Attribut, das den Namen des Tags angibt, auch dessen Typ über das `type`-Attribut angegeben werden. Das `type`-Attribut verweist dabei auf einen entweder vordefinierten oder

abgeleiteten Datentyp. Die Verwendung von `<element>` als nicht-leeres Tag erfordert die Deklaration untergeordneter Element zwischen Start- und Endtag [SWW11].

3.2.4.2.1.3 Lokale vs. globale Deklaration

Sowohl für Elemente als auch für Attribute existieren in XML Schema zwei unterschiedliche Formen der Deklaration. *Lokale* Deklarationen liegen dann vor, wenn ein Element oder ein Attribut innerhalb eines komplexen Typs deklariert wird. In diesem Fall ist die Deklaration nur innerhalb des komplexen Typs gültig, eine Referenzierung durch andere Komponenten außerhalb der Datenstruktur ist nicht möglich. Dies wiederum ist bei *global* deklarierten Elementen und Attributen sehr wohl der Fall. Der Vorteil einer globalen Deklaration zeigt sich dann, wenn ein Element an verschiedenen Stellen innerhalb eines Schemas eingesetzt wird. In diesem Fall erhöht die globale Deklaration die Wartbarkeit des Schemas, da Änderungen nur einmalig durchgeführt werden müssen.

Im Zusammenhang mit XML Schema wird darüber hinaus auch die Wiederverwendbarkeit als große Stärke genannt [SWW11]. Durch die Möglichkeit, verschiedene Schemadokumente miteinander zu kombinieren, können auch Element- oder Attributdeklarationen aus fremden Dokumenten referenziert und wiederverwendet werden. Soll eine solche Komponente referenziert werden, verwendet man das Attribut `ref` und gibt den Namen der Zielkomponente an. Befindet sich diese in einem anderen Namespace, so ist eine Qualifikation mit dem Namespace-Präfix erforderlich.

3.2.4.2.1.4 Datentypen in XML Schema

Eine der größten Stärken von XML Schema ist das umfangreiche Datentypkonzept. Dieses Konzept basiert auf zwei Säulen. Zum einen enthält der Standard eine ganze Reihe vordefinierter Datentypen. Zu diesen gehören verschiedene primitive Datentypen, die auch als *Urtypen* bezeichnet werden. Zum andern bietet die Sprache durch ihren Vererbungsmechanismus die Möglichkeit, neue Datentypen auf der Basis bereits vorhandener zu konstruieren. Der XML Schema-Standard enthält zusätzlich zu den primitiven Datentypen eine Reihe weiterer Datentypen, die durch Ableitung aus diesen konstruiert sind.

Bevor auf diese Konzepte eingegangen wird, sollen zunächst die primitiven Datentypen vorgestellt werden. Diese lassen sich in die Kategorien *Zeichenketten*, *Logik*, *Zahlen*,

Zeitangaben und *Binärdaten* einteilen. Zu den Zeichenketten gehört unter anderem der Typ `xs:string`, der beliebige Zeichenketten enthalten kann. Weitere Typen aus dieser Kategorie beschreiben bestimmte Formen von Zeichenketten, so zum Beispiel der Typ `xs:anyUri`, der Angaben von *Uniform Ressource Identifiern* (URI) in Form absoluter oder relativer Pfade speichert. Die Logikkategorie besteht nur aus dem Typ `xs:boolean`. Dieser Datentyp speichert Zeichenketten, die die zwei unterschiedlichen Werte der booleschen Algebra ausdrücken, also „`TRUE`“ und „`FALSE`“. Die dritte Kategorie enthält Datentypen, die Zahlenwerte repräsentieren. Dazu gehört unter anderem der Typ `xs:decimal`, der Dezimalzahlen mit einer beliebigen Genauigkeit speichert. Der Typ `xs:float` dagegen beschränkt die Länge der gespeicherten Gleitpunktzahlen auf 32-Bit. In der Kategorie Zeitangaben finden sich verschiedene Datentypen mittels derer man Zeitpunkte und Zeitspannen darstellen kann. Diese unterscheiden sich unter anderem in der Formatierung und der Granularität der Zeitangabe. So speichert `xs:time` Uhrzeiten ohne Datumsangabe im Format `hh:mm:ss.sss`, `xs:dateTime` enthält dagegen ein vollständiges Datum im Format `CCYY-MM-DDThh:mm:ss`. Die Datentypen der letzten Kategorie repräsentieren Binärdaten. So speichert beispielsweise `xs:hexBinary` die repräsentierten Zeichenketten in einer hexadezimalen Kodierung [SWW11].

Basierend auf diesen primitiven Datentypen enthält XML Schema eine ganze Reihe weiterer vordefinierter Datentypen, die durch Ableitung aus den Urtypen erzeugt wurden. Diese abgeleiteten Datentypen stammen von `xs:string` und `xs:decimal` ab. Als Beispiel für einen von `xs:string` abgeleiteten Typ sei `xs:normalizedString` genannt. Hierbei handelt es sich um normalisierte Zeichenketten, die keinerlei Leerzeichen enthalten dürfen.

Ein gutes Beispiel für einen von `xs:decimal` abgeleiteten Datentyp ist `xs:integer`. Der Wertebereich dieses Datentyps umfasst die Menge der ganzen Zahlen. Diese enthalten im Gegensatz zum Basistyp `xs:decimal` keinen gebrochenen Anteil.

Abbildung 2 zeigt sämtliche in XML Schema vordefinierten Datentypen, dies umfasst sowohl die Urtypen, als auch sämtliche von diesen durch Ableitung erzeugten weiteren Datentypen.

3.2.4.2.1.5 Deklaration einfacher Datentypen

Einfache Datentypen unterscheiden sich von den nachfolgend vorgestellten komplexen Datentypen dadurch, dass sie nur reine Werte enthalten dürfen, verschachtelte Strukturen

mit Kindelementen und Attributdeklarationen sind nicht zulässig [Sh02]. Dabei gilt auch für einfache Datentypen der Unterschied zwischen lokaler und globaler Deklaration, auf den vorab für Elemente und Attribute eingegangen wurde. Deklariert man einen einfachen Datentyp lokal, also als Kind eines Elements, so ist dieser Datentyp nur innerhalb des Elements sichtbar. Eine globale Deklaration dagegen erfolgt außerhalb einer Elementdeklaration. Derart deklarierte Datentypen sind von beliebigen Stellen referenzierbar und erhöhen die Wiederverwendbarkeit des Schemas. Bei der Deklaration globaler Datentypen ist die Angabe eines Namens über das `name`-Attribut verpflichtend. XML Schema stellt zwei Mechanismen bereit, mittels derer einfache Datentypen deklariert werden, einerseits die Verwendung von Facetten und andererseits verschiedene Formen der Ableitung.

Der Einsatz von Facetten erlaubt eine genaue Festlegung, welche Eigenschaften ein bestimmter Datentyp haben soll. Hierfür greift man auf eine Reihe vordefinierter Attribute zurück, mittels derer man den Wertebereich eines Datentyps einschränken kann. Welche Einschränkungen vorgenommen werden können, hängt vom Basistyp ab. So ist es nicht möglich, bei einem `xs:string`-basierten, einfachen Datentyp die Anzahl an Nachkommastellen zu beschränken.

Die zweite Technik für die Erzeugung einfacher Datentypen ist die Ableitung. Bei dieser deklariert man Datentypen entweder durch *Einschränkung*, *Auflistung* oder *Vereinigung*. Die Verwendung von Facetten zur spezifischen Festlegung des Wertebereichs eines Datentyps stellt dabei eine Form der Ableitung durch Einschränkung dar.

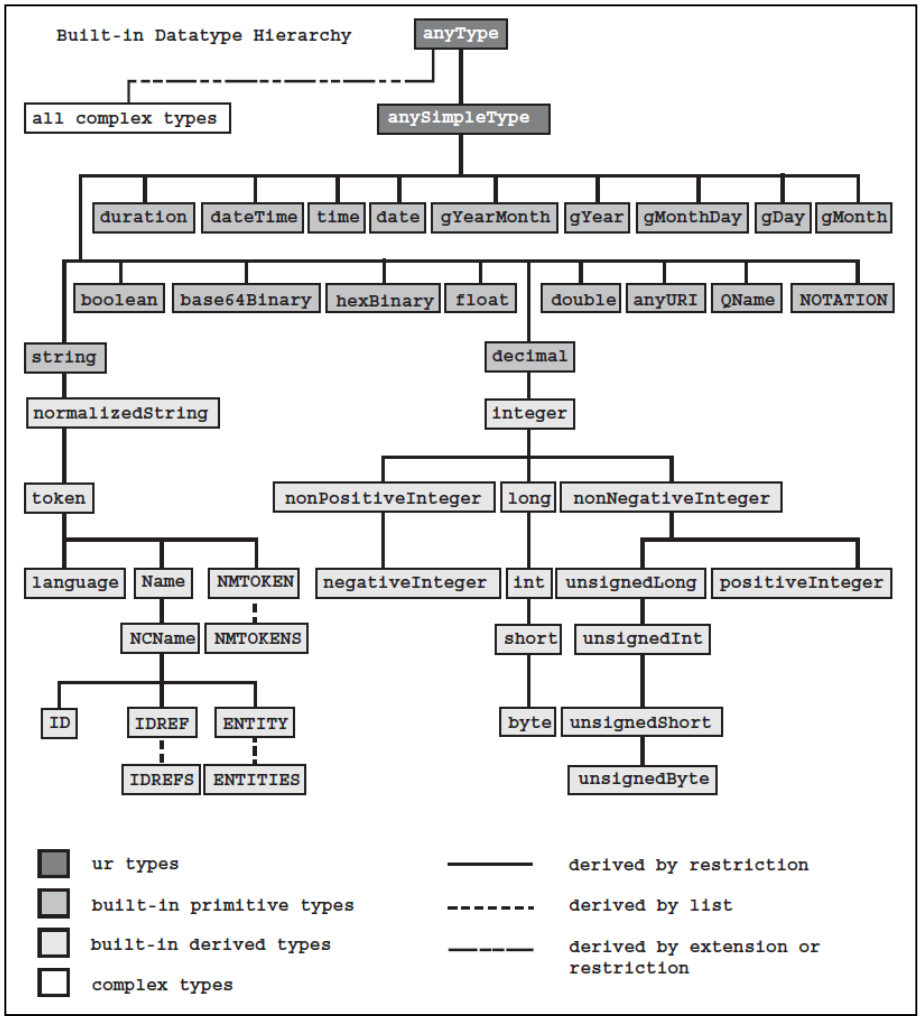

Abbildung 2: Vordefinierte Datentypen in XML Schema [W304b]

3.2.4.2.1.5.1 Facetten – Ableitung durch Einschränkung

In XML Schema existieren zwei unterschiedliche Arten von Facetten. *Fundamentale* Facetten sind festgelegte Eigenschaften eines Datentyps, die nicht durch den Nutzer verändert werden können. Als Beispiel sei die `numeric`-Facette genannt, die angibt, ob es sich bei dem jeweiligen Datentyp um einen numerischen Datentyp handelt. Offensichtlich ist dies eine unveränderbare Eigenschaft des Datentyps, es ist nicht möglich, aus einem Zeichenkettendatentyp einen numerischen zu erzeugen. Andere fundamentale Facetten sind `equal`, `bounded`, `cardinality` und `ordered` [SWW11].

Die zweite Facettenart bilden die *einschränkenden* Facetten, die es ermöglichen, spezielle Eigenschaften eines Datentyps festzulegen. Die Angabe einschränkender Facetten erfolgt innerhalb eines `xs:simpleType`-Elements. Innerhalb dieses Containers deklariert man ein `xs:restriction`-Tag, dessen jeweilige Kindelemente die Einschränkungen enthalten. Das

`xs:restriction`-Element erfordert die Angabe eines Wertes für das obligatorische `base`-Attribut. Der Wert dieses Attributs definiert den Basisdatentyp, der durch die Deklaration eingeschränkt wird, beispielsweise `xs:string` oder `xs:integer`. Die Kindelemente des `xs:restriction`-Elements definieren dann die eigentlichen Facetten. Welche Facetten zur Verfügung stehen, hängt vom gewählten Basisdatentyp ab. Wie die Datentypen lassen sich auch die vorhandenen Facetten in unterschiedliche Kategorien einordnen [SWW11]. Diese Kategorien werden nachfolgend vorgestellt und exemplarisch Facetten aus diesen erläutert.

Längenangaben können bei zeichenkettenbasierten Datentypen dafür verwendet werden, die Länge des Textes festzulegen. Dies kann mit einer festen Länge erfolgen (`xs:length`) oder durch die Angabe eines Bereichs mittels Unter- (`xs:minLength`) und Obergrenze (`xs:maxLength`). Eine andere sehr mächtige Facettenkategorie bilden *Wertmuster* [SWW11]. Eine Variante der Wertmuster sind reguläre Ausdrücke, mittels derer man beschreibt, wie Werte aussehen müssen, damit es sich um gültige Wertzuweisungen handelt. Auf die genaue Syntax wird an dieser Stelle nicht eingegangen, sie stellen aber im Bereich der Informationsverarbeitung ein weit verbreitetes und wirkungsvolles Werkzeug dar, um beliebig komplexe Muster zu definieren. Wertmuster können sowohl für numerische als auch nicht-numerische Datentypen deklariert werden. Ein einfaches Beispiel für ein Wertmuster, das für die Validierung von Postleitzahlen eingesetzt werden kann, ist `[0-9]{5}`. Dieser Ausdruck akzeptiert alle Zeichenketten, die aus genau 5 Zahlen (`{5}`) und aus dem Wertebereich von 0 bis 9 (`[0-9]`) bestehen. Eine andere Form der Wertmuster stellt die Deklaration von Aufzählungen (`xs:enumeration`) dar. Bei dieser Facette legt man eine Liste mit möglichen Werten an. Elemente mit diesem Datentyp dürfen dann nur solche Werte erhalten, die Teil der Aufzählungsliste sind. Die nächste Kategorie umfasst Facetten, mittels derer sich Grenzen und Schranken definieren lassen. Bei numerischen Datentypen ist es mittels solcher Facetten möglich, den Wertebereich durch die Angabe von Ober- und Untergrenzen festzulegen. Am Beispiel des vordefinierten Datentyps `xs:nonNegativeInteger` lässt sich gut die Funktionsweise solcher Schranken erläutern. Der Basistyp ist `xs:integer`. Durch den Einsatz der `xs:minInclusive`-Facette kann man aus diesem Grundtyp sehr leicht den `xs:nonNegativeInteger`-Typ erzeugen, indem man dieser den Wert `0` zuweist. Der neu definierte Datentyp erlaubt dann nur noch Werte größer gleich `0` [va02]. Die letzte Kategorie wird durch Facetten gebildet, mittels derer man die Stellenanzahl numerischer Datentypen festlegen kann. Das gilt sowohl für die Gesamtzahl

an Stellen (`xs:totalDigits`) als auch für die Anzahl an Nachkommastellen (`xs:fractionDigits`).

3.2.4.2.1.5.2 Ableitung durch Liste

Die Verwendung von Facetten stellt die erste von insgesamt drei Ableitungstechniken dar, die XML Schema bereitstellt. Die zweite Ableitungsvariante ist die Ableitung durch Listen. Bei dieser Technik deklariert man einen Listendatentyp, dessen einzelne Elemente alle vom gleichen Typ sein müssen. Die Deklaration erfolgt durch das `xs:list`-Element. Den Datentyp der Listenelemente gibt man entweder als Referenz auf einen global deklarierten Typ über das `itemType`-Attribut oder durch eine lokale Deklaration an [va02]. Innerhalb der XML-Dokument-Instanz werden die einzelnen Elemente getrennt durch Leerzeichen notiert. Leerzeichen sind das einzige gültige Trennzeichen. Es ist nicht möglich, eigene Zeichen als Trenner zu definieren, die weitverbreitete Komma-separierte Liste kann in XML Schema demnach nicht 1:1 umgesetzt werden. Darüber hinaus impliziert die exklusive Verwendung des Leerzeichens als Trenner, dass auch die Elemente einen Typ haben müssen, der keine Leerzeichen enthält, da sonst Elemente fälschlicherweise zerteilt werden. XML Schema stellt eine Reihe vordefinierter Datentypen bereit, die diese Forderung erfüllen, neben den numerischen Datentypen sind dies beispielsweise `xs:normalizedString` oder `xs:IDREF`. Natürlich können auch eigene Datentypen für die Listenelemente verwendet werden, beispielsweise Elemente eines Wertmuster-basierten Datentyps [SWW11].

3.2.4.2.1.5.3 Ableitung durch Vereinigung

Die Kernidee der Ableitung durch Vereinigung ist die Konstruktion eines Datentyps, dessen Wertebereich durch die Vereinigung der Wertebereiche verschiedener Datentypen gebildet wird. Die Deklaration erfolgt durch das `xs:union`-Element. Auch bei dieser Ableitungstechnik kann die Angabe der zugrunde liegenden Datentypen durch Referenzen auf global deklarierte Typen oder durch eine lokale Deklaration erfolgen. Auch eine Kombination dieser beiden Ansätze ist möglich. Referenzen deklariert man als Leerzeichen-getrennte Liste als Wert des `memberTypes`-Attributs des `xs:union`-Elements. Lokale Deklarationen erfolgen durch die Angabe mehrerer `xs:simpleType`-Elemente als Kinder des `xs:union`-Elements.

Ein Beispiel, an dem sich die Funktion von Vereinigungs-Datentypen gut verdeutlichen lässt, ist die Deklaration eines Datentyps, der neben den Zahlen von 0-100 auch den Wert `undefined` zulassen soll. Einen solchen Datentyp implementiert man durch die Vereinigung eines eingeschränkten `xs:integer`-Datentyps in Kombination mit einem durch ein Wertmuster eingeschränkten `xs:string`-Typs, der nur den Wert `undefined` zulässt. Die Verwendung eines solchen Vereinigungs-Datentyps ist nicht die einzige Möglichkeit, um einen Datentyp mit den genannten Eigenschaften zu deklarieren, eine Alternative wäre die Verwendung regulärer Ausdrücke als Facette. Welche Lösung man wählt, ist in diesem Fall Geschmackssache, tendenziell ist es leichter, die vereinigungsbasierte Lösung zu nutzen, als einen regulären Ausdruck zu formulieren, der den gleichen Wertebereich realisiert. Dies gilt besonders für Nutzer, die im Umgang mit regulären Ausdrücken wenig Erfahrung besitzen.

3.2.4.2.1.6 Deklaration komplexer Datentypen

Nachdem bereits mehrfach auf die Fähigkeit von XML Schema eingegangen wurde, komplexe Datenstrukturen zu definieren, werden nun die hierfür erforderlichen Strukturen aufgezeigt. Die Deklaration erfolgt über das Element `<xs:complexType>`. Dieses Element kann nicht nur einfache Elemente als Kindelemente enthalten, sondern auch selber wieder aus einer Menge verschachtelter, komplexer Datentypen bestehen [Vo07]. Dadurch ist es möglich, beliebig komplexe Strukturen zu deklarieren.

Innerhalb der Deklaration eines komplexen Datentyps kann man eine Menge weiterer Kindelemente beliebigen Typs angeben, die Teil dieses Datentyps sind. Dies entspricht der Angabe des Inhaltsmodells bei DTDs. Allerdings bietet XML Schema mehr Freiheiten sowohl bezüglich der Kardinalitätsdeklaration als auch bezüglich möglicher Inhaltsalternativen. Die am häufigsten verwendete Container-Variante ist die Deklaration einer Elementsequenz durch das `xs:sequence`-Element. Kindelemente eines `xs:sequence`-Elements müssen im XML-Dokument, das durch das deklarierte Schema validiert werden soll, in exakt der vorgegebenen Reihenfolge auftauchen.

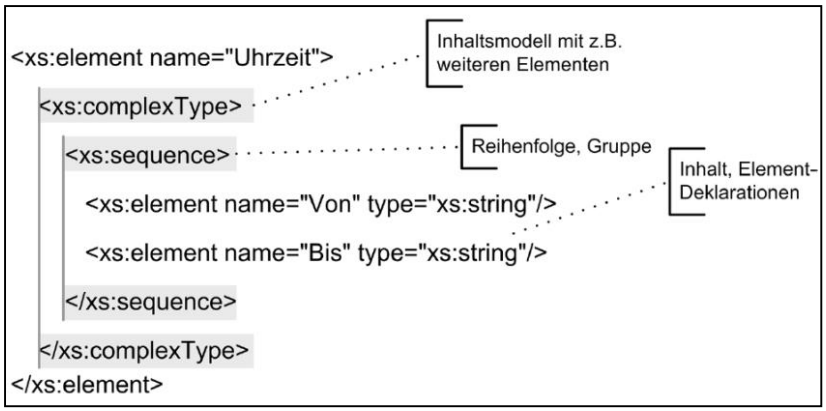

Abbildung 3: Elementdeklaration mit komplexem Datentyp [SWW11]

Abbildung 3 zeigt ein Beispiel für eine solche Elementdeklaration, die als Kindelement die Deklaration eines komplexen Datentyps enthält. Durch die Verwendung von `xs:sequence` ist die Reihenfolge, in der die Elemente „`Von`“ und „`Bis`“ deklariert sind, verpflichtend für jede Instanz des Datenmodells. Die Verschachtelungstiefe einer solchen Deklaration ist dabei theoretisch unbegrenzt. So könnten auch die Elementdeklarationen unterhalb der Sequenz selber nicht leer sein und weitere komplexe Datentypen deklarieren.

Zusätzlich zur Deklaration einer Reihenfolgebeziehung mittels `xs:sequence` bietet XML Schema über die `xs:choice`-Komponente die Deklaration einer Element-Auswahl. Befinden sich nur `xs:element`-Kinder unterhalb von `xs:choice`, so darf nur genau eines dieser Kinder innerhalb eines Instanzdokuments vorkommen. Im Gegensatz zum `xs:sequence`-Element ist beim `xs:all`-Element die Reihenfolge der Elementdeklaration für das Instanzdokument nicht bindend. Hier können die Elemente in einer beliebigen Reihenfolge notiert werden.

All diesen Elementen ist gemeinsam, dass bei der Validierung eines Instanzdokuments neben der Art des Containers auch die Häufigkeitsangaben (`minOccurs` bzw. `maxOccurs`) der Elemente berücksichtigt werden müssen. So ist zwar die Reihenfolge der Elementdeklaration bei `xs:sequence`-Elementen bindend, allerdings ist es möglich, Elemente als optional zu kennzeichnen, indem man ihnen eine Kardinalität von `0` zuweist. In diesem Fall kann das Element weggelassen werden.

Den Schemakomponenten `xs:choice` und `xs:sequence` ist darüber hinaus gemeinsam, dass sie sich selber als Kindelemente enthalten dürfen. So ist es beispielsweise möglich, unterhalb einer `xs:choice`-Deklaration mehrere Sequenzen über `xs:sequence` zu deklarieren. Innerhalb des Instanzdokuments muss dann eine dieser Sequenzen ausgewählt

werden. Dies unterscheidet die beiden Komponenten von `xs:all`, da dieses Element ausschließlich Elemente und keine weiteren Containerdeklarationen als Kinder enthalten darf [W304b].

Neben der Deklaration von Elementen, die Teil eines komplexen Typs sind, ist auch die Angabe von Attributen möglich. Diese werden durch das Element `<xs:attribute>` angegeben, welches neben den Attributen `name` und `type` noch weitere Attribute enthalten kann, beispielsweise die Angabe eines Vorgabewerts durch `default` oder die Festlegung, ob es sich um ein optionales oder obligatorisches Attribut handelt (`use`) [va02].

3.2.4.2.1.6.1 Komplexe Datentypen mit einfachem Inhaltsmodell

Einfache Inhaltsmodelle erlauben nur Attribute und direkten Inhalt. Aus Sicht der hierarchischen Baumstruktur von XML Schema handelt es sich bei solchem Inhalt um einen Blattknoten. Dieser kann keine weiteren Kinder haben und somit keine zusätzlichen Elemente deklarieren. Komplexe Datentypen können solche einfachen Inhaltsmodelle enthalten. Die Deklaration des Modells erfolgt direkt unterhalb des `xs:complexType`-Elements durch das `xs:simpleContent`-Element. Dieses ermöglicht die Deklaration einfachen Inhalts durch Verwendung einer speziellen Ableitungstechnik, der Ableitung durch Erweiterung. Im Kontext einfacher Inhaltsmodelle gestattet dieser Ansatz ausschließlich das Hinzufügen von Attributen zu einem vorhandenen Basistyp. Die Erweiterung wird mit Hilfe des Elements `xs:extension` festgelegt, das als Kindelemente eine Liste der hinzuzufügenden Attribute enthält. Außerdem definiert man innerhalb des `base`-Attributs des `xs:extension`-Elements den Datentyp, der erweitert werden soll und als Basisdatentyp dient. Diese Erweiterung ist nur für solche Basisdatentypen möglich, die selber komplexe Datentypen sind, allerdings ein einfaches Inhaltsmodell besitzen, also in ihrer Deklaration selber den Container `xs:simpleContent` enthalten. Konzeptuell erlaubt diese Form der Erweiterung nur das Erweitern durch neue Attribute. Elemente, die den deklarierten Datentyp als Typ verwenden, können nach der Erweiterung weiterhin nur Inhalte besitzen, die dem Basisdatentyp entsprechen, allerdings ist die Verwendung neuer Attribute möglich. Das Hinzufügen neuer Elemente ist in dieser Erweiterungstechnik nicht erlaubt, da diese in einfachen Inhaltsmodellen nicht vorkommen dürfen [SWW11].

Eine Alternative zur Ableitung durch Erweiterung ist bei einfachen Inhaltsmodellen die Ableitung durch Einschränkung. Diese gestattet es, einen komplexen Basisdatentyp mit

einfachem Inhaltsmodell in Bezug auf die Attributverwendung und seine Facetten einzuschränken. Diese Einschränkung wird durch Verwendung des `xs:restriction`-Elements als Kindelement des `xs:simpleContent`-Elements deklariert. Die Syntax zur Facetteneinschränkung entspricht dem Vorgehen, das im Zusammenhang mit der Ableitung durch Einschränkung bei einfachen Datentypen vorgestellt wurde. Bezüglich der Attributverwendung kann man innerhalb der Einschränkung beispielsweise festlegen, dass ein innerhalb des Basistyps vorhandenes Attribut für den eingeschränkten Datentyp nicht mehr verwendet werden darf oder angegeben werden muss.

3.2.4.2.1.6.2 Komplexe Datentypen mit komplexem Inhaltsmodell

Im Gegensatz zum einfachen Inhaltsmodell kann ein komplexes Inhaltsmodell weitere Kindelemente deklarieren und legt dadurch eine verschachtelte Hierarchie fest. Auch für komplexe Inhaltsmodelle bietet XML Schema die beiden vorab genannten Ableitungsansätze. Diese unterscheiden sich allerdings sowohl syntaktisch als auch inhaltlich von ihrer Umsetzung bei einfachen Inhaltsmodellen. Für die Deklaration eines komplexen Inhaltsmodell verwendet man das `xs:complexContent`-Element, das wiederum als Kindelemente `xs:restriction` und `xs:extension` zulässt [va02].

Zunächst sei auf die Ableitung durch Erweiterung eingegangen. Diese basiert auf einem komplexen Basisdatentyp, der über ein komplexes Inhaltsmodell verfügt. Diesem können weitere Attribute und Elemente hinzugefügt werden, die an bereits vorhandene Attribute und Elemente angehangen werden. Eine Änderung der Reihenfolge der Elemente im Basisdatentyp kann durch die Ableitung nicht erreicht werden, nur das Hinzufügen neuer Strukturen ist möglich. Die Erweiterungsdeklaration verwendet das `xs:extension`-Element, innerhalb dessen man den eindeutigen Namen des zu erweiternden komplexen Datentyps angibt. Anschließend deklariert man unter Verwendung der bereits vorgestellten unterschiedlichen Container `xs:sequence`, `xs:all` oder `xs:choice` eine Menge von Elementen und Attributen, die dem Basisdatentyp hinzugefügt werden sollen.

Die Ableitung durch Einschränkung ist die zweite Ableitungstechnik. Diese unterscheidet sich konzeptuell stark von der verwandten Technik bei einfachen Inhaltsmodellen. Anstatt wie dort beispielsweise durch eine Redefinition von Elementkardinalitäten bestimmte Elemente auszuschließen oder die Attributverwendung zu regeln, entspricht die Ableitung

durch Einschränkung einer vollständigen Neudeklaration des gesamten Inhaltsmodells, bei der man nicht benötigte Komponenten weglässt [SWW11].

3.2.4.2.1.7 Mechanismen zur Auslagerung und Wiederverwendung

Bei komplexen Gegenstandsbereichen, die in XML Schema modelliert werden, können auch die erzeugten Schema-Dokumente schnell komplex und umfangreich werden. Mit wachsender Komplexität sind darum Mechanismen wünschenswert, die es gestatten, Schemata zu gestalten, die auf mehrere Dokumente aufgeteilt werden können. Kern solcher Aufteilungsmechanismen sind Sprachmittel, über die nachträglich aus den verschiedenen Einzeldokumenten wieder ein logisches Gesamtdokument erzeugt werden kann. XML Schema bietet hierfür den *Inklusions*- und den *Import*-Mechanismus, die im folgenden Abschnitt vorgestellt werden.

Prinzipiell sind die Auslagerungsmechanismen ein Kernkonzept für die Wiederverwendbarkeit von XML Schema-Dokumenten. Erst die Möglichkeit, vorhandene Strukturen aus externen Dokumenten in eigene Schemata einzubinden und dort zu verwenden, macht XML Schema wiederverwendbar. Diese Technologien erlauben beispielsweise den Aufbau von Datentypsammlungen für bestimmte Anwendungsbereiche, die zentral gepflegt und in beliebigen anderen Schemata eingebunden werden können. Auch das hier vorliegende System macht intensiven Gebrauch von diesen Möglichkeiten, um dadurch die Wartbarkeit der Schemata zu erhöhen und gleichzeitig unnötige Deklarationsreplikationen zu vermeiden.

Die Auslagerung von Schemata ist unabhängig von der Art und Weise, wie diese in anderen Dokumenten eingebunden werden. Die ausgelagerten Schemata können vollwertige XML Schema-Dokumente sein, die zur Validierung von Instanzdokumenten eingesetzt werden können. Dies muss aber nicht der Fall sein. Es ist ebenso möglich, Schemadokumente zu erzeugen, die wohlgeformt sind, allerdings nur Datentypdeklarationen enthalten, seien es nun einfache oder komplexe Datentypen. Die Deklaration eines Elements ist nicht zwingend erforderlich. Solche reinen Datentypsammlungen ohne Elementdeklarationen können offensichtlich keine XML-Instanzdokumente validieren, die deklarierten Datentypen können aber nach der Einbindung in ein anderes Schemadokument innerhalb diesem verwendet werden.

3.2.4.2.1.7.1 Einbindung durch Inklusion

Die Inklusion ist die einfachere Form der Einbindung eines externen Dokuments in ein Schema. Diese deklariert man durch das `xs:include`-Element, das über ein Attribut `schemaLocation` verfügt, das die URI des einzubindenden Dokuments erwartet. Die Einbindung mittels Inklusion kann man sich vorstellen wie eine einfache Ersetzung des `xs:include`-Tags durch den Inhalt der einzubindenden Datei. Anschließend kann man beispielsweise die innerhalb des eingebundenen Dokuments deklarierten Datentypen wie gewohnt verwenden. Dazu gehört auch die Möglichkeit, eingebundene Datentypen zu überschreiben. Hierfür stellt XML Schema das Schema-Element `xs:redefine` zur Verfügung [SWW11]. Auch dafür existieren die zwei vorab diskutierten Ableitungsformen, der grundsätzliche Ansatz ist dabei sehr ähnlich. Da innerhalb des vorliegenden Systems auf diese Form der Ableitung verzichtet wird, soll auch an dieser Stelle nicht weiter darauf eingegangen werden.

3.2.4.2.1.7.2 Einbindung durch Import

Der Unterschied zwischen der Einbindung durch Inklusion und der Einbindung durch Import liegt im Umgang mit Namensräumen. Auf diese wird im späteren Abschnitt „Namensräume in XML Schema“ detailliert eingegangen, im Zusammenhang mit der Einbindung ausgelagerter Strukturen ist es allerdings zentral, ob das einbindende und das einzubindende Dokument innerhalb des gleichen Namensraumes liegen. Ein solcher Namensraum wird über das `targetNamespace`-Attribut des `xs:schema`-Elements festgelegt. Diesem kann entweder ein Verweis auf einen Namensraum in Form einer URI zugewiesen werden oder es wird explizit festgelegt, dass kein Namensraum verwendet werden soll. Unterscheiden sich die Namensraumdeklarationen zwischen den beiden Dokumenten, so ist eine Einbindung durch Inklusion nicht möglich. In diesem Fall muss auf die Import-Einbindung zurückgegriffen werden.

Das `xs:import`-Element verfügt über zwei Attribute. Dies ist zum einen das `schemaLocation`-Attribut, über das auch das `xs:include`-Element verfügt. Auch beim `xs:import`-Element legt dieser Wert über eine URI fest, wo das einzubindende Dokument zu finden ist. Darüber hinaus gibt man über das `namespace`-Attribut den Namensraum des einzubindenden Dokuments an. Dadurch wird implizit festgelegt, dass die Strukturen, die aus dem Zieldokument eingebunden werden, ihren eigenen Namensraum behalten und nicht

in den Namensraum des einbindenden Dokuments integriert werden [W304b]. Sobald Namensräume deklariert werden, müssen sämtliche globalen Strukturen innerhalb des Dokuments durch Verwendung des Namensraum-Präfixes qualifiziert werden. Dies gilt auch beim Import eines externen Schema-Dokuments.

3.2.4.2.1.8 Namensräume in XML Schema

Im Kontext der Wiederverwendbarkeit von Schemadeklarationen und der Auslagerung von Schemabestandteilen in unterschiedliche Schemadateien spielt das Konzept der Namensräume eine wichtige Rolle. Diese Technik existiert nicht nur in XML Schema, sondern ist ebenso wichtiger Bestandteile vieler Programmiersprachen. Die grundsätzliche Idee des Namensraum-Konzepts zielt darauf ab, Kollisionen in der Benennung einzelner Elemente zu verhindern. Dabei erhöht sich die Kollisionswahrscheinlichkeit mit steigender Anzahl an Schemata, die zu einem logischen Dokument zusammengefasst werden. Kollisionen entstehen durch Mehrdeutigkeiten, die aus der gleichen Benennung einzelner Schemakomponenten resultieren. Verwendet man nur eine einzige Schemadatei und inkludiert keinerlei externe Komponenten, so können auch keine Mehrdeutigkeiten entstehen, sofern man auf eine eindeutige Benennung achtet. Bei umfangreichen Schemata, die über Importe eine potentiell große Anzahl von weiteren Schemadokumenten einbinden, kann die Kollisionsvermeidung dagegen zu einem komplexen Problem werden. Diese Problematik verschärft sich, sobald es sich um Dokumente handelt, die der Schemaautor nicht selber verfasst hat. Namensräume stellen eine Lösung für diese Problematik dar und gestatten die Kombination verschiedener externer Vokabulare innerhalb eines logischen Dokuments [va02].

Das zentrale Sprachmittel für die Deklaration von Namensräumen in XML Schema ist das `xmlns`-Attribut. Diesem kann als Wert ein beliebiger Bezeichner zugewiesen werden, der als Name für den Namensraum fungiert. Dabei sollte der Bezeichner so gewählt werden, dass er innerhalb der Anwendung, für die das Schema verfasst wird, eindeutig ist. Eine gängige Praxis ist es darum, URIs als Namensraum-Bezeichner zu verwenden, dies ist aber durch den Standard nicht explizit vorgeschrieben. Zusätzlich zum Namensraumbezeichner kann eine Kurzform für diesen angeben werden, die innerhalb der Schema- und Instanzdokumente anstelle des vollständigen Bezeichners verwendet werden kann.

Deklariert man einen Namensraum über das `xmlns`-Attribut innerhalb eines Elements, so befinden sich alle Kindelemente dieses Elements automatisch innerhalb dieses Namensraumes. Dies impliziert, dass ein im Wurzelelement eines XML Schemas angegebener Raum für alle Elemente innerhalb des Dokuments gilt. Dabei ist es möglich, Namensräume hierarchisch zu schachteln, indem Kindelemente wiederum eigene Namensräume deklarieren. Deren Kindelemente befinden sich dann im neu deklarierten Namensraum.

Die Angabe von Kurzformen für deklarierte Namensräume bietet sich an, da diese Kurzformen als Präfixe für die Qualifizierung von Elementen dienen. Eine solche Qualifizierung drückt durch die Präfixverwendung explizit die Zugehörigkeit des jeweiligen Elements zu einem bestimmten Namensraum aus.

Grundsätzlich ist es möglich, auf die Angabe einer Kurzform zu verzichten. In diesem Fall ist auch eine Qualifizierung der Kindelemente nicht erforderlich. Dabei ist allerdings zu beachten, dass es in jedem Schemadokument nur einen einzigen Standardnamensraum ohne Kurzformangabe geben darf. Elemente, die zu diesem Namensraum gehören, benötigen keine weitere Qualifikation [SWW11]. Sobald allerdings weitere Namensräume innerhalb des Schemas verwendet werden, ist eine Qualifikation zwingend erforderlich, da es sonst zu Mehrdeutigkeiten kommen kann. In einem solchen Fall ist es ratsam, Namensraumkurzformen für sämtliche vorkommenden Namensräume zu verwenden. Üblicherweise erledigt man dies im Wurzelelement des XML Schema-Dokuments, dadurch können sämtliche Elemente innerhalb des Dokuments über die angegebenen Kurzformen qualifiziert werden. Eine solche Elementqualifikation empfiehlt sich speziell bei großen und komplexen Dokumenten, da direkt erkennbar ist, zu welchem Namensraum ein Element gehört. Für Attribute gilt dabei, dass diese sich im gleichen Namensraum befinden, wie das Element, zu dem sie gehören.

Bezüglich der Namensraumdeklaration gelten für das `xs:schema`-Element eine Reihe von Besonderheiten. Dieses verfügt über zusätzliche Attribute, die im Zusammenhang mit Namensräumen eine Rolle spielen. Zunächst ist das `targetNamespace`-Attribut zu nennen. Durch die Angabe eines Namensraum-Bezeichners für dieses Attribut legt man fest, welcher Namensraum durch das aktuelle Schema beschrieben wird [va02]. Für sämtliche Instanzdokumente dieses Schemas führt die Angabe eines Zielnamensraums dazu, dass alle verwendeten Elemente und Attribute innerhalb dieses Namensraums vorkommen müssen. Neben der Angabe des Zielnamensraumes verfügt das `xs:schema`-Element unter anderem

noch über die Attribute `elementFormDefault` und `attributeFormDefault`. Diese beiden Attribute können jeweils die Werte „`qualified`“ oder „`unqualified`“ annehmen [W304a]. Sie legen fest, ob Elemente und Attribute in Instanzdokumenten des Schemas standardmäßig qualifiziert oder unqualifiziert vorkommen müssen. Die Verwendung dieser Attribute im `xs:schema`-Element legt ein Standardverhalten für sämtliche Komponenten des Schemas fest. Für bestimmte Komponenten kann es allerdings wünschenswert sein, von diesem Standardverhalten abzuweichen und eine alternative Festlegung zu verwenden. Dies kann man auf der Ebene von Element- und Attributdeklarationen über das `form`-Attribut erreichen. Auch dieses kann die Werte „`qualified`“ oder „`unqualified`“ zugewiesen bekommen und deklariert dadurch, ob die jeweilige Komponente in Instanzdokumenten qualifiziert werden muss oder ob dies nicht der Fall ist.

Das übliche Vorgehen, das auch im vorliegenden System für die Erzeugung der XML Schema-Strukturen gewählt wurde, verwendet für jede einzelne Schemadatei einen einzelnen Namensraum. Die Zusammenführung dieser verteilten Dateien erfolgt dann durch die vorab erläuterte Import-Technik, bei der man sowohl die deklarierten Strukturen als auch den angegebenen Namensraum aus der einzubindenden Datei übernimmt. Bei einem solchen Import muss beachtet werden, dass der Namensraum innerhalb des namespace-Attributs des `xs:import`-Elements exakt mit dem Namensraumbezeichner übereinstimmen muss, der im eingebundenen Dokument als Zielnamensraum deklariert wurde. Sofern dies der Fall ist, funktioniert die Strukturübernahme problemlos [SWW11].

4 Basistechnologien für die prozedurale Inhaltsgenerierung

Der Semantic Building Modeler gehört zur Gruppe der prozeduralen Ansätze, die sich dadurch auszeichnen, dass sie Gebäudemodelle anhand eines nutzerdefinierten Konstruktionsprozesses automatisiert erzeugen. Verfahren dieser Art werden im Abschnitt „Prozedurale Ansätze“ an späterer Stelle detailliert vorgestellt und dabei in *prozedurale* und *regelbasierte* Technologien unterteilt. Prozedurale Systeme verwenden Programmiersprachen, um Algorithmen zur Gebäudekonstruktion zu implementieren. Demgegenüber stehen regelbasierte Verfahren, bei denen der Nutzer innerhalb eines Ersetzungssystems eine Menge von Regeln definiert, mittels derer die Gebäudekonstruktion erfolgt. Bevor der Einsatz sowohl prozeduraler als auch regelbasierter Verfahren für die automatisierte Berechnung von 3D-Gebäudemodellen thematisiert wird, werden zunächst die theoretischen Konzepte erörtert, auf denen diese Verfahren basieren. Darum befasst sich dieser Abschnitt mit der Diskussion der zentralen Merkmale von Programmiersprachen im Allgemeinen und den speziellen Fähigkeiten der hier verwendeten Sprache Java, bevor im darauffolgenden Abschnitt die Konzepte und Formalismen von Ersetzungssystemen thematisiert werden.

4.1 **Programmiersprachen**

4.1.1 **Entwicklung unterschiedlicher Programmierparadigmen**

Programmiersprachen stellen eine Möglichkeit dar, einem Computer Befehle zu erteilen, die dieser anschließend ausführt. Dabei werden die einzelnen Befehlsschritte als Text eingegeben. Jede Programmiersprache besteht aus einer Menge von Funktionen, Operatoren und Konstanten. Die *Syntax* einer Programmiersprache legt dabei fest, wie Anweisungen aufzubauen und zu kombinieren sind, damit gültige Befehle entstehen, die vom Computer verstanden werden können. Frühe Programmiersprachen zeichneten sich durch ihre große Nähe zur Hardware des Zielsystems aus, auf dem die erstellten Programme ausgeführt werden sollten. Zu dieser Zeit bildeten die Sprachen die Operationen der zugrundeliegenden Prozessoren ab und ermöglichten dadurch die Formulierung von Programmen auf Maschinenebene. Die zugrunde liegende Maschinensprache gibt dem Nutzer einen direkten Zugang zur gesamten vorhandenen Hardware des Rechners, dazu gehören neben der *Central Processing Unit* (CPU) beispielsweise der Hauptspeicher und angeschlossene Peripheriegeräte. Der Vorteil solcher hardwarenahen Programmiersprachen liegt in der großen Kontrolle, die der Nutzer über die Ausführung des Programms hat, da sie es

ermöglichen, zeitkritische Bereiche innerhalb des Programmcodes direkt in Anweisungen der CPU zu formulieren und dadurch potentiell große Laufzeitgewinne zu erreichen. Demgegenüber steht aber eine Reihe großer Nachteile, die dazu geführt haben, dass Maschinensprache nur noch in speziellen Anwendungsbereichen wie der Betriebssystemprogrammierung intensiv eingesetzt wird.

Der potentiell größte Nachteil von maschinensprachlichen Konstrukten ist die fehlende Portierbarkeit. Dies hängt damit zusammen, dass die zugrundeliegenden Operationen der Sprache für einen bestimmten Prozessortyp implementiert sind. Diese Operationen werden durch Mikroprogramme umgesetzt, so dass Programme in Maschinensprache zumindest auf neuen Versionen eines bestimmten CPU-Typs lauffähig bleiben, eine Ausführung auf einer anderen Architektur ist dagegen aufgrund der großen Hardwarenähe nicht möglich [GS02]. Neben der fehlenden Portierbarkeit fehlen maschinennahen Sprachen eine Reihe von Konstrukten, durch die sich höhere Programmiersprachen auszeichnen, beispielsweise Kontrollstrukturen und Schleifen. Solche Konstrukte müssen in Maschinensprache durch Sprunganweisungen simuliert werden, was dazu führt, dass solche Programme schwer lesbar und somit fehleranfällig sind.

Moderne Programmiersprachen orientieren sich darum weniger an der zugrunde liegenden Hardware, sondern stärker an den Problemstellungen, die durch ihren Einsatz gelöst werden sollen. Hierfür bieten sie Sprachfeatures, die eine abstrakte Formulierung des Lösungsweges ermöglichen und dabei die konkrete Hardware außen vor lassen. Durch diese Abstraktion soll es möglich sein, das gleiche Programm auf unterschiedlichen Systemen auszuführen.

Im Zusammenhang mit Programmiersprachen gibt es eine Reihe von Unterscheidungskriterien. Ein gängiges Kriterium orientiert sich am zugrundeliegenden Berechnungsmodell der verwendeten Sprache und unterteilt diese in *imperative* und *deklarative* Sprachen, wobei deklarative Sprachen meist noch in *funktionale* und *logische* Sprachen untergliedert werden. „Imperative Sprachen beschreiben eine Berechnung als eine Folge von Zustandsübergängen einer Menge von Zustandsvariablen, die in erster Näherung den Speicher des benutzten Rechners repräsentieren“ [Re06]. Dabei wird jede Zustandsüberführung durch einen einzelnen Befehl ausgelöst. Zu dieser Menge von Sprachen gehören unter anderem die erwähnten Maschinensprachen und auch die für das hier vorgestellte System verwendete Sprache Java.

Bei funktionalen Sprachen stehen dagegen mathematische Abbildungen im Zentrum. Ein Programm besteht aus einer Menge von Funktionsdefinitionen und geschachtelten Funktionsaufrufen. Theoretisches Fundament solcher Sprachen ist das *Lambda-Kalkül*. Demgegenüber basieren logische Sprachen auf den mathematischen Konzepten der *Prädikatenlogik*. In solchen Programmiersprachen „bestätigt man eine Formel der mathematischen Logik, indem man das Gegenteil, die Negation der Formel, zu widerlegen versucht. Die speziellen Werte der vorkommenden logischen Variablen, die zu dieser Widerlegung führen, bilden das Ergebnis der Berechnung“ [Re06].

Deklarative Sprachen zeichnen sich meist durch ein Variablenkonzept aus, das der Mathematik entlehnt ist. Dabei definiert eine Variable einen Namen für einen bestimmten Wert, der innerhalb eines bestimmten Kontextes fest und somit unveränderlich ist. Aus diesem Grunde verfügen deklarative Sprachen nicht über Zuweisungsoperatoren und Kontrollstrukturen.

Neben der Unterscheidung von Programmiersprachen anhand ihres Berechnungsmodells existiert eine weitere Untergliederung für die Klasse der imperativen Sprachen, die sich an der verwendeten Programmiermethodik orientiert. Diese Einteilung klassifiziert befehlsorientierte Sprachen in prozedurale, modulare und objektorientierte Sprachen. Nachfolgend wird die Entwicklung von prozeduralen Sprachen bis hin zu den heute weit verbreiteten objektorientierten Sprachen aufgezeigt.

Zentral aus Sicht des vorliegenden Systems sind *Wartbar*- und *Erweiterbarkeit*. Aus diesem Grund wurde Java als objektorientierte Sprache für die Implementation des Prototyps verwendet. Besondere Sprachfeatures wie *Reflection* und die gute Portierbarkeit von Java durch die Verwendung einer virtuellen Maschine, die Java von anderen objektorientierten Sprachen wie C++ unterscheidet, werden nach der Erörterung der theoretischen Konzepte objektorientierter Sprachen am Ende des Abschnittes erläutert.

4.1.2 **Prozedurale und modulare Programmierung**

Das prozedurale Programmierparadigma ist sowohl Vorläufer der modularen als auch der objektorientierten Programmierung. Der wichtigste Abstraktionsmechanismus ist die Prozedur. Prozedurale Programme bestehen aus einer Menge von Prozeduren, die Algorithmen zur Lösung eines Problems implementieren. Im Zuge der Verarbeitung rufen sich diese Prozeduren wechselseitig auf und verändern den globalen Programmzustand. Die

zentrale Unterscheidung zur objektorientierten Programmierung ist die Unabhängigkeit von Prozeduren und den jeweiligen Daten, die von diesen verarbeitet werden. Daten können an beliebigen Stellen im Programm vorkommen und von beliebigen Prozeduren verändert werden. Dadurch werden Daten und Prozeduren voneinander getrennt und die Daten konzeptuell dem Algorithmus untergeordnet [GS02], weshalb in der prozeduralen Programmierung der Einsatz globaler Daten eine zentrale Rolle spielt [Pu07]. Die Daten selber werden als *passiv* aufgefasst, die von den *aktiven* Elementen in Form von Prozeduren verändert werden.

Objektorientierte bietet im Gegensatz zur prozeduralen Programmierung aufgrund der Zusammenfassung von Daten und Methoden innerhalb eines Objekts eine Möglichkeit, die Programmausführung selber anhand der Zustandsänderungen einzelner Objekte zu beschreiben, ohne dabei globale Änderungen der Programmzustände berücksichtigen zu müssen. Diese Zusammenfassung von Daten und Methoden in einem Objekt bezeichnet man als *Kapselung*. Durch die Kapselung soll garantiert werden, dass auf Objekte nur noch über ihre Methoden zugegriffen werden kann, direkte Zugriffe auf die Datenfelder werden dagegen vermieden [GS02]. Dadurch ist es beispielsweise möglich, Datenänderungen innerhalb von Setter-Methoden zu validieren oder auch abzulehnen, wenn eine solche Änderung nicht durchgeführt werden soll. Im prozeduralen Paradigma sind Programmzustände somit „nur global bewertbar, im objektorientierten Paradigma lokal gekapselt“ [Pu07].

Prozedurale Programme weisen aus Sicht der Wartbarkeit verschiedene Nachteile auf. Dies resultiert daraus, dass der Einsatz globaler Daten dem Prinzip der Lokalität entgegensteht, das fordert, dass der „Effekt jeder Programmänderung auf einen kleinen Programmteil beschränkt“ [Pu07] sein sollte. Änderungen globaler Variablen, speziell dann, wenn viele verschiedene Prozeduren auf diese lesend und schreibend zugreifen, sind für den Programmierer nur schwer nachvollziehbar, was die Fehleranfälligkeit solcher Software erhöht. Der Kern des Problems ist darin zu sehen, dass a priori keinerlei Aussagen darüber getroffen werden können, von welcher Stelle innerhalb eines Programms eine Änderung der Daten vorgenommen werden kann. Außerdem hat die Verwendung globaler Daten den Nachteil, dass eine Änderung der Struktur der Daten im Rahmen der Weiterentwicklung des Programms häufig eine Anpassung sämtlicher Prozeduren erfordert, die auf diese zugreifen.

Darüber hinaus verlangt die prozedurale Programmierung die korrekte Initialisierung sämtlicher verwendeter Datenobjekte und der Programmierer muss darauf achten, dass die

an eine Prozedur übergebenen Daten korrekt sind. Dies macht prozedurale Programme fehleranfällig und kann sich negativ auf deren Wartbarkeit auswirken [PK02]. Objektorientierte Programmierung kann die Wartbarkeit und Erweiterbarkeit von Software durch den Einsatz des *Datenabstraktionsprinzips* deutlich erhöhen, so dass Änderungen nur noch lokal und nicht an vielen unterschiedlichen Stellen vorgenommen werden müssen. Das Datenabstraktionsprinzip beschreibt das Verstecken der Implementation zusammen mit dem bereits erwähnten Konzept der Kapselung, da durch die Anwendung dieses Prinzips die Daten des Objekts selber nicht mehr sichtbar und von außen manipulierbar sind, sondern abstrakt.

Die prozedurale Programmierung erwies sich im Laufe der Zeit aufgrund der immer umfangreicheren und komplexeren Programme speziell für die Entwicklung im Team als unzureichend. Große Softwareprojekte erfordern die Zusammenarbeit einer Gruppe von Entwicklern. Diese sollten in der Lage sein, die Komponenten eines Systems, für die sie zuständig sind, unabhängig von anderen Entwicklern zu implementieren und zu testen. Man benötigte also einen Weg, um ein Gesamtsystem in einzelne, möglichst unabhängige Teile zu zerlegen, die dann von verschiedenen Programmierern entwickelt und zum Abschluss des Projekts zusammengefügt werden konnten.

Das Konzept des modularen Programmierens stellt eine solche Möglichkeit dar und wurde in den 70er Jahren vorgestellt. In der prozeduralen Programmierung waren Prozeduren der zentrale Abstraktionsmechanismus. Ein Modul im Sinne der modularen Programmierung beinhaltet neben den Prozeduren darüber hinaus Konstanten, Variablen und Typen und fasst diese zu einer Einheit zusammen. Die Zusammenfassung von Prozeduren und Daten bietet eine Möglichkeit, ein komplexes System in eine Menge von Teilsystemen zu unterteilen, die dann von verschiedenen Entwicklern umgesetzt werden können. Ein Modul selber kann als Zusammenfassung von logisch zusammengehörigen Komponenten in einem einzelnen Baustein aufgefasst werden.

Man unterscheidet die *Exportschnittstelle* eines solchen Moduls von seinem *Rumpf*. An der Exportschnittstelle gibt das Modul die Dienste bekannt, die es anderen Modulen zur Verfügung stellt. Exportierte Dienste können sowohl Prozeduren sein, aber auch Datenobjekte wie Variablen, Konstanten oder Datentypen. Der Rumpf stellt die Realisierung der Dienste bereit, die nach außen nicht sichtbar ist. Man spricht hier auch von unterschiedlichen *Sichten* auf die Komponenten des Systems. Die Exportschnittstelle entspricht der *Klientensicht*, die für den Nutzer des Moduls interessant ist. Konzeptionell

legt diese Sicht ein Protokoll für den Umgang mit diesem Baustein fest, das von den Klienten eingehalten werden muss. Demgegenüber stellt die *Herstellersicht* die Sicht dar, die der Entwickler des Moduls einnimmt, da er für die Implementation der Exportschnittstelle zuständig ist [Na99]. Die Trennung der verborgenen internen Repräsentation von den nach außen sichtbaren Diensten wird als *Information Hiding* bezeichnet und spielt auch in der objektorientierten Programmierung eine zentrale Rolle.

Die Kommunikation der Module untereinander erfolgt über deren Exportschnittstellen. Ein Modul, das auf Dienste eines anderen Moduls zugreifen möchte, muss nur dessen Exportschnittstelle kennen, die Implementierung und die interne Struktur bleiben dagegen verborgen. Dadurch erbringt der Einsatz von Modulen für die Strukturierung von Software einen großen Vorteil. Durch die Festlegung und Bekanntmachung der Dienste eines Moduls an seiner Exportschnittstelle ist es möglich, die Implementierung des Moduls selber auszutauschen, ohne dabei andere Module modifizieren zu müssen.

Auch aus Sicht der Wiederverwendung von Code spielt der Einsatz von Modulen als Strukturierungswerkzeug eine wichtige Rolle. Prinzipiell gilt, dass sich der Entwicklungsaufwand von Softwareprojekten durch die Wiederverwendung existierenden Codes reduzieren lässt, da es nicht erforderlich ist, alle Komponenten eines Systems neu zu entwickeln, was zu einer Zeit- und somit Kostenersparnis führt. Außerdem ist es sinnvoll, bereits entwickelten und getesteten Code einzusetzen, da dieser tendenziell weniger fehleranfällig sein wird, als eine zunächst ungetestete Neuimplementation. Aufgrund der Kommunikation der Module über ihre Exportschnittstellen wird die Kopplung der Module auf das unbedingt Notwendigste reduziert, was wiederum die Austauschbarkeit einzelner Module und die Wartbarkeit des gesamten Systems verbessert.

Allerdings löst die Verwendung von Modulen das Problem des Datenzugriffs, das vorab im Rahmen der prozeduralen Programmierung bereits erwähnt wurde, nur oberflächlich. Module stellen zunächst ein Strukturierungskonzept zur Verfügung, das verwendet werden kann, um ein komplexes System in eine Menge einzelner Teilbausteine aufzuteilen. Durch den gezielten Einsatz des Information Hidings ist es möglich, die Module auszutauschen, so lange sich die Exportschnittstelle und somit die bereitgestellten Dienste nicht ändern. Es verschiebt aber die Probleme, die aus der Verwendung globaler Daten resultieren, nämlich Fehleranfälligkeit und Erschwernis von Wartung und Erweiterung, konzeptionell um eine Ebene nach unten, nämlich auf Modulebene. Innerhalb eines Moduls sind Daten allerdings immer noch global. Bei umfangreichen Modulen kann man demnach in die gleiche Situation

geraten, die auch bei der prozeduralen Programmierung auftritt. Es ist nicht ohne Weiteres feststellbar, von welchen Stellen eines Moduls Änderungen der *modulglobalen* Daten auftreten. Dieses Problem soll durch die noch engere Kopplung von Daten und Methoden gelöst werden, die von der objektorientierten Programmierung angestrebt wird [MS02]. Ein Zugriff auf Datenelemente ist dann nur noch über Methoden des Objekts möglich, Direktzugriffe auf Daten werden verhindert. Dadurch haben Änderungen an der Datenstruktur nur noch lokale Auswirkungen, da eine Anpassung der Objektzugriffsmethoden ausreicht, die die Aufrufer der Zugriffsmethoden im Normalfall nicht betrifft.

Ein wichtiger Punkt zur Abgrenzung der modularen und der objektorientierten Programmierung ist die Betrachtung von Klassen und Objekten. Während die objektorientierte Programmierung hier eine klare Unterscheidung vornimmt und Objekte als Instanzen von Klassen betrachtet, setzt die modulare Programmierung diese gleich. Module entsprechen Objekten, die ohne Verwendung von Klassen direkt vom Compiler erzeugt werden. Eine weitere Abgrenzung betrifft den Fokus der beiden Programmierparadigmen. Während bei der modularen Programmierung auf die „Zerlegung von komplexen Programmieraufgaben in unabhängige Module“ [GS02] abgezielt wurde, geht es bei der objektorientierten Programmierung um einen abstrakten Datenbegriff. Kern dieses Datenbegriffs ist die Zusammenfassung von Datenfeldern eines Datenobjekts mit den Operationen, die auf diesen Datenfeldern definiert sind. Im objektorientierten Sprachgebrauch bezeichnet man diese Operationen üblicherweise als Methoden und die Datenfelder als Attribute. In der objektorientierten Programmierung spielt die bereits erwähnte Datenkapselung eine zentrale Rolle. Der Einsatz dieses Prinzips führt in der Praxis dazu, dass die Attribute eines Objekts, immer nur über dessen Methoden gelesen und geschrieben werden können und die eigentliche Struktur des Objekts nach außen nicht sichtbar ist.

4.1.3 **Charakteristika des objektorientierten Programmierparadigmas**

Kern des objektorientierten Paradigmas ist das Objekt. Ein solches Objekt besitzt während seiner Existenz einen Zustand, der definiert ist durch die zu diesem Zeitpunkt zugewiesenen Werte seiner Attribute sowie durch die zu anderen Objekten aufgebauten Objektbeziehungen. Ändern sich entweder die Attributwerte oder die Objektbeziehungen, so ändert sich auch der Objektzustand.

Das Verhalten eines Objekts wird durch eine Menge von Operationen beschrieben, die das Objekt anderen Objekten zur Verfügung stellt. Ist das Objekt vollständig gekapselt, verbirgt also seine interne Repräsentation nach außen, so ist ein lesender oder schreibender Zugriff ausschließlich über die Objektmethoden möglich, was zu den bereits erwähnten Vorteilen der Kapselung führt. Aufgrund des abstrakten Datenbegriffs der objektorientierten Programmierung bilden Zustand und Verhalten des Objekts eine Einheit. Ist das Objekt in diesem Sinne gekapselt, so realisiert es das *Geheimnisprinzip*, es versteckt seine interne Repräsentation nach außen [Ba05]. Dies entspricht dem vorab erwähnten Konzept des Information Hidings.

Jedes Objekt besitzt eine eindeutige *Objektidentität*. Es handelt sich dabei um eine Eigenschaft, die ein Objekt von allen anderen Objekten unterscheidet. Zwei Objekte bleiben durch ihre Identität unterscheidbar, auch wenn sie die gleichen Attributwerte besitzen. Damit dies der Fall ist, darf sich die Identität eines Objekts nicht ändern. Man unterscheidet demnach in der Objektorientierung auch zwischen identischen und gleichen Objekten. Zwei Objekte sind demnach *identisch*, wenn sie die gleiche unveränderliche Identität besitzen. Sie sind *gleich*, wenn sie dieselben Attributwerte für ihre Attribute aufweisen. Somit sind identische Objekte zwangsläufig auch gleich, gleiche Objekte müssen aber nicht identisch sein. Dieser Fall tritt beispielsweise auf, wenn ein Objekt durch eine Kopie aus einem anderen Objekt hervorgeht. Im Moment der Erzeugung sind die Objekte gleich, aber nicht identisch, da jedes Objekt eine eigene Identität besitzt [Ba05].

Eine Klasse im objektorientierten Paradigma ist eine Abstraktion einer Menge von Objekten, deren Gemeinsamkeiten sie beschreibt. Ziel ist die Reduktion auf das Wesentliche und nicht eine Darstellung sämtlicher Eigenschaften der Objekte. Stattdessen sollen nur die Attribute, Methoden und Beziehungen definiert werden, die für die konkrete Problemstellung relevant sind. Eine Klasse definiert die Struktur ihrer Objekte durch die Attribute. Das Verhalten wird beschrieben durch die Botschaften, auf die Objekte dieser Klasse reagieren können. Konzeptionell aktiviert eine Botschaft, die an ein Objekt dieser Klasse gesendet wurde, eine Methode gleichen Namens. Eine Klasse stellt das Muster dar, nach dem Objekte dieser Klasse instanziiert werden. Man verwendet darum auch den Begriff *Instanz* als Synonym für ein Objekt, da ein Objekt eine Instanz einer Klasse ist [Ba05].

In der Literatur findet man meist drei Charakteristika, durch die sich objektorientierte Sprachen auszeichnen. Neben der bereits erwähnten *Datenkapselung* werden *Vererbung* und *Polymorphismus* genannt.

Vererbung oder auch Generalisierung beschreibt eine Anordnung von Klassen in einer Hierarchie. Hierbei handelt es sich wiederum um ein Abstraktionskonzept objektorientierter Programmierung. Eine Klasse erbt sowohl die Attribute als auch die Methoden ihrer Oberklasse. Sie wird dann als Unterklasse bezeichnet und kann den ererbten Methoden und Attributen neue Komponenten hinzufügen. Sie ist somit spezialisierter als ihre Oberklasse, weshalb man auch von *Spezialisierung* spricht, wenn man aus einer Oberklasse eine Unterklasse ableitet [Ba05]. Da eine Unterklasse die Attribute ihrer Oberklasse erbt, sind auch die Methoden, die auf diese Attribute zugreifen, in der Unterklasse verfügbar. Bewegt man sich von der Oberklasse zu einer Unterklasse, so wird das Abstraktionsniveau niedriger, Unterklassen sind somit konkreter als ihre Oberklassen.

Eine Instanz einer Unterklasse steht zu ihrer Oberklasse in einer *is-a*-Beziehung. Dies hängt damit zusammen, dass eine Instanz der Unterklasse sowohl die Eigenschaften als auch das Verhalten ihrer Oberklasse erbt und somit auch Instanz dieser Oberklasse ist [KS09]. In Java beispielsweise manifestiert sich diese *is-a*-Beziehung darin, dass die Erzeugung eines Objekts der Unterklasse immer zuerst den Erzeugungsmechanismus ihrer Oberklasse aufruft und erst dann den eigenen. Das Konzept der Vererbung macht keinerlei Vorschriften darüber, wie viele Ebenen eine solche Hierarchie aufweisen darf. Somit kann eine Klasse Unterklasse beliebig vieler Oberklassen sein. Instanzen der Unterklasse stehen dann zu allen Oberklassen, von denen sie erben, in einer *is-a*-Beziehung.

Eine *is-a*-Beziehung sollte dabei die Semantik der realen Welt berücksichtigen. Dies kann durch objektorientierte Programmiersprachen nicht überwacht werden, da sie die Semantik einer Vererbungsbeziehung nicht überprüfen können. Es ist Aufgabe des Entwicklers, dafür zu sorgen, dass die Aussage „ein Element der Unterklasse A IST ein Element der Oberklasse B“ Sinn macht.

Ein großer Vorteil der objektorientierten Programmierung ist die hohe Wiederverwendbarkeit von Code, die man durch den konsequenten Einsatz objektorientierter Konzepte erreichen kann. Die Vererbung ist ein solches Konzept, das sowohl die Wartung der Software als auch die Wiederverwendung unterstützt. Die Wiederverwendung von Code im Kontext der Vererbung besteht darin, dass Unterklassen die Methoden ihrer Basisklassen erben. Dadurch ist der Code der Methoden nur in der Oberklasse implementiert, kann aber auch von allen Instanzen der Unterklassen ausgeführt werden, da diese die Methoden erben und aufrufen können. Der Vorteil, der aus Wartungssicht besteht, ist auch gleichzeitig ein Nachteil der Vererbung. Möchte man den

Code einer Methode ändern, so muss man dies nur in der Basisklasse tun. Die Codeänderungen wirken sich durch die Vererbung auch auf alle Unterklassen aus, weitere Anpassungen sind nicht erforderlich.

Allerdings gibt es Situationen, in denen dieser Effekt unerwünscht ist. In diesem Fall möchte man nicht, dass sich Änderungen der Oberklasse direkt auf die Unterklasse auswirken. Das Sprachmittel, das objektorientierte Programmiersprachen zur Verfügung stellt, um dieses Problem zu lösen, ist das *Überschreiben* von Methoden [KS09]. Eine Unterklasse kann die ererbten Methoden überschreiben, das heißt eine eigene Implementation der ererbten Methode vornehmen. Ein Objekt der Unterklasse ruft dann im Normalfall immer die eigene Implementation einer Methode auf, sofern eine solche existiert. Ansonsten führt sie die ererbte Methode aus.

Vererbung bietet also ein Konzept, um Coderedundanz zu vermeiden. Änderungen an Methoden in der Vererbungshierarchie können lokal ausgeführt werden und wirken sich auf alle Unterklassen der Klasse aus, in der die Modifikation vorgenommen wird.

Auch aus Sicht der Erweiterbarkeit eines Softwaresystems spielt Vererbung eine wichtige Rolle. So ist es möglich, neue Klassen an eine bestehende Klassenhierarchie anzufügen, dabei die benötigten Attribute und Methoden der Oberklasse zu erben und zu verwenden, ohne Anpassungen in diesen Oberklassen vornehmen zu müssen.

Die volle Mächtigkeit erreicht Vererbung in Kombination mit *Polymorphismus*. Diese Technik gestattet es, Programmcode einer neu hinzugefügten Klasse aus einem „alten“ Teil des Programms aufzurufen, ohne dass dieser alte Teil modifiziert werden muss. Der Begriff Polymorphismus stammt aus dem Griechischen und bedeutet *Vielgestaltigkeit*. Man spricht von Polymorphismus, wenn eine „Variable oder eine Routine gleichzeitig mehrere Typen haben kann.“ [Pu07] Polymorphismus ist dabei ein Überbegriff für eine Reihe von Technologien in der objektorientierten Programmierung.

4.1.4 **Charakteristika der Programmiersprache Java**

Nachdem nun aus theoretischer Sicht die wichtigsten Eigenschaften objektorientierter Programmierung ausführlich erläutert wurden, soll abschließend auf spezielle Sprachfeatures eingegangen werden, die die verwendete Programmiersprache Java von anderen objektorientierten Sprachen wie C++ unterscheidet.

Einer der wichtigsten Punkte ist dabei die *Portierbarkeit*. Diese wird in Java durch die Verwendung einer speziellen Laufzeitumgebung erreicht, innerhalb derer Java-Programme ausgeführt werden. Java-Anwendungen sind darum keine reinen kompilierten Sprachen, wie dies bei Sprachen wie C++ der Fall ist. Bei kompilierten Sprachen wird ein Übersetzungsprogramm (engl. *Compiler*) verwendet, das den in der Programmiersprache formulierten Programmcode in Maschinensprache übersetzt. Das Ergebnis ist ein ausführbares Programm im Binärcode, das auf dem jeweiligen Zielsystem lauffähig ist. Das Problem an diesem Ansatz ist nun aber genau die direkte Übersetzung in Maschinensprache, da diese Übersetzung wiederum für eine bestimmte Zielarchitektur erfolgt. Ein Programm, das mittels eines bestimmten Compilers für eine Zielarchitektur übersetzt wurde, ist nur auf dieser lauffähig. Soll es auf andere Architekturen übertragen werden, muss es auf diesen im besten Fall nur neu übersetzt werden, sofern innerhalb des Programmcodes keine architektur- oder betriebssystemspezifischen Konstrukte verwendet werden [GS02].

Einen anderen Ansatz bieten interpretierte Sprachen. Bei solchen Sprachen wird der Binärcode während der Ausführung des Programms durch einen Interpreter erzeugt. Der Interpreter ist plattformabhängig und erzeugt Maschinencode für seine jeweilige Zielplattform. Aufgrund der Übersetzung während der Laufzeit sind interpretierte Sprachen gegenüber kompilierten Sprachen zum Teil deutlich langsamer in der Programmausführung, weshalb für zeitkritische Anwendungen kompilierte Sprachen immer noch das Mittel der Wahl sind.

Java befindet sich zwischen kompilierten und interpretierten Sprachen. Java-Programme werden durch einen *Bytecode*-Compiler in Bytecode übersetzt. Dieser wird dann zur Laufzeit der Anwendung innerhalb einer virtuellen Maschine interpretiert. Dadurch verbindet Java die Vorteile kompilierter mit denen interpretierter Sprachen. Durch die Interpretation des plattformunabhängigen Bytecodes innerhalb der Laufzeitumgebung ist der vorkompilierte Java-Code portierbar und durch die Vorübersetzung in Bytecode darüber hinaus auch schneller als die vollständige Programmcodeübersetzung, die in interpretierten Sprachen erforderlich ist. Trotzdem erreicht Java noch nicht die gleiche Performance kompilierter Sprachen wie C++, wobei der Geschwindigkeitsunterschied mit der Verbesserung der virtuellen Maschine und der Verwendung optimierender *Hot-Spot*-Compiler in den letzten Jahren immer kleiner geworden ist [KS09]. Da das vorliegende System nicht als Renderengine für die konstruierten Gebäude konzipiert ist, sondern diese stattdessen für die nachfolgende Weiterverarbeitung in 3D-Formate exportiert, wurde die

Portierbarkeit gegenüber der Performance stärker gewichtet. Außerdem verfügt Java durch Reflection über ein Sprachfeature, das in C++ nicht Teil des Sprachstandards, allerdings für die Architektur von großer Bedeutung ist.

Reflection ist die Fähigkeit einer Programmiersprache, zur Laufzeit Informationen über die Struktur der verwendeten Klassen abzufragen. Bei der Entwicklung des *Java Development Kit* (JDK) in der Version 1.1 sahen sich die Entwickler mit einer Schwäche des bis dato implementierten Java-Sprachstandards konfrontiert, die darin bestand, dass die Struktur von Klassen und Objekten statisch war. Um ein Objekt einer bestimmten Klasse zu erzeugen oder dessen Methoden aufzurufen, war es erforderlich, dass die Klasse dieses Objekts zum Zeitpunkt der Kompilierung vollständig bekannt ist. Für die meisten Anwendungen reicht dies vollkommen aus, allerdings existieren Anwendungsarchitekturen, bei denen dies nicht der Fall ist. Dazu gehören beispielsweise Anwendungen, die durch Plug-Ins nachträglich erweiterbar sein sollen. Ein gutes Beispiel für ein solches System ist die weit verbreitete Java-Entwicklungsumgebung *Eclipse*[2] [Si04]. Allgemein spielen solche Fähigkeiten für hochkonfigurierbare Anwendungen eine wichtige Rolle, bei denen die Möglichkeit bestehen soll, die Klassenstruktur nachträglich zu erweitern und neue Klassen ohne Anpassungen des Codes zu verwenden. Um die Entwicklung solcher Programme zu erleichtern, wurde im JDK 1.1 die Reflection-API eingeführt.

Kern der Java-Reflection ist die Metaklasse `Class`, die es ermöglicht, Klassendefinitionen selber wiederum als Objekte zu verwenden. Der Zugriff auf ein solches Klassenobjekt erfolgt über die `getClass()`-Methode, die in der Klasse `Object` definiert ist. Da sämtliche definierten Klassen automatisch von dieser Klasse abgeleitet werden, ist jedes instanziierte Objekt in der Lage, Zugriff auf sein Klassenobjekt zu erhalten. Das Klassenobjekt selber wiederum ermöglicht unter anderem die Abfrage sämtlicher definierter Methoden, eventuell vorhandener Superklassen oder innerhalb der Klasse verfügbarer Felder. Weiterhin kann man über ein solches Klassenobjekt Zugriff auf Methoden und Konstruktoren erhalten, die anschließend ausgeführt werden können. Klassenobjekte kann man dabei nicht nur über Methoden bereits erzeugter Objekte erhalten, sondern auch über die Methode `forName()` der `Class`-Klasse. Dieser Methode übergibt man den vollständigen Namen der gesuchten Klasse als Zeichenkette. Daraufhin durchsucht der *Classloader* den vorhandenen *Classpath* nach einer Klasse mit dem übergebenen Namen und lädt diese in die virtuelle Maschine [KS09].

[2] Eclipse IDE: http://eclipse.org/

War die Suche erfolgreich, gibt die Methode ein `Class`-Objekt zurück, mittels dessen es anschließend möglich ist, Instanzen dieser Klasse dynamisch zu erzeugen.

Dieser Ansatz wird innerhalb des vorliegenden Systems beispielsweise derart verwendet, dass der Nutzer Namen von Klassen innerhalb der Konfigurationsdateien angegeben kann, die für die Erzeugung von Gebäudekomponenten, Grundrisstypen oder ganzen Gebäuden zur Verfügung stehen. Der derart angegebene Name wird anschließend verwendet, um Instanzen der jeweiligen Klasse während der Laufzeit zu erzeugen und gestattet dadurch die nachträgliche Erweiterung des Systems um weitere Klassen, die ohne Anpassungen an der Erzeugungslogik direkt eingebunden und verwendet werden können. Hierdurch bietet Java ein Sprachfeature, das in anderen Sprachen nicht vorhanden ist und für die Architektur des Systems viele Vorteile bringt. Konzeptionell stellt das System dadurch ein *Framework* für die Gebäudeerzeugung dar, das von Nutzern sukzessive erweitert werden kann.

Der vorherige Abschnitt befasste sich mit Programmiersprachen, Möglichkeiten der Klassifizierung und der Entwicklungsgeschichte von prozeduralen hin zu objektorientierten Sprachen. Da der vorliegende Prototyp in Java implementiert wurde, wurden zentrale Konzepte von objektorientierten Sprachen im Allgemeinen und anschließend besondere Sprachfeatures von Java erläutert. Das nachfolgende Kapitel befasst sich mit einem anderen Ansatz für die prozedurale Geometrieerzeugung.

Die Trennung zwischen der Darstellung von Programmiersprachen und den nachfolgend thematisierten Ersetzungssystemen soll dabei nicht suggerieren, dass Ansätze, die auf Ersetzungssystemen basieren, nicht ebenfalls durch Programmiersprachen umgesetzt sind. Auch bei solchen Systemen muss die Regelauswahl und –ausführung durch ein Programm erfolgen, das in einer beliebigen Programmiersprache implementiert ist. Allerdings bietet der Formalismus des Ersetzungssystems dem Nutzer einen anderen Ansatz für die Geometrieerzeugung. Während er dies bei Systemen wie dem vorliegenden entweder durch die Implementation neuer Routinen tut oder die bereits implementierten parametrisierten Algorithmen verwendet, um durch Konfigurationsdateien die Konstruktion zu steuern, definiert er in regelbasierten Systemen eine Menge von Ersetzungsregeln, die das System anschließend ausführt. Um den Vergleich der beiden Ansätze zu erleichtern, wird darum zunächst der theoretische Formalismus der *Lindenmayer*-Systeme und anschließend seine Anwendung für die Gebäudegenerierung exemplarisch vorgestellt.

4.2 **Ersetzungssysteme**

4.2.1 **Lindenmayer-Systeme**

Lindenmayer-Systeme wurden 1968 von dem ungarischen Biologen Aristid Lindenmayer entwickelt. Er suchte nach einem mathematischen Formalismus, mittels dessen er biologische Entwicklungen, konkret das Wachstum und die Teilung fadenförmiger Organismen, beschreiben konnte. Inspiriert durch die Arbeiten des norwegischen Mathematikers Axel Thue [Li1968b], der Anfang des 20. Jahrhunderts eine erste formale Definition für ein Ersetzungssystem lieferte, entwickelte Lindenmayer ein ähnliches Konzept, das heute den Namen seines Erfinders trägt und als Lindenmayer- oder kurz *L-System* bezeichnet wird. Da diese Systeme die theoretische Grundlage der Shape-Grammatiken bilden, die Basis einer Reihe von Systemen zur prozeduralen Gebäudeerstellung sind, sollen diese hier zunächst vorgestellt und anschließend ihre Anwendung für die automatisierte Gebäude- und Fassadendefinition diskutiert werden.

4.2.2 **Historischer Hintergrund**

Intensive Forschungen im Bereich der Ersetzungssysteme begannen Ende der 1950er Jahre aufgrund der Arbeiten von Noam Chomsky, der sich mit der Definition formaler Grammatiken befasste. Chomsky verwendete das Konzept der Ersetzungssysteme, um einen Formalismus zu entwickeln, mittels dessen er die syntaktischen Eigenschaften natürlicher Sprachen beschreiben konnte. Verwandte Arbeiten stammten von Backus und Naur, die wenige Jahre nach Chomsky die sogenannte Backus-Naur-Form entwickelten, die sie für die formale Definition der Programmiersprache *ALGOL-60* verwendeten. Wenig später wurde die Äquivalenz dieses Formalismus zu den kontextfreien Grammatiken festgestellt, die Teil der von Chomsky entwickelten Grammatiken sind [PL96]. Die daraus resultierenden Forschungsbemühungen befassten sich mit der Definition von Formalismen unterschiedlicher Mächtigkeit, die sich in der Struktur der erlaubten Ersetzungen unterscheiden und in der Lage sind, basierend auf einem Alphabet Elemente der beschriebenen formalen Sprache zu erzeugen. Solche Formalismen werden nicht nur zur Erzeugung von Worten aus einer solchen Sprache verwendet, sondern auch um zu prüfen, ob ein Eingabewort Teil dieser Sprache ist.

Der zentrale Unterschied zwischen den Ersetzungssystemen von Chomsky und dem von Lindenmayer entwickelten System besteht in der Art der Anwendung der Produktionen auf die nicht-terminalen Zeichen. Chomskys Grammatiken wenden die Ersetzungsregeln

sequentiell auf ein Wort an. Dabei wird zu jedem Zeitpunkt immer ein Zeichen oder eine Zeichenfolge ersetzt, anschließend kann die Ersetzung erneut auf das modifizierte Wort angewendet werden oder es greifen andere Regeln. Der Prozess terminiert, sobald das Wort nur noch aus terminalen Zeichne besteht. Lindenmayer entwickelte sein System dagegen als Formalismus zur Beschreibung biologischer Wachstumsprozesse, die sich dadurch auszeichnen, dass viele Prozesse gleichzeitig ablaufen. Aus diesem Grund werden die Ersetzungen in Lindenmayer-Systemen parallel ausgeführt [GS02]. Kommt eine Zeichenfolge innerhalb eines Wortes mehrfach vor, so würde eine Ersetzungsregel alle Vorkommen simultan in einem Schritt ersetzen. Bei L-Systemen terminiert der Ersetzungsprozess nicht wie bei Chomsky sobald nur noch terminale Zeichen vorkommen, sondern nachdem eine festgelegte Anzahl von Ableitungen durchgeführt wurde. Bei Organismen entspricht dies dem Erreichen eines vordefinierten „*terminal age*“ [Pr01]. Darin zeigt sich wiederum die ursprüngliche Zielsetzung, die Lindenmayer bei der Entwicklung des L-System-Formalismus verfolgte. Die L-System-Produktionen sollten in der Lage sein, eine Entwicklung der einzelnen Organismen- oder Pflanzenbestandteile über einen Zeitraum hinweg zu beschreiben [Pr01].

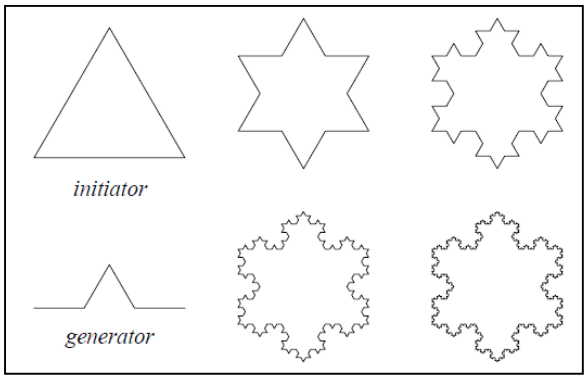

Abbildung 4: Kochsche Schneeflocke [PL96]

Ersetzungssysteme sind nicht nur auf Zeichenketten anwendbar, sondern können verwendet werden, um graphische Objekte zu erzeugen. Das klassische Beispiel für ein solches Ersetzungssystem ist die *Kochsche Schneeflocke*, die 1905 von dem schwedischen Mathematiker Helge von Koch vorgestellt wurde.

Abbildung 4 demonstriert die rekursive Konstruktion der Schneeflocke basierend auf einem Startzustand, der in der Abbildung als *Initiator* bezeichnet wird. Neben dem Initiator existiert ein *Generator*. In jeder Iteration wird jede gerade Linie durch den Generator ersetzt,

so dass Start- und Endpunkt der Ausgangslinie nach der Ersetzung mit dem Start- und Endpunkt des Generators identisch sind. Die rechte Seite der Grafik zeigt die Ergebnisse des rekursiven Ersetzungsprozesses nach jeder Iteration. Die Erzeugung der Kochschen Schneeflocke erfolgt durch ein L-System mit nur einer einzigen Ersetzungsregel.

Aufgrund der Konstruktion besitzt die Kochsche Schneeflocke einen hohen Grad an *Selbstähnlichkeit*. Selbstähnliche Strukturen sind Gegenstandsbereich der *fraktalen Geometrie*, die sich mit Gebilden befasst, die aus Teilen bestehen, die ihrerseits dem gesamten Gebilde ähneln [MB11]. Solche Gebilde bezeichnet man als *Fraktale*. Die Selbstähnlichkeit wird dabei unterschieden in *geometrische* und *statistische* Selbstähnlichkeit. Ein Objekt besitzt eine geometrische Selbstähnlichkeit, wenn seine Teile bei Vergrößerung nicht vom Ganzen zu unterscheiden sind. Bei einer statistischen Selbstähnlichkeit betrachtet man dagegen die statistischen Charakteristika, beispielsweise Erwartungswert und Varianz und vergleicht diese auf unterschiedlichen Auflösungsstufen. Ein oft zitiertes Beispiel für selbstähnliche Strukturen sind Küstenlinien. Diese zeichnen sich typischerweise durch eine Selbstähnlichkeit auf, bei der die Gezacktheit des Linienverlaufs zwischen unterschiedlichen Betrachtungsmaßstäben Ähnlichkeiten aufweist.

4.2.3 **Deterministisch kontextfreie Lindenmayer-Systeme**

Nach diesem Beispiel soll nun eine Definition von Lindenmayer-Systemen gegeben werden. Zunächst wird die einfachste Form dieser Textersetzungssysteme vorgestellt, die deterministisch kontextfreien oder auch D0L-Systeme. Ein solches System wird durch eine formale Grammatik G festgelegt, die durch ein Triple $G = (V, \omega, P)$ definiert ist. Dabei gilt [PL96]:

(1) Vist eine endliche und nichtleere Menge, die auch als *Alphabet* bezeichnet wird. V^* ist die Menge aller Wörter über V, V^+ die Menge aller nichtleeren Wörter über V

(2) $\omega \in V^+$ ist das nicht leere Startwort. Dieses wird auch *Axiom* genannt

(3) Pist eine endliche Menge von Ersetzungsregeln (*Produktionen*)

(4) Eine Produktion $(a, \chi) \in P$ wird geschrieben als $a \to \chi$. a wird als *Predecessor* bezeichnet, das Wort χ als *Successor*. Es gilt $a, \chi \in V^+$. Bei kontextfreien L-Systemen gilt darüber hinaus $|a| = 1$, wobei $|a|$ die Länge des Wortes, also die Anzahl an Zeichen beschreibt, aus denen das Wort besteht

(5) Für jeden Buchstaben $a \in V^*$existiert mindestens ein Wort $\chi \in V^*$, so dass gilt: $a \rightarrow \chi$

(6) Für ein deterministisches L-System gilt, dass für jedes $a \in V$ genau ein $\chi \in V^*$ mit $a \rightarrow \chi$ existiert

(7) Sei $\mu = a_0 a_1 \dots a_{m-1}$ ein Wort über das Alphabet V, Das Wort $v = \chi_0 \chi_1 \dots \chi_{m-1} \in V^*$ wurde genau dann aus dem Wort μ generiert (geschrieben als $\mu \Rightarrow v$), wenn für alle $i = 0, \dots, m-1$ eine Produktion $a_i \rightarrow \chi_i$ existiert.

(8) Ein Wort v wird von einem Textersetzungssystem G in einem Ableitungsprozess der Länge n erzeugt, sofern eine Folge von Wörtern $\mu_0, \mu_1, \dots, \mu_n$ existiert, wobei $\mu_0 = \omega, \mu_n = v$ und $\mu_0 \Rightarrow \mu_1 \Rightarrow \cdots \Rightarrow \mu_n$ gilt

Anhand eines einfachen Beispiels sei die Funktionsweise eines D0L-Systems verdeutlicht. Gegeben sei ein Alphabet V mit zwei Buchstaben $a, b \in V$. Weiterhin existieren zwei Produktionen $a \rightarrow ab$ und $b \rightarrow a$ und ein Startwort $\omega = b$.

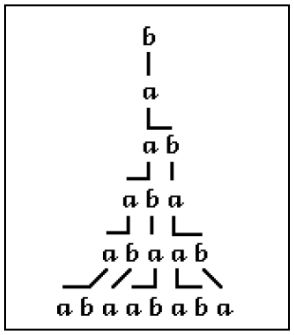

Abbildung 5: Einfaches D0L-System [PL96]

Abbildung 5 stellt exemplarisch fünf Durchläufe des Systems dar. Da es sich um ein Lindenmayer-System handelt, werden in jedem Iterationsschritt gleichzeitig alle Vorkommen durch die Produktionen ersetzt. Lindenmayer nutzte ein solches einfaches System zur Beschreibung der Entwicklung von Fadenorganismen. Hierfür benötigte er lediglich vier unterschiedliche Produktionen.

4.2.4 **Graphische Interpretation der Zeichenketten**

Wie gelangt man nun über ein formal beschriebenes D0L-System zu der oben dargestellten Kochschen Kurve? Zunächst handelt es sich bei L-Systemen um Textersetzungssysteme, die auf Zeichenketten agieren. Um aus den Worten, die durch ein Ersetzungssystem erzeugt

werden, eine geometrische Darstellung zu generieren, benötigt man einen Interpreter, der diese Überführung leisten kann. Als Paradigma für einen solchen grafischen Interpreter verwenden L-Systeme häufig die *Turtle*-Grafik. Hierbei handelt es sich um Konzept, das von Hal Abelssson und Andrea diSessa [AD86] entwickelt wurde. Dabei sollte die Turtle-Geometrie nicht nur zur bloßen Visualisierung mathematischer Theoreme und Beweise, sondern auch zur explorativen Forschung eingesetzt werden und als Werkzeug für neue Forschungsfelder dienen.

Grundlegend ist dabei die Vorstellung einer Schildkröte, die sich auf einem Blatt Papier, also in einer Ebene bewegt. Die Schildkröte kann eine festgelegte Menge von Schritten vorwärts gehen und sich um die eigene Achse drehen, wodurch sich ihre Ausrichtung ändert. Jede Bewegung der Schildkröte in eine bestimmte Richtung hinterlässt eine Spur. Dieses Konzept lässt sich bereits mit einem minimalen Befehlssatz realisieren, konzeptuell benötigt man nur einen Befehl zum Vorwärtsgehen und einen für die Drehung um einen bestimmten Winkel. Die von Abelsson und diSessa entwickelte Turtle-Geometrie ist dabei deutlich mächtiger, die Sprache enthält Schleifen, erlaubt prozedurale Abstraktion und das Zwischenspeichern von Zuständen, an die an späterer Stelle zurückgesprungen werden kann. Für die Interpretation der Ergebnisse eines einfachen D0L-Systems reicht dieser rudimentäre Befehlssatz allerdings bereits aus.

Die Idee besteht in der Definition eines Alphabets, dessen Elemente Aktionen der Schildkröte kodieren. Auf diesem Alphabet legt man anschließend eine Menge von Produktionen und ein Startwort fest. Nachdem das System terminiert oder die Verarbeitung nach einer vorab festgelegten Anzahl von Schritten abgebrochen wurde, verwendet man einen Turtle-Grafik-Interpreter, um das Ergebniswort grafisch darzustellen.

4.2.5 **Deterministisch kontextsensitive Lindenmayer-Systeme**

Kontextsensitive Lindenmayer-Systeme stellen eine Erweiterung der vorab vorgestellten kontextfreien Systeme dar. Bei kontextfreien Systemen werden Produktionen unabhängig vom Kontext eines Predecessors ausgeführt. Bei kontextsensitiven Systemen kann man dem Predecessor zusätzliche Informationen darüber geben, in welchem Kontext er sich befinden muss, damit die Produktion ausgeführt wird. Innerhalb eines Textersetzungssystems wird der Kontext eines Buchstaben a gebildet durch dessen direkte Nachbarn. Hierbei können sowohl einzelne Zeichen als auch Zeichenketten für die Definition eines Kontextes

eingesetzt werden. Kontextsensitive Systeme unterscheiden sich dabei darin, ob der linke, rechte oder beide Kontexte für die Produktionsauswahl relevant sind. Wird nur eine Seite berücksichtigt, nennt man solche deterministischen kontextsensitiven Systemen D1L-Systeme, können zwei Kontexte definiert werden, handelt es sich um D2L-Systeme. Kontextsensitive Systeme stellen eine Erweiterung kontextfreier Systeme dar und können demnach mindestens alle Wörter erzeugen, die durch ein kontextfreies System erzeugt werden. Die Definition eines solchen Systems ähnelt strukturell der eines D0L-Systems, allerdings muss die Definition der Produktionen erweitert werden, um die Kontexte abbilden zu können.

(1) In einem kontextsensitiven System hat eine Produktion $(a_l, a_r, a, \chi) \in P$ die Form $a_l < a > a_r \rightarrow \chi$. Dabei gilt $a \in V$sowie $\chi, a_l, a_r \in V^*$

Das folgende Standardbeispiel verdeutlicht die Funktionsweise eines einfachen kontextsensitiven Systems. Es sei gegeben das Alphabet $V = \{a, b\}$, das Axiom $\omega = baaa$ sowie die Produktionsmenge $P = \{b < a \rightarrow b, b \rightarrow a\}$. Die Anwendung der Produktionen auf das Startwort erzeugt die folgenden Iterationen.

1. Iterationsschritt: baaa
2. Iterationsschritt: abaa
3. Iterationsschritt: aaba
4. ….

Zwar heben kontextsensitive Systeme die vorab genannte Einschränkung auf, dass für jeden Buchstaben a immer genau eine Produktion zur Verfügung steht, sie sind aber trotzdem weiterhin deterministisch. Dies liegt daran, dass die Kontexte, die nun im Predecessorteil einer Produktion angegeben werden können, die Regelauswahl weiterhin eindeutig machen. Eine Regel wird nur dann gewählt, wenn sie die Kontextbedingung erfüllt. Dadurch führen auch kontextsensitive Systeme für das gleiche Axiom immer zum gleichen Resultat.

4.2.6 **Verzweigende Lindenmayer-Systeme**

Die bisher vorgestellten L-Systeme sind aufgrund ihrer Struktur nur in der Lage, Wörter zu generieren, die nach der Interpretation durch den Turtle-Grafik-Interpreter eine einzige zusammenhängende Linie beschreiben. Solche Strukturen bilden in der Natur und speziell der Pflanzenwelt die Ausnahme. Vielmehr zeichnen sich Pflanzen durch eine „baumartige“ Struktur aus. Von der Wurzel zu jedem Blatt gibt es genau einen Pfad, Zyklen existieren

nicht. Innerhalb eines solchen Pfades existieren Verzweigungen, beispielsweise wenn kleinere Äste vom Hauptstamm abgehen und dann weiter verzweigen, bis der Wachstumsprozess schließlich in den Blättern terminiert. Eine solche Struktur mit Verzweigungen ist durch die bisherigen Systeme nicht umsetzbar, da sie erfordert, Zwischenzustände temporär zu speichern und später wiederherzustellen. Im Pflanzenbeispiel wäre jede Verzweigung ein solcher Zwischenzustand. Die Realisierung eines Lindenmayer-Systems, das Zwischenzustände unterstützt, benötigt die Einführung eines Zwischenspeichers. Hierfür verwenden verzweigende L-Systeme einen *Kellerspeicher* (engl. *Stack*). Ein Stack ist eine spezielle Datenstruktur, die konzeptuell nur zwei Operationen benötigt. Die *Push*-Operation fügt dem Stack ein neues Element hinzu und legt dieses oben auf den Stapel. Die *Pop*-Operation holt das letzte hinzugefügte Element vom Stapel und entfernt es von diesem. Zu jedem Zeitpunkt ist immer nur das oberste Element auf dem Stapel sicht- und zugreifbar [GS02]. Damit ein solcher Stapelspeicher innerhalb eines L-Systems eingesetzt werden kann, muss das vorhandene Alphabet erweitert werden, um die Push- und Pop-Operation umzusetzen. Hierfür verwendet man meist eckige Klammern (engl. *brackets*), weshalb verzweigende L-Systeme im Englischen *Bracketed 0L*-Systems genannt werden. Der Turtle-Interpreter verarbeitet diese neuen Zeichen wie folgt [PL96]:

[Der aktuelle Zustand der Schildkröte wird auf einem Stack abgelegt. Dabei werden mindestens Position und Ausrichtung der Schildkröte zum jeweiligen Zeitpunkt gespeichert, weitere Informationen wie Farbe o.ä. sind denkbar.

] Der letzte auf dem Stack abgelegte Zustand wird geladen, vom Stack gelöscht und zum aktuellen Zustand gemacht.

Das folgende Beispiel soll die Verwendung eines verzweigenden L-Systems verdeutlichen. Es stammt aus dem vorab bereits mehrfach zitierten Werk von Lindenmayer und Prusinkiewicz [PL96]. Gegeben sei das Alphabet $V = \{X, F, -, +, [,]\}$, die Produktionsmenge $P = \{X \to F[+X][-X]FX, F \to FF\}$ sowie das Axiom X. Der Turtle-Interpreter führt zusätzlich zu den vorab bereits beschriebenen Stackmethoden folgende Operationen aus, sobald er auf einen der Buchstaben trifft:

F Bewege die Schildkröte um eine vordefinierte Anzahl von Schritten entsprechend ihrer aktuellen Ausrichtung. Dabei wird eine Linie

zwischen der Ausgangsposition (x, y) und der Endposition (x', y') gezeichnet. Ihr Zustand verändert sich von (x, y, α) zu (x', y', α)

\+ Drehe die Schildkröte um den Winkel δ nach links. Ihr Zustand ändert sich von (x, y, α) nach $(x, y, \alpha + \delta)$

\- Drehe die Schildkröte um den Winkel δ nach rechts. Ihr Zustand ändert sich von (x, y, α) nach $(x, y, \alpha - \delta)$

Das beschriebene System erzeugt durch die Turtle-Interpretation die in Abbildung 6 dargestellte Grafik für eine Schrittanzahl von $n = 7$ und einem Winkel $\delta = 25{,}7°$. Dabei wird das Zeichen X des Alphabets vom Interpreter „überlesen“, das heißt er führt keine Operation aus und geht zum nächsten Zeichen über. Die Produktion für X erzeugt durch die Verwendung der Operationen des Kellerspeichers eine verzweigte Struktur, da durch die Ersetzung jeweils neue Verzweigungen erzeugt entstehen.

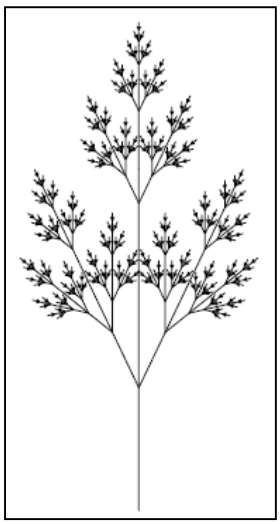

Abbildung 6: Ergebnis eines verzweigenden L-Systems [PL96]

4.2.7 **Stochastische L-Systeme**

Die bisherigen L-Systeme sind in ihrer Struktur deterministisch. Bei gleichem Axiom führt die gleiche Produktionsmenge immer zum selben Endwort. Dies liegt daran, dass zu jedem Zeitpunkt immer nur eine einzige Regel ausgewählt werden kann. Verwendet man solche Systeme im Kontext der Pflanzengenerierung zur Erzeugung einer Vielzahl von Pflanzen auf einer Wiese, so wird das Ergebnis sehr unrealistisch wirken, da keinerlei Variationen

innerhalb des Systems vorhanden sind. Stochastische L-Systeme können hier Abhilfe schaffen. Die Kernidee ist es, den Determinismus in der Ersetzungsabfolge zu stören, indem nun für einen Predecessor mehr als ein Successor zur Auswahl steht. Dabei kann jeder Produktion eine Wahrscheinlichkeit zugewiesen werden, mit der diese gewählt und ausgeführt wird. Hierfür muss die formale Definition der L-Systeme erweitert werden.

Ein stochastisches L-System ist definiert durch ein Quadrupel $G_\pi = (V, \omega, P, \pi)$. Dabei sind V, P und ω analog zu den vorab gegebenen Festlegungen definiert. Neu ist dagegen die *Abbildungsfunktion* $\pi: P \rightarrow (0,1]$. Diese Wahrscheinlichkeitsverteilung bildet die Menge der Produktionen P auf die Menge der Produktionswahrscheinlichkeiten ab. Dabei gilt, dass für alle Produktionen mit gleichem Predecessor die Summe der Wahrscheinlichkeiten der Successoren gleich 1 ist. Verwendet man kontextsensitive stochastische L-Systeme, so wird der Kontext als differenzierender Teil des Predecessors betrachtet. Bei solchen Systemen müssen ebenfalls die Produktionswahrscheinlichkeiten aller Produktionen mit gleichem Successor $a_l < a > a_r$ in der Summation 1 ergeben. Um die Wahrscheinlichkeiten in ein solches System zu integrieren, muss die Notation der Produktionen erweitert werden. Nachfolgend wird dies für kontextsensitive stochastische Systeme angeben, die gleiche Notation wird aber auch für kontextfreie Systeme eingesetzt. Eine Produktion hat in einem solchen System die Form $a_l < a > a_r \overset{\pi}{\rightarrow} \chi$ mit $\pi \in (0,1]$. Liegen für einen Successor mehrere Produktionen vor, so werden diese zufallsbasiert mit der Wahrscheinlichkeit π ausgewählt. Aufgrund der Parallelität des Ersetzungsprozesses in Lindenmayer-Systemen wird bei einem wiederholten Vorkommen eines Successors mit unterschiedlichen Produktionen für jedes Vorkommen erneut entschieden, welche Ersetzung ausgeführt wird. Somit können sich innerhalb einer Iteration des Systems die ausgewählten Produktionen selbst bei gleichen Successoren unterscheiden.

Das nachfolgende Beispiel stammt wiederum aus der Arbeit von Prusinkiewicz und Lindenmayer [PL96]. Gegeben sei ein stochastisches L-System mit dem Axiom $\omega: F$ und den Produktionen:

$$p_1: F \overset{.33}{\rightarrow} F[+F]F[-F]F$$

$$p_2: F \overset{.33}{\rightarrow} F[+F]F$$

$$p_3: F \overset{.34}{\rightarrow} F[-F]F$$

Abbildung 7 zeigt Turtle-Grafik-Interpretationen der Worte, die durch das beschriebene System erzeugt wurden. Da es sich um ein stochastisches System handelt, ist das Verhalten nichtdeterministisch, für die gleiche Eingabe entstehen unterschiedliche Ergebnisse. Die stochastische Variation führt dabei zu realistischeren Ergebnissen, die konkreten Ausprägungen des generierten Pflanzentyps zeigen teils starke Abweichungen voneinander, gehören aber immer noch zum gleichen Typ.

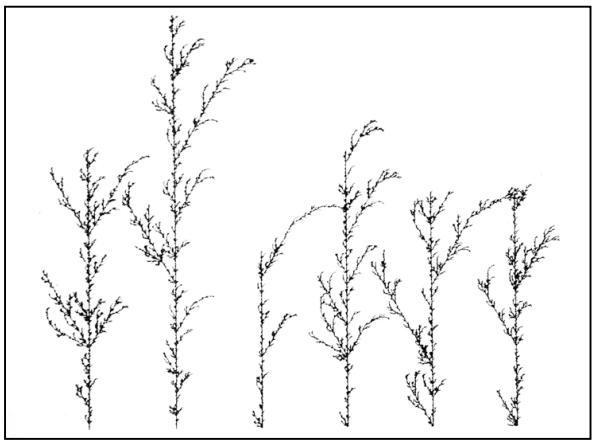

Abbildung 7: Unterschiedliche Ergebnisse des gleichen stochastischen Systems [PL96]

4.2.8 **Parametrische L-Systeme**

Die bisherigen L-Systeme bieten bereits eine Vielzahl von Möglichkeiten zur Generierung komplexer Strukturen, trotzdem unterliegen sie einer Reihe von Einschränkungen, die ihre Mächtigkeit reduzieren. Vergleichsweise offensichtlich ist die Problematik, Zahlenwerte durch das System auszuwerten und zu modifizieren. Aufgrund des Ersetzungsprozesses ist es nur möglich, ganzzahlige Vielfache eines Initialwertes zu erzeugen. In den vorab vorgestellten Beispielen betrifft dies beispielsweise die Länge der erzeugten Äste oder auch den Winkel, in dem neue Verzweigungen von einem Ast entstehen. Eine Ersetzungsregel erlaubt hierbei nur die Erzeugung von Vielfachen der Startwerte, andere Modifikationen sind aufgrund des Fehlens arithmetischer Operationen nicht möglich.

Aus diesem Grund schlug Lindenmayer 1974 [Li74] eine Erweiterung der bestehenden L-Systeme vor, die diese Nachteile beheben sollte. Hierfür können in parametrischen L-Systemen allen Zeichen zusätzliche Parameter angehangen werden. Dabei gehören die Zeichen weiterhin zum definierten Alphabet *V*, die Parameter zur Menge reeller Zahlen $\mathbb{R}$. Ein Modul mit dem Buchstaben $A \in V$ und den Parametern $a_1, a_2, \dots, a_n \in \mathbb{R}$ wird geschrieben als $A(a_1, a_2, \dots, a_n)$. Weiterhin gilt [PL96]:

(1) Jedes Modul gehört zur Menge $M = V \times \mathbb{R}^*$. Dabei ist $\mathbb{R}^*$ definiert als Menge aller endlichen Folgen von Parametern

(2) Die Menge aller Zeichenketten von Modulen ist definiert als $M^* = (V \times \mathbb{R}^*)^*$, die Menge aller nicht-leeren Zeichenketten als $M^+ = (V \times \mathbb{R}^*)^+$

(3) Es wird unterschieden zwischen formalen Parametern, die zur Spezifikation eines Moduls dienen, und reellen Parametern. Reelle Parameter sind formale Parameter, denen eine reelle Zahl zugewiesen wurde.

(4) $\sum$ ist eine Menge formaler Parameter, $C(\sum)$ beschreibt eine logische Operation auf den formalen Parametern aus $\sum$, $E(\sum)$ eine arithmetische Operation auf diesen.

(5) In parametrischen L-Systemen stehen arithmetische Operatoren $+, -, *, /;$, relationale Operatoren $<, >, =;$, logische Operatoren $!, \&, |$, der Potenzoperator ^ sowie Klammern $(,)$; zur Verfügung

Zusammenfassend ist ein parametrisches L-System definiert durch ein Quadrupel $G = (V, \sum, \omega, P)$. Dabei ist $\omega \in (V \times \mathbb{R}^*)^+$ das Axiom des Systems und P eine endliche Menge an Produktionen. Um die formalen Parameter und die neu hinzugefügten Operatoren in die Notation der Produktionen zu integrieren, muss diese erweitert werden. Eine Produktion besteht in einem parametrischen L-System aus drei Komponenten, dem Predecessor, dem Successor und einem konditionalen Bedingungsteil. Weiterhin enthalten die Produktionen nun die vorab definierten Module und nicht mehr nur die Zeichen des Alphabets. Eine Produktion wird dabei genau dann ausgewählt, wenn

1. das Zeichen des Moduls mit dem Zeichen des Predecessors übereinstimmt
2. die Anzahl der reellen Parameter im Modul der Anzahl der formalen Parameter im Predecessor entspricht
3. die Bedingung, die in der Produktion definiert ist, erfüllt ist

Die Funktionsweise solcher Systeme sei an einem einfachen Beispiel demonstriert. Es handelt sich um ein System, das ausgehend von einem Axiom den Parameter des Moduls verdoppelt, bis er einen Grenzwert überschreitet. Gegeben sei das Alphabet $V = \{A(x)\}$, das Axiom $\omega = A(2)$ sowie die Produktionsmenge $P = \{A(x): x > 0 \,\&\, x < 512 \rightarrow A(x * 2), A(x): x \geq 512 \rightarrow A(0)\}$. Die Produktionen enthalten nun neben dem Successor, der als Modul definiert ist, einen Bedingungsteil, der durch „:“ vom Successor-Modul getrennt wird. Im Beispiel ist die Bedingung für die erste Produktion, dass der Wert des Parameters $x > 0$ ist. Nur wenn diese Bedingung erfüllt ist, wird die Produktion ausgeführt.

Die Anwendung der Produktionen auf das Axiom führt zunächst dazu, dass der formale Parameter x in jeder Iteration verdoppelt wird. Erreicht er einen Wert ≥ 512, so greift die zweite Produktion und setzt den Parameter zurück auf 0. Damit terminiert die Verarbeitung des Systems, da nunmehr keine Produktionsbedingung mehr erfüllt wird.

Um parametrische L-Systeme mittels Turtle-Interpreter zeichnen zu können, muss der Befehlssatz erweitert werden. Auch der Interpreter arbeitet dann auf Modulen und nicht mehr länger auf einzelnen Zeichen, die zu Operationen führen. Die Parameter dieser Module werden dann wie in einer Programmiersprache als Übergabeparameter interpretiert, die bestimmte Aspekte des Verhaltens steuern. So kann beispielsweise die zu überbrückende Distanz bei einer Vorwärtsbewegung übergeben werden oder auch der Winkel, um den rotiert werden soll. Durch die Erweiterung um arithmetische Operationen sind in solchen Systemen nun auch Längen und Winkel möglich, die keine ganzzahligen Vielfachen der initial gesetzten Parameter sind.

4.2.9 **Umgebungssensitive / Offene L-Systeme**

Innerhalb dieses Kapitels wurde sukzessive die Weiterentwicklung der L-Systeme beginnend von einem deterministischen kontextfreien D0L-System bis hin zu den zuletzt vorgestellten parametrischen L-Systemen erläutert. Parametrische L-Systeme können Elemente stochastischer und kontextsensitiver L-Systeme enthalten und sind dadurch sehr mächtig in Bezug auf die Komplexität der Strukturen, die sie erzeugen können. Solche komplexen Systeme können durch Parameter globale Eigenschaften eines Systems berücksichtigen. Ein Beispiel für eine globale Eigenschaft ist die Bodenqualität, die den Wachstumsprozess einer Wurzel und somit der gesamten Pflanze beeinflussen kann. Durch die Kontextsensitivität kann man die Ersetzungsprozesse an Vorbedingungen innerhalb des bisherigen Wachstumsprozesses koppeln, um beispielsweise Verzweigungen erst ab einer bestimmten Höhe entstehen zu lassen. Solche Bedingungen sind immer an die bisherige Entwicklung des Ersetzungsprozesses gebunden und hängen nur von diesem selbst, aber nicht von der direkten Umwelt ab. Der verbleibende Schwachpunkt solcher Systeme ist die nicht stattfindende Interaktion mit der direkten Umgebung. So ist es nicht möglich festzustellen, ob der Wuchs einer Pflanze durch Hindernisse in der Umgebung eingeschränkt ist, da hierfür eine Interaktion stattfinden müsste. Das System müsste der Umgebung die Position eines Astes und dessen Wachstumsrichtung übermitteln und würde daraufhin

mitgeteilt bekommen, ob ein weiteres Wachstum mit diesen Parametern überhaupt stattfinden kann.

Genau darin besteht nun die Erweiterung, die Měch und Prusinkiewicz in ihrer Arbeit [PJM94] vorgestellt haben. Dazu erweitern sie parametrische L-Systeme um sogenannte *Anfragemodule* (engl. *query modules*). Jedes Modul enthält formale Parameter. Diese unterscheiden sich allerdings darin von denen in parametrischen Systemen, dass sie während der eigentlichen Ersetzung noch nicht mit konkreten Werten belegt sind. In einem parametrischen System ist dagegen während des Ersetzungsprozesses jedem formalen Parameter ein konkreter Wert zugewiesen, der beispielsweise durch eine arithmetische Operation bestimmt wurde. In umgebungssensitiven Systemen findet diese Zuweisung erst nach Abschluss des Ersetzungsprozesses innerhalb einer Iteration statt. Nachdem alle möglichen Produktionen ausgeführt wurden, wird das entstehende Wort interpretiert. Während dieses Interpretationsschrittes stellen die Anfragemodule Anfragen an die Umwelt des Systems und übermitteln dieser den aktuellen Status der Schildkröte. Die Antworten des Systems werden dann den undefinierten Parametern zugewiesen. Diese nun konkreten Werte können im nächsten Iterationsschritt verwendet werden. Erst nach der Terminierung des Ersetzungsprozesses wird das entstandene Wort grafisch dargestellt.

Umgebungssensitive L-Systeme ermöglichen eine unidirektionale Kommunikation des Systems mit seiner Umgebung. Dadurch kann diese direkten Einfluss auf das System nehmen, eine umgekehrte Einflussnahme ist dagegen nicht möglich. Offene L-Systeme, die in einer späteren Arbeit von Měch und Prusinkiewicz [PJM94] eingeführt wurden, erweitern das Konzept der Anfragemodule derart, dass sowohl die lokale Umwelt das System beeinflussen kann, als auch umgekehrt. Hierfür erhalten die Module die Fähigkeit, nicht nur Informationen der Umwelt durch Anfragen ermitteln zu können, sondern ebenso der Umwelt Informationen zu übermitteln. Mittels eines solchen Systems mit bidirektionaler Kommunikation sind die Autoren in der Lage, verschiedene komplexe Wachstumsprozesse zu simulieren. Dazu gehört beispielsweise das Wurzelwachstum in Abhängigkeit des Wassers, das in Bodenbereichen zur Verfügung steht. Der bidirektionale Einfluss besteht dabei darin, dass Wurzeln immer in Richtung der größten Wasserkonzentration streben, dem Boden aber gleichzeitig Wasser entziehen und dadurch direkten Einfluss auf diese Konzentration nehmen. Ein anderes Beispiel befasst sich mit dem Wachstum von Bäumen in Abhängigkeit der Lichtmenge, die diese erhalten. Hierfür simulierten die Autoren das Wachstum zweier Bäume in direkter Nachbarschaft. Jeder der beiden Bäume strebt danach,

möglichst viel Licht zu erhalten und wächst darum in eine Richtung, in der eine möglichst hohe Lichtkonzentration zur Verfügung steht. Gleichzeitig beeinflusst aber das Wachstum jedes Baumes die Lichtmenge, die der jeweils andere Baum erhält und nimmt darum umgekehrt wiederum Einfluss auf seinen Nachbarn.

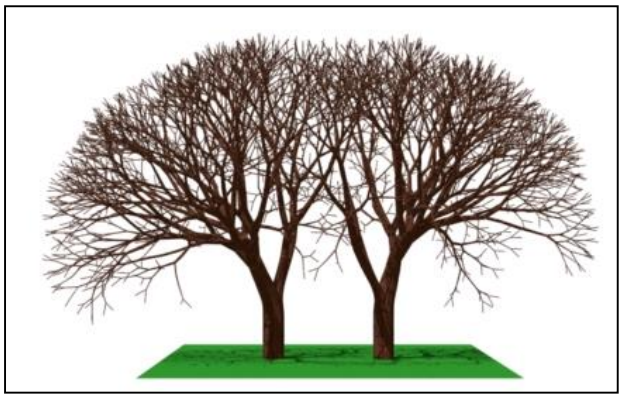

Abbildung 8: Beispiel eines offenen L-Systems [PJM94]

Abbildung 8 zeigt das Ergebnis der Berechnungen eines L-Systems, das die erhaltene Lichtmenge als zentralen Faktor für das Wachstum der benachbarten Bäume berücksichtigt. Man erkennt gut, wie die wechselseitige Beschattung der Bäume dazu führt, dass an den einander zugewandten Seiten deutlich weniger Äste erzeugt werden.

4.3 **Shape Grammars**

L-Systeme bieten einen mächtigen Formalismus zur Beschreibung von Wachstumsprozessen. Die vorgestellten Arbeiten erzielten sehr gute Resultate im Bereich der prozeduralen Erzeugung von Pflanzen durch eine Kombination von L-Systemen mit Turtle-Grafik-Interpretationen. Müller verwendete eine erweiterte Form der umgebungssensitiven L-Systeme zur automatischen Generierung von Straßennetzen, die die Basis der von ihm entwickelten *CityEngine* darstellt [Mü01]. L-Systeme eignen sich demnach gut zur Modellierung von Wachstumsprozessen, seien es nun Pflanzen oder Straßennetze, bei denen durch Ersetzungsprozesse neue Verzweigungen entstehen. Solche Prozesse sind dabei typischerweise rekursiv.

Für die Modellierung von Gebäuden sind solche Systeme dagegen nur bedingt einsetzbar, da sich beispielsweise Gebäudefassaden nicht durch einen Wachstumsprozess beschreiben lassen. Die Modellierung von Gebäudefassaden lässt sich im Kontext von

Ersetzungssystemen besser als Unterteilungsprozess formalisieren, bei dem eine initiale Grundform fortlaufend unterteilt und durch weitere Formen ersetzt wird. Eine solche Methodik ist Basis der von Müller entwickelten *CityEngine*, die einen rekursiven Unterteilungsprozess verwendet. Dieser Ansatz basiert auf einer Arbeit von Stiny und Gips [GJ71], die einen Formalismus vorstellten, mittels dessen sie Zeichnungen und ihre Erzeugung formal beschreiben konnten. Inspiriert wurde ihre Arbeit dabei wiederum durch die Forschungen von Chomsky im Bereich formaler Sprachen. Während bei Chomsky allerdings auf eindimensionalen Strings gearbeitet wird, basieren Shape Grammars auf einem Formen- statt auf einem Zeichenalphabet. Für Stiny und Gips besteht die Spezifikation einer Zeichnung aus

1. der Definition einer Sprache aus zweidimensionalen Formen
2. der Auswahl einer Form aus dieser Sprache, die gezeichnet werden soll
3. der Definition eines Schemas, das festlegt, welche Bereiche innerhalb der Zeichnung wie zu zeichnen sind (beispielsweise das Füllen bestimmter Bereiche mit einer bestimmten Farbe)
4. der Festlegung von Position und Skalierung einer Form die auf einer Zeichenoberfläche mit festgelegter Größe gezeichnet werden soll

Die formale Definition ähnelt der Definition einer Grammatik bei Chomsky und somit auch der Festlegung von L-Systemen. Ein Shape Grammar (SG) ist ein Quadrupel $SG = (V_T, V_M, R, I)$ wobei gilt:

(1) V_T ist eine endliche Menge von Formen (vergleichbar dem Alphabet bei L-Systemen)

(2) ${V_T}^*$ definiert die Menge von Formen, die durch eine endliche Menge von Formen aus V_T erzeugt werden. Dabei dürfen Elemente aus V_T mehrfach vorkommen und eine beliebige Skalierung und Orientierung besitzen

(3) V_M ist eine endliche Menge von Formen, so dass gilt: ${V_T}^* \cap V_M = \emptyset$. Anschaulich ist V_M also eine Menge von Formen, die nicht durch eine Kombination von Elementen aus V_T darstellbar sind

(4) R ist eine Menge von Regeln, vergleichbar den Produktionen in L-Systemen. Sie werden als $u \rightarrow v$ geschrieben

(5) I bildet die Startform des Ersetzungsprozesses, in L-Systemen als Axiom bezeichnet. I ist dabei eine Form, die sich aus Elementen aus ${V_T}^*$ und V_M zusammensetzt

Im Gegensatz zu L-Systemen erfolgt der Ersetzungsprozess in dem von Stiny und Gips vorgestellten Shape-Grammar sequentiell und nicht parallel. In jedem Iterationsschritt wird genau eine Form basierend auf einer Regel ersetzt, die nachfolgenden Iterationen arbeiten dann mit der modifizierten Form. Die Auswahl einer anzuwendenden Regel aus R ist ein mehrschrittiger Prozess. Zunächst muss innerhalb der Zeichnung nach einer Form gesucht werden, die einer gegebenen linken Seite (bei L-Systemen Predecessor genannt) geometrisch ähnelt. Wurde eine solche Form gefunden, müssen die erforderlichen Transformationen (Translation, Skalierung, Rotation oder Spiegelung) bestimmt werden, mittels derer die gefundene Form in die Form auf der linken Seite der Regel transformiert werden kann. Dieselbe Abfolge von Transformationen wird anschließend auch auf die rechte Seite der Regel angewendet. Wurde beispielsweise die zu ersetzende Form um 50% skaliert, um zu der linken Regelseite zu passen, so wird diese Skalierung auch auf die rechte Seite angewendet. Abschließend wird dann die ausgewählte Form in der Ausgangszeichnung durch die transformierte rechte Regelseite ersetzt. Die Anwendung dieser Regeln erfolgt dabei so lange, bis es keine anwendbare Regel mehr gibt und der Ersetzungsprozess terminiert. Dabei kann die von einem Shape Grammar SG erzeugte Sprache $L(SG)$ eine endliche oder unendliche Menge endlicher Formen darstellen [GJ71].

Da ein Shape Grammar eine Sprache mit potentiell unendlich vielen Formen definieren kann, führen Stiny und Gips einen Mechanismus ein, mittels dessen sie in der Lage sind, bestimmte Formen aus der jeweiligen Sprache für das Zeichnen auszuwählen. Diesen Mechanismus bezeichnen die Autoren als *Selection Rule*, die Basis der Selection Rule bildet das *Level* einer Form. Während des Fortschreitens des Ersetzungsprozesses werden den einzelnen Bestandteilen der Zeichnung solche Levels zugewiesen. Wird durch eine Regel eine Form ersetzt, so werden die Levels der beteiligten Strukturen inkrementiert. Dabei können unterschiedliche Teile der gleichen Form während des Prozesses unterschiedliche Level zugewiesen bekommen, abhängig von der Abfolge des Ersetzungsprozesses.

Eine Selection Rule besteht aus einem Tupel (m, n), wobei $m, n \in \mathbb{N}_0$. Dabei bezeichnet m das Minimumlevel, n das Maximumlevel, das innerhalb einer erzeugten Form vorkommen darf, damit diese Mitglied der Klasse ist, die durch das Shape Grammar definiert ist.

Neben dem regelbasierten Ersetzungsprozess ist es möglich, festzulegen, wie bestimmte Bereiche einer Form eingefärbt werden. Dabei werden Bereiche mit gleichem Level identisch eingefärbt. Kommt es innerhalb der Zeichnung zur Überlagerung von Formen mit unterschiedlichen Levels, so werden *Venn*-Diagramme eingesetzt, um zu entscheiden, wie

die überlappenden Bereiche eingefärbt werden. Die Festlegung, welche Bereiche wie gefärbt werden, erfolgt durch die Definition von Regeln, die die Autoren als *Painting Rules* bezeichnen [GJ71].

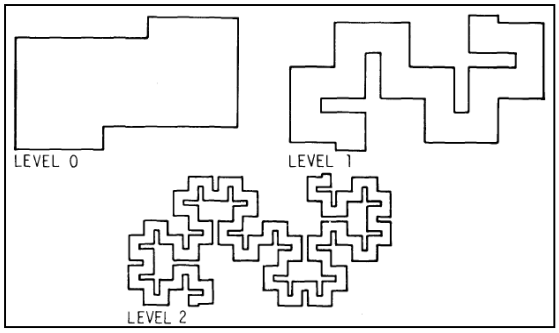

Abbildung 9: Formen mit unterschiedlichen Levels [GJ71]

Nachdem in den vorherigen Abschnitten verschiedene Basistechnologien und theoretische Formalismen vorgestellt wurden, befasst sich das nachfolgende Kapitel mit unterschiedlichen Ansätzen, die zur Erzeugung von Gebäude- und Stadtmodellen eingesetzt werden und stellt exemplarisch verschiedene Arbeiten vor, die sich mit dieser Thematik auseinandersetzen.

5 Verwandte Arbeiten - Technologien zur Erstellung von 3D-Gebäudemodellen

Innerhalb dieses Abschnittes werden drei unterschiedliche Klassen von Technologien erörtert, die zur Erzeugung von 3D-Gebäudemodellen verwendet werden können. Diese werden zunächst grob anhand des Erstellungsaufwands für neue Gebäude und der vorhandenen Freiheitsgrade bei der Modellierung unterschieden.

Die erste große Technologiegruppe stellen Modellierungswerkzeuge aus dem Bereich des *Computer Aided Designs* (CAD) bzw. *Computer Aided Architectural Designs* (CAAD) dar. Hierbei handelt es sich um Softwarepakete, die dem Nutzer Werkzeuge zur Verfügung stellen, um 3D-Modelle „per Hand“ zu erzeugen. Ein speziell in der Architektur weit verbreitetes System ist die Software *AutoCAD* der Firma *Autodesk*[3]. Bei Softwaresystemen dieser Art ist der Aufwand für den Nutzer typischerweise sehr groß, da er alle Details manuell modellieren muss. Dafür ist die Komplexität der entstehenden Modelle theoretisch unbegrenzt, hängt allerdings von der Expertise des Nutzers ab.

Weniger Nutzerintervention erfordert die zweite Gruppe, die als *fotogrammetrische Modellierungstechnologien* bezeichnet werden. Kernidee dieser Ansätze ist die Extraktion geometrischer Details von Objekten aus Fotografien. Die Vorteile solcher Ansätze liegen auf der Hand. Sofern solche Systeme in der Lage sind, beliebig komplexe Gebäudemodelle aus Fotos abzuleiten, besitzt man ein Werkzeug, das mit minimaler Nutzerintervention in der Lage ist, Modelle mit großer Komplexität zu generieren. Eine offensichtliche Einschränkung besteht allerdings darin, dass die Systeme auf Fotografien angewiesen sind, um eine automatisierte Rekonstruktion durchzuführen. Dies beschränkt den Einsatzbereich naturgemäß auf Gebäude, von denen Fotografien zur Verfügung stehen. Historische Bauwerke aus frühen Epochen können somit nicht abgebildet werden.

Die letzte Technologiegruppe zur Gebäudegenerierung sind prozedurale Technologien. Diese lassen sich wiederum untergliedern in zwei große Teilbereiche. *Regelbasierte* Verfahren nutzen meist Ersetzungssysteme, bei denen der Nutzer ein Regelsystem spezifiziert, durch das die Geometrie des Gebäudes erzeugt wird. *Algorithmische* Verfahren kodieren die Geometriestruktur in Form von Methoden, die in Programmiersprachen implementiert werden. Prozedurale Technologien stellen eine Abstraktion der Modellerzeugung dar, da sie den Prozess der Geometrieerzeugung durch Regeln oder

[3] Autodesk: AutoCAD. http://www.autodesk.de/adsk/servlet/pc/index?siteID=403786&id=14659756

Algorithmen beschreiben. Solche Abstraktionen sind im Idealfall wiederverwend- und mit bereits vorhandenen Beschreibungen kombinierbar, um möglichst vielfältige Gebäude zu erzeugen. Durch den Einsatz stochastischer Elemente können prozedurale Systeme aus wenigen Beschreibungen viele unterschiedliche Gebäude des gleichen Typs generieren. Im Gegensatz zu CAD-basierten und fotogrammetrischen Technologien sind allerdings häufig die Komplexität und der Detailreichtum der erstellten Gebäude geringer.

In den nachfolgenden Abschnitten werden die unterschiedlichen Modellierungstechnologien erläutert und ihre Funktionsweise anhand existierender Softwaresysteme beschrieben.

5.1 **3D-Modellierungswerkzeuge / Computer-Aided Design (CAD)**

3D-Modellierungswerkzeuge stellen die am weitesten verbreitete Technologie zur Erzeugung von 3D-Modellen dar. Eingesetzt werden solche Systeme in der Konstruktion und Planung beliebiger Objekte, vom komplexen Bauteil bis hin zu großen Gebäudekomplexen. Außerdem finden sie Anwendung in der Unterhaltungsindustrie, beispielsweise zur Erzeugung dreidimensionaler Welten für Computerspiele oder Kinofilme. Von allen vorgestellten Technologien zur Gebäudeerzeugung bieten solche Softwarepakete dem Nutzer die größte Freiheit, da er jeden Aspekt der Modelle festlegen kann. Daraus ergibt sich allerdings auch ein großer Nachteil, da die Erzeugung komplexer Modelle auch für erfahrene Nutzer sehr aufwendig ist und für Einsteiger zunächst eine lange Einarbeitungszeit erfordert. Dabei unterscheiden sich die Systeme in ihrer Komplexität und den Werkzeugen, die zur Verfügung stellen. Hierbei spielt auch der Anwendungsbereich eine wichtige Rolle, so sollten Systeme, die in der Architektur zur Konzeption und Planung von Gebäuden verwendet werden, naturgemäß andere Möglichkeiten bieten, als Systeme, in denen Gebäude für eine künstliche Computerspielewelt entwickelt werden. Diese Abgrenzung ist auch für das hier vorliegende System von Bedeutung, da es nicht den Anspruch erhebt, „baufähige“ Gebäudemodelle zu berechnen, sondern vielmehr Wert auf den visuellen Eindruck dieser Modelle legt. Die Verwendung der Grundrisse und Strukturen zur Realisierung in der „wirklichen Welt“ kann und soll durch das System nicht geleistet werden.

Kommerzielle Systeme wie die Software AutoCAD der Firma Autodesk bieten dem Nutzer Funktionalitäten, die auf die jeweiligen Anwendungsbereiche ausgerichtet sind. AutoCAD selber wird in verschiedenen Bereichen für den Entwurf eingesetzt, Autodesk vertreibt die

Software in unterschiedlichen Paketen, die bereichsspezifische Funktionen besitzen. So adressiert AutoCAD Architecture[4] die Bedürfnisse von Architekten und erlaubt die automatische Erstellung von Schnitten, Grundrissen und Ansichten basierend auf den vom Nutzer erstellten 2D- und 3D-Zeichnungen. Darüber hinaus bietet das Architecture-Paket eine umfangreiche Bibliothek vorgefertigter 3D-Komponenten, die im Bereich der Gebäudeentwicklung häufig benötigt werden, dazu gehören 3D-Modelle für Wände, Türen oder Dächer. Dadurch wird die Entwicklung von Gebäudemodellen vereinfacht und beschleunigt. Ähnlich angepasste Softwarepakete für andere Kontexte wie den Maschinenbau[5] werden analog zum Architecture-Paket durch Autodesk vertrieben. Gemein ist diesen Systemen, dass sie den Nutzer dabei unterstützen, möglichst exakte Modelle zu erstellen, da es möglich sein muss, direkte Baupläne aus den modellierten Objekten abzuleiten. Hierin wird deutlich, dass die AutoCAD-Produktreihe für erfahrene Nutzer im kommerziellen Bereich ausgelegt ist, was sich sowohl in der Komplexität als auch in den Lizenzkosten der Produkte manifestiert. Für fachfremde Anwender sind solche Systeme nicht zu bedienen, für diese Nutzergruppe sind sie allerdings auch nicht konzipiert.

Eine kostengünstige Alternative mit einem vergleichsweise einfachen Einstieg bietet beispielsweise *Trimble Sketchup*[6]. Hierbei handelt es sich um ein Modellierungswerkzeug, das für nicht kommerzielle Anwendungen kostenfrei verwendet werden kann. Eine kommerzielle Version steht ebenfalls zur Verfügung und bietet unter anderem eine Reihe von Exportwerkzeugen, mittels derer erstellte Modelle in unterschiedliche Dateiformate exportiert und anschließend in anderen Modellierungswerkzeugen weiterverwendet werden können. Außerdem ermöglicht die kommerzielle Version den Import von 3D-Objekten aus anderen Modellierungsumgebungen, die anschließend in SketchUp-Modelle integriert werden können. Nachfolgend sei kurz auf die grundsätzlichen Modellierungsmöglichkeiten eingegangen, die SketchUp in der Version 13 bietet. Die beiden geometrischen Basisprimitive, mit denen der Nutzer seine Objekte erstellt, sind Linien und Flächen. Eine Fläche ist begrenzt von einem geschlossenen Linienzug, der entweder aus einer Menge vorgegebener geometrischer Primitive (Kreis, Rechteck etc.) stammt oder mittels des zur Verfügung stehenden Linientools selber erstellt wird.

Grundsätzlich ähneln die Schritte zur Erstellung eines Gebäudes in SketchUp dem Verfahren, das in dem hier vorgestellten System implementiert wird. Zunächst erstellt der

[4] Autodesk: AutoCAD Architecture. http://www.autodesk.de/adsk/servlet/pc/index?siteID=403786&id=14607160
[5] Autodesk: AutoCAD Mechanical. http://www.autodesk.de/adsk/servlet/pc/index?siteID=403786&id=14571830
[6] Trimble Buildings: SketchUp | 3D for Everyone. http://www.sketchup.com/de

Nutzer einen beliebigen Grundriss. Anschließend wird dieser Grundriss extrudiert, wodurch die Basisform des Gebäudes definiert ist. Da SketchUp allerdings nicht auf den Einsatzzweck der Gebäudemodellierung beschränkt, sondern als allgemeines Werkzeug für die Erstellung von 3D-Objekten vorgesehen ist, existieren keinerlei Automatismen für die Definition von gebäudetypischen Operationen wie der Erstellung von Stockwerken, Dächern oder Gesimsen. Die Integration solcher Komponenten muss manuell durch den Nutzer vorgenommen werden. So erstellt man Stockwerke, indem man den Grundkörper des Gebäudes unterteilt, für die Erstellung von Dächern muss man auf die vorhandenen Modellierungsoperationen zurückgreifen. Speziell die Erstellung von realistischen Dachstrukturen für komplexe Gebäudegrundrisse kann dadurch zu einer schwierigen Aufgabe werden, bei der der Nutzer nicht durch das System unterstützt wird. Gleiches gilt für die Erzeugung und Anbringung von Gesimsen. Auch hier muss zunächst ein Gesimsprofil definiert und anschließend extrudiert werden. Schwierigkeiten entstehen dabei unter anderem an Hausecken, an denen extrudierte Gesimse aufeinandertreffen. Auf diese Problematik wird im Abschnitt „Erzeugung von Gesimsstrukturen“ an späterer Stelle detailliert eingegangen. Dort werden die im Semantic Building Modeler für diesen Zweck implementierten Verfahren erläutert.

Sobald der Nutzer das grobe Gebäudemodell durch die Extrusion des Grundrisses erstellt hat, kann er dem Gebäude durch die Verwendung der SketchUp-Zeichentools weitere Details hinzufügen. So ist es möglich, beliebige Linienzüge direkt auf bereits bestehende Oberflächen zu zeichnen und diese anschließend zu extrudieren. Mittels dieses Verfahrens können Erker oder Mauervorsprünge leicht in ein Gebäude integriert werden. Weiterhin besitzt SketchUp eine Bibliothek mit Texturen, die in unterschiedlichen Kategorien vorliegen (beispielsweise Dachziegel, Mauertexturen etc.) und für die Texturierung der Gebäude verwendet werden können. Eine weitere Möglichkeit zur Texturierung besteht in der Verwendung von Gebäudefotografien, die direkt aus *Googles Street View*-Dienst[7] geladen und anhand der Geolokalisierung des Gebäudes auf dieses aufgetragen werden können. Dies ist eines der Werkzeuge, mit denen SketchUp den Nutzer bei der Rekonstruktion existierender Gebäude unterstützt. Weiterhin ist es möglich, Satellitenaufnahmen von Gebäuden als Hintergrundbild zu verwenden, anhand derer es möglich ist, Grundrisse schnell und einfach in SketchUp nachzuzeichnen. Die anschließend extrudierten Grundrisse können dann mit den aus Google Street View importierten

[7] Google: Street View: Erkunden Sie die Welt auf Straßenebene. http://maps.google.de/intl/de/help/maps/streetview/

Aufnahmen des Gebäudes texturiert und nach Abschluss der Modellierung das vollständige Model nach *Google Earth*[8] exportiert werden, um es online mit anderen Nutzern zu teilen.

Zusammenfassend ist SketchUp ein innovatives Werkzeug für die Modellierung von 3D-Objekten, da es im Vergleich zu anderen Werkzeugen speziell für Einsteiger deutlich einfacher zu verwenden ist. Dies zeigt sich bereits im Vergleich zwischen dem User-Interface von SketchUp und dem von Autodesk 3ds Max, einem ebenfalls sehr weit verbreiteten Modellierungswerkzeug, das hier exemplarisch gezeigt werden soll. Abbildung 10 zeigt einen Screenshot des User-Interfaces von SketchUp in der Version 13. Im Vergleich zu Abbildung 11, einem Bildschirmfoto aus der Software 3ds Max 2012, ist das Interface deutlich übersichtlicher, die Werkzeugpalette auf einige wenige Werkzeuge beschränkt.

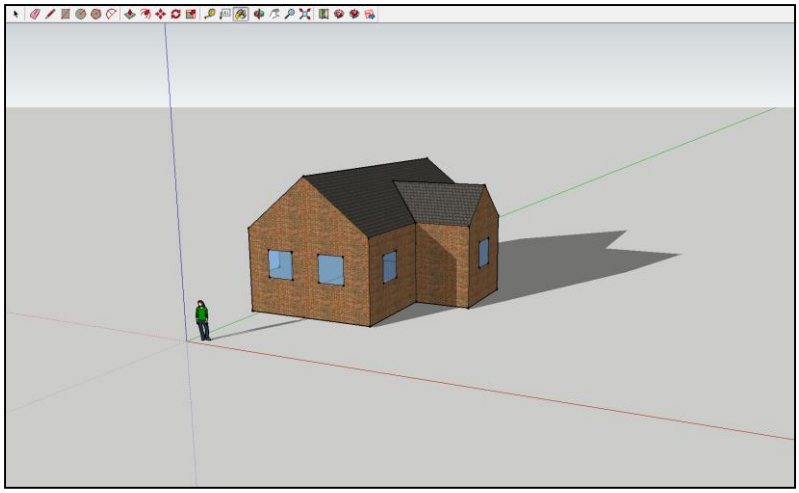

Abbildung 10: Trimble SketchUp 13 User Interface

Der Screenshot aus 3ds Max zeigt dagegen eine typische Benutzeroberfläche „professioneller“ Modellierungswerkzeuge, beginnend bei den unterschiedlichen Sichten (engl. *Viewports*) auf die gleiche Szene. Die einfachere Interfacestruktur in SketchUp macht es Einsteigern deutlich leichter, sich zurechtzufinden. Das erforderliche Vorwissen im Bereich der 3D-Modellierung ist deutlich geringer als bei Systemen wie 3ds Max. Somit ist SketchUp auch primär als Einsteigerprogramm zu betrachten, das dafür verwendet werden kann, um mit vergleichsweise geringem Aufwand 3D-Modelle zu erstellen. Die Fähigkeiten von Softwareumgebungen wie 3ds Max, die neben unterschiedlichen Modellierungstechnologien (CSG, Patches, Polygonnetze) auch die Erstellung und das

[8] Google: Google earth. http://www.google.de/intl/de/earth/index.html

Rendering animierter Szenen ermöglichen, erreicht das System nicht, darauf ist es allerdings auch nicht ausgerichtet.

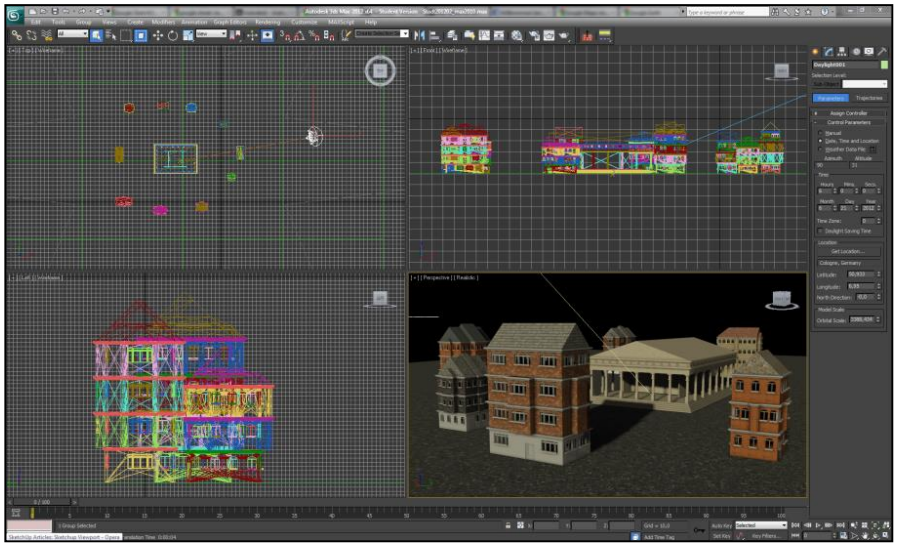

Abbildung 11: Autodesk 3dsMax 2012 User Interface

Obwohl SketchUp zusammenfassend ein sehr gut bedienbares Programm ist, mit dem man schnell und einfach Modelle beliebiger Gebäude erstellen kann, ist es nicht in der Lage, die Ziele zu erfüllen, die durch das vorgestellte System erreicht werden sollen. Zwar ist die Bedienbarkeit speziell für unerfahrene Nutzer vergleichsweise gut, allerdings hat es auch die Schwäche, die alle Modellierungswerkzeuge dieser Art haben, sei es nun AutoCAD, 3ds Max oder eben SketchUp. Konkret ist dies zum einen die nicht vorhandene Wiederverwendbarkeit des Modellierungsprozesses, zum anderen die Tatsache, dass es nach der Erstellung der Modelle nur mit großem Aufwand möglich ist, Änderungen an diesen vorzunehmen.

Zunächst sei kurz auf den Aspekt der fehlenden Wiederverwendbarkeit eingegangen. Typischerweise beschränkt sich diese auf die Nutzung einzelner Komponenten eines Gebäudes, wie Modelle für Fenster, Türen oder Säulen. Diese können in einer Modellbibliothek zusammengefasst und somit in anderen Projekten erneut eingesetzt werden. Einen solchen Ansatz verwendet auch das hier vorliegende System, indem Komponentenmodelle in beliebigen Modellierungsumgebungen erstellt und anschließend in das System geladen und von diesem automatisiert in die Gebäude integriert werden. Die Wiederverwendbarkeit in Bezug auf den Modellierungsprozess des gesamten Gebäudes ist dagegen nicht gegeben. Möchte der Nutzer beispielsweise 10 Gebäude eines ähnlichen

Gebäudestils erzeugen, so muss er in Modellierungssystemen 10 Mal die gleichen Schritte vornehmen. Nach der Definition eines Grundrisses erfolgt dessen Extrusion, dann die manuelle Unterteilung in Stockwerke, die Erzeugung eines Dachs etc. Je komplexer und detailreicher die Gebäude sind, desto größer wird auch der Aufwand. Prinzipiell wäre es möglich, ein Ausgangsgebäude zu modellieren, das anschließend vervielfältigt wird. An den Kopien würde man dann Modifikationen durchführen, um die Vielfalt zu erhöhen. Solche Modifikationen sind aber naturgemäß in Modellierungsumgebungen vergleichsweise aufwendig, da diese Oberflächendarstellungen der modellierten Objekte verwenden. Die Anpassungen des Grundrisses erfordern dadurch weitreichende Änderungen an sämtlichen Teilen der Geometrie und der darin positionierten Komponenten bis hinauf zum Dach. Dies führt dazu, dass der Zeitaufwand für die Erstellung einer Vielzahl von Gebäuden von deren Anzahl abhängt und konstant zunimmt. Abhilfe können hier nur Verfahren schaffen, die bestimmte Aufgaben im Kontext der Gebäudeerzeugung automatisieren und den Nutzer dadurch entlasten. Besonders wichtig ist dabei die Fähigkeit des Systems, automatisch Variationen zu erzeugen, sei es in Bezug auf die Grundrisse, die verwendeten Komponentenmodelle oder auf die Dachkonstruktion. Modellierungsumgebungen fehlt die Fähigkeit solcher automatisierter Modifikationen.

Trotz der genannten Nachteile von Modellierungsumgebungen spielen sie in der Produktion von digitalen Gebäudekomplexen und ganzen Städten eine wichtige Rolle und sind für diese zwingend erforderlich. Für die Erstellung einer Stadt benötigt man sowohl die Fähigkeiten von Modellierungswerkzeugen als auch die Algorithmen, die das vorliegende System dem Nutzer zur Verfügung stellt. Es handelt sich somit um komplementäre Ansätze. Dies liegt zum einen daran, dass ein Hauptansatz zur Erzeugung von Vielfalt bei der Konstruktion eines bestimmten Gebäudetyps im Laden und Verwalten von Komponentenobjekten besteht. So existieren für viele Stilepochen innerhalb der Architektur typische Fenster- oder Türformen. Auch Säulenstrukturen, die beispielsweise in Tempeln positioniert werden, sind einer bestimmten Stilrichtung zuordenbar. Solche Komponenten lassen sich beispielsweise anhand ihres Baustils gruppieren und können anschließend bei der Konstruktion von Gebäuden des jeweiligen Stils eingesetzt werden. Je größer die Bibliothek an Komponentenobjekten im Laufe der Zeit wird, desto größer wird auch die Vielfalt, die durch die Variationen in den verwendeten Komponenten erzeugt wird. Für die Modellierung solcher Komponenten, seien es nun Türen, Fenster oder auch Regenrinnen, greift man auf Modellierungssysteme zurück, da sich solche Komponenten meist nur mit großem Aufwand

prozedural erzeugen lassen. Hier ist die Verwendung von Softwarepaketen wie SketchUp oder 3ds Max der einfachere Weg. Allgemein gilt, dass auch die später vorgestellten prozeduralen Ansätze nicht ohne Modellierungssysteme auskommen, da prozedurale Systeme nur solche Komponenten erzeugen können, die durch Regeln oder Algorithmen beschrieben werden können. Je komplexer das zu modellierende Bauteil oder Gebäude, desto komplexer müssen naturgemäß auch die Prozeduren sein, die es erzeugen. Auf die Möglichkeiten und Einschränkungen prozeduraler Ansätze wird an späterer Stelle detailliert eingegangen, da auch diese Arbeit zu dieser Gruppe zählt.

Im nachfolgenden Kapitel werden fotogrammetrische Modellierungstechnologien vorgestellt, die eine halb-automatische Extraktion der Gebäudegeometrie aus Gebäudefotografien anstreben und dadurch versuchen, den Nutzer bei der Gebäudekonstruktion zu entlasten.

5.2 **Fotogrammetrisches Modellieren / Image Based Modeling (IBR)**

Wie in der Einleitung bereits kurz dargestellt, befassen sich Technologien aus dem Bereich des fotogrammetrischen Modellierens mit der Fragestellung, wie man aus Fotografien von Gebäuden deren Geometrie automatisiert extrahieren und als 3D-Modell repräsentieren kann. Debevec et al. entwickelten einen Prototyp für ein System, das semi-automatisch mit Hilfe des Nutzers aufgrund einer relativ geringen Anzahl von Bildern in der Lage ist, 3D-Modelle der dargestellten Gebäude zu errechnen [DTM96]. Bevor auf diese Arbeit eingegangen wird, seien kurz die Hauptprobleme erläutert, die durch Verfahren in diesem Bereich gelöst werden müssen.

Die Erzeugung von 3D-Modellen basierend auf Fotografien erfordert die Zuordnung von Punkten innerhalb eines 2D-Bildraums zu Punkten innerhalb eines dreidimensionalen Koordinatensystems. Das Verfahren muss demnach die dritte Dimension der Punktkoordinaten aus den Fotografien ableiten. Die Koordinate liegt dabei irgendwo auf einem Strahl ausgehend vom Kamerablickpunkt durch den aufgenommenen Punkt in der Szene. Ohne zusätzliche Informationen ist es nicht möglich, den Wert des Punktes bezüglich der dritten Koordinatenachse zu berechnen. Aus diesem Grund benötigen solche Ansätze mehrere Fotografien des gleichen Gebäudes, die aus unterschiedlichen Positionen angefertigt wurden. Theoretisch ist es dann möglich, aus einer ausreichenden Menge von Bildpunkten, die in allen Fotografien identifiziert werden, die Koordinaten der jeweiligen

Punkte innerhalb des dreidimensionalen Koordinatenraums zu errechnen. Außerdem lassen sich anhand dieser Berechnungen die Kamerapositionen und Blickrichtungen der Kamera rekonstruieren. Eine solche Identifikation der Punkte muss häufig durch den Nutzer erfolgen. Ein erster Schritt in solchen Systemen besteht darum typischerweise darin, dem System eine Hilfestellung zu geben, indem Bezugspunkte definiert werden, anhand derer anschließend die Berechnungen erfolgen können. Diese erste manuelle Intervention durch den Nutzer ist aufgrund des zweiten großen Problems erforderlich, mit dem fotogrammetrische Verfahren konfrontiert sind. Dieses wird als *Übereinstimmungsproblem* (engl. *Correspondance Problem*) bezeichnet und beschreibt die Schwierigkeit, automatisiert identische Punkte der Geometrie in unterschiedlichen Bildern zu identifizieren [DTM96]. Die Fotografien stellen dabei Projektionen der gleichen Punkte auf unterschiedlichen Projektionsebenen dar. Die Ermittlung der Ausgangspunkte wird dabei durch verschiedene Aspekte erschwert. Dazu gehören zunächst sämtliche Aspekte, die die Kamera und somit die Aufnahmeposition betreffen. Ein geänderter Aufnahmepunkt mit anderem Blickwinkel kann zu perspektivischen Verzerrungen innerhalb des entstehenden Bildes führen, die sich beispielsweise in unterschiedlichen Größen und Formen des Punktes manifestieren. Bewegt man sich beispielsweise auf das fotografierte Objekt zu, so ändert sich der Maßstab des Bildes. Weitere Probleme resultieren aus Licht- und Schattenverhältnissen, die zu unterschiedlichen Farbwerten eines Punktes in den Fotografien führen können. Ähnliche Effekte werden auch durch Glanzlichter ausgelöst, die auf spekularen Oberflächen entstehen, wenn der Blickwinkel der Reflektionsrichtung einer einfallenden Lichtquelle ähnelt.

Vereinfacht gesagt gibt es eine Vielzahl von Phänomenen, die dazu führen, dass die Farben der Punktprojektionen auf unterschiedlichen Fotos unterschiedlich sind. Ein naiver Ansatz zur Bestimmung solcher identischen Punkte besitzt aber zunächst nur die Farbinformationen und muss versuchen, für einen bestimmten Farbwert im ersten Bild einen Punkt im zweiten Bild zu finden, der einen identischen oder zumindest ähnlichen Farbwert besitzt. Offensichtlich ist ein solches naives Vorgehen ungeeignet, um brauchbare Ergebnisse zu produzieren. Aus diesem Grund versucht man, dem Computer Zusatzinformationen zu geben, die er für die Auswertung der Eingabebilder nutzen kann. Ein gängiger Ansatz hierfür ist die Verwendung kalibrierter Kameras. Die Kenntnis über die Brennweite und weitere Parameter wie die Verzerrungen, die durch die Kameraoptik entstehen, erleichtern die Zuordnung der projizierten Punkte zu ihren dreidimensionalen Pendants [DTM96].

Andere Ansätze verwenden zusätzlich festgelegte Kamerapositionen und Blickwinkel, um das Übereinstimmungsproblem zu lösen oder zumindest zu vereinfachen.

5.2.1 **Façade [DTM96]**

Nachdem vorab die Schwierigkeiten erörtert wurden, die bei Ansätzen zum Image Based Modeling gelöst werden müssen, soll nun ein System vorgestellt werden, das Debevec et al. 1996 entwickelt haben. Das *Façade* genannte System ist ein interaktives Modellierungsprogramm, mittels dessen der Nutzer in der Lage ist, aus einer Menge von Eingabebildern semi-automatisch 3D-Modelle der dargestellten Gebäude erzeugen zu lassen. In ihrer Arbeit stellen die Autoren drei neue Technologien vor. Dies sind ein Ansatz zum fotogrammetrischen Modellieren, eine Technologie zum ansichtsanhängigen Textur-Mapping und eine Methodik zur Lösung des Übereinstimmungsproblems, die als *modellbasiertes Stereo* bezeichnet wird. Sie sehen einen großen Beitrag ihrer Arbeit in der Untergliederung des Bildmodellierungsprozesses in eine Menge von Schritten, die entweder sehr gut durch den menschlichen Nutzer oder durch automatisierte Algorithmen ausgeführt werden können [DTM96]. Nachfolgend wird auf die Implementation der fotogrammetrischen Modellierung bei Debevec et al. eingegangen, bevor anschließend aktuellere Ergebnisse innerhalb dieses Forschungsfeldes vorgestellt werden.

Der Modellierungsprozess bei Debevec et al. ist ein Prozess, in dem der Nutzer unter Verwendung einer vorgegebenen Menge geometrischer Grundkörper (bsw. Quader oder Zylinder) zunächst die ungefähre Struktur des Gebäudes nachbaut. Auf geometrische Details und Ausdehnungen muss er in dieser Phase keinerlei Rücksicht zu nehmen. Anschließend verbindet er die Kanten der volumetrischen Grundkörper in seinem vorläufigen Modell mit Kanten innerhalb der Fotografien und stellt dadurch die Verbindung zwischen dem zweidimensionalen Bildraum und dem dreidimensionalen Modellierungsraum her, innerhalb dessen später das fertige Modell erzeugt wird. Dadurch ist es möglich, einen vorhandenen Grundkörper an verschiedene Blickwinkel auf das gleiche Gesamtgebilde anzupassen. Aufgrund der Verwendung solcher geometrischer Grundkörper kann dann ein vollständiges geometrisches Modell aus den Bildern abgeleitet werden, auch wenn auf diesen unter Umständen Teile dieses Modells verdeckt und somit für den Rechner nicht sichtbar sind. Der Eingriff des Nutzers in Form der Modellierung der Grundstruktur macht diese Ergänzung potentiell unsichtbarer Gebäudebestandteile dadurch möglich, dass er die Grundformen bereits als dreidimensionale Körper modelliert [Wa02]. Diese sind innerhalb

des Systems parametrisiert, ein Quader beispielsweise über seine Länge, Breite und Höhe. Diese Parameter müssen nicht durch den Nutzer angegeben werden, da deren Bestimmung durch Auswertung der Eingabebilder erfolgt. Allerdings ist es möglich, die Parameterwerte in ihrem Wertebereich einzuschränken und dadurch Einfluss auf die Berechnungsergebnisse des Algorithmus zu nehmen.

Der Detailgrad des errechneten Modells hängt von verschiedenen Punkten ab. Dazu gehört die Grundstruktur des Gebäudes unter Berücksichtigung der Frage, wie gut es durch die zur Verfügung stehenden Grundkörper modelliert werden kann. Weiterhin spielt auch die Anzahl der Details, die der Nutzer in den Bildern und dem Modell markiert, eine wichtige Rolle. Je mehr Details der Nutzer dem Rechner zur Berücksichtigung gibt, desto detaillierter wird das entstehende Modell sein.

Ein weiterer wichtiger Schritt innerhalb des Façade-Systems ist die Ermittlung der Kamerapositionen basierend auf dem erstellten Modell und den Fotografien des zu modellierenden Gebäudes. Aufgrund der Zuordnung der Kanten der geometrischen Grundkörper zu den Kanten innerhalb der Eingabebilder kann der Rechner die Positionen interpolieren, aus denen die Bilder aufgenommen wurden. Konzeptuell handelt es sich hierbei um ein Minimierungsproblem, bei dem der Computer basierend auf Vorinformationen zunächst Kameraposition und Blickrichtung rät. Anschließend wird das errechnete 3D-Modell unter Verwendung der so bestimmten Kameraparameter auf die vermutete Bildebene projiziert. Ein Vergleich des virtuell erstellten Bildes mit dem Eingabefoto gibt nun Aufschluss über die Qualität des vorherigen „Ratens“, indem die Position und Ausrichtung der markierten Kanten in beiden Bildern miteinander verglichen werden. Liegen die Abweichungen oberhalb eines vorab gesetzten Grenzwertes, verwendet das System die Abweichungsdaten zur Neuberechnung der vorab bestimmten Kameraparameter und führt so lange weitere Iterationen durch, bis die Werte den Qualitätsvorgaben genügen.

Eine möglichst exakte Bestimmung der Kamerapositionen der Quellaufnahmen benötigen die Autoren für das ansichtsabhängige Texturmapping [DTM96], das sie für die Erzeugung neuer Ansichten des Gebäudes während des Renderings verwenden. Dabei wird das erzeugte geometrische Modell des Gebäudes unter Verwendung der Ansichtsparameter (Kameraposition, Blickwinkel etc.) erneut projiziert. Anschließend werden die Fotografien des Gebäudes als Texturmaps eingesetzt. Diese Maps nutzen die Autoren, um Pixel im neu erzeugten Bild einzufärben. Typischerweise stehen dabei zwei oder mehr Ansichten des

gleichen Punktes zur Verfügung, die allerdings aufgrund des Eingangs erwähnten Problems nicht zwingend dieselbe Farbe besitzen. Durch die vorab durchgeführten Berechnungen des Modells, der Kantenzuordnung und der Errechnung der Kamerapositionen ist es allerdings vergleichsweise einfach, das Übereinstimmungsproblem zu lösen. Dadurch kann bestimmt werden, welchem Bildpunkt in Eingabebild A welcher Bildpunkt in weiteren Eingabebildern entspricht. Das System sammelt nun die Farbwerte aus den Eingangsbildern und mischt diese, wobei der Beitrag eines Punktes zum finalen Farbwert durch den Winkel gewichtet wird, den der neue Blickwinkel vom Blickwinkel im Referenzbild abweicht. Je größer der Unterschied zwischen dem neuen Blickwinkel und dem Blickwinkel im Referenzbild, desto kleiner ist der Beitrag des Referenzbildes zum neu berechneten Bild.

Abbildung 12 zeigt die einzelnen Schritte, die bei der Rekonstruktion eines Gebäudes durch das Façade-System durchlaufen werden. Das erste Bild zeigt ein Foto des *Campanile* (Glockenturm in Berkley), innerhalb dessen der Nutzer bereits die einzelnen Kanten markiert hat. Das zweite Bild zeigt das erzeugte 3D-Modell. Im dritten Bild sieht man die Ergebnisse der Rückprojektion des Modells auf das Quellfoto unter Verwendung der vorab errechneten Kameraparameter. Das letzte Bild zeigt nun ein Rendering des Modells mittels ansichtsabhängigem Texturmapping. Der hohe Detailreichtum in der Darstellung resultiert aus der Verwendung der Eingangsbilder als Texturmaps.

Abbildung 12: Aufnahmen aus dem Façade-System von Debevec et al. [DTM96]

Inzwischen existiert eine Reihe von kommerziellen Produkten für das fotogrammetrische Modellieren, die aufgrund aktueller Forschungsergebnisse in der Lage sind, komplexere Modelle aus Eingabefotografien abzuleiten. Diese Technologie ist dabei nicht nur im

Bereich der Gebäuderekonstruktion einsetzbar, sondern auch in anderen Kontexten. Die Software *PhotoModeler* der Firma *EOS*[9] beispielsweise kann nach eigener Aussage nicht nur für die Architekturrekonstruktion eingesetzt werden, sondern auch in anderen Forschungsbereichen wie der Archäologie, der Geologie oder der Produktion von 3D-Modellen für Filme. *RhinoPhoto*[10] ist ein Plug-In für das CAD-System *Rhinoceros*[11] der Firma *McNeel* mittels dessen es möglich ist, automatisiert 3D-Koordinaten aus Fotos zu extrahieren und innerhalb der Rhinoceros-Modeling-Umgebung zu modifizieren und weiterzuverarbeiten.

Trotz der Tatsache, dass seit der Entwicklung von Façade durch Debevec et al. und aktuellen fotogrammetrischen Lösungen mehr als 15 Jahre vergangen sind, bleibt das Image Based Modeling ein semi-automatischer Prozess. Zwar gibt es in den verschiedenen Softwarepaketen Ansätze, die erforderliche Nutzerintervention zu reduzieren, vollständig ohne menschliches Eingreifen kommen aber auch diese Systeme nicht aus. Dabei steigt der Aufwand mit der Komplexität des zu digitalisierenden Modells. Ein modernes Hochhaus, dessen Grundform relativ leicht durch eine Menge quaderförmiger Grundkörper angenähert werden kann und sich nicht durch viele Details auszeichnet, ist für ein solches System einfacher und erfordert weniger Nutzeraufwand als eine gotische Kathedrale.

Unterschiede existieren zwischen den Systemen allerdings in der Art, wie der Nutzer den Computer bei der Modellerzeugung unterstützt. PhotoModeler ähnelt hierin dem Ansatz von Debevec et al., indem der Nutzer für eine Menge von Eingabebildern zunächst Features in den Bildern markiert. Durch die Markierung eines Punktes oder einer Kante in allen Bildern löst der Nutzer das Übereinstimmungsproblem und erleichtert dadurch die Modellerzeugung [Ab01]. Rhinophoto geht einen anderen Weg, um dieses Problem zu lösen und benötigt vor der Anfertigung der Eingabefotos das Anbringen von Referenzpunkten auf dem zu digitalisierenden Modell. Anschließend erstellt man eine Reihe von Fotografien des präparierten Objekts. Das Plug-In ist dann in der Lage, aus den Referenzpunkten automatisiert 3D-Punkte zu erstellen, die anschließend in das Rhinoceros CAD-System importiert und weiterverarbeitet werden können.

[9] EOS: PhotoModeler. http://www.photomodeler.com/
[10] RhinoPhoto. http://www.rhinophoto3d.com/
[11] McNeel: Rhinoceros. http://www.de.rhino3d.com/

5.2.2 **Diskussion fotogrammetrischer Modellierungswerkzeuge**

Fotogrammetrische Modellierung stellt einen Ansatz dar, mittels dessen man basierend auf Fotografien semi-automatisch 3D-Modelle der dargestellten Objekte erzeugen kann. Die Einsatzbereiche solcher Werkzeuge sind dabei nicht auf die Rekonstruktion von Gebäuden beschränkt, vielmehr handelt es sich um einen allgemeinen Ansatz, der auch für andere Einsatzzwecke nützlich ist. Aufgrund der einleitend vorgenommenen Einteilung von Technologien für die automatisierte Erzeugung von Gebäudemodellen aufgrund einerseits des Aufwands der Nutzerintervention und andererseits der Komplexität der erstellten Modelle soll zunächst der Aufwand für den Nutzer betrachtet werden.

Dieser unterscheidet sich zwischen den verschiedenen Systemen, so benötigt Rhinophoto das Anbringen von Referenzpunkten im Ausgangsmodell, PhotoModeler lässt den Nutzer diese Aufgabe direkt auf den Fotografien durchführen. Wie bei CAD-Systemen wächst somit der Aufwand für den Nutzer in Abhängigkeit von der Komplexität des Modells, das digitalisiert werden soll. Je mehr Details vorhanden sind, desto mehr Aufwand muss durch den Nutzer betrieben werden. Trotzdem ist der Aufwand während des Erstellungsprozesses als geringer einzuordnen, als dies bei einer manuellen Konstruktion mittels CAD-Systemen der Fall ist, da zumindest Teile der Architektur durch die Systeme automatisiert extrahiert und in 3D-Modelle überführt werden können.

Auf der anderen Seite ist aber auch die Komplexität der entstehenden 3D-Modelle geringer, da sehr feine Strukturen, wie sie beispielsweise bei Gesimsen oder Säulenstrukturen vorhanden sein können, nur mit großem Nutzeraufwand extrahierbar sind. Ein weiterer Nachteil besteht in der Einschränkung des Einsatzbereiches solcher Systeme auf real existierende Objekte oder zumindest auf solche, von denen Fotografien existieren. Zunächst können somit nur Rekonstruktionen existierender Architektur erstellt werden, die Konstruktion neuer Gebäude ist nicht möglich. Weiterhin sind die Ergebnisse des Modellierungsprozesses nicht wiederverwendbar.

Wie bei der CAD-basierten Modellierung ist das Ergebnis ein 3D-Modell des fotografierten Objekts, der Modellierungsprozess ist nur für dieses eine Objekt einsetzbar. Modifikationen an der zugrundeliegenden Geometrie sind aufwendig, da sie direkt auf den Drahtgittermodellen aufsetzen und mit steigender Modellkomplexität ebenfalls aufwendiger werden. Das Wissen über den Modellierungsprozess, beispielsweise die Kantenzuordnungen oder das Anbringen von Markern auf dem Objekt, sind nur für das jeweilige Objekt gültig. Solche Systeme sind nicht in der Lage, das im Erstellungsprozess erlangte Wissen

wiederzuverwenden, um ähnliche Gebäude mit Variationen zu berechnen. Die Zielsetzung des vorliegenden Systems, automatisiert Gebäude aufgrund nutzerdefinierter Eingaben zu erzeugen, kann darum durch rein fotogrammetrische Technologien nicht erreicht werden. Darüber hinaus gewinnen Systeme dieses Typs kein neues Wissen aus der Digitalisierung, das für die Modellerzeugung für weitere Objekte einsetzbar ist.

5.2.3 **Build-by-Number [BA05]**

Genau an diesem Punkt setzt nun die Arbeit von Bekins und Aliaga an [BA05]. Sie kombinieren den Ansatz von Debevec et al. mit einer Technik zur Extraktion von Strukturmustern aus den rekonstruierten Gebäuden. Analog zum Façade-System modelliert der Nutzer im ersten Schritt zunächst die groben Gebäudestrukturen unter Verwendung volumetrischer Grundkörper. Anschließend ordnet er Kanten in den Eingabebildern den Kanten dieser Körper zu und kann für die Berechnung der Grundkörperparameter Einschränkungen vornehmen. Das System bestimmt daraufhin durch die Analyse der Eingabefotografien Parameter für die einzelnen Grundkörper, die die Form des Gebäudes beschreiben. Dabei verwenden Bekins und Aliaga den gleichen Algorithmus zur Parameterbestimmung wie er ursprünglich von Debevec et al. vorgeschlagen wurde. Das so errechnete Modell wird intern durch eine Menge dreidimensionaler Grundkörper repräsentiert, die innerhalb eines Szenegraphen angeordnet sind. Die Autoren bezeichnen diese Grundkörper als *building blocks* [BA05]. Jeder Knoten ist ein Block, die Kanten zwischen den Knoten beschreiben Transformationen der Blöcke relativ zu ihrem Eltern- oder Kindknoten. Dadurch werden Position und Ausrichtung der Grundkörper immer relativ zu Blöcken spezifiziert, mit denen eine räumliche Beziehung existiert. Die Blöcke selber bestehen aus einer Menge von Vertices und einer einfachen geometrischen Struktur (bsw. Quader oder Zylinder).

Um die Unterteilung der Grundkörper durchzuführen, unterstützt das System Unterteilungsoperationen auf zwei verschiedenen Ebenen. Bei der *Blockunterteilung* wird ein Block durch eine Unterteilungsebene in Subblöcke aufgespalten. Diese Operation wird beispielsweise eingesetzt, um Stockwerke innerhalb eines Gebäudes zu erzeugen. Die zweite Unterteilungsoperation erfolgt auf der Ebene einzelner Faces, also Seitenflächen der Blöcke. Im Gegensatz zur Blockunterteilung wird bei der *Faceunterteilung* nur eine Seitenfläche in mehrere Subseitenflächen unterteilt, die restlichen Flächen bleiben dagegen unangetastet.

Die Festlegung solcher Unterteilungen ist im System von Bekins und Aliaga Aufgabe des Benutzers.

Abbildung 13 illustriert die unterschiedlichen Unterteilungsoperationen. Das linke Bild zeigt das Ergebnis einer wiederholten Blockunterteilung angewendet auf das erstellte Gebäudemodell, welche die einzelnen Stockwerke des Gebäudes erzeugt. Die rechte Seite visualisiert die Ergebnisse der Flächenunterteilung durch den Nutzer.

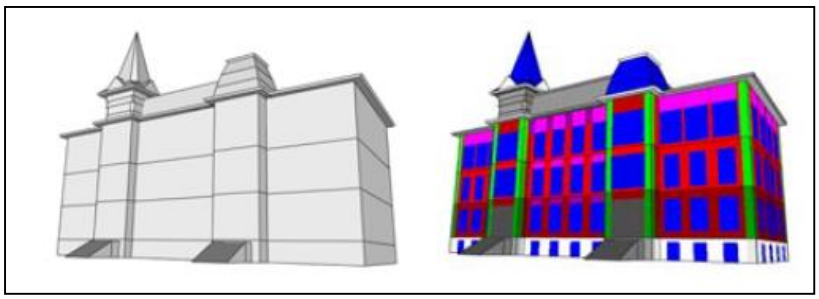

Abbildung 13: Unterschiedliche Unterteilungsoperationen [BA05]

Die Generierung von wiederverwendbarem Wissen durch die Extraktion von Strukturregeln anhand der nutzergenerierten Unterteilungen erfolgt auf verschiedenen Ebenen, die als *Schemata* bezeichnet werden. Beim *Face-Schema* bewegt man sich auf der untersten Ebene der möglichen Unterteilungen. Hier geht es um die Unterteilungen von Seitenflächen. Dabei untergliedert der Nutzer Oberflächen rekursiv in verschiedene Spalten. Diese einzelnen Spalten werden bestimmten Kategorien zugeordnet, im rechten Bild in Abbildung 13 werden diese Zuordnungen durch unterschiedliche Farbzuweisungen verdeutlicht. Kodiert man nun die Zuordnungen durch Buchstaben, so könnte eine Fassade mit drei verschiedenen Oberflächentypen (beispielsweise *A, B* und *C*) ein Unterteilungsmuster aufweisen, das durch eine Zeichenkette *ABCBCBCBA* beschrieben werden kann. Die Idee ist es nun, sich wiederholende Muster in derart kodierten Fassadenunterteilungen zu entdecken und dadurch allgemein einsetzbare Musterbeschreibungen zu erhalten.

Wiederholt sich eine Zeichenfolge innerhalb eines Musters, so verwenden die Autoren den *Kleene-Stern* (*), um die Häufigkeit des Vorkommens zu kodieren. Diese Form der Notation findet in vielen Bereichen der Informatik Anwendung, beispielsweise im Kontext der regulären Ausdrücke. Sei L eine Menge von Buchstaben oder Zeichenketten, so definiert L^* die Menge aller Zeichenketten, die sich durch Konkatenation von 0 oder mehr Elementen aus L erzeugen lassen [EEP02]. Angewendet auf das vorab verwendete Beispiel für ein Fassadenmuster ließe sich die Zeichenkette *ABCBCBCBA* unter Verwendung des Kleene-

Sterns formuliert als Ersetzungsregel schreiben als *AB(AB)*A*. Eine derart abgeleitete Ersetzungsregel kann anschließend eingesetzt werden, um Oberflächen anderer Gebäude ähnlich zu unterteilen und zu texturieren.

Neben der reinen Unterteilung in unterschiedliche Typen von Flächenelementen muss der Nutzer für jede derart kreierte Gruppe angeben, ob diese eine feste oder potentiell variable Größe besitzt. Der Sinn dieser Festlegung wird deutlich, sobald man sich die Zielsetzung des Verfahrens vor Augen führt. Diese besteht darin, Strukturmuster in der Geometrie, aber auch in den Texturen zu kennzeichnen und diese Muster anschließend wiederzuverwenden. Dabei

werden die Unterteilungen im *Face-Schema* nicht auf die Geometrie übertragen, sondern unterteilen viel mehr die Textur, die auf diese gemappt wird. Da es sich bei dem Verfahren um ein Hybridverfahren aus fotogrammetrischen und prozeduralen Komponenten handelt, stammen diese Texturen wiederum aus den Referenzbildern, die der Nutzer zu Beginn von dem zu rekonstruierenden Gebäude angefertigt hat. Das Übertragen der Muster auf ein neues Gebäude besteht anschließend darin, dass man Teile der Textur aus dem Originalbild ausschneidet und anhand des Strukturmusters auf das neue Model aufträgt. Bei Ziegelsteinen oder anderen Wandtexturmustern ist es dabei nicht zwingend erforderlich, dass die Größe zwischen Quelle und Ziel exakt übereinstimmt, diese können bei Bedarf mehrfach auf die Oberfläche aufgetragen (engl. *Tiling*) oder beschnitten werden. Bei Fenstern oder Türen im Original sollte die Ausdehnung dagegen beibehalten werden, um das Muster wiederverwenden zu können.

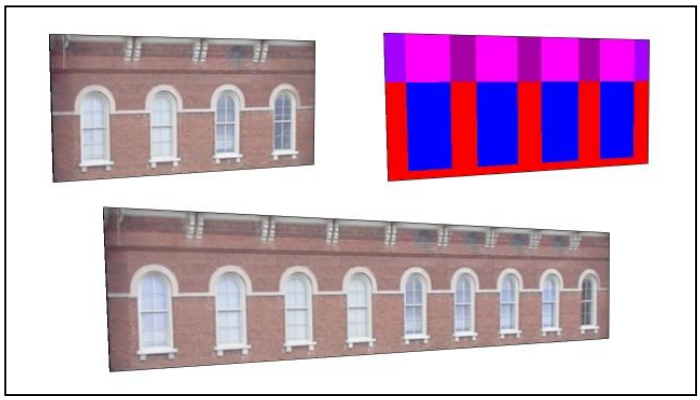

Abbildung 14: Anwendung des Face-Schemas [BA05]

Abbildung 14 zeigt exemplarisch die verschiedenen Schritte bei der Erstellung und Anwendung von Ersetzungsregeln für das Face-Schema. Links oben ist zunächst ein Teil der Fassade eines rekonstruierten Gebäudes zu sehen. Der Nutzer untergliedert rekursiv in verschiedene Oberflächenelemente, die er jeweils einer bestimmten Gruppe zuweist. Diese Zuordnung ist im rechten oberen Bild durch die Verwendung unterschiedlicher Farben kodiert. Aus dieser Unterteilung leitet das System nun eine Ersetzungsregel ab. Diese Ersetzungsregel wird im unteren Bild angewendet, um eine deutlich längere Fassade im gleichen Stil zu unterteilen und zu texturieren. Dabei werden die mittleren Elemente mehrfach wiederholt und füllen dadurch den zur Verfügung stehenden Raum aus.

Die zweite Ebene auf der Strukturregeln abgeleitet werden, ist das Floor-Schema, das sich mit der Beschreibung von Strukturen innerhalb von Stockwerken befasst. Ein Stockwerk besteht aus einer Menge von Oberflächenelementen, die durch Face-Schemata beschrieben werden. Die interne Repräsentation eines Stockwerks basiert auf der Verwendung eines ungerichteten zyklischen Graphs, dessen Knoten jeweils einzelne Oberflächenelemente darstellen. Jeder Knoten besitzt einen linken und einen rechten Nachbarn, mit denen er über die Kanten im Graphen verbunden ist. Die Kanten speichern darüber hinaus auch die Winkel, in denen die adjazenten Elemente zueinander stehen.

Auf der obersten Ebene wird das Gebäudemodel als Ganzes durch das Model-Schema repräsentiert. Auch hierfür wird eine Graphenstruktur verwendet, bei der die einzelnen Stockwerke die Knoten innerhalb des Graphen bilden. Jeder Knoten mit Ausnahme von Erd- und Dachgeschoss besitzt Kanten zu seinem Vorgänger- und Nachfolgerstockwerk. Bei der Anwendung eines Model-Schemas auf ein neues Gebäude können dann die Zwischenstockwerke innerhalb des Gebäudes mehrfach generiert werden.

Der Ansatz von Benkins und Aliaga [BA05] schwächt einen der vorab genannten Kritikpunkte ab, da durch die Extraktion von Strukturmustern auf verschiedenen Modellebenen eine Wiederverwendbarkeit in den Modellierungsprozess integriert wird. Dadurch ist es möglich, für neue geometrische Grundmodelle sowohl die Unterteilungen als auch die Gebäudefotografien für das neue Gebäude zu nutzen. In früheren Arbeiten bot der fotogrammetrische Modellierungsprozess dagegen keine Möglichkeit, Teile des Prozesses wiederzuverwenden. Damit löst sich diese Technik automatisch auch von ihrer Einschränkung, dass sie nur Gebäude modellieren kann, von denen Fotos existieren, somit auf reine Rekonstruktionsanwendungen beschränkt ist. Der Build-by-Number-Ansatz erlaubt

durch den Transfer von Strukturmustern auf beliebige geometrische Grundmodelle die Erschaffung von Gebäuden, die in der Realität nicht existiert haben müssen.

Bezüglich des Nutzeraufwandes erkennt man eine Tendenz, die man häufig im Bereich prozeduraler Technologien findet. Hier muss zunächst vergleichsweise viel Zeit in die Formulierung beziehungsweise Erzeugung der Strukturregeln investiert werden. Sind diese aber einmal erstellt, kann die Anwendung solcher Regelsysteme neue Modelle deutlich schneller und effizienter erstellen. Einen solchen Gewinn zieht man dagegen aus CAD-basierten Ansätzen nicht, sofern man die Erfahrung, die der Nutzer im Umgang mit einem solchen System gewinnt, außer Acht lässt. Betrachtet man einen sehr erfahrenen Nutzer, der sich mit einem CAD-System seiner Wahl sehr gut auskennt, so wird die Zeit, die er in die Erstellung eines Gebäudes investieren muss, typischerweise direkt von dessen Komplexität abhängen, selbst wenn dieses Gebäude Ähnlichkeiten zu bereits modellierten Gebäuden besitzt. Diese Ähnlichkeiten sind im manuellen Modellierungsprozess nicht nutzbar, abgesehen von der Möglichkeit, vollständige Teile eines Gebäudes zu kopieren oder als Ausgangspunkt für die Modellierung neuer Varianten zu verwenden. Prozedurale Ansätze, die in ihrer reinen Form im nachfolgenden Abschnitt detailliert diskutiert werden, besitzen ihre Stärken speziell in solchen Bereichen, in denen Ähnlichkeiten bestehen, wie man bereits ansatzweise am Build-By-Number-System erkennt.

Eines der Ziele dieser Arbeit ist die Erzeugung von Gebäuden mit einer hohen geometrischen Komplexität. Betrachtet man fotogrammetrische Technologien unter dieser Vorgabe, so muss festgehalten werden, dass solche Systeme nicht in der Lage sind, sehr feine Strukturen wie sie beispielsweise bei Gesimsen oder Säulen vorkommen, abzubilden. Diese Problematik zeigt sich sowohl bei Debevec et al. [DTM96] als auch in Nachfolgearbeiten wie der von Bekins und Aliaga [BA05]. Grundprinzip ist die Erzeugung von Gebäudestrukturen durch die Verwendung vergleichsweise einfacher Grundkörper, die zur Unterstützung des fotogrammetrischen Modellierungsprozesses durch den Nutzer platziert werden. Dadurch werden zunächst die groben Formen des Gebäudes abgebildet, anhand derer sich anschließend Parameter wie die Kameraposition aus den Eingabefotografien ableiten lassen. Dies ist aus dem Grund erforderlich, da die vorgestellten Arbeiten ansichtsabhängiges Texturmapping als Rendering-Technik einsetzen. Durch diese IBR-Technologie ist es möglich, aus einer Menge von vorhandenen Blickwinkeln neue Blickwinkel auf ein Objekt zu erzeugen [Wa02]. Ein solches Verfahren wurde bereits im Kontext von Façade thematisiert. Vereinfacht gesagt verwendet man eine Überabtastung, bei

der für einen Punkt im neu zu errechnenden Bild Farbwerte des gleichen Punktes aus verschiedenen Referenzbildern gewichtet kombiniert werden. Daraus ergibt sich dann der finale Farbwert im neu gerenderten Bild [DTM96]. Solche Verfahren können je nach Qualität der Referenzbilder gute Renderingergebnisse erreichen, haben aber naturgemäß den Nachteil, dass sie geometrische Komplexität nur vortäuschen, indem sie Texturen einsetzen, die komplexe Details auf die darunterliegende, einfache Geometrie auftragen. Für Systeme, die Gebäude nur aus mittlerer bis großer Distanz darstellen, reichen solche Ansätze aus und bieten darüber hinaus Performancevorteile aufgrund der meist geringen Polygonanzahl der darunterliegenden Modelle. Speziell die Renderingperformance spielt für das hier vorgestellte System aber nur eine untergeordnete Rolle, da es als reines Erzeugungssystem und nicht als Renderengine konzipiert ist. Zusammenfassend sind fotogrammetrische Technologien aufgrund der Einschränkung der geometrischen Komplexität nicht geeignet, die gesetzten Ziele zu erreichen, können aber in anderen Kontexten gute Ergebnisse erzielen.

5.3 **Prozedurale Ansätze**

Prozedurale Technologien werden bereits seit Mitte der 1980er Jahre in der Computergrafik in unterschiedlichen Kontexten eingesetzt. Ebert definiert Ansätze dieser Art als „code segments or algorithms that specify some characteristics of a computer-generated model or effect“ [Eb02]. Im Mittelpunkt solcher Technologien steht somit die Prozedur, die durch die Implementation eines Algorithmus in der Lage ist, bestimmte Objekte zu erzeugen. Die ersten Arbeiten gehen zurück auf Perlin, der 1985 eine Rauschfunktion vorstellte, die für die Erzeugung realistischer Texturen für Marmor oder Holzmaserungen einsetzbar ist [Pe85]. Die Kernidee dieses Ansatzes ist, eine regelmäßige Funktion durch eine auf Rauschelementen basierende Turbulenzfunktion zu stören. Für die Erzeugung von Texturen indiziert man anschließend eine Farb-Tafel mit Werten, die aus der gestörten Basisfunktion berechnet werden und erhält dadurch realistische Texturen für unterschiedliche Arten von Materialien. Der Einsatz einer Turbulenzfunktion ist erforderlich, da die regelmäßige Basisfunktion zu regelmäßigen Mustern und Strukturen in der Ergebnistextur führen würde. Solche Regelmäßigkeiten stellen in der Natur die Ausnahme dar. Dies ist der Grund, warum Texturen für Oberflächen, die Unregelmäßigkeiten und Störungen enthalten deutlich weniger „synthetisch“ wirken, als dies bei sehr regelmäßigen Strukturen der Fall ist. Solchen Texturen sieht man ihre synthetische Herkunft sehr leicht an. In seinem Marmor-Beispiel verwendete Perlin eine regelmäßige Sinusfunktion, deren Eingabeparameter x mit der

Ausgabe einer Turbulenzfunktion kombiniert wurde [Pe85]. Abbildung 15 zeigt ein Gefäß mit einer Marmortextur, die mittels der skizzierten Technik erstellt wurde.

Anhand dieses Beispiels lassen sich die Vorteile des prozeduralen Modellierens gut erläutern. Zunächst ist hier die Abstraktion zu nennen, die durch die Verwendung von Prozeduren erreicht wird. Diese Abstraktion besteht in der Formulierung einer Berechnungsvorschrift, die in der Lage ist, über Parameter bestimmte Ausgaben zu erzeugen. Am Beispiel von Perlins Marmortextur muss diese nicht vorab generiert und als Rastergrafik gespeichert werden, vielmehr ist es mittels der definierten Funktionen möglich, die Farbe jedes Punktes zu errechnen. Offensichtlich spart ein solches Vorgehen Speicherplatz, da es nicht notwendig ist, das Texturbild vorzuhalten, stattdessen kann es bei Bedarf dynamisch erzeugt werden.

Abbildung 15: Gefäß mit Perlin-Noise-Marmor-Textur [Pe85]

Übertragen auf die Erzeugung komplexer Gebäudestrukturen zeigt sich diese Ersparnis ebenfalls zunächst im reduzierten Speicherbedarf, da die Gebäude nicht als fertige Modelle vorliegen müssen, sondern durch Algorithmen erzeugt werden. Dadurch steigt auf der anderen Seite der Berechnungsaufwand, da das Laden eines Gebäudemodells oder einer Textur weniger Rechenzeit erfordert, als deren Generierung. Am Beispiel der Textur zeigt sich ein weiterer Vorteil der prozeduralen Berechnung, da die Auflösung, mit der die Textur erzeugt wird, beliebig groß sein kann. Sofern die Rechenleistung des Computers ausreicht, ist es möglich, die Textur in beliebigen Größen während der Laufzeit einer Renderengine zu berechnen und beispielsweise an die Entfernung des Nutzers vom Objekt anzupassen. Dadurch ist das System nicht mehr länger auf die Auflösung festgelegt, die bei der Erstellung der Textur verwendet wurde.

Die dynamische Änderung der Auflösung einer Textur ist eine Variante eines Parameters, der in einem prozeduralen System verwendet werden kann, um die Ergebnisse des Berechnungsvorgangs zu beeinflussen. Die Möglichkeit, solche Parameter in Prozeduren zu integrieren, ist ein allgemeiner Vorteil solcher Systeme. Parameter bieten dem Nutzer eine Kontrolle auf einer höheren Abstraktionsebene. Dies sei am Beispiel eines einfachen Gebäudes erläutert, für das der Nutzer Stockwerke modellieren möchte. Tut er dies in einer Modellierungssoftware wie dem vorab vorgestellten SketchUp, so agiert er auf einer sehr niedrigen Abstraktionsebene, da er die Unterteilung des Gebäudes selbst vornehmen muss. So würde er wahrscheinlich das extrudierte Grundmodell durch eine Menge von Ebenen unterteilen, die parallel zum Gebäudegrundriss verlaufen und das Gebäude in mehrere Stockwerke zerlegen.

Auf der untersten Ebene kann ein prozedurales System ähnlich vorgehen, das hier vorgestellte System arbeitet ebenfalls mit Grundrissextrusionen, diese Ebene wird aber vor dem Nutzer versteckt. Dieser legt Werte für bestimmte Parameter fest und das System führt die Unterteilung durch. Eine Stockwerksunterteilung könnte beispielsweise auf der Angabe der Parameter `Gebäudehöhe` und `Stockwerksanzahl` basieren. Daraus kann das System anschließend automatisch die erforderlichen Unterteilungen vornehmen.

Kombiniert man die Parametrisierbarkeit mit stochastischen Elementen, so entstehen Systeme, die in der Lage sind, vielfältige Formen und Strukturen basierend auf vorhandenen Prozeduren zu erzeugen. Die Mächtigkeit solcher Ansätze wurde bereits ausführlich im Kontext der Ersetzungssysteme thematisiert. Auch die von Lindenmayer und Prusinkiewicz vorgestellten L-Systeme [PL96] in ihren verschiedenen Ausprägungen gehören zur Gruppe der prozeduralen Ansätze. Die Systeme, die die Autoren mit Hilfe dieser theoretischen Konzepte konstruiert haben und die Vielfalt der Pflanzen, die diese erzeugen können, geben einen Eindruck von den Möglichkeiten, die prozedurale Ansätze bieten.

Ein weiterer großer Einsatzbereich prozeduraler Ansätze in der Computergrafik sind *Partikelsysteme*. Solche Systeme werden im Bereich der Computeranimation verwendet, um Phänomene wie Wolken oder Feuer zu simulieren. Grundidee ist die Modellierung von Objekten durch eine Vielzahl sehr einfacher geometrischer Primitive mit einer sehr kleinen räumlichen Ausdehnung. Diese Primitive besitzen eine Reihe fester Eigenschaften, die während der Partikelanimation durch das ausführende Animationsskript verändert werden. Vorreiter auf diesem Gebiet war William T. Reeves, der in seiner Arbeit [Re83] ein Verfahren vorstellte, mittels dessen er „unscharfe“ (engl. *fuzzy*) Objekte darstellen konnte.

Dies war erforderlich, da sich die bis dahin verwendeten Oberflächendarstellungen aus Polygonnetzen nicht eigneten, um irreguläre, komplexe Oberflächen darzustellen. Reeves Ansatz verwendet eine Vielzahl geometrischer Grundelemente mit einer sehr kleinen räumlichen Ausdehnung, die als *Partikel* bezeichnet werden. Deren Größe ist meist so gering, dass viele dieser Elemente beim Rendering auf das gleiche Pixel projizieren. Dies ermöglicht im Gegensatz zu polygonnetzbasierten Modellen die Gestaltung flexibler Oberflächen, deren Form sich stetig ändert. Die Formänderung wird durch Partikelskripte gesteuert, die ebenfalls Teil des Systems sind. Hierbei existiert nicht für jedes Partikel ein eigenes Skript, typischerweise verwendet man ein allgemeines Skript mit stochastischen Elementen, über das die Partikelparameter in jedem Frame angepasst werden. Typische Partikelparameter sind Farbe, Position und Lebensdauer, beliebig viele andere Parameter können zusätzlich für ein solches System definiert werden. Reeves beschreibt in seiner Arbeit fünf Schritte [Re83], die in jedem Frame des Renderings eines solchen Systems durchgeführt werden müssen.

1. Erzeugung neuer Partikel und Integration in das System
2. Initialisierung sämtlicher Parameter für die neu erzeugten Partikel
3. Entfernen aller Partikel, die ihre Lebensdauer überschritten haben
4. Animation sämtlicher Partikel durch das übergeordnete Partikelskript
5. Rendering der aktuellen Partikel zur Erzeugung eines neuen Bildes

In einer späteren Arbeit [RB85] erweiterten Reeves und Blau das Konzept der Partikelsysteme um volumenfüllende Objekte zu modellieren, die sich durch eine konstante Form auszeichnen. Als Beispiel für die Anwendung dieser Technik entwickelten sie ein System, mittels dessen sie im Wind wogende Grashalme darstellen konnten.

Prozedurale Ansätze stellen auch für die Erzeugung von Gebäudemodellen einen guten Ansatzpunkt dar, mittels dessen sich die vorab diskutierten Probleme in Bezug auf die Erzeugung von Variationen lösen lassen. Während man beim Einsatz von Modellierungswerkzeugen nur mit großem Aufwand in der Lage ist, Varianten eines Gebäudetyps zu erstellen, so ist dies in prozeduralen Systemen durch die Verwendung von stochastischen Parametern leicht umsetzbar, wobei dem Nutzer meist Parameter zur Steuerung des stochastischen Prozesses zur Verfügung gestellt werden.

Die größere Vielfalt durch stochastische Berechnungskomponenten hat allerdings aus Sicht des Nutzers einen Nachteil. Speziell bei Systemen, bei denen die Algorithmen in einer

Programmiersprache implementiert werden, ist es für den Nutzer schwer, die einzelnen Berechnungsschritte nachzuvollziehen. Dies erschwert die Vorhersage des Berechnungsergebnisses und lässt das System wie eine *Black-Box* erscheinen, deren Innenleben unbekannt ist. An dieser Stelle zeigt sich der Gegensatz zwischen Modellierungswerkzeugen auf der einen und prozeduralen Systemen auf der anderen Seite am deutlichsten. Während der Nutzer bei ersteren über eine Vielzahl von Freiheitsgraden in Bezug auf den Konstruktionsprozess verfügt, ist dies bei letzteren durch die Abstraktion des Prozesses nicht mehr der Fall. Offensichtlich reduziert sich mit der Anzahl der Freiheitsgrade auch der Aufwand, den der Nutzer für die Gebäudekonstruktion aufbringen muss, dafür besitzt er weniger Möglichkeiten, in den Konstruktionsprozess einzugreifen. Ziel eines prozeduralen Systems muss es demnach sein, einen Mittelweg zu finden, bei dem der Nutzer flexibel wählen kann, wie viel Freiheit benötigt wird, um die Gebäude und ihren Konstruktionsprozess festzulegen. Das hier vorliegende System bietet dem Nutzer unterschiedliche Modi, die sich unter anderem in den verfügbaren Freiheitsgraden unterscheiden. Dabei hängen die zur Verfügung stehenden Parameter auch vom Typ des zu erzeugenden Gebäudes ab, so benötigt ein griechischer Tempel naturgemäß andere Parameter als ein modernes Hochhaus. Trotzdem existieren auch zwischen solchen stark unterschiedlichen Gebäudetypen gemeinsame Parameter, beispielsweise die Gebäudehöhe.

5.3.1 **Prozedurale vs. regelbasierte Systeme**

Im Kontext prozeduraler Systeme soll eine Unterscheidung eingeführt werden, die unterschiedliche Arten solcher Systeme charakterisiert. Dies ist für die Abgrenzung des vorliegenden Systems hilfreich und erleichtert die Diskussion der Vor- und Nachteile der jeweiligen Ansätze. Oliver Deussen unterscheidet in seinem Buch [De03] prozedurale von regelbasierten Systemen zur Erzeugung von Pflanzen.

Prozedurale Systeme sind in seiner Einteilung solche Ansätze, bei denen parametrisierte Algorithmen eingesetzt werden, um beispielsweise einen bestimmten Pflanzentyp zu erzeugen. Spezifische Ausprägungen der jeweiligen Spezies werden durch Parametermodifikationen erzeugt. Demgegenüber basieren regelbasierte Verfahren auf der Anwendung eines Regelsystems. In diese Klasse fallen die vorab erörterten Ersetzungssysteme. Diesen ist gemein, dass sie eine Menge von nutzerdefinierten Regeln auf ein Startwort anwenden und so lange Ersetzungen innerhalb des zu verarbeitenden Wortes vornehmen, bis ein definierter Endzustand erreicht wird oder keine weitere

Ersetzung mehr durchgeführt werden kann. Nachfolgend werden die Definitionen von Deussen verwendet, um zwischen den verschiedenen Arten prozeduraler Ansätze zu unterscheiden.

Beide Ansätze besitzen Vor- und Nachteile. Prozedurale Systeme implementieren die Algorithmen zur Objekterzeugung meist in einer höheren Programmiersprache, das vorliegende System ist beispielsweise in Java programmiert. Im Gegensatz dazu folgen Regelsysteme einem festem Formalismus und der Logik der sukzessiven Ersetzung von Wortteilen durch neue Komponenten. Programmiersprachen besitzen in diesem Zusammenhang eine größere Mächtigkeit, da sie dem Entwickler mehr Freiheiten für die Umsetzung beliebiger Verfahren und Berechnungen bieten und ihn dabei nicht in den Formalismus eines Textersetzungssystems zwängen. Die Implementation eines Verfahrens erfordert aber auch ein deutlich größeres Vorwissen, der Nutzer muss die Programmiersprache beherrschen und die zu implementierenden Algorithmen umsetzen können. Hier sind Regelsysteme deutlich einfacher zu bedienen, allerdings bei komplexen Systemen ebenfalls alles andere als trivial.

Ähnlich wie Programmiersprachen erfordern auch Regelsysteme vom Entwickler ein gutes Verständnis der zugrunde liegenden Strukturen und die Fähigkeit, sich das Ergebnis der Ausführung des Algorithmus auf der einen beziehungsweise die Anwendung des Regelsystems auf der anderen Seite vorzustellen, um die jeweiligen Ziele zu erreichen. Bezogen auf den Kontext der Gebäudeerzeugung stellen beide Ansätze vergleichsweise hohe Anforderungen an den Nutzer, wobei das Erlernen einer Programmiersprache als deutlich schwieriger anzusehen ist. Der Entwickler eines solchen Systems, sei es nun regelbasiert oder prozedural, muss in der Lage sein, die einzelnen Schritte der Gebäudekonstruktion entweder durch Algorithmen oder Regeln auszudrücken. Die Umsetzung ist dabei auf einem sehr niedrigen Abstraktionsniveau angesiedelt, in beiden Fällen erfordert beispielsweise die Unterteilung eines Gebäudes in einzelne Stockwerke geometrische Operationen. Dabei ist festzuhalten, dass auch regelbasierte Systeme in den Ersetzungsprozessen ein Abstraktionsniveau verwenden, das nicht mehr auf der untersten Geometrieebene agiert. So existieren beispielsweise im Shape Grammar der an späterer Stelle vorgestellten CityEngine Produktionen, die komplexe Unterteilungsoperationen vornehmen, deren Implementation selber aber vor demjenigen versteckt wird, der die Regelsysteme definiert. Regelsysteme zur Gebäudeerzeugung sind somit auf einer Ebene angesiedelt, die oberhalb der rein prozeduralen Systeme angesiedelt ist. Die Anwendung der

Regeln auf geometrische Objekte erfolgt im Hintergrund aber auch durch den Aufruf implementierter Routinen.

In prozeduralen und regelbasierten Systemen modelliert der Nutzer somit nicht länger durch die direkte Anwendung geometrischer Operationen auf geometrische Primitive, wie dies bei Softwareanwendungen wie SketchUp der Fall ist. Vielmehr beschreibt er den Modellierungsprozess durch eine Menge von Regeln beziehungsweise Prozeduren. Dies erhöht sowohl die Wiederverwendbarkeit als auch die Modifizierbarkeit der daraus erzeugten Gebäude. Im Gegensatz zu manuell erstellten Gebäudemodellen passt man die Bauvorschrift und nicht das Gebäude selbst an. Eine Änderung des Bauplans wirkt sich direkt auf die Gebäudestrukturen aus, ohne dass es erforderlich ist, das vorab berechnete Gebäude auf der Ebene seiner Oberflächendarstellung zu modifizieren. Dies erlaubt die sukzessive Modifikation des zugrundeliegenden Systems, bis die konstruierten Gebäude den eigenen Vorstellungen entsprechen. Eine solche iterative Adaption ist auf der Ebene der Konstruktionsvorschriften deutlich schneller durchführbar als auf der des erstellten Objekts. Dies gilt umso mehr, je komplexer dieses ist, da mit der zunehmenden Anzahl primitiver Oberflächenelemente in der Objektrepräsentation auch die Modifikation einzelner Elemente aufwendiger wird. Wiederum gilt, dass Anpassungen am Regelsystem für den Entwickler tendenziell einfacher sind als im Programmcode prozeduraler Systeme. So ist davon auszugehen, dass regelbasierte Anwendungen iterative Modellierungsansätze besser unterstützen als prozedurale.

Allerdings sollte hier zunächst zwischen Anwender und Entwickler unterschieden werden. Dies drückt sich auch in der Konzeption des vorliegenden Systems aus. Allgemein zeichnen sich prozedurale und regelbasierte Systeme gegenüber den vorab vorgestellten Ansätzen dadurch aus, dass durch die Abstraktion, die sie vornehmen, der Übergang zwischen Anwender und Entwickler zunehmend verschwimmt. Während der Anwender bei Modellierungssystemen das Modell direkt durch die bereitgestellten Methoden „entwickelt“, verschiebt sich seine Aufgabe bei regelbasierten Systemen auf die Entwicklung eines Regelsystems, das die konkrete Umsetzung seines Gebäudes für ihn vornimmt. In prozeduralen Systemen tritt der Anwender dann bereits in der Rolle eines Softwareentwicklers auf, der Bauvorschriften in Form von Quelltexten implementiert.

Dass ein System, das vom Nutzer verlangt, eine Programmiersprache zu beherrschen, um Anpassungen an Gebäuden vorzunehmen, nicht als intuitiv zu bezeichnen ist, ist offensichtlich. Aus diesem Grund wird innerhalb des vorliegenden Systems versucht, dieses

sowohl für Anwender als auch für Entwickler nutzbar zu machen. Entwickler mit Programmierkenntnissen finden eine klar strukturierte Klassenhierarchie, in die sie sowohl Klassen für die Erzeugung von Gebäudekomponenten als auch ganzer Gebäudetypen integrieren und durch das System erzeugen lassen können. Für Anwender stehen dagegen vorgefertigte Gebäudeklassen zur Verfügung, die über Konfigurationsdateien den eigenen Vorstellungen angepasst werden können. Beispielsweise existiert eine Klasse, die dazu dient, Instanzen eines Gebäudetyps zu erstellen, der von griechischen Tempelbauten inspiriert ist. Bei der Erzeugung solcher Tempel besitzt der Nutzer nur wenige Freiheitsgrade, da die Struktur der Gebäude stark festgelegt ist. So sind die Grundriss- und Innenraumstruktur und weitere strukturelle Parameter durch den Gebäudetyp festgelegt. Dies reduziert die Anzahl der Parameter, für die der Nutzer konkrete Werte festlegen muss. Tut er dies nicht, greift das System auf Standardwerte zurück, die für einen Teil der Eigenschaften Wertebereiche definieren, innerhalb derer zufallsbasiert Werte ausgewählt werden.

Um dem Nutzer weiterhin die Definition neuer Gebäudetypen zu ermöglichen, die nicht als vorgefertigte Klassen zur Verfügung stehen, existiert außerdem ein Gebäudetyp mit einer hohen Anzahl an Freiheitsgraden. Diese Klasse bietet dem Nutzer die größtmöglichen Freiheiten bei der Gebäudegestaltung, verlangt dadurch allerdings auch mehr Parameterfestlegungen. Die Grundidee hinter dieser Architektur ist ein zweigleisiger Ansatz für die Definition von Gebäuden, der darauf ausgelegt ist, dass sukzessive weitere Komponenten- und Gebäudeklassen in das System integriert werden. Bei der Definition neuer Gebäudetypen basierend auf reinen Konfigurationsdateien kann man beliebige Instanzen sowohl der Komponenten als auch der Gebäudeklassen durch das Reflection-Sprachfeature von Java erzeugen lassen. Die Konfiguration ähnelt somit in Ansätzen den Möglichkeiten von Scriptsprachen wie *Lua* [Ro96], die es mit vergleichsweise einfachen Sprachmitteln ermöglichen, komplexe Abläufe in Systemen zu steuern, die typischerweise in Hochsprachen implementiert sind.

Für eine klarere Abgrenzung regelbasierter von prozeduralen Systemen werden in den nächsten Abschnitten Vertreter beider Seiten ausführlich diskutiert. Zunächst wird darum das aktuell umfangreichste System für die Generierung von Stadtmodellen, die CityEngine, vorgestellt, um daran exemplarisch zu veranschaulichen, wie regelbasierte Systeme an welchen Punkten der automatisierten 3D-Stadtentwicklung eingesetzt werden können.

5.3.2 **CityEngine**

Die CityEngine ist ein kommerzielles Produkt zur prozeduralen Generierung sowohl einzelner Gebäude als auch ganzer Städte. Sie basiert auf verschiedenen Arbeiten von Pascal Müller, der sich sowohl während seines Studiums als auch im Laufe seiner anschließenden Dissertation intensiv mit dem Thema der prozeduralen Stadterzeugung befasste. Aus diesen wissenschaftlichen Arbeiten entstand die Firma *Procedural Inc*., die sich mit der Weiterentwicklung und dem Vertrieb der Software befasste. Procedural Inc. wurde 2011 von *Esri* übernommen [Es11]. Ziel dieser Übernahme ist laut Esri die Integration der CityEngine in das GI-System ArcGIS, das eines der größten kommerziellen GI-Systeme am Markt darstellt. Dadurch soll es den Nutzern ermöglicht werden, Stadtmodelle automatisch anhand bereits vorhandener GIS-Daten zu generieren. Aufgrund der großen Bedeutung der CityEngine und der hohen Relevanz der Arbeiten, auf denen sie basiert, soll hier nun die Entstehungsgeschichte des Systems und der zentralen Technologien näher erläutert werden.

5.3.2.1 **Erzeugung des Straßennetzes**

Die Ursprünge der CityEngine liegen in einer Semesterarbeit, die von Pascal Müller 1999 an der ETH Zürich verfasst wurde [Mü99]. Dort wurden zwei Verfahren vorgestellt, die zum Kern der CityEngine wurden.

Dies ist zunächst ein Verfahren zur automatischen Generierung von Straßennetzen basierend auf einer Erweiterung der vorab vorgestellten L-Systeme. Hierbei handelt es sich um ein *sequentiell selbstsensitives L-System*. Dieses System unterscheidet sich darin von anderen L-Systemen, dass es Kollisionen von Verzweigungen im Laufe der Verarbeitung zulässt und auf diese reagieren kann. Dies ist bei Straßennetzen relevant, da diese Zyklen bilden können. Bei der Modellierung des Pflanzenwachstums wachsen dagegen aufgespaltete Äste immer getrennt voneinander weiter, das entstehende Konstrukt entspricht einer Baumstruktur, während das L-System von Müller zyklische Graphstrukturen erzeugen kann. In Müllers Arbeit reagieren die einzelnen Straßensegmente, die während der Berechnung erzeugt werden, sowohl auf Kollisionen mit bereits vorhandenen Segmenten als auch auf Kollisionen mit Wasserflächen.

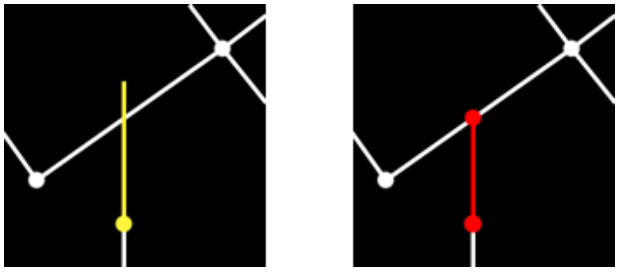

Abbildung 16: Kollisionsbehandlung im selbstsensitiven L-System [Mü99]

Abbildung 16 zeigt ein Beispiel für die Kollision des gelben Straßensegments mit dem bereits vorhandenen Segment. Das L-System erkennt diese Kollision und reagiert, indem die Länge des vorab behandelten Straßensegments neu berechnet und derart modifiziert wird, dass das gelbe auf das rote Segment reduziert wird. Dadurch entsteht eine Kreuzung innerhalb des Straßenverlaufs. Schneidet ein Straßensegment mehrere Segmente, so wird die Länge auf die Distanz bis zum ersten geschnittenen Segment reduziert. Nach Auftreten einer solchen Kollision kann festgelegt werden, ob das Wachstum für das Segment beendet oder fortgesetzt werden soll.

Um mit Hilfe eines solchen Systems in der Lage zu sein, realistische Straßenzüge zu erzeugen, hat Müller sein System bezüglich der Kollisionsbestimmung und –bearbeitung zusätzlich um das Konzept des *Nearest Neighbours* erweitert. Dieses Konzept soll die Erzeugung neuer Kreuzungen verhindern, sofern die Distanz zur nächsten Kreuzung zu gering ist. Die Kernidee dabei ist, dass für wachsende Straßensegmente geprüft wird, ob sich in einem festgelegten Radius um ihren Endpunkt Kreuzungspunkte befinden. Sofern dies der Fall ist, wird der Kreuzungspunkt zum Endpunkt des wachsenden Endsegments. Abbildung 17 illustriert diese Berechnungsschritte.

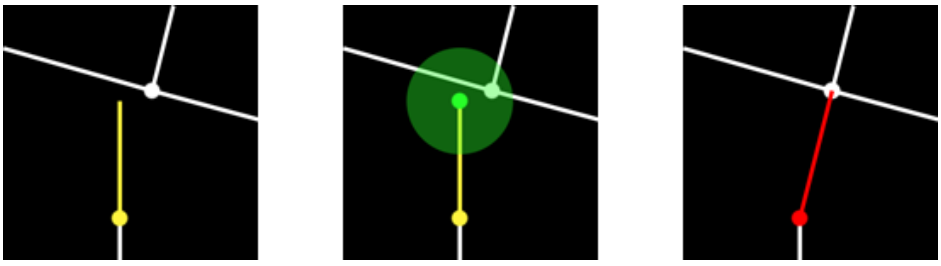

Abbildung 17: Nearest Neighbour-Verarbeitung [Mü99]

Eine letzte zentrale Fähigkeit des selbstsensitiven L-Systems von Müller ist die Reaktion des Wachstums auf Wasser. Dabei gibt es zwei mögliche Reaktionen auf eine Wasserfläche, innerhalb derer der Endpunkt eines wachsenden Straßensegments liegt. Das einfachere

Vorgehen kürzt das Straßensegment schrittweise so lange, bis der Endpunkt des Segments entweder auf festem Boden liegt oder eine Untergrenze für die Segmentlänge unterschritten wird. Tritt der zweite Fall ein, so ändert das System den Winkel des ursprünglichen Straßensegments innerhalb eines vordefinierten Intervalls und tastet dadurch die Umgebung nach trockenen Flächen ab. Wird über diesen Abtastvorgang ein gültiger Endpunkt für das Segment gefunden, so wird es gebaut, ansonsten verworfen.

Abbildung 18 zeigt die Reaktion eines wachsenden Segments auf die Kollision mit einer Wasserfläche innerhalb der Umgebung. Zunächst wird in der linken Abbildung versucht, das Segment zu kürzen, schlägt dies fehl, wird die Umgebung im rechten Segment nach trockenen Bereichen abgesucht.

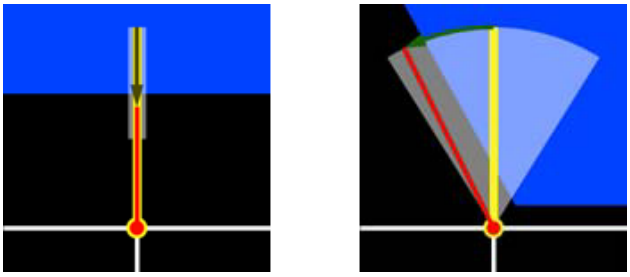

Abbildung 18: Kollision mit einer Wasserfläche [Mü99]

Diese Komponenten verwendet Müller innerhalb seines Systems, um eine möglichst realistische Nachbildung des Straßennetzes von New York zu realisieren. Hierfür unterscheidet er zwischen siedlungsorientierten Straßen und Highways. Highways verbinden Ballungszentren innerhalb der Stadt und unterteilen diese in Bezirke und Quartiere. Von diesen Hauptverkehrsadern verzweigen dann die siedlungsorientierten Straßenzüge, die aufgrund ihres Verlaufs die jeweiligen Bezirke in Häuserblocks untergliedern. Innerhalb des selbstsensitiven L-Systems unterscheiden sich die Highways bezüglich ihres Wachstumsverhaltens von den siedlungsorientierten Straßen unter anderem in Bezug auf die Länge der einzelnen Wachstumssegmente und in der Art des Verhaltens beim Auftreffen auf Wasser. Bei Highways wird vor dem Test auf Kollisionen der Endknoten mit Wasserflächen ein weiterer Test vorgeschaltet, der dafür sorgen soll, dass Highways an Küstenlinien entlang verlaufen. Zunächst wird dafür die Länge des Ausgangssegments verdreifacht. Liegt der Endknoten dieses Segments innerhalb einer Wasserfläche, so variiert man den Ausgangswinkel so lange, bis das derart verlängerte Segment nicht mehr im Wasser endet. Den so bestimmten Winkel halbiert Müller und verwendet ihn, um das tatsächliche Segment in seiner Ausrichtung zu modifizieren. Durch dieses Vorgehen vermeidet das Verfahren

Highways, die plötzlich an Wasserflächen enden oder stark abknicken. Stattdessen nähern sich die Segmente der Küstenlinie an und laufen an dieser entlang. Sollte das verlängerte Segment nicht auf eine Wasserfläche getroffen sein, so bedeutet dies zunächst, dass ohne Anpassung der Richtung in den nächsten Iterationsschritten keine Wasserfläche getroffen wird. In diesem Fall sucht der Algorithmus die Seiten des Segments nach Wasserflächen ab. Werden solche Flächen gefunden, wird das Segment in Richtung dieser Wasserflächen gelenkt, wodurch Highways automatisch in Richtung von Küstenstreifen gezogen werden.

Für beide Straßentypen stellt Müller eine Menge von Produktionen auf, die zur automatischen Erzeugung eines Straßennetzes eingesetzt werden können. Dabei stellt ein Highwaysegment das Ausgangswort dar, auf das die Produktionen des Regelsystems angewendet werden. Die siedlungsorientierten Straßen werden als Abzweigungen von den Highwaysegmenten durch die Produktionen des Highwayregelsystems erzeugt. Dadurch erfolgt der Eintritt in den Regelsatz, der diesen Straßentyp erzeugt.

Abbildung 19: Mögliches Resultat des L-Systems [Mü99]

Abbildung 19 zeigt ein mögliches Ergebnis der Anwendung des entwickelten L-Systems auf ein Highwaystartsegment. Dieses Startsegment ist in der Abbildung durch einen roten Kreis markiert. Da das von Müller genutzte L-System stochastische Komponenten enthält, Produktionen also zufallsbasiert aufgrund ihnen zugewiesener Wahrscheinlichkeiten ausgewählt werden, ist die Anwendung des Systems auf ein Startwort nicht deterministisch. Die Eingabe des gleichen Axioms führt zu unterschiedlichen Ergebnissen.

In seiner nachfolgenden Arbeit [Mü01] erweiterte Müller das System zur Erzeugung des Straßennetzwerks, um dem Nutzer einen stärkeren Einfluss auf die Ergebnisse der Berechnungen zu geben. Zu diesem Zweck definiert er eine Folge von Funktionen, die in Kombination mit dem L-System eingesetzt werden, um den tatsächlichen Verlauf einer Straße zu bestimmen.

Abbildung 20 zeigt die verschiedenen Schritte, die für die Berechnung einer Straße durchgeführt werden, um die erforderlichen Parameter zu bestimmen. Mit steigender Mächtigkeit der Produktionen eines L-Systems steigt zwangsläufig auch dessen Komplexität, was mit der wachsenden Anzahl an Produktionen zusammenhängt. Aus diesem Grund verwendet Müller sein L-System als Template, während die Bestimmung der erforderlichen Parameter von externen Funktionen übernommen wird. Das L-System selber enthält dann „nur noch die für die Wachstumssteuerung nötigen Komponenten“ [Mü01].

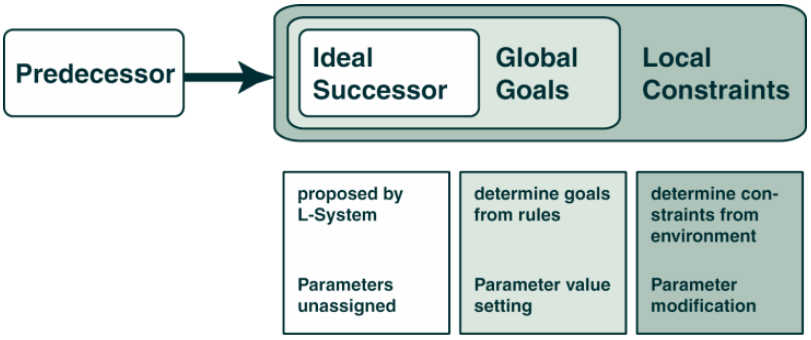

Abbildung 20: Berechnung der Parameter zum Bau einer Straße basierend auf hierarchischen Funktionen [Mü01]

Der Ablauf des Straßenbaus besteht aus drei Schritten, die in Abbildung 20 durch die drei unteren Rechtecke in ihrer Abfolge dargestellt werden. Zunächst werden durch den *Ideal Successor* neue Straßen initiiert. Dabei werden am Ende des Vorgängersegments drei neue Straßen erstellt, die jeweils durch ein einziges Modul repräsentiert werden. Jedes Modul enthält Parameter, die allerdings zu diesem Zeitpunkt noch keine konkreten Werte besitzen. Für jedes erzeugte Modul verwendet Müller nun die externe Funktion `globalGoals`, die die Wachstumsziele auf einer globalen Ebene repräsentiert. Die Verwendung dieser Funktion sorgt dafür, dass das Wachstum einem vorab festgelegten Muster folgt. Das Ergebnis der Berechnungen weist anschließend den Parametern des Moduls Werte zu. Abschließend prüft die externe Funktion `localGoals`, ob die im zweiten Schritt zugewiesenen Modulparameter innerhalb der lokalen Umgebung überhaupt umsetzbar sind und realisiert dadurch den

umgebungssensitiven Teil des Systems. Ist dies nicht der Fall, versucht die Funktion die Parameter anzupassen, ansonsten wird das Modul verworfen.

5.3.2.1.1 Die Umsetzung der globalen Wachstumsziele

Kernidee der Umsetzung der globalen Wachstumsziele ist die Steuerung des Straßenwachstums basierend auf vier globalen Regeln. Jede dieser Regeln definiert eine bestimmte Form bzw. ein bestimmtes Verhalten des Wachstums für ein Straßensegment. Für jede im System vorhandene Regel, konkret *Western-*, *Grid-*, *Radial-* und *Topographical-Rule* kann der Nutzer ein Kontrollbild in Form eines Graustufenbildes in das System eingeben. Dabei beschreibt jedes Kontrollbild Werte für die Parameter jeder Regel an einem bestimmten Ort des Eingabebereichs. Somit müssen alle Kontrollbilder die gleiche Ausdehnung besitzen, da sie über die Koordinaten des aktuell verarbeiteten Straßenabschnitts indiziert werden. Die Auswertung der globalen Regeln testet anhand der Kontrollbilder zunächst, welche Regeln für das aktuelle Modul aktiv sind und berechnet anschließend dessen Parameter. Die Western-Rule verwendet beispielsweise ein Kontrollbild, dass Populationsdichten innerhalb des Eingabebereichs darstellt und wird für die Richtungsbestimmung des Verlaufes von Highways eingesetzt. Diese verbinden Bereiche hoher Population miteinander. Um dies innerhalb des Systems umzusetzen, wird die Richtung eines Highway-Segments derart angepasst, dass sie auf Zentren hoher Dichte zustrebt. Weiterhin werden durch die Western-Rule siedlungsorientierte Straßen parametrisiert, diese berücksichtigen allerdings nicht die Populationsdichte, sondern entstehen durch Abzweigungen dieses Straßentyps von den Highways. Dies entspricht vereinfacht dem Vorgehen im ursprünglichen Entwurf von Müller.

Um dem Nutzer einen größeren Einfluss auf die Ausrichtung und Struktur des Schachbrettmusters zu geben, wird die Grid-Rule verwendet. Diese sorgt dafür, dass sich das Wachstum von siedlungsorientierten Straßen und Highways an geometrischen Regeln orientiert. Der Nutzer definiert dabei Position, Orientierung und Straßenlängen. Dadurch wachsen die Straßen vergleichbar zur Western-Rule, allerdings ohne Winkelabweichungen, wodurch das entstehende Muster noch regelmäßiger wird.

Die Radial-Rule erzeugt radiale bzw. konzentrische Straßenmuster, die in ihrem Verhalten der Western Rule ähneln, da auch hier die Straßen Schachbrettmuster bilden. Die Highways werden dagegen abhängig von ihrem Winkel zum Zentrum tangential oder radial abgelenkt.

Der Einsatz dieser Regel ist für die Erzeugung von Straßennetzen geeignet, wie man sie beispielsweise in europäischen Städten wie Paris findet.

Die letzte verwendete Regel ist die Topographical-Rule, welche den Höhenverlauf einer Region berücksichtigt. Zu diesem Zweck kodiert das zugehörige Kontrollbild Höhenwerte innerhalb des Bereiches, in dem das Straßennetz erzeugt wird. Die Kernidee dieser Regel ist die Tatsache, dass größere bzw. längere Straßen typischerweise den Höhenlinien der Region folgen, in der sie sich befinden, sie folgen also der geringsten Steigung an einem konkreten Punkt innerhalb der Topographie. Umgekehrt folgen kurze Straßen häufig der größten Steigung, um dadurch Höhen zu überbrücken, sofern diese Steigung nicht zu stark ist.

Als letzte Komponente der globalen Funktion verwendet Müller einen Ansatz, der die Ergebnisse unterschiedlicher Rules kombiniert, sofern diese aktiv sind. Eine Regel ist genau dann aktiv, wenn der indizierte Wert innerhalb ihres Kontrollbildes einen Wert > 0 besitzt (also nicht vollständig schwarz ist). Diese Kombination unterschiedlicher Regeln bezeichnet Müller als *Blending*. Im ersten Schritt entscheidet diese Komponente, welche Module, also Abzweigungen einer Vorgängerstraße überhaupt existieren. Lehnt eine Regel eine Abzweigung ab, wird der Kennwert über das aus dem Kontrollbild geladene Regelgewicht dekrementiert, ansonsten inkrementiert. Ist der Kennwert am Ende der Auswertung positiv, so wird die Abzweigung erzeugt.

Im zweiten Schritt wird für die existierenden Abzweigungen über die Auswertung der Regeln ermittelt, ob es sich um einen Highway oder eine Straße handelt. Abschließend werden dann die Parameter aller Regeln, die für die Abzweigung gestimmt haben, beispielsweise Länge, Winkel etc. linear kombiniert und als Parameter innerhalb des Moduls gespeichert, das die Abzweigung repräsentiert.

5.3.2.1.2 Die Umsetzung der Umgebungssensitivität

Nachdem durch das System die globalen Wachstumsziele berücksichtigt und in Form von Parametern in den einzelnen Verzweigungsmodulen gespeichert wurden, erfolgt der letzte Schritt der Straßennetzerzeugung, der in der Überprüfung der aktuellen Umgebung besteht. Dieser Berechnungsschritt ähnelt dem vorab bereits erläuterten Vorgehen zur Überprüfung des Straßenverlaufs auf Wasserflächen. Dazu gehört auch die Anpassung von Straßenverläufen an solche Flächen, beispielsweise um Highways an Küstenlinien entlang verlaufen zu lassen.

5.3.2.2 **Erzeugung der Grundstücke**

Nachdem ein Straßennetz mit dem vorgestellten Verfahren erzeugt wurde, besteht der nächste Schritt in der Identifikation von Grundstücken innerhalb dieses Netzes, die als Ausgangspunkt für die Konstruktion von Häusern eingesetzt werden können. Auch für diese Problematik schlägt Müller ein Verfahren vor. Dieses basiert auf der Repräsentation des Straßennetzes durch einen Graphen, bei dem die Straßen die Kanten und die Kreuzungen die Knoten bilden. Aus diesem Graphen leitet Müller mittels eines vergleichsweise einfachen Verfahrens Grundstücke ab.

Betrachtet man einen Häuserblock in einer amerikanischen Großstadt wie New York, so wird dieser durch eine Menge von Straßen definiert, die den Block an jeder Seite begrenzen. Aus dem Ausgangsgraphen kann man somit eine Menge von Blockpolygonen ermitteln, indem man alle Kantenzüge berechnet, deren Länge minimal ist. Jeder Kantenzug dieser Art bildet dann den Grundriss eines Häuserblocks. Der Algorithmus für diese Berechnung startet in jedem Knoten des Graphen eine rekursive Suche nach Zyklen. Überschreitet die Länge des aktuellen Zyklus einen gesetzten Maximalwert, so wird der Rekursionszweig abgebrochen. Wird ein Zyklus entdeckt, prüft man seine Länge. Ist er länger als ein bereits ermittelter Zyklus, wird er verworfen. Ist er dagegen Teil eines anderen aber längeren Zyklus, wird dieser verworfen. Als Ergebnis des Verfahrens erhält man eine Menge von Zyklen innerhalb des Graphen, die für die jeweiligen Ausgangsknoten eine minimale Länge besitzen.

Die Ermittlung der eigentlichen Block-Polygone aus diesen Zyklen erfordert die Subtraktion der Straßenbreiten von den Zyklen. Da es sich bei dem Straßennetz um einen ungewichteten Graphen handelt, kann man die Straßenbreite nicht aus diesem ableiten. Aus diesem Grund schlägt Müller ein einfaches Vorgehen vor, das zu guten Ergebnissen führt. Dazu werden die Zykluspolygone auf 75% ihrer ursprünglichen Ausdehnung geschrumpft, indem die Polygoneckpunkte auf den Schwerpunkt des Polygons zuwandern.

Nachdem auf diese Weise die Polygone zur Beschreibung der Häuserblöcke aus dem Straßennetzgraphen ermittelt wurden, erfolgt als letzter Schritt vor der Gebäudegenerierung die Festlegung ihrer Grundrisse. Zu diesem Zweck werden die entstehenden Block-Polygone rekursiv in kleinere Polygone unterteilt. Das Ziel dieser Unterteilungsberechnung besteht in der Erzeugung möglichst rechtwinkliger Gebäude. Das Verfahren versucht zunächst, die

längste vorhandene Kante zu unterteilen. Sind die beiden Nachbarkanten dieser Kante rechtwinklig, so erzeugt man eine Kante, die die Ausgangskante halbiert und senkrecht auf dieser steht. Konnte keine Kante gefunden werden, die diese Anforderungen erfüllt, prüft man verschiedene Abschwächungen der Vorgabe. Kann auch über diese Abschwächungen kein Kandidat für die Teilung ermittelt werden, so wählt das Verfahren einfach die längste vorhandene Kante aus und unterteilt diese. Das Vorgehen des Verfahrens ist in Abbildung 21 exemplarisch dargestellt. Da die zweitlängste Kante innerhalb des Ausgangspolygons über zwei Nachbarkanten verfügt, die senkrecht auf ihr stehen, wird sie ausgewählt und unterteilt. Anschließend startet die Rekursion und unterteilt die entstehenden Subpolygone entsprechend.

An diesem Punkt verfügt man sowohl über ein Straßennetz, das durch ein selbstsensitives L-System erzeugt wurde als auch über eine Menge von Gebäudegrundrissen. Im nächsten Schritt verwendet man die berechneten Gebäudegrundrisse und extrudiert diese in Richtung der Grundrissnormalen. Die Extrusionshöhe wird aus einer *Heightmap* ermittelt, die als Graustufen-Bild in das System geladen und anschließend über die Position des Gebäudes indiziert wird. Basierend auf dem Graustufenwert an der jeweiligen Position wird die Extrusionshöhe bestimmt. Das Ergebnis der Extrusion ist ein Masse-Modell, das in der ersten Version von Müllers Arbeit nicht weiter bearbeitet, sondern texturiert wurde, um Fassaden zu simulieren.

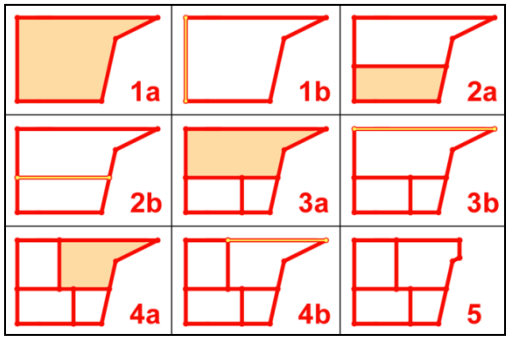

Abbildung 21: Polygonsubdivision [Mü01]

Neben diesem algorithmischen Ansatz, der auf der automatisierten Erzeugung von Straßennetzen und der daraus abgeleiteten Berechnung von Grundrissen basiert, bietet die CityEngine die Möglichkeit, geographische Daten aus anderen Systemen zu laden und für die Gebäudeerstellung zu verwenden. Dazu gehören beispielsweise GIS-Daten wie Parzellierungen, Grundrisse oder Straßennetze. Der Import solcher Daten aus GI-Systemen

erlaubt die einfache Rekonstruktion existierender Städte unter Ausnutzung vorhandener struktureller Daten.

Nachdem entweder unter Verwendung des L-System-basierten Ansatzes oder durch Laden von GIS-Daten Grundrisse bestimmt und Masse-Modelle durch Extrusion erzeugt wurden, ermöglicht die CityEngine die automatisierte Erstellung von Fassaden unter Verwendung von CGA-Shape, einer von Müller et. al entwickelten Formengrammatik. Deren Struktur und Entwicklungsgeschichte ist Thema des nächsten Abschnitts.

5.3.2.3 **Erzeugung komplexer Fassaden**

In der ersten Version der CityEngine war die Fassadenerstellung auf die Auswahl geeigneter Texturen und deren Mapping auf die erzeugte Geometrie beschränkt. Ein solcher Ansatz hat zwar den Vorteil, dass er mit relativ wenigen Polygonen auskommt und dadurch effizient renderbare Modelle erzeugt, allerdings eignen sich solche Ansätze nur dann, wenn der Betrachter ausreichend weit von diesen entfernt ist und somit nicht den Unterschied zwischen vorhandenen geometrischen und durch Texturen simulierten Details wahrnimmt. Aus diesem Grund wurde die CityEngine in späteren Arbeiten um eine Formengrammatik erweitert, mittels derer ein Regelsystem für die Fassadenerstellung definiert werden kann. Die als CGA-Shape bezeichnete Grammatik ist inspiriert durch die im Abschnitt „Shape Grammars“ beschriebene Shape-Grammatik nach Stiny et. al. [GJ71] und folgt dem dort definierten sequentiellen Formalismus. CGA-Shape basiert unter anderem auf der Arbeit von Wonka et al. [Wo03], der *Split*- und *Control*-Grammatiken entwickelte, mittels derer er in der Lage war, Fassaden regelbasiert zu unterteilen. Da diese Technologien einen wichtigen Bestandteil der von Müller et al. [Mü06] entwickelten CGA-Shape-Grammatik bilden, soll detaillierter darauf eingegangen werden. Kern von Wonkas Ansatz sind drei Komponenten. Hierbei handelt es sich um die Split-, die Control-Grammatik und ein *Attribut-Matching*-System.

Formal definiert Wonka eine Split-Grammatik als eine Grammatik über einem Vokabular B mit bestimmten Arten von Produktionsregeln. Das Vokabular besteht aus einer Menge von einfachen Basisformen wie Kuben, Zylindern oder Prismen. Ein Split zerlegt ein solches Basisobjekt in eine Menge von Objekten, die ebenfalls Bestandteile von B sind. So wird beispielsweise ein Kubus in eine Menge von $n \times m \times k$ Kuben zerlegt, die in einem Raster aus parametrisierten Split-Ebenen angeordnet sind. Die zur Verfügung stehenden Regeln

einer Split-Grammatik lassen sich in zwei unterschiedliche Typen aufteilen. Für Split-Regeln $a \rightarrow b$ ist a eine verbundene Untermenge des Vokabulars B. Die durch Anwendung der Regel entstehende Form b enthält die gleichen Elemente wie die Ausgangsform a bis auf ein Element aus a, auf das die Zerlegung angewendet wurde.

Als zweiter Typ existieren *Conversion*-Regeln, die eine Basisform in eine andere Basisform transformieren. Auch diese Regeln haben die Form $a \rightarrow b$, wobei a eine verbundene Untermenge des Vokabulars darstellt, die eine Basisform enthält. Diese Basisform wird durch die Conversion-Regel in eine andere Basisform transformiert, so dass das erzeugte b die gleichen Elemente wie a enthält bis auf die Ersetzung der Basisform durch eine andere Basisform.

Ein zentraler Unterschied zwischen Split- und Conversion-Regeln betrifft das Volumen, das die erzeugten Formen einnehmen. Bei Split-Regeln nimmt die Form, die durch die Regel erzeugt wurde, exakt den gleichen Raum ein wie die Ausgangsform. Bei Conversion-Regeln ist dies dagegen nicht zwingend der Fall. Ein Beispiel hierfür ist die Ersetzung eines Kubus durch ein dreiseitiges Prisma mit dem Ziel, ein Dach zu modellieren. Offensichtlich sind das Volumen der Ausgangsform und der erzeugten Form nicht identisch.

Abbildung 22 zeigt ein Beispiel einer einfachen Split-Grammatik. Die erste Regel zerteilt die Startform in vier gleich große Elemente, die dann durch Anwendung der weiteren Regeln in weitere Subelemente unterteilt werden.

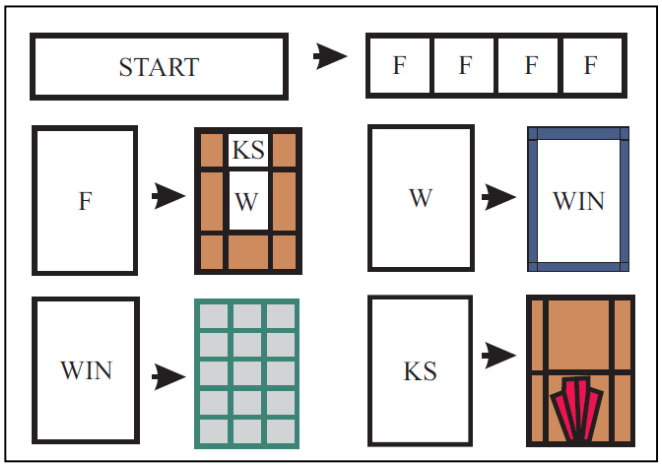

Abbildung 22: Beispiel einer einfachen Split-Grammatik [Wo03]

Die Anwendung von Split-Grammatik-Regeln unterteilt eine Basisform in eine Menge hierarchisch angeordneter, struktureller Elemente. Die derart erstellte Unterteilung besitzt

zunächst keinerlei Informationen darüber, wie die einzelnen Elemente dargestellt werden. So würde beispielsweise die wiederholte Anwendung von Split-Regeln auf einen Quader eine Menge von Unterquadern erzeugen, die selber wiederum Gegenstand weiterer Unterteilungen sein können. Die Größe und Position dieser Unterelemente wird dabei durch die Ausrichtung der parametrisierten Split-Ebenen definiert, die für die Unterteilung verwendet wurden [Wo03].

Die Erzeugung einer Unterstruktur ist somit nur ein erster Schritt für die Generierung von Fassaden, für die weiteren Schritte spielen Attribute eine zentrale Rolle, die Formen zugewiesen werden können und bei der Regelanwendung für die Regelselektion eingesetzt werden. Dabei werden solche Attribute innerhalb Wonkas System [Wo03] für unterschiedliche Zwecke eingesetzt. So können sie beispielsweise auf einer niedrigen Abstraktionsebene zur Festlegung von Materialfarben oder ähnlichem eingesetzt werden, auf höheren Ebenen aber auch spezifizieren, dass das Gebäude in einem bestimmten Stil erbaut werden soll.

Für die Festlegung von Attributwerten existieren drei verschiedene Wege. Die einfachste Variante ist die Zuweisung von Werten durch den Nutzer. Solche Attributwerte können anschließend bei jedem Zerlegungsschritt vom Elternelement in die Kindelemente kopiert werden.

Neben diesen beiden einfachen Varianten wird eine Control-Grammatik eingeführt, die für die Festlegung von Designzielen während der Regelanwendung eingesetzt werden kann. Wonka nennt als Beispiel unter anderem die unterschiedliche Gestaltung des Erdgeschosses eines Gebäudes im Vergleich zu den anderen Stockwerken oder die Verwendung unterschiedlicher Fenstertypen innerhalb des gleichen Stockwerks. Solche Designfestlegungen müssen dabei derart getroffen werden, dass sie mit architektonischen Richtlinien übereinstimmen, sonst kann die Regelanwendung leicht unrealistische Gebäude erzeugen.

Die nicht-terminalen Elemente der Control-Grammatik bestehen aus deskriptiven Symbolen, wie beispielsweise `WOOD`, terminale Symbole dagegen aus Befehlen zur Festlegung von Attributen für bestimmte Elemente der aktuellen Form. Diese Befehle haben die Form $< c, a, v >$. Hierbei definiert c den Scope, legt also fest, für welche Ergebniselemente einer Unterteilung die Attributfestlegung durchgeführt wird, a ist der Bezeichner des Attributs und v der Wert, der diesem zugewiesen wird.

Der Einsatz von Attributen ist dabei nicht nur auf die Festlegung bestimmter Eigenschaften der Unterteilungselemente beschränkt, sondern wird auch für die Auswahl von Regeln eingesetzt. Diese Regelauswahl ist ein kritischer Schritt während der Unterteilung, der mit zunehmender Größe der verfügbaren Regelbasis an Bedeutung gewinnt. Dabei müssen die Regeln derart gewählt werden, dass ausreichend Variationen der vorhandenen Gebäude erzeugt werden, dabei aber die Plausibilität der berechneten Strukturen garantiert ist. Zusätzlich müssen die nutzerdefinierten Designziele berücksichtigt werden, damit sich diese im fertigen Modell widerspiegeln. Um dies zu gewährleisten, verwendet Wonka einen Ansatz, bei dem sowohl Symbolen als auch Regeln der Grammatik Attribute zugewiesen werden. Die Regelauswahl basiert auf einem Vergleich der Attributwerte des Symbols mit den Attributwerten der zur Auswahl stehenden Regeln. Dabei werden die Attributwerte auf beiden Seiten als Intervall angegeben. Nur, wenn sich die Intervalle aller Attribute zwischen Symbol und Regel überlappen, kann die Regel ausgewählt werden. Darum stellt die Auswahl, welche Regeln diese Bedingungen erfüllen, einen ersten Selektionsschritt dar. Dabei wird gleichzeitig basierend auf den Intervallen und ihrer Ähnlichkeit eine Priorität der einzelnen Regeln bestimmt und zurückgereicht. Stehen nach diesem Schritt weiterhin mehrere gültige Regeln gleicher Priorität zur Auswahl, wird zufallsbasiert ausgewählt, welche Regel angewendet werden soll. Abbildung 23 zeigt exemplarisch die Auswahl einer Regel für das Symbol `WIN`. Zur Auswahl stehen die von 1 bis 6 durchnummerierten Regeln inklusive ihrer Attributintervalle. Dabei steht das Intervall $[-,+]$ für $[-\infty,+\infty]$. Die Regeln 4 bis 6 werden durch das Attributmatching abgelehnt, da sie bezüglich der rot umkreisten Attribute das Intervallkriterium nicht erfüllen.

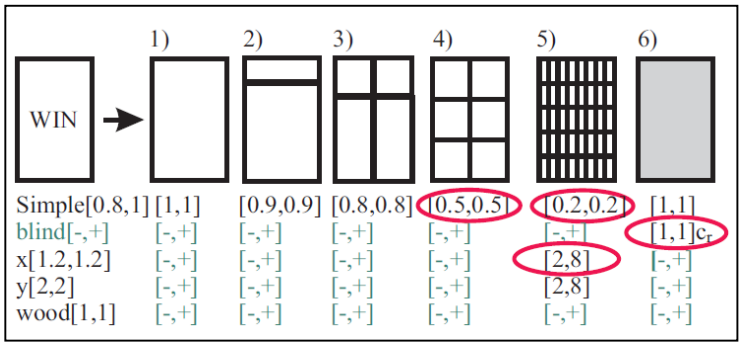

Abbildung 23: Regelauswahl durch Attributmatching [Wo03]

Abbildung 24 zeigt exemplarisch ein mittels der von Wonka et. al. entwickelten Grammatiken erstelltes Gebäude.

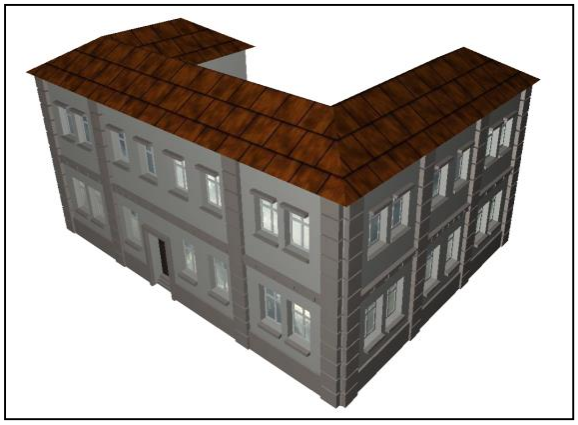

Abbildung 24: Mittels Split- und Control-Grammar erzeugtes Gebäude [Wo03]

Der vorab vorgestellte Ansatz wurde in einer späteren Arbeit [Mü06] um verschiedene Konzepte erweitert, um die jeweiligen Stärken der Ansätze von Parish und Müller sowie der Arbeit von Wonka et al. zu kombinieren. Dabei benennen die Autoren zunächst die folgenden Schwächen in den beiden Ausgangsansätzen.

Parish und Müller [Mü06] waren in der Lage, einfache Massemodelle zu erzeugen. Diese entstanden durch lineare Transformationen geometrischer Grundkörper. Details wurden nachfolgend mittels unterschiedlicher Shader hinzugefügt, spiegelten sich aber nicht in der eigentlichen Geometrie wider. Dagegen konnten Wonka et al. [Wo03] durch die Anwendung ihrer Split- und Control-Grammatiken komplexe Geometrien erzeugen, deren Anwendbarkeit allerdings auf vergleichsweise einfache Massemodelle beschränkt blieb. Diese Schwächen sollten durch die Entwicklung der CGA-Shape-Grammatik behoben werden. Diese Grammatik ist analog zu den Shape Grammatiken von Stiny [GJ71] und der Arbeit von Wonka et al. eine sequentielle Grammatik. Solche Ansätze eignen sich besser für die regelbasierte Erzeugung von Architektur als dies bei L-Systemen der Fall ist. Aufgrund der Parallelität biologischer Wachstumsprozesse sind diese das Mittel der Wahl, wenn es um die prozedurale Erzeugung von Pflanzen geht. Die Generierung von Gebäuden ist dagegen ein schrittweiser Prozess, bei dem beispielsweise die von Wonka et al. entwickelten Unterteilungsschritte nacheinander durchgeführt werden.

Jeder Erzeugungsschritt besteht in CGA-Shape aus drei verschiedenen Unterschritten ausgehend von einer Startkonfiguration. Eine Startkonfiguration ist definiert durch eine Menge von Ausgangsformen. Aus diesen Ausgangsformen wird zunächst ein aktives Objekt mit Symbol B ausgewählt. Anschließend wird aus der vorhandenen Regelmenge eine

Produktion bestimmt, die B als nicht-terminales Symbol enthält. Die Anwendung dieser Regel auf B erzeugt eine Menge neuer Formen $BNEW$. Nachdem $BNEW$ erzeugt wurde, wird die Ausgangsform B als inaktiv gekennzeichnet und $BNEW$ zur Konfiguration hinzugefügt. Anschließend beginnt der Prozess erneut mit der Auswahl einer aktiven Form aus der Konfiguration und terminiert erst, sobald die Konfiguration ausschließlich nicht-terminale Objekte enthält. Die Regelauswahl erfolgt dabei wie bei Wonka et al. basierend auf Prioritätszuordnungen zu den Regeln in der Produktionsmenge.

Produktionen werden in der Form $id: predecessor: cond \rightsquigarrow successor: prob$ angegeben. Diese Notation ähnelt der im Bereich der Lindenmayer-Systeme vorgestellten. Dabei ist $predecessor$ das Ausgangssymbol, das bei Erfüllung der Bedingung $cond$ mit der Wahrscheinlichkeit $prob$ durch das Symbol $successor$ ersetzt wird. Als Beispiel geben die Autoren die folgende Produktion an: $fac(h): h > 9 \rightsquigarrow floor\left(\frac{h}{3}\right) floor\left(\frac{h}{3}\right) floor\left(\frac{h}{3}\right)$, die das Ausgangssymbol fac durch drei Formen mit dem Symbol $floor$ ersetzt, sofern der Parameter h größer als 9 ist.

Für die Modifikation vorhandener Formen stellt CGA-Shape Transformationen, Rotationen sowie Skalierungen zur Verfügung, bei denen das Konzept des *Scopes* eine zentrale Rolle spielt. Die Autoren definieren einen solchen Scope als *objektausgerichtete Boundingbox* mit Position *P*, die das Zentrum des Boundingboxkoordinatensystems kennzeichnet, drei orthogonalen Koordinatenachsen sowie den Ausdehnungen der Box in Bezug auf diese Achsen. Abbildung 25 zeigt ein Beispiel für den Scope eines einfachen Körpers.

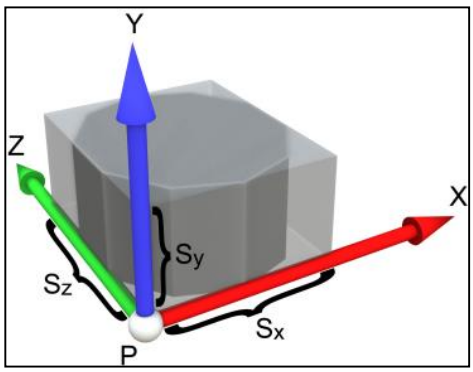

Abbildung 25: Scope eines einfachen Körpers [Mü06]

Die genannten Transformationen beziehen sich immer auf Parameter dieses Scopes, so wird beispielsweise bei einer Translation der Translationsvektor auf die Position P des Scopes aufaddiert und dadurch die neue Position des Objekts im Koordinatenraum festgelegt. Auch

die Split-Regeln werden in Bezug auf solche Scopes definiert. So unterteilt ein Basissplit den Scope eines Objekts in Richtung einer seiner Koordinatenachsen und erzeugt eine Menge von Unterobjekten. Als Erweiterung der Split-Regeln führen die Autoren sogenannte *Repeat*-Regeln ein. Auch diese Regeln unterteilen den Scope, allerdings ist die Anzahl der Unterteilungen nicht a priori gegeben. Vielmehr erzeugt die Anwendung einer solchen Regel auf ein Objekt eine Menge von Unterteilungen einer bestimmten Größe in Richtung einer bestimmten Scopeachse. Je größer das unterteilte Objekt ist, desto mehr Unterteilungen werden durch die Regelanwendung durchgeführt.

Bis zu diesem Punkt wurden alle Unterteilungen sowohl in der Ausgangsgrammatik von Wonka et al. als auch in der darauf basierenden CGA-Shape-Grammatik auf dreidimensionale Objekte angewendet und erzeugten darum auch als Ergebnis neue dreidimensionale Objekte. Hier enthält CGA-Shape mittels des *Component*-Split eine Erweiterung der bisherigen Unterteilungsverfahren, indem Komponenten des dreidimensionalen Körpers, also beispielsweise Faces oder Kanten, Gegenstand einer Unterteilung werden können. Dabei ist es sowohl möglich, sämtliche vorhandenen Komponenten eines Objekts zu unterteilen, als auch nur eine Untermenge auszuwählen und die Berechnungsschritte auf deren Elemente zu beschränken. Dadurch können beispielsweise gezielt sämtliche Seitenflächen eines Quaders unterteilt werden.

Die Erzeugung von Massemodellen kann in der CGA-Shape-Grammatik auf zwei unterschiedliche Arten erfolgen. Die einfachste Variante ist dabei die Verwendung von geometrischen Primitiven wie Quadern und Zylindern, die miteinander kombiniert und durch Transformationen in ihrer Position und Ausrichtung modifiziert werden. Zusätzlich zu diesen Grundprimitiven enthält CGA-Shape eine Reihe von einfachen Dachprimitiven, die für die Grundobjekte eingesetzt werden können und als Standardtypen zur Verfügung stehen. Dazu gehören beispielsweise Giebel- oder Mansardendächer. Eine Auswahl der vorhandenen Dachtypen ist in Abbildung 26 zu sehen.

Abbildung 26: Dachtypen in der CGA-Shape-Grammatik [Mü06]

Sind die Gebäudegrundstücke vorgegeben, beispielsweise weil sie aus einem geographischen Informationssystem importiert wurden, verwenden die Autoren einen zweischrittigen Ansatz. Zunächst wird versucht, die importierten Massemodelle anhand der Basisprimitive zu klassifizieren, um anschließend auf die vorhandenen Dachprimitive zurückgreifen zu können. Gelingt dies nicht, wird das Massemodell durch Footprintextrusion und das Dach durch den *Straight-Skeleton-Algorithmus* erstellt.

Abbildung 27 zeigt ein einfaches Gebäude, das mittels CGA-Shape erstellt wurde. Abbildung 28 zeigt einen Auszug aus dem Regelsatz, der für die Berechnung zum Einsatz kam.

Als Beispiel für die Ausdrucksfähigkeit von CGA-Shape entwickelten die Autoren eine Regelmenge zur Rekonstruktion des antiken Pompei vor der Zerstörung durch den Ausbruch des Vesuvs im Jahre 79 n.Chr.. Basis dieser Rekonstruktion waren sowohl Grundrisspläne als auch beispielhafte Skizzen von Gebäuden, wie sie zu dieser Zeit üblich waren. Die etwa 190 Regeln umfassende Produktionsmenge wurde in Zusammenarbeit mit Archäologen entwickelt. Basierend auf diesem System wurde ein Stadtmodell in drei verschiedenen LoDs erzeugt, das in der höchsten Auflösungsstufe aus etwa 1,4 Milliarden Polygonen bestand. Abbildung 29 zeigt gerenderte Ansichten des durch das Regelsystem erstellten Modells.

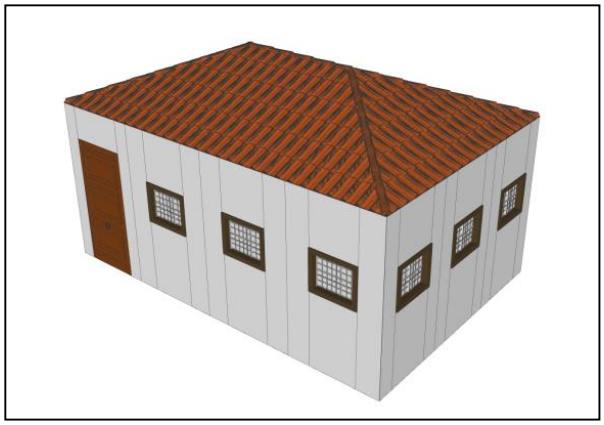

Abbildung 27: Mittels CGA-Shape erstelltes, einfaches Gebäude [Mü06]

```
PRIORITY 1:
1:  footprint ⇝ S(1r,building_height,1r) facades
      T(0,building_height,0) Roof("hipped",roof_angle){ roof }
PRIORITY 2:
2:  facades ⇝ Comp("sidefaces"){ facade }
3:  facade : Shape.visible("street")
      ⇝ Subdiv("X",1r,door_width*1.5){ tiles | entrance } : 0.5
      ⇝ Subdiv("X",door_width*1.5,1r){ entrance | tiles } : 0.5
4:  facade ⇝ tiles
5:  tiles ⇝ Repeat("X",window_spacing){ tile }
6:  tile ⇝ Subdiv("X",1r,window_width,1r){ wall |
      Subdiv("Y",2r,window_height,1r){ wall | window | wall } | wall }
7:  window : Scope.occ("noparent") != "none" ⇝ wall
8:  window ⇝ S(1r,1r,window_depth) I("win.obj")
9:  entrance ⇝ Subdiv("X",1r,door_width,1r){ wall |
      Subdiv("Y",door_height,1r){ door | wall } | wall }
10: door ⇝ S(1r,1r,door_depth) I("door.obj")
11: wall ⇝ I("wall.obj")
PRIORITY 3:
12: roof ⇝ Comp("sidefaces"){ covering }
      Comp("sideedges"){ roofedge } Comp("topedges"){ roofedge }
13: covering ⇝
      Repeat("XY",flatbrick_width,brick_length){ flatbrick }
      Subdiv("X",flatbrick_width,1r){ ε |
        Repeat("X",flatbrick_width){ roofedge } }
```

Abbildung 28: Auszug aus einer CGA-Shape Produktionsmenge [Mü06]

Abbildung 29: Rekonstruktion des antiken Pompei mittels CGA-Shape [Mü06]

5.3.2.4 **Diskussion der CityEngine**

Die vorgestellten Arbeiten legten den Grundstein für die Entwicklung der CityEngine, dem einzigen in dieser Arbeit vorgestellten prozeduralen System, das kommerziell erfolgreich vertrieben wird. Ausgehend vom ersten durch Müller entwickelten Prototyp wurde die CityEngine zunächst durch die Firma Procedural Inc. weiterentwickelt, bevor diese von der amerikanischen Softwarefirma Esri übernommen wurde. Die Arbeiten von Müller et al. waren nicht nur entscheidend für die Entwicklung der CityEngine, sondern auch für viele andere Ansätze im Bereich der regelbasierten, prozeduralen Erzeugung von Gebäude- und Stadtmodellen. Die aktuellen Versionen des Softwaresystems sind in der Lage, den Nutzer

durch die Bereitstellung von Nutzerinterfaces bei der Regelerzeugung zu unterstützen und bieten durch die Integration des Systems mit dem GI-System ArcGIS eine umfangreiche Werkzeugpalette zur Rekonstruktion von Städten basierend auf GIS-Daten. Die Oberfläche des Systems stellt dem Nutzer verschiedene Möglichkeiten bereit, Regeln zu definieren. So ist es beispielsweise möglich, Bilder existierender Gebäude als Basis für die Erstellung von Unterteilungsvorschriften zu verwenden. Das System ist dann in der Lage, erstellte Vorschriften mittels der vorab vorgestellten Comp- und Repeat-Regeln zu verallgemeinern, wodurch die Wiederverwendbarkeit der Regelbasis gesteigert wird. Dadurch lassen sich Produktionsregeln erstellen und anschließend auf weitere Massemodelle übertragen.

Um den grundsätzlichen Ansatz der CityEngine zu verdeutlichen und dadurch eine Abgrenzung zu dem hier vorgestellten System zu erreichen, soll ein kurzer Blick auf das Nutzerinterface des Softwaresystems geworfen werden.

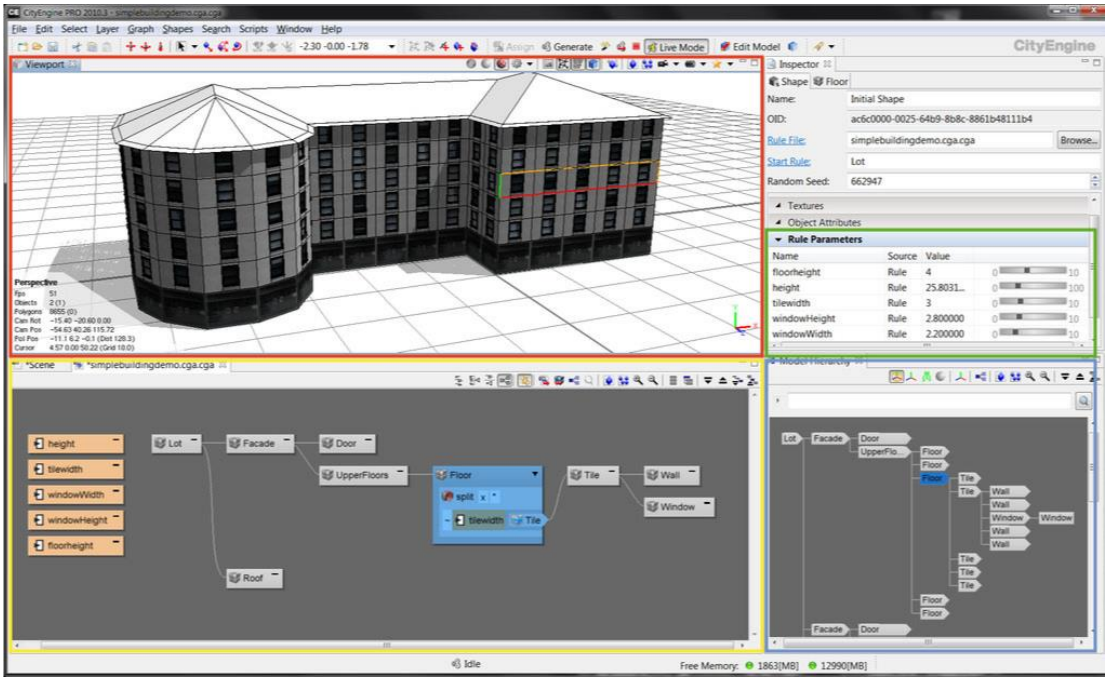

Abbildung 30: CityEngine PRO 2010.3 Nutzeroberfläche (farbige Kästen wurden nachträglich in die Abbildung eingefügt)

Abbildung 30 zeigt einen Screenshot dieser Oberfläche in der Version *PRO 2010.3*. Das Interface besteht aus mehreren Teilen, das größte Fernster (roter Kasten) bietet eine Ansicht des aktuell durch die Regelmenge erstellten Gebäudes. Dieses Fenster gleicht den verschiedenen perspektivischen Sichten auf eine Szene, wie sie auch von den meisten 3D-Modellierungswerkzeugen zur Verfügung gestellt werden. Sie ermöglicht eine

Rundumansicht der 3D-Modelle unter Verwendung einer perspektivischen Projektion, die es dem Nutzer erlaubt, das erstellte Gebäude von allen Seiten zu betrachten.

Im unteren Teil des Fensters stellt das System das Regelsystem (gelber Kasten) und die Erzeugungshierarchie (blauer Kasten) grafisch dar. Jedes in den unteren beiden Fenstern dargestellte Label entspricht dabei einem terminalen oder nicht-terminalen Symbol der Regelmenge. Steuerparameter für die bedingungsabhängige Ausführung von Regeln definiert der Nutzer auf der rechten Seite im Roll-Out *Rule Parameters* (grüner Kasten). Dort stellt ihm das Interface Regler für die Anpassung der Parameter zur Verfügung.

Diese Oberfläche versteckt die potentiell komplexen Regeln vor dem Nutzer, die intern für die Gebäudeerzeugung eingesetzt werden. Prinzipiell ist es aber auch möglich, diese direkt zu editieren oder per Hand zu verfassen. Die Oberfläche ähnelt von ihrem Konzept dem Ansatz, den auch Sven Havemann in seiner GML vertritt, die im nachfolgenden Abschnitt „Generative Modeling Language [Ha05] ausführlich vorgestellt wird. Dieser sieht starke Parallelen zwischen der Entwicklung von Softwaresystemen durch einen Programmierer und der Erstellung von 3D-Modellen durch einen Modellierer, da er die Erzeugung von 3D-Modellen als Lösungsprozess eines komplexen Problems betrachtet, das einen strukturierten Lösungsansatz erfordert [Ha05]. Aufgrund der prozeduralen Basis der CityEngine manifestiert sich diese Parallele im Vergleich zu Modellierungswerkzeugen noch deutlich stärker. Hier definiert der Anwender beispielsweise Variablen, die zur Steuerung der Modellerzeugung eingesetzt werden. Diese Variablen werden im Laufe des Ersetzungsprozesses ausgewertet und für die Erzeugung der Masse- und Fassadenmodelle eingesetzt.

Dabei muss festgehalten werden, dass der Umgang mit Variablen nicht zu vergleichen ist mit der Festlegung von Parametern in einem Modellierungssystem wie 3ds Max. In solchen Systemen besitzen die Parameter eine feste Bedeutung, die aus dem Systemkontext heraus vorgegeben ist. Betrachtet man als Beispiel einen Quader als einfachen parametrischen Grundkörper, so besitzt dieser in 3ds Max unter anderem die Parameter `Länge`, `Breite` und `Höhe`. Hierbei handelt es sich um Parameter, die nicht vom Nutzer definiert werden, dieser kann nur die Werte der Parameter ändern. Die Parameter selber besitzen eine festgelegte Bedeutung, die durch den Anwender nicht geändert werden kann. Die Regelparameter der CityEngine ähneln dagegen deutlich stärker Variablen, wie man sie in Programmiersprachen verwendet. Die Deklaration einer Variablen definiert deren Namen und Typ. Dieser Variablen können nach ihrer Deklaration Werte zugewiesen werden. In Abbildung 30 findet

man beispielsweise die Variable `floorheight` innerhalb des Rule-Parameter-Rollouts. Die Bedeutung dieses Wertes wird erst in den Regeln selber deutlich, beispielsweise bei der Unterteilung des Massemodells in eine Menge von Stockwerken. Hierfür muss der Nutzer eine Split-Regel festlegen, die auf diese Variable zurückgreift und sie für die Stockwerkserstellung auswertet. Ohne die Angabe einer Regel und die Verwendung der Variablen für die Regelauswertung besitzt diese Variable keine semantische Bedeutung.

Für einen menschlichen Benutzer wird diese nur durch die Benennung deutlich, da diese die beabsichtigte Verwendung ausdrückt, für das System ist die Semantik der Benennung dagegen vollkommen irrelevant, die Variable könnte auch durch eine willkürliche, alphanumerische Zeichenkette benannt werden. Gleiches gilt für deren Verwendung. Der Nutzer ist vollkommen frei in der Entscheidung, wie er sie einsetzt. So macht es aus Sicht der Wartbarkeit und eines möglichst leichten Verständnisses Sinn, eine Variable wie `floorheight` für die Festlegung der Höhe der durch einen Split erstellten Gebäudeunterteilungen einzusetzen. Dies ist aber nicht zwingend festgelegt, da eine Variable für das System nur einen Behälter mit einem Wert darstellt. Genauso gut könnte man die gleiche Variable zur Festlegung der Wanddicke oder als Symbol für erstellte Unterteilungen verwenden. Im Gegensatz zu Systemen wie 3ds Max ist die Verwendung einer Variablen somit nicht klar festgelegt, sondern vom Nutzer abhängig. In 3ds Max beschreibt der Parameter `Länge` dagegen immer die Länge eines parametrischen Objekts, dies kann durch den Nutzer nicht beeinflusst werden. Dadurch besitzt er eine klare und unveränderliche Semantik.

Das hier vorliegende System gibt dem Nutzer außerhalb des Quelltextes nicht die Möglichkeit, eigene Variablen zu definieren. Es stellt einen allgemeinen Parameter `floorHeight` zur Verfügung, der vom System direkt umgesetzt wird, um Stockwerke zu erzeugen. Hierfür greift es auf implementierte Methoden zurück. Während der Nutzer also in der CityEngine eine Variable definiert und dem System durch die Verwendung dieser Variablen innerhalb von Regeln mitteilt, wie diese einzusetzen ist, besitzen die Variablen im hier vorgestellten System eine semantische Bedeutung.

Dies ist ein Beispiel für einen wichtigen konzeptuellen Unterschied zwischen prozeduralen und regelbasierten Systemen. In regelbasierten Systemen würde der Nutzer für die Erzeugung von Stockwerken verschiedene Schritte manuell festlegen. Am Beispiel der vorab vorgestellten Operationen der CGA-Shape-Grammatik würde er zunächst angeben, dass ein initialer Grundriss (im Screenshot als *Lot* bezeichnet) zunächst in Richtung seiner

Oberflächennormalen extrudiert werden muss. Anschließend erfolgt die Anwendung der Split-Regel auf das derart erzeugte Massemodell, wobei die Unterteilung Elemente der Höhe `floorheight` erzeugt. Der Nutzer muss den Konstruktionsprozess also in Form einer Abfolge geometrischer Operationen spezifizieren (Extrusion, Split).

Im Semantic Building Modeler wird der Konstruktionsprozess dagegen durch eine Menge parametrisierter Algorithmen durchgeführt. Anstatt dem System mitzuteilen, dass das Massemodell durch eine Unterteilungsebene vorgegebener Ausrichtung in mehrere Teile bestimmter Höhe untergliedert werden soll, legt er fest, dass ein Gebäude mit einer bestimmten Anzahl von Stockwerken erstellt wird. Die Konstruktion erfolgt in der Semantik des Gebäudebaus und greift nur intern auf die erforderlichen geometrischen Operationen zurück. Dies versteckt die mathematische Komplexität der Konstruktion vor dem Nutzer und erleichtert dadurch unerfahrenen Anwendern den Einstieg in die prozedurale Gebäudegenerierung. Durch die Kapselung der geometrischen Operationen in den implementierten Algorithmen des Semantic Building Modelers ist es für einen Nutzer nicht erforderlich, sich mit linearen Transformationen und anderen geometrischen Operationen auseinanderzusetzen. Dadurch wird die Nutzung für unerfahrene Anwender deutlich vereinfacht, da sie die Gebäudeerstellung durch leicht verständliche Parameter steuern können. Auf der anderen Seite bieten Systeme wie die CityEngine durch die Verwendung von CGA-Shape eine größere Flexibilität durch eine umfangreiche Konfigurierbarkeit des Regelsystems.

Der nächste Ansatz für die prozedurale Erstellung von 3D-Modellen ist ein Hybridansatz zwischen regelbasierten und prozeduralen Ansätzen. Die *Generative Modeling Language* ermöglicht die Beschreibung von 3D-Modellen durch die einzelnen Konstruktionsschritte anstelle der Speicherung der Ergebnismodelle in Form von Punkten, Kanten und Oberflächenelementen. Ziel dieses Paradigmenwechsels ist unter anderem die Steigerung der Wiederverwendbar- und Modifizierbarkeit von 3D-Modellen. Dabei ist die Beschreibungssprache in ihrem Anwendungsbereich nicht wie CGA-Shape auf die Konstruktion von Gebäuden und Bauwerken eingeschränkt, sondern für beliebige Arten von 3D-Modellen einsetzbar.

5.3.3 **Generative Modeling Language [Ha05]**

Die *Generative Modeling Language* (GML) wurde von Sven Havemann im Verlauf seiner Promotion [Ha05] entwickelt. Dabei handelt es sich um ein Framework, das neben einer funktionalen Sprache für die Shape-Generierung auch einen Interpreter inklusive angeschlossener Renderengine enthält. Havemanns Motivation für die Entwicklung der GML resultierte aus den verschiedenen Schwächen speziell manueller Modellierungsansätze für 3D-Modelle, mit denen er während seiner Arbeit in verschiedenen Projekten konfrontiert wurde. Vor allem das bereits mehrfach diskutierte Problem der fehlenden Wiederverwendbarkeit sowohl des eigentlichen Modellierungsprozesses als auch des Modells selber empfand er als großen Nachteil solcher Ansätze.

Aufgrund seiner Tätigkeit in Projekten, die sich mit dem Aufbau digitaler Bibliotheken befassten, stieß er auf ein weiteres Problem solcher Systeme. Sobald man beginnt, einmal gebaute 3D-Modelle in umfangreichen Bibliotheken zu sammeln, um sie archivieren und einer breiten Nutzergruppe verfügbar zu machen, gewinnen Fragen der Dateigröße und des verwendeten Dateiformats zunehmend an Bedeutung. Mit zunehmender Komplexität der Modelle wächst auch deren Dateigröße. Dies hängt damit zusammen, dass die Speicherung keinerlei Abstraktion vornimmt, die Dateien müssen sowohl die Geometrie als auch die Topologie der jeweiligen Modelle vollständig enthalten, um diese wiederherstellen zu können [Ha05].

Wie in anderen Bereichen auch wurden innerhalb der Computergrafik Ansätze entwickelt, die sich damit befassen, wie man die Dateigröße solcher Modelle reduzieren kann. Dabei unterscheidet man verlustbehaftete von verlustfreien Kompressionsansätzen [GS02]. Bei verlustfreien Kompressionsverfahren ist es möglich, das Modell vollständig wiederherzustellen, durch die Kompression geht keinerlei Information verloren. Verlustbehaftete Ansätze arbeiten bei 3D-Modellen meist mit einer Simplifikation der Ausgangsstrukturen, beispielsweise durch das Entfernen von Punkten und Kanten aus dem Modell. Solche Ansätze verwenden Grenzwertverfahren, bei denen beispielsweise benachbarte Oberflächen zusammengefasst werden können, falls die Abweichung ihrer Ausrichtung unterhalb eines vorgegebenen Grenzwerts liegt. Offensichtlich führen solche Ansätze zu einer Veränderung des Quellmodells und erlauben dabei typischerweise keine Wiederherstellung der Ausgangsstrukturen [Wa02].

Neben der Größenproblematik wird man im Bereich digitaler Bibliotheken mit einem weiteren Problem konfrontiert, das sich im Zusammenhang mit der Verwaltung von 3D-

Modellen stellt. Hierbei handelt es sich um Technologien für das *Markup*, das *Indexing* und das *Retrieval* solcher Modelle. Zusammengefasst lassen sich diese Fragestellungen auf die Frage reduzieren, wie man nach 3D-Modellen suchen kann.

Die Schwierigkeit sei durch den Vergleich mit dem Retrieval von Texten aus einer digitalen Bibliothek veranschaulicht. Texte werden für die Suche meist durch einen Indexing-Mechanismus vorverarbeitet. Das Ergebnis dieser Indizierung ist eine Datenstruktur, die für die Beantwortung von Suchanfragen durch den Nutzer verwendet wird. Bei der Indizierung existieren verschiedene Ansätze, um die Qualität des Retrievals zu optimieren, so werden häufig bestimmte Wörter ignoriert (Verwendung von *Stopwordlisten*) oder Wörter auf ihre gemeinsame Grundform zurückgeführt (Algorithmen wie *Stemming* etc.). Eine Datenstruktur zur Speicherung des Ergebnisses eines solchen Indizierungsprozesses sind *Inverted Index Files*. Für jedes im Text vorkommende und während der Indizierung verarbeitete Wort wird eine Liste der Positionen aller Vorkommen dieses Wortes im Text erzeugt. Wird ein bestimmter Term durch den Nutzer angefragt, so benötigt man nur Zugriff auf diesen Index und erhält schnell alle Dokumente, in denen dieser Term zu finden ist [BR99].

Übertragen auf 3D-Modelle wird dieser Ansatz deutlich schwieriger. Was sollte ein Indexer beispielsweise indizieren, damit anschließend durch den Nutzer danach gefragt werden kann? Die Verwendung von Vertexkoordinaten oder Oberflächenstrukturen ist hier nicht zielführend. Eine Möglichkeit in diesem Bereich sind sogenannte *Query-by-Example*-Ansätze, bei denen der Nutzer dem System ein Modell zur Verfügung stellt und das System Modelle liefern soll, die diesem ähneln. Typischerweise arbeiten solche Systeme mit multidimensionalen Featureräumen, innerhalb derer Modelle aufgrund unterschiedlicher Eigenschaften angeordnet sind. Je näher sich zwei Modelle innerhalb dieses Raumes sind, desto ähnlicher sind sie [BR99]. Abgesehen von der Retrievalqualität ist allerdings problematisch, dass man zur Suche nach einem bestimmten Modell immer bereits über ein Referenzmodell verfügen muss, von dem ausgehend die Ähnlichkeitssuche startet.

Eine Alternative zu solchen ähnlichkeitsbasierten Ansätzen ist die Verwendung textueller Metadaten. Die Nutzung zusätzlicher Daten, die die Semantik des jeweiligen Modells beschreiben, gestattet den Einsatz der vorab beschriebenen Technologien zum Textretrieval. Dies kann durch die Verwendung einer Markup-Sprache wie XML (s. Abschnitt „Die eXtensible Markup Language (XML)“) erfolgen, oder aus einer einfachen Menge von Tags bestehen, vergleichbar den Metatags in HTML (s. Abschnitt „Die Hypertext Markup

Language (HTML)“), die Schlüsselwörter für die Beschreibung des Seiteninhalts enthalten. Dies erlaubt es dem Nutzer durch eine schlüsselwortbasierte Suche gezielt Modelle zu finden. Der große Nachteil an diesem Ansatz ist dann aber wiederum die Frage, wie die Metadaten in ein solches System gelangen. Mit steigender Größe der digitalen Bibliothek wird der Aufwand der manuellen Vergabe solcher Terme zu einem großen Problem. Ansätze, die es erlauben, wichtige Retrievalterme direkt aus der Repräsentation der 3D-Modelle abzuleiten, wären hier ein großer Fortschritt [BR99]. Havemann sieht darum den Schritt weg von der Repräsentation der Objekte und hin zur Repräsentation der Methoden, die diese Objekte erzeugen, als guten Ansatz an, um eine semantische Indizierung von 3D-Modellen zu ermöglichen [Ha05].

Havemann entwickelte 1998 ein erstes Toolkit für die interaktive, generative Modellierung von 3D-Modellen. Dieses Toolkit bestand aus einem C++-Kern, der die Datenstrukturen und zur Verfügung stehenden Operationen implementierte und über ein Python-Interface angesprochen werden konnte. Dies ermöglichte den Aufruf von Methoden des C++-Moduls und dadurch eine skriptbasierte Arbeitsweise.

5.3.3.1 **Subdivion-Surfaces**

Ein wichtiges Konzept, das innerhalb dieses frühen Systems implementiert wurde, war eine Erweiterung der *Catmull-Clark-Subdivision-Surfaces* [CC98], die auf die Arbeit von Tony deRose et al. [DKT98] zurückgeht. *Subdivision Surfaces* sind ein allgemeiner Ansatz zur rekursiven Unterteilung beliebiger Oberflächen. Die einzelnen Verfahren verwenden unterschiedliche Schemata, die festlegen, wie die Unterteilung vorgenommen wird. Subdivision Surfaces ermöglichen die Modellierung von Oberflächen mit weichen Kantenverläufen, ähnlich wie dies bei *Bézier-Patches* der Fall ist. Die Verwendung solcher Patches zur Modellierung komplexer Oberflächen kann äußerst kompliziert werden, wenn Artefakte und Löcher vermieden werden sollen. Aufgrund der mathematischen Definition der einzelnen Patches müssen Einschränkungen vorgenommen werden, um an gemeinsamen Patchkanten die Kontinuität der Oberfläche zu wahren und Löcher zu vermeiden. Die Verwendung solcher Einschränkungen geht automatisch zu Lasten der Freiheitsgrade, die der Nutzer bei der Modellierung besitzt und erschweren dadurch den Modellierungsprozess [Wa02]. Subdivison Surfaces können dagegen für beliebige Oberflächen eingesetzt werden und den Modellierungsprozess durch ihre algorithmische Definition deutlich vereinfachen.

5.3.3.1.1 Unterteilungsschemata

Die Kontinuität von Subdivision Surfaces ist eine Eigenschaft des Unterteilungsschemas, das die Oberflächen aus dem initialen Kontrollpolyeder erzeugt. Nähert man gekrümmte Oberflächen durch Polyeder an, so erscheint das Ergebnis kantig, da sich die Krümmung zwischen zwei benachbarten Polygonen abrupt ändert. Dies ist in der Realität nicht der Fall, hier verläuft die Krümmung kontinuierlich. Mathematisch drückt man dies durch die Differenzierbarkeit entlang der Fläche aus. Eine zusammenhängende Fläche ohne Sprünge besitzt eine C^0-Stetigkeit. Ist die Fläche überall mindestens einmal stetig differenzierbar, so ist die Oberfläche C^1-stetig. Allgemein ist eine Fläche C^n-kontinuierlich, wenn sie überall mindestens *n*-mal differenzierbar ist. Die Stetigkeit eines Subdivision Surface Schemas ist somit ein wichtiger Kennwert dafür, ob es in der Lage ist, gekrümmte Oberflächen zu erzeugen.

Weiterhin unterscheidet man zwischen *approximativen* und *interpolierenden* Schemata. Beiden Ansätzen ist gemein, dass sie ausgehend von einem Kontrollpolyeder starten, auf den dann der rekursive Unterteilungsalgorithmus angewendet wird. Bei approximativen Ansätzen liegen die Punkte des erzeugten Objekts nicht auf diesem Kontrollpolyeder. In jedem Iterationsschritt werden die bereits vorhandenen Punkte näher an die finale Oberfläche heranbewegt. Bei interpolierenden Schemata liegen die Punkte des Startpolyeders dagegen auch auf der Oberfläche des erzeugten Objekts. Die vor einer Iteration vorhandenen Punkte werden während der eigentlichen Berechnung nicht weiter bewegt.

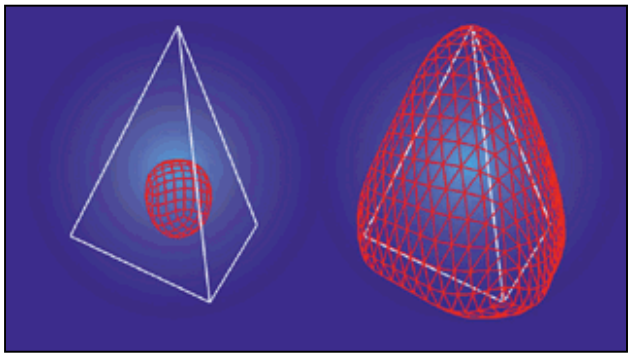

Abbildung 31: Subdivison Surface Schemata: Approximativ vs. Interpolierend [Sh00]

Abbildung 31 illustriert die Unterschiede zwischen den verschiedenen Ansätzen für den gleichen Kontrollpolyeder. Während die Vertices des weißen Kontrollpolyeders bei einem approximativen Schema auf der linken Seite nicht auf der berechneten, roten Objektoberfläche liegen, ist dies beim interpolierenden Schema auf der rechten Seite der Fall. Approximative Verfahren erzeugen auch für Kontrollpolyeder mit sehr scharfen Kanten sehr weiche Oberflächen. Ihr Nachteil liegt darin, dass die Modellierung des Ausgangspolyeders speziell bei komplexen Zielobjekten sehr schwierig sein kann, da das Berechnungsergebnis schwer vorhersagbar ist. Diese Vorhersagbarkeit ist bei interpolierenden Ansätzen größer, allerdings kann es in manchen Situationen ebenfalls problematisch sein, den Kontrollpolyeder derart zu konstruieren, dass die Berechnung das gewünschte Zielobjekt erzeugt.

Die Kernidee aller Schemata für Subdivision Surfaces ist die rekursive Unterteilung eines initialen Kontrollnetzes bis eine gewünschte Unterteilungstiefe erreicht wurde. Die Art, wie diese Unterteilung vorgenommen wird und was genau unterteilt wird, unterscheidet die verschiedenen Schemata voneinander.

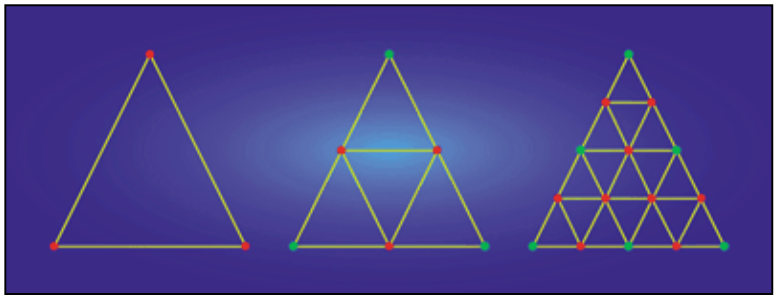

Abbildung 32: Polyedrisches Schema [Sh00]

Abbildung 32 zeigt ein sehr einfaches polyedrisches Schema, bei dem jedes Dreieck im Quellpolygon in vier neue Dreiecke aufgebrochen wird. Hierfür erzeugt man in der Mitte aller Dreieckskanten ein neues Vertex. Diese Punkte bilden dann die Eckpunkte eines Dreiecks, das das Ausgangsdreieck in vier neue Dreiecke unterteilt.

Abbildung 33 zeigt das Ergebnis der Anwendung des polyedrischen Schemas auf einen einfachen Kontrollpolyeder. Man erkennt, dass es sich um ein interpolierendes Schema handelt, da die Eckpunkte des Kontrollpolyeders auf der Oberfläche des unterteilten Objekts liegen. Oberflächen, die mittels des polyedrischen Schemas unterteilt wurden, sind C^0-stetig, es kann somit keine gekrümmten Oberflächen erzeugen.

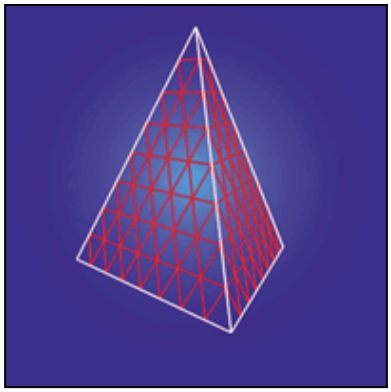

Abbildung 33: Unterteilung mit polyedrischem Schema [Sh00]

5.3.3.1.2 Catmull-Clark Oberflächen

Catmull-Clark Oberflächen stellen eine Erweiterung des einfachen polyedrischen Schemas dar. Der erste Unterschied besteht in den Komponenten, die es unterteilt. Während das polyedrische Schema auf Dreiecken arbeitete, werden bei Catmull-Clark Vierecke unterteilt. Hierfür erzeugt das Schema zunächst ein neues Vertex in der Mitte des zu unterteilenden Vierecks, wobei diese als Mittelwert der Ausgangsvertices errechnet wird. Anschließend wird für jede Kante des Vierecks ein neuer Punkt berechnet. Auch dieser entsteht als Mittelwert aus vier Quellpunkten. Hierbei handelt es sich um die Endpunkte der Kante, sowie der berechneten Mittelpunkte der adjazenten Faces der verarbeiteten Kante.

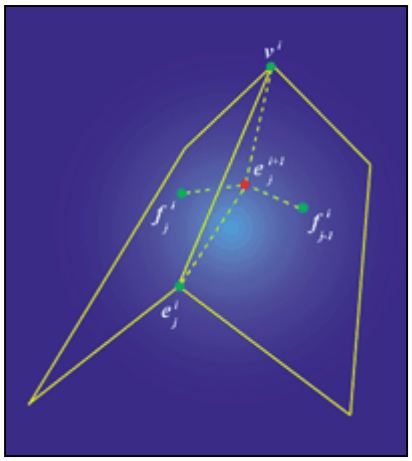

Abbildung 34: Catmull-Clark Edge-Vertex Berechnung [Sh00]

Abbildung 34 illustriert das Einfügen neuer Edge-Vertices. Das rote Vertex entsteht als Mittelwert aus den beiden Face-Vertices und den Endpunkten der Ausgangskante. Im letzten Schritt bewegt man alle Vertices des vorherigen Kontrollnetzes unter Verwendung der

neuen Face- und der alten Edge-Vertices. Auch hier wird eine gewichtete Verschiebung durchgeführt. Der Vollständigkeit halber soll hier kurz die Formel gezeigt werden, anhand derer die Verschiebung durchgeführt wird. Diese lautet:

$$v^{i+1} = \frac{N-2}{N} v^i + \frac{1}{N^2} \sum_{j=0}^{N-1} (e_j^i + f_j^{i+1})$$

Dabei ist v^{i+1} die Position des Vertex v nach der $(i+1)$-ten Iteration. N beschreibt die Valenz des Vertex, also die Anzahl adjazenter Kanten. e_j^i sind die Endvertices aller zu v adjazenten Kanten, f_j^{i+1} die neu berechneten Face-Vertices der Faces, die jeweils adjazent zur aktuell verarbeiteten Kante e_j^i liegen. Nachdem auf diese Art alle neuen Vertices berechnet und die alten Vertices verschoben wurden, werden neue Kanten berechnet, indem alle neuen Face-Vertices mit ihren adjazenten, neuen Edge-Vertices verbunden werden. Außerdem werden neue Kanten zwischen den verschobenen Vertices und den zu diesen adjazenten Edge-Vertices eingefügt. Abbildung 35 zeigt eine Oberfläche, die mittels Catmull-Clark Subdivision erstellt wurde.

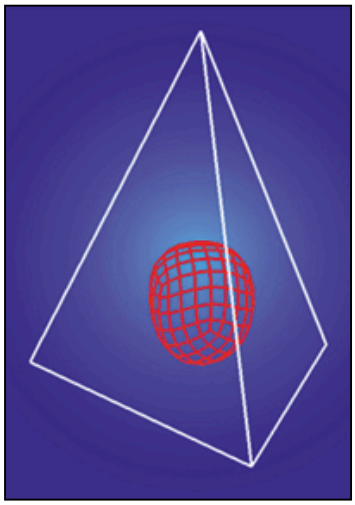

Abbildung 35: Ergebnis der Catmull-Clark-Unterteilung [Sh00]

Subdivision Surfaces sind eine zentrale Komponente des von Havemann entwickelten Python-Prototyps. Hierfür verwendete er allerdings eine Erweiterung des ursprünglichen Schemas, das von deRose et al. für die *Pixar Animation Studios*[12] entwickelt wurde, um den Kurzfilm *Geri´s Game* [Pi97] zu produzieren. Die Kernidee dieser Arbeit ist die Verwendung von Kanten- und Vertexgewichten für den Kontrollpolyeder, die festlegen, wie

[12] Pixar: http://www.pixar.com/

das Unterteilungsschema angewendet wird. In der einfachsten Variante weist man einer Kante ein ganzzahliges Kantengewicht $s < \infty$ zu. Solange diese Kante ein Gewicht mit $s > 0$ besitzt, wird sie mittels des polyedrischen Schemas unterteilt und bleibt dadurch scharf. Nach jeder Unterteilung wird s dekrementiert, sobald $s = 0$ erreicht wird, erfolgt die Unterteilung mittels des vorab beschriebenen Catmull-Clark-Schemas. Dadurch ist es möglich, die Krümmung der Oberfläche an unterschiedlichen Kanten zu variieren.

Abbildung 36 zeigt den Einfluss unterschiedlicher Kantengewichtungen im Ansatz von deRose et al. [DKT98]. Die gelben Kanten im Kontrollpolyeder besitzen Kantengewichte von 0, erzeugen also gekrümmte Oberflächen. Im rechten Bild besitzen die roten und magentafarbenen Kanten jeweils ein Gewicht von 4. Man erkennt deutlich, wie mit steigendem Kantengewicht das erzeugte Objekt zunehmend schärfere Kanten besitzt.

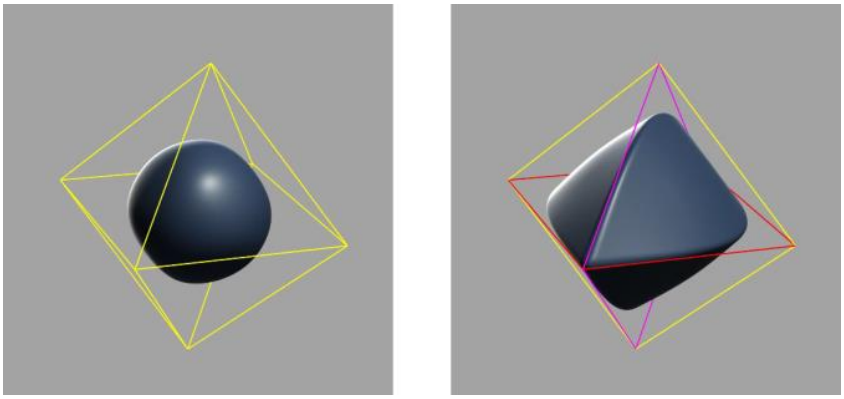

Abbildung 36: Erweiterte Catmull-Clark Subdivision mit Kantengewichten [DKT98]

5.3.3.2 **Subdivision Surfaces in der Generative Modeling Language**

Havemann verwendet ein ähnliches Konzept in seinem ersten Prototyp [Ha05]. Dieses Verfahren eignet sich hervorragend für die Erzeugung gekrümmter Oberflächen, die der Nutzer direkt innerhalb des Prototyps gestalten und manipulieren kann. Wichtige Erkenntnisse, die Havemann aus der Entwicklung dieses ersten Prototyps zog, waren neben der Erfahrung im Umgang mit Catmull-Clark-Subdivision Surfaces auch ein Ansatz zur Aufzeichnung des Modellierungsprozesses.

Hierfür protokollierte das Werkzeug die Useroperationen auf der Ebene des Python-GUIs, das verwendet wurde, um die in C++ implementierte Kernfunktionalität aufzurufen. Zentral hierfür sind sowohl die Aufzeichnung der Auswahloperationen durch den Nutzer, also die Bestimmung der Faces, auf die geometrische Operationen angewendet werden sollen, als

auch die durchgeführten Operationen selber. Eine solche Aufzeichnung gestattet neben der Wiederverwendung des Modellierungsprozesses auch die Implementation von Undo- / Redo-Funktionalität, die für die Arbeit mit Werkzeugen dieser Art von großer Bedeutung für den Nutzer ist. So sind Anwender meist experimentierfreudiger, wenn sie in der Lage sind, durchgeführte Operationen rückgängig zu machen. Weiterhin ermöglicht die Protokollierung als textbasierte Abfolge von Python-Befehlen deren nachträgliche Modifikation innerhalb eines einfachen Texteditors, so dass der Nutzer auch nach Fertigstellung des Modells an beliebigen Punkten des Modellierungsprozesses Änderungen vornehmen kann, die sich direkt auf das finale Modell auswirken. Trotzdem war dieser Ansatz wenig intuitiv, die direkte Erstellung eines Modells ausschließlich innerhalb eines Texteditors erfordert die Programmierung desgleichen in Python selber. Diese Schwierigkeit bezeichnet Havemann als „*Code Generation Problem*“ [Ha05]. Hierin zeigte sich das Kernproblem solcher Ansätze, da die Python-Befehle selber auf einer sehr niedrigen Ebene angesiedelt sind und direkte geometrische Operationen auf den jeweils ausgewählten Komponenten ausführen, was die Vorhersagbarkeit des Resultats der jeweiligen Operationen für den Nutzer sehr schwierig macht.

5.3.3.2.1 Selektive Patchtesselation

Aus der Entwicklung des Prototyps ergaben sich wichtige Erkenntnisse, speziell in Bezug auf die Bedeutung der verwendeten Datenstrukturen zur Verwaltung der Geometriedaten. Zunächst entwickelte Havemann eine Struktur, die er als *Subdivision Mesh* (*SMesh*) bezeichnete [Ha05]. Eine Kernanforderung an diese Struktur ist die Möglichkeit, die Unterteilung eines initialen Patches mit unterschiedlicher Tiefe durchzuführen, also die Anzahl der Unterteilungsschritte je nach Bedarf zu variieren. Um die Ergebnisse einer solchen selektiven Tesselation zu speichern, eignen sich hierarchische Baumstrukturen sehr gut, die nachfolgend auch für das Rendering eingesetzt werden können. Jeder Knoten innerhalb des Baumes repräsentiert dabei genau ein Face innerhalb der Unterteilungshierarchie. Die Kinder dieser Faces sind die Ergebnisse der nächsten Unterteilungsstufe. Die Speicherung der unterschiedlichen Unterteilungsebenen ermöglicht das dynamische Anpassen der Unterteilungsstufe während der Laufzeit, indem man den Baum entweder auf- oder absteigend durchläuft.

Bei der selektiven Tesselation mittels Catmull-Clark-Subdivision unterscheidet Havemann zwischen vollständiger und unvollständiger Unterteilung. Die Unterscheidung basiert darauf,

ob bei einer Patchunterteilung alle Kinder eines Patches oder nur ein konkret angefordertes berechnet werden. Betrachtet man das Catmull-Clark-Subdivision-Schema, so wird deutlich, dass die Berechnung einer Unterteilungsebene Vertexinformationen der direkten Nachbarschaft des zu unterteilenden Patches benötigt, um die Berechnung durchzuführen. Dabei muss die Nachbarschaft sich auf der gleichen Unterteilungsebene befinden, wie der aktuell bearbeitete Patch. Dadurch ist es häufig erforderlich, zur Unterteilungsberechnung eines Patches direkt benachbarte Patches vorab zu unterteilen, damit diese dasselbe Ausgangslevel besitzen. Bei einer unvollständigen Unterteilung berechnet man in diesem Fall nur die Patchkinder, die benötigt werden, sonst bestimmt man die nächste Stufe vollständig. Speziell bei der vollständigen Unterteilung kann dadurch leicht der Fall auftreten, dass sich eine initiale Unterteilungsanfrage immer weiter fortpflanzt und eine Vielzahl weiterer Unterteilungen auslöst. Dafür ist allerdings der administrative Aufwand zur Verwaltung des Patchnetzes geringer.

Welche Form der Unterteilung am sinnvollsten ist, sollte laut Havemann darum auch von der verwendeten Hardware abhängen [Ha05]. Unterstützt diese eine schnelle Traversierung des Baumes, ist aber auf der anderen Seite in Bezug auf die Unterteilungsberechnungen vergleichsweise langsam, so eignet sich ein unvollständiges Unterteilungsschema besser, ansonsten greift man zur vollständigen Variante. Die ersten Versuche mit der SMesh-Datenstruktur waren laut Havemann zwar recht vielversprechend, trotzdem besitzt die Datenstruktur speziell für das interaktive Rendering eine Reihe großer Nachteile.

Das größte Problem besteht in der Vermeidung von Brüchen an Patchkanten mit Patches unterschiedlicher Unterteilungstiefe, da diese keine gemeinsamen Kanten besitzen. Dieses Problem ähnelt dem vorab diskutierten Schwierigkeiten bei der Verwendung von Bézier-Patches als Modellierungselemente, auch bei diesen ist es unter Umständen schwierig, Brüche an den Verbindungskanten benachbarter Patches zu vermeiden, ohne dabei zu viele Freiheitsgrade in der Modellierung zu verlieren [Wa02]. Für die von Havemann verwendeten Patches besteht die einzige Möglichkeit, solche Brüche zu verhindern, in der Projektion der Unterteilungsvertices auf die begrenzende Oberfläche des Objekts. Da die derart berechnete Position allerdings von allen Vertices aller Faces abhängt, die das Vertex verwenden, ist der Berechnungsaufwand enorm.

Ein weiteres Problem resultiert aus der Modifikation der Ausgangs-Patches. Führt man Änderungen an diesen durch, so muss die rekursive Unterteilung erneut durchgeführt werden, vorab berechnete Faces und Vertices müssen freigegeben und wieder alloziiert

werden. Dadurch wird die Verwendung des rekursiven Unterteilungsalgorithmus zu einem kostspieligen Prozess, der für Echtzeitanwendungen ungeeignet ist.

Aus diesem Grund verwendet Havemann ein anderes Vorgehen, das die rekursive Unterteilung unnötig macht. Der Ansatz basiert auf der Verwendung von Basisfunktionen, die über die ursprünglichen Kontrollpunkte des Kontrollpolyeders gewichtet werden. Die Bestimmung der Basisfunktionen kann offline durchgeführt und ihre Ergebnisse gespeichert werden. Einen solchen Ansatz verfolgen Bolz und Schröder [BS02], die offline eine Tesselation der Catmull-Clark-Basisfunktionen vorberechnen und die Ergebnisse anschließend verwenden, um Catmull-Clark-Unterteilungen in Echtzeit durchführen zu können. Die Bestimmung der Basisfunktionen soll für einen regulären Patch exemplarisch dargestellt werden.

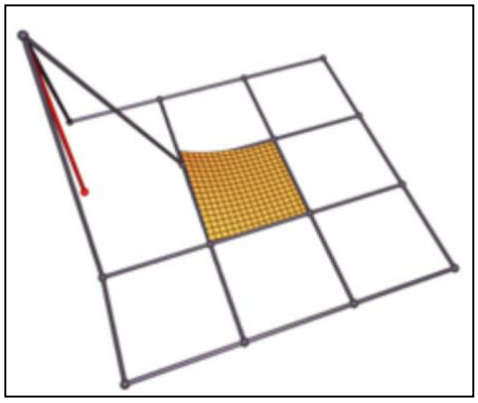

Abbildung 37: Subdivision Surface mittels Basisfunktionsevaluation [Ha06]

Abbildung 37 zeigt ein reguläres Patchnetz in der xy-Ebene. Die Berechnung der Basisfunktionen basiert darauf, dass man für jeweils einen Kontrollpunkt dessen z-Komponente auf 1 setzt, die anderen Kontrollpunkte verbleiben in der xy-Ebene. Betrachtet man nun die Oberfläche, die durch die Unterteilung erzeugt wird (in der Abbildung gelb hinterlegt), so kann man den Einfluss jedes der 16 Kontrollpunkte auf beliebige Punkte auf der Oberfläche bestimmen als Abweichung dieser Punkte von der xy-Ebene. Dabei wird ein Oberflächenpunkt innerhalb des Parameterraums des Patches dargestellt durch seine Parameter $(x, y) \in [0,1] \times [0,1]$. Je größer der Einfluss eines Kontrollpunktes auf einen Oberflächenpunkt, desto größer auch die Abweichung der z-Koordinate dieses Oberflächenpunktes. Über diesen Ansatz kann man die Basisfunktionen definieren mittels derer man anschließend in der Lage ist, innerhalb des Parameterraums des Patches Punkte auf beliebigen Unterteilungsebenen durch eine Kombination der gewichteten

Basisfunktionen direkt zu errechnen, ohne dabei den rekursiven Unterteilungsalgorithmus einzusetzen.

Ein Problem der Vorberechnung der Kontrollpunktgewichte besteht in der Abhängigkeit des Unterteilungsprozesses von der Valenz der beteiligten Vertices. Bei einer maximalen Valenz m innerhalb des Ausgangspolyeders ist es erforderlich, $m \times m$ verschiedene Valenzkombinationen zu berechnen, bei der jede Kombination eine eigene Menge an Gewichten für die jeweiligen Vertices benötigt. Um dieses Problem zu lösen, setzt Havemann Vertex- und Face-Ringe ein. Für eine detaillierte Erläuterung dieses Konzepts sei auf Havemann verwiesen [Ha05].

5.3.3.2.2 Repräsentationsformen für Subdivison Surfaces

Nachdem vorab Möglichkeiten zur effizienten Berechnung von Subdivision Surfaces erläutert wurden, soll nun auf die Datenstrukturen eingegangen werden, die Havemann für deren Repräsentation einsetzt [Ha05]. Diese muss eine Reihe von Anforderungen erfüllen. Eine der wichtigsten ist die Fähigkeit, polygonale und Freiform-Oberflächen innerhalb eines Objekts verwalten und darstellen zu können. Da die verwalteten Objekte durch den Nutzer interaktiv modifiziert werden können, müssen die Strukturen selektive Updates an den geometrischen Daten erlauben. Ändert der Nutzer beispielsweise einen kleinen Teil der Oberfläche durch die ihm zur Verfügung gestellten Operatoren, so ist es für eine Echtzeitanwendung nicht akzeptabel, wenn die Oberflächenunterteilung vollständig neuberechnet werden muss. Stattdessen sollten lokale Modifikationen auch nur lokale Updates nach sich ziehen, um die Interaktivität des Systems nicht durch aufwendige Neuberechnungen zu gefährden.

Um diese Ziele zu erreichen, verwendet Havemann eine Klassenhierarchie bestehend aus drei aufeinander aufbauenden Repräsentationsklassen. Auf der untersten Ebene stehen *Boundary-Representation Meshes* (*B-Rep-Meshes*), die als Containerdatenstruktur dazu dienen, die Verbindungen innerhalb eines Meshes zu repräsentieren. Dabei ist die Datenstruktur generisch implementiert und in der Lage, beliebige planare Graphen zu repräsentieren. Aufbauend auf den B-Rep-Meshes folgen die *Combined B-Rep-Meshes*. Diese enthalten Vertices, Triangulationen polygonaler Oberflächenelemente sowie Subdivision Surfaces, mittels derer gekrümmte Oberflächen dargestellt werden können. Combined B-Rep-Meshes stellen somit die Verbindung zwischen polygonalen und

Freiform-Oberflächen her und ermöglichen deren Verwaltung in der gleichen Datenstruktur. *Progressive Combined B-Reps* stellen die höchste Ebene dieser Hierarchie her, da sie die Datenstruktur der Combined B-Reps um ein Interface zur Modifikation der Geometriedaten erweitern. Eine wichtige Fähigkeit dieser Combined B-Reps ist neben den Operationen auch deren Logging. Jede auf dem Mesh ausgeführte Operation wird kodiert geloggt und kann dadurch rückgängig gemacht oder wiederholt werden. Eine derartige Repräsentation der Operation kann auch für die flexible Wahl unterschiedlicher LoDs eingesetzt werden, indem Details weggelassen oder wieder hinzugefügt werden. Nachfolgend soll kurz auf die wichtigsten Details der unterschiedlichen Mesh-Klassen eingegangen werden.

Für die Repräsentation der Meshdaten implementieren die B-Rep-Meshes eine Erweiterung der vorab erörterten Winged-Edge-Datenstruktur, die als *Half-Edge-Datenstruktur* bezeichnet wird. Dabei handelt es sich um eine kantenbasierte Datenstruktur, die eine schnelle Iteration von Flächen und Knoten erlaubt und dabei einen vergleichsweise geringen Speicherbedarf besitzt. Allerdings ist die Datenstruktur nur zur Repräsentation von *Manifold-Meshes* geeignet. Bei einem solchen Mesh sind alle Faces der Oberfläche von einem beliebigen Startface aus erreichbar, indem man von diesem ausgehend immer auf die jeweils adjazenten Faces wechselt, alle Faces also direkt oder indirekt miteinander verbunden sind. Darüber hinaus darf jede Kante nur adjazent zu einem oder zwei Faces sein [SE02].

Die Kernidee der Half-Edge-Datenstruktur geht bereits aus ihrem Namen hervor. Kanten werden zerteilt und als gerichtete Halbkanten gespeichert. Dabei sind die zwei Halbkanten, aus denen eine Kante besteht, einander entgegengerichtet. Je nach Implementation unterscheiden sich die Informationen, die die einzelnen Komponenten speichern, abhängig davon, ob der Speicher- oder der Rechenbedarf minimiert werden soll. Üblicherweise speichert jede Half-Edge Zeiger auf das Face, zu dem sie gehört, ihrem Startvertex, ihrer Zwillings-Half-Edge sowie auf den Nachfolger in der zirkulären Kantenliste, die das Face beschreibt. Soll die Iteration über die einzelnen Faces möglichst schnell erfolgen, kann zusätzlich auch noch ein Zeiger auf den Vorgänger in der zirkulären Kantenliste implementiert werden, dies ist für die Basisiteration allerdings nicht notwendig. Knoten in dieser Struktur enthalten nur ihre Position und einen Zeiger auf die Kante, die den Knoten als Ausgangsknoten verwendet. Faces wiederum speichern in der kleinsten Version einen Zeiger auf eine beliebige Außenkante, von dieser ausgehend können anschließend alle weiteren Außenkanten basierend auf der zirkulären Kantenliste durchlaufen werden [Be08].

Um die Performance der Datenstruktur zu maximieren und dadurch interaktive Modifikationen der Meshs zu erlauben, verwendet Havemann zusätzlich *Skipvector* und -*chunk-Datenstrukturen*, um die einzelnen Elemente zu speichern. Die grundsätzliche Idee sei am Skipvector kurz erläutert. Skipvectoren sind Containerdatenstrukturen vergleichbar zu Vector-Containern, wie sie in vielen verschiedenen Programmiersprachen zur Verfügung stehen, in C++ beispielsweise durch Verwendung der *Standard Template Library* (STL) [Jo99]. Vector-Container werden üblicherweise durch Arrays implementiert. Sie erlauben einen $O(1)$-Zugriff auf ihre Elemente, ermöglichen das Hinzufügen und Entfernen von Elementen allerdings nur am Ende der Datenstruktur. Skipvectoren stellen nun ebenfalls eine array-basierte Implementation dar, recyceln im Gegensatz zu normalen Vector-Strukturen aber ihre Elemente. Dadurch werden die kostspieligen Konstruktor- und Destruktor-Berechnungen vermieden. Soll ein Objekt aus dem Container entfernt werden, wird es nur logisch gelöscht, indem es auf inaktiv gesetzt wird. Bei der Iteration über die Objekte werden dann inaktive Objekte übersprungen. Der Container selber verwaltet Informationen darüber, welches Objekt zuletzt gelöscht wurde. Werden neue Objekte benötigt, liefert er eine Referenz auf das jeweilige Objekt, so dass dieses wiederverwendet werden kann. Dadurch wird zusätzlich zur Destruktoreinsparung auch die kostspielige Erzeugung neuer Objekte eingespart, was Skipvectoren zu einer effizienten Containerstruktur macht.

Die Combined-B-Rep-Strukturen bauen auf den B-Rep-Strukturen auf. Sie nutzen deren Half-Edge-Datenstrukturen für die Repräsentation der Meshdaten und erweitern sie um die Fähigkeit Subdivision Surfaces zu verwalten. Dadurch wird die Trennung zwischen Polygon- und Freiformmodels überwunden. Kern des Ansatzes ist die Verwendung eines Flags, das für jede Kante des Meshs gesetzt werden muss. Dieses *Sharpness Flag* wird für die Unterscheidung zwischen scharfen und weichen Kanten eingesetzt. In Regionen eines Meshs, in denen nur scharfe Kanten vorkommen, wird das Mesh mittels eines Standard-Polygon-Renderings dargestellt, in Bereichen mit weichen Kanten wird die verwaltete Geometrie als Kontrollpolyeder für eine Catmull-Clark-Subdivision eingesetzt, die mittels der vorab vorgestellten Ansätze unterteilt wird. Bei gemischten Kantentypen, also sowohl weichen als auch scharfen Kanten, werden die einzelnen Vertices anhand ihrer adjazenten Kanten klassifiziert. Ein *Smooth-Vertex* ist dabei ein Vertex, das ausschließlich weiche Kanten verbindet. Ist dagegen genau eine Kante scharf, die restlichen Kanten weich, so wird das Vertex als *Dart-Vertex* bezeichnet. Zwei scharfe Kanten machen das Vertex zu einem

Crease-Vertex, drei oder mehr scharfe Kanten definieren ein *Corner-Vertex*. Anhand der Klassifizierung werden bei der Unterteilung unterschiedliche Schemata für die Berechnung der unterteilten Patches eingesetzt. Beispielsweise verbleiben Corner-Vertices über alle Unterteilungsebenen hinweg immer an der gleichen Position.

Das Sharpness-Flag gibt dem Nutzer ein zusätzliches Werkzeug an die Hand, das für die Modellierung eingesetzt werden kann. Neben den erweiterten Modellierungsoptionen bieten Combined-B-Reps die Möglichkeit eines adaptiven LoD während der Laufzeit für die Freiform-Oberflächenelemente. Durch die Möglichkeit der selektiven Patchunterteilung kann die Unterteilungsstufe so gewählt werden, dass eine gute Balance zwischen visueller Qualität und hoher Renderperformance erreicht wird. Ist ein Objekt beispielsweise weit entfernt vom Betrachter, so kann die Unterteilungstiefe niedriger gewählt werden, was zu einer deutlich kleineren Anzahl an Patches führt. Umgekehrt erlauben es die optimierten Unterteilungsschemata, die Unterteilungstiefe zu erhöhen, wenn sich der Betrachter nähert.

Für die Bestimmung der erforderlichen Subdivisiontiefe verwendet Havemann eine Reihe von Heuristiken [Ha05]. Dazu gehört die Projektionsgröße der Quads. Alle Quads sollten eine ähnliche Projektionsgröße im Bildschirmraum besitzen. Bei einem Patch, der beispielsweise doppelt so weit vom Betrachter entfernt ist, kann man die Unterteilungstiefe um eine Stufe reduzieren. Als zweites Kriterium nennt er die Krümmung der Oberfläche zwischen benachbarten Patches gemessen über die Winkelabweichung der Normalen der benachbarten Oberflächensegmente. Übersteigt diese einen Grenzwert, werden die Patches erneut unterteilt, um zu starke Steigungsunterschiede zu reduzieren. Ein weiteres Kriterium betrifft die Silhouetten der betrachteten Objekte. Eine Silhouette entsteht, wenn ein Patch nicht vollständig sichtbar ist, ein Teil also verdeckt ist. Bei solchen Patches erhöht man die Unterteilungstiefe um eine Ebene. Die hierfür erforderlichen Berechnungen und die daraus resultierenden Unterteilungen werden durch die Combined-B-Rep-Mesh-Datenstrukturen implementiert und zur Verfügung gestellt. Innerhalb dieser Klasse stehen auch Methoden für das Rendering der Meshes bereit, ebenso wie solche für das Löschen und Erzeugen sämtlicher vorhandener Grundelemente, also beispielsweise Vertices, Edges oder Faces.

Aufbauend auf dieser Klasse folgt nun die letzte Klasse in der erläuterten Hierarchie, die Progressive Combined B-Rep-Meshes. Sie erweitern die vorab beschriebenen Klassen um eine Menge von Operatoren zur interaktiven Modifikation der Meshes. Dabei dürfen Änderungen nicht die Konsistenz der verarbeiteten Meshes gefährden. Im Idealfall sollten die Basisoperatoren so gewählt werden, dass ihre Anwendung unabhängig von dem

jeweiligen Anwendungskontext immer einen gültigen Zustandsübergang erzeugt, so dass man nie von einem konsistenten in einen inkonsistenten Meshzustand gerät. Genau diese Forderung wird durch die Euler-Operatoren erfüllt. Zur Verfügung stehen Methoden zum Einfügen und Entfernen von Vertices, Kanten und Faces in ein Mesh-Objekt. Zu jeder Operation existiert eine inverse Operation mittels derer man das Ergebnis rückgängig machen kann. Durch die Verwendung der Euler-Operatoren als Basis für die Modifikationen ist darum die Implementation eines Undo-Mechanismus sehr gut umsetzbar, indem man die angewendeten Operationen loggt. Soll eine Operation rückgängig gemacht werden, führt man deren inverse Operation auf die Übergabeparameter aus und erhält dadurch den Ausgangszustand. Zu diesem Zweck loggen Progressive Combined-B-Rep Meshes jede ausgeführte Nutzeroperation auf der Ebene der Euler-Operatoren. Daher resultiert auch die Bezeichnung als Progressive Mesh.

Progressive Meshes gehen zurück auf eine Arbeit von Hugues Hoppe [Ho96]. In dieser beschreibt er eine Datenstruktur zur Speicherung von Meshes, die durch eine Mesh-Optimierung aus einem Quell-Mesh erzeugt wurden. Die Kernidee dabei ist es, das Meshnetz sukzessive zu vereinfachen, indem Kanten entfernt werden. Diesen Schritt bezeichnet Hoppe als *Edge Collapse*-Operation. Das Gegenstück zum Edge Collapse ist der sogenannte *Vertex Split*. Bei dieser Operation wird ein Vertex durch zwei neue Vertices ersetzt, die durch eine Kante miteinander verbunden sind. Das Verfahren von Hoppe startet nun bei einem Mesh und führt solange Edge Collapses durch, bis keine weiteren Vereinfachungen mehr durchgeführt werden können, ohne die Form des Meshes zu stark zu verändern. Die Auswahl der zu entfernenden Kanten erfolgt über die Minimierung einer Energiefunktion, deren Werte die Änderungen der Meshform aufgrund der Kantenentfernungen beschreibt. Diese Funktion wird für jede Kante innerhalb des Meshes berechnet, anschließend wird diejenige Kontraktion ausgeführt, die zur geringsten Änderung führt. Während der Berechnung wird jede Kontraktion aufgezeichnet und gespeichert. Die so entstandene Vereinfachungshistorie kann anschließend mittels Standardalgorithmen komprimiert werden und nimmt dadurch deutlich weniger Platz ein. Das Progressive Mesh besteht dann aus dem Mesh, das durch die Unterteilungen entstanden ist und der Kontraktionsfolge, mittels derer man das Ausgangsmesh verlustfrei wiederherstellen kann. Dadurch reduziert sich zum einen der Speicherplatz gegenüber einem Standard-LoD-Ansatz deutlich, zum anderen kann man kontinuierliche Anpassungen der Mesh-Auflösung

vornehmen und vermeidet das plötzliche Auftauchen oder Verschwinden geometrischer Details, wie es beim Wechsel zwischen vorberechneten LoDs entsteht.

Einen solchen Ansatz verwendet auch Havemann in seiner Arbeit, anstatt allerdings ausschließlich Edge Collapse-Operationen zu verwenden und diese aufzuzeichnen, protokollieren die Progressive Combined B-Rep-Meshes sämtliche angewendeten Euler-Operatoren. Allerdings gruppiert Havemann eine Menge aufeinanderfolgender Euler-Operationen, die gleiche Teile des Meshs editieren, in sogenannten Euler-Makro zusammen. Diese werden bei Undo-Redo-Anfragen eingesetzt, um eine Folge von Euler-Operationen rückgängig zu machen oder zu wiederholen. Dabei kann ein solches Makro eine beliebige Anzahl von Euler-Operationen enthalten und zusammenfassen.

Für den Nutzer eines solchen Systems sind Euler-Operatoren nicht gut geeignet, da sie auf einem zu niedrigen Abstraktionslevel angesiedelt sind. Bei der Mesherstellung innerhalb gängiger Modellierungswerkzeuge stehen dem Nutzer typischerweise Operationen auf einem deutlich höheren Abstraktionsniveau zur Verfügung. Ein gutes Beispiel für eine solche Operation ist die Extrusion eines polygonalen Querschnitts in Richtung seiner Oberflächennormalen um einen festgelegten Betrag. Diese Operation ermöglicht die Erstellung dreidimensionaler Objekte aus zweidimensionalen Formen und stellt eine vergleichsweise intuitive Modellierungstechnik dar. Havemanns Ansatz innerhalb des GML-Frameworks ist es nun, Operationen wie die Faceextrusion durch eine Folge von Euler-Operatoren auszudrücken. Dadurch kapselt er die vergleichsweise komplexen Operationen auf der Ebene der Basisoperatoren vor dem Nutzer und macht es für diesen unnötig, sich auf der Ebene der geometrischen Primitive zu bewegen, um die gewünschte Modellierung vorzunehmen. Exemplarisch stellt er unter anderem ein einfaches Extrusionswerkzeug und eine Path-Extrusion vor, bei der ein Querschnitt entlang eines beliebigen Spline-Pfades geführt und dabei skaliert wird. Jedes dieser komplexen Werkzeuge wird durch C++-Code umgesetzt, die Modifikationen der Meshes selber erfolgen allerdings nur unter Verwendung der Euler-Operatoren, die dadurch eine Zwischenschicht zwischen der Geometrie und den komplexen Bearbeitungswerkzeugen bilden [Ha05].

5.3.3.3 **Implementation der GML**

Auf diesen Schichten setzt nun die eigentliche Generative Modeling Language (GML) auf, wobei die Bezeichnung GML laut Havemann sämtliche vorab vorgestellten Konzepte und

Technologien umfasst. Die GML selber ist eine stackbasierte Scriptsprache inspiriert durch die Programmiersprache *PostScript*. PostScript selber wurde 1985 von der Firma *Adobe*[13] als Sprache für die prozedurale Beschreibung von Seiten für Drucker entwickelt. Sie ist in der Lage, eine Vielzahl unterschiedlicher zweidimensionaler Objekte zu beschreiben. Dazu gehören neben Text beispielsweise auch Vektor- und Rastergrafiken [Ad94]. Dadurch ermöglicht PostScript die Erzeugung multimedialen Outputs und ist dabei plattformunabhängig. Havemann ließ sich bei der Entwicklung der GML von PostScript inspirieren und wollte eine Sprache mit ähnlicher Struktur entwerfen, die für die Beschreibung von dreidimensionalen Objekten und ihre automatische Erzeugung durch einen GML-Interpreter geeignet ist. Während bei PostScript das Ergebnis der Interpretation eine gedruckte Seite oder ein am Computer für menschliche Nutzer lesbares Dokument ist, soll der GML-Interpreter dreidimensionale Objekte basierend auf dem GML-Programmcode erzeugen.

Die Kernidee dieses Ansatzes ist es, dreidimensionale Objekte nicht mehr durch das Ergebnis eines Modellierungsprozesses in Form von Vertices, Kanten und Oberflächen zu beschreiben, sondern stattdessen den Erzeugungsprozess zu protokollieren. Dies erlaubt die Reproduktion des jeweiligen Modells durch Interpretation der Modellierungsschritte ausgedrückt in der GML. Havemann spricht hier von einem „Paradigm shift from objects to operations“ [Ha05]. Um diesen zu erreichen, muss die GML die erforderlichen Operatoren zur Erzeugung beliebiger Objekte zur Verfügung stellen. Diese Operatoren können anschließend verwendet werden, um Objekte direkt innerhalb der Sprache zu programmieren. Dabei greift der GML-Interpreter auf C++-Implementationen der vorhandenen Operatoren zurück, dies gilt sowohl für die Low-Level-Euler-Operatoren als auch die komplexeren High-Level-Operatoren wie die erwähnte Pfad-Extrusion. Dadurch reduziert sich aus technischer Sicht die Aufgabe des Interpreters darauf, Daten von einer Funktion in die nächste zu transferieren und die Ergebnisse der jeweiligen Operationen für die Weiterverarbeitung bereitzuhalten.

Die Struktur der GML selber ist vergleichsweise einfach. Jede Eingabe in den Interpreter wird von diesem zunächst tokenisiert, also in eine Reihe von Einzelteilen zerlegt. Anschließend werden die unterschiedlichen Tokens verarbeitet. Dabei unterscheidet man zwischen Literalen und ausführbaren Befehlen. Die Bezeichner ausführbarer Befehle werden innerhalb einer *Dictionary-Datenstruktur* abgelegt und ausgeführt, sobald sie innerhalb des

[13] Adobe Systems Inc.: http://www.adobe.com/

GML-Programmcodes auftauchen. Die Berechnungen selber werden durch die C++-Implementation vorgenommen, der unter anderem eine Referenz auf den aktuellen Stack des GML-Interpreters übergeben wird. Dadurch können die Methoden ihre benötigten Parameter direkt vom Stack nehmen und ihre Berechnungsergebnisse auf diesem ablegen. Dabei werden Operatoren und Operanden in der *Postfix-Notation* angegeben, die ohne Klammerungen bei arithmetischen Ausdrücken auskommt. Für die Operanden stehen innerhalb der GML eine Reihe vorgefertigter Datentypen zur Verfügung. Dazu gehören die Standarddatentypen Integer, Float und String. Weiterhin existieren Datentypen zur Verwaltung zwei- und dreidimensionaler Vektoren sowie Typen zur Speicherung von Bezeichnern und Container wie Dictionarys oder Arrays. Die Typenmenge kann bei Bedarf auf der Ebene der C++-Implementation erweitert werden.

Bezüglich des integrierten Array-Datentyps unterscheidet die GML zwischen *ausführbaren* und *literalen* Arrays. Hierfür stehen zwei unterschiedliche Marker zur Verfügung. Eckige Klammerpaare definieren literale Arrays. Sobald eine schließende eckige Klammer als Token verarbeitet wird, sucht der Interpreter die zugehörige öffnende Klammer. Alle dazwischen liegenden Tokens werden vom Stack gepoppt und innerhalb eines literalen Arrays gespeichert. Analog geht der Interpreter mit ausführbaren Arrays um, diese werden allerdings durch geschweifte Klammern definiert. Solche Arrays stellen den Mechanismus dar, mittels dessen Funktionen innerhalb der GML deklariert werden, da die Ausführung eines ausführbaren Arrays die Ausführung sämtlicher innerhalb des Arrays gespeicherten Tokens bedeutet. Kommen innerhalb eines solchen Arrays literale Tokens vor, werden diese auf den Stack gepushed, bei ausführbaren Tokens wird dagegen deren verbundene Funktion ausgeführt.

Die GML folgt dem Paradigma funktionaler Programmierung und stellt somit die Funktion in das Zentrum ihres Sprachkonstrukts. Dabei spielen die vorab erläuterten ausführbaren Arrays eine zentrale Rolle, da sie das Werkzeug zur Definition von Funktionen darstellen. Darüber hinaus bietet die Sprache die Möglichkeit, Arrays vom einen in den jeweils anderen Typ zu konvertieren, also beispielsweise aus einem literalen ein ausführbares Array zu machen. Verwaltet wird dies durch die Verwendung eines Flags, das ein Array als ausführbar kennzeichnet. Dieses Sprachkonstrukt bietet mächtige Möglichkeiten im Umgang mit Arrays und Funktionen. So ist es beispielsweise möglich, Arrayoperationen auch auf Funktionen anzuwenden und diese dadurch zur Laufzeit zu verändern. Ein Beispiel für eine solche Operation wäre das Anhängen eines Arrays an ein anderes, auf

Funktionsebene entspräche dies dann der Verkettung zweier vorab unabhängiger Funktionen.

Ein zentrales Sprachfeature der GML sind die bereits erwähnten Dictionaries, die intern durch *Maps* umgesetzt werden, innerhalb derer eine beliebige Anzahl von *Key-Value-Paaren* gespeichert werden können. Dictionaries können während der Laufzeit erzeugt und modifiziert werden und dienen unter anderem der Umsetzung eines Scoping-Ansatzes. Dazu verwaltet der Interpreter einen Dictionaries-Stack für das *Look-Up* von Variablen oder Funktionsnamen. Wird ein ausführbares Token verarbeitet, sucht der Interpreter nach diesem immer innerhalb des Dictionaries, das aktuell auf dem Dictionary-Stack als oberstes Element gespeichert ist. Dadurch können Konzepte wie lokale Variablen umgesetzt werden. Weiterhin können Dictionaries auch für die Umsetzung objektorientierter Programmierung eingesetzt werden, da sie in der Lage sind, Daten und Methoden zusammenzufassen und dadurch Klassen im Sinne der OOP zu realisieren. Im Vergleich zu objektorientierten Programmiersprachen wie C++ besitzt die GML hier allerdings eine größere Flexibilität, da sie die Modifikation der Klassen zur Laufzeit durch Veränderung der Dictionaries ermöglicht.

Neben den erläuterten Sprachkonstrukten enthält die GML auch Konstrukte für die Flusskontrolle von GML-Programmen. Dazu gehören verschiedene Varianten konditionaler Ausdrücke, die Programmteile nur dann ausführen, wenn eine festgelegte Bedingung erfüllt ist sowie verschiedene Schleifenkonstrukte, die eine wiederholte Ausführung von Programmteilen bis zum Eintreten eines Abbruchkriteriums ermöglichen.

5.3.3.4 **Anwendungsbeispiele und Diskussion der GML**

Nachdem die wichtigsten Merkmale der GML erläutert wurden, sollen abschließend Beispiele für den Einsatz des Systems gezeigt und anschließend seine Anwendbarkeit für das Ziel dieser Arbeit erörtert werden.

Allen prozeduralen Ansätzen und somit auch der GML ist gemein, dass sie ihre Stärken in solchen Modellierungsbereichen besitzen, in denen sich die Struktur der modellierten Objekte durch Regeln beschreiben lässt, die also eine wie auch immer geartete Regelmäßigkeit besitzen. Havemann wählt darum in seiner Arbeit als *Proof-of-Concept* die Modellierung gotischer Architektur mittels der GML.

Laut Havemann zeichnet sich diese dadurch aus, dass sie teilweise hochkomplexe Strukturen enthält, deren Konstruktion aber durch die Kombination einer relativ geringen Anzahl von Grundelementen erfolgt [Ha05]. Dies zeigt sich gut am Beispiel gotischer Fenster, bei denen sämtliche Freiformflächen ausschließlich aus Kreisen und Kreissegmenten bestehen. Die Modellierung solcher Fenster mittels GML erfordert zunächst deren Analyse. Ziel ist die Extraktion vorhandener Muster innerhalb der Konstruktion, die effizient mittels der GML-Sprachfeatures ausgedrückt werden können.

Abbildung 38: Gotische Fenster mit unterschiedlichen Rosetten-Stylezuweisungen [Ha05]

Abbildung 38 zeigt verschiedene gotische Fenster, die mittels GML erstellt wurden. Die einzelnen Fenster unterscheiden sich dabei in Bezug auf den *Style*, mittels dessen die Rosettendekoration erzeugt wurde. Diese Stylefestlegung ist ein komplexer globaler Parameter, den man bei der Konstruktion angeben kann und der auf eine ebenfalls in der GML formulierte Stylebibliothek zugreift. Abbildung 39 zeigt den GML-Code, der für die Erzeugung der farbigen Flächen innerhalb des ersten Fensters in Abbildung 38 eingesetzt wird. Durch die Materialzuordnungen mittels des GML-Befehls `setcurrentmaterial` ist die Zuordnung der einzelnen Codesegmente zu den unterschiedlichen Teilen der erzeugten Geometrie leicht möglich.

Anhand dieses Beispiels lassen sich die Vor- und Nachteile des Ansatzes gut verdeutlichen. Der größte Vorteil besteht in der Erweiter- und Wierderverwendbarkeit des GML-Codes. Durch den Einsatz von Bibliotheken, die problemlos in den GML-Interpreter geladen werden können, ist es möglich, einmal erstellte GML-Programme in anderen Kontexten erneut einzusetzen. Diese Wiederverwendbarkeit von Code ist im Bereich der Softwareentwicklung ein wichtiges Qualitätskriterium von Softwaresystemen. Je höher die

Wiederverwendbarkeit, desto höher auch die Qualität des erstellten Codes. Der Grund liegt in der signifikanten Reduktion der Entwicklungszeit neuer Anwendungssysteme, die durch die Wiederverwendung bereits entwickelter und getesteter Codebestandteile erreicht wird. Genau hierin lag das bereits vorab mehrfach diskutierte Problem im Bereich der 3D-Modellierung. Der Einsatz von Modellierungswerkzeugen wie 3ds Max bietet kaum Möglichkeiten zur Wiederverwendung und zur nachträglichen Modifikation erstellter Modelle.

Hier sind prozedurale Ansätze wie die GML deutlich besser geeignet, da die Veränderung des Codes zu direkten Änderungen des daraus erzeugten Modells führt. Allerdings steigert dieser Ansatz auch die Komplexität der Modellerzeugung, da der Nutzer nicht mehr länger in einer Modellierungsumgebung seine Modelle mittels vorgegebener Werkzeuge erstellt. Vielmehr programmiert er diese in der GML, die dann aus dem Programmcode die beschriebenen Modelle automatisch erzeugt. Darin zeigt sich ein großer Nachteil dieses Ansatzes in Bezug auf die Zielsetzung des Semantic Building Modelers. Diese besteht in der Entwicklung eines Systems, das unerfahrenen Nutzern mit vergleichsweise einfachen Mitteln die Möglichkeit bietet, Gebäude unterschiedlichen Typs automatisch zu erzeugen. Somit ist der Anwendungsbereich a priori kleiner als der der GML, die in der Lage ist, beliebige Arten von 3D-Objekten zu modellieren und nicht auf den Bereich der Gebäudeerzeugung beschränkt ist. Der Vorteil einer Spezialisierung auf einen bestimmten Anwendungsbereich besteht dabei in der Möglichkeit, für diesen Anwendungsbereich relevante Werkzeuge zu implementieren und diese dem Nutzer zur Verfügung zu stellen.

```
Gothic-Window.Styles.Style-1 begin

{ usereg !edgeBack !edgeWall
        !wallSetback !bdOuter !poly

  /stdGrey setcurrentmaterial
  :poly 5 poly2doubleface dup edgemate
  :edgeWall killFmakeRH
  :bdOuter 4 div :wallSetback neg 5 vector3
  extrude
} /style-main-arch exch def

{ usereg !edgeBack !edgeWall !poly
  /stdGreen setcurrentmaterial
  :poly 5 poly2doubleface !edge
  /gold setcurrentmaterial
  :edge edgemate :edgeWall killFmakeRH
  [ :edge ] [ (0.05,-0.3,1) ]
  extrudestable pop
} /style-fillet exch def

{ usereg !rad !mid !edgeBack !edgeWall !poly
  /stdRed setcurrentmaterial
  :poly 5 poly2doubleface !edge
  :edge edgemate :edgeWall killFmakeRH
  /gold setcurrentmaterial
  :edge (0.05,-0.3,5) extrude !edge
} /style-rosette exch def

{ usereg !bh !arcL !edgeBack !edgeWall !poly
  /stdBlue setcurrentmaterial
  :poly 5 poly2doubleface !edge
  :edge edgemate :edgeWall killFmakeRH
  /gold setcurrentmaterial
  :edge (0.05,-0.3,5) extrude !edge
} /style-sub-arch exch def

end
```

Abbildung 39: GML -Code für die Definition von Style 1 [Ha05]

Dadurch können die Operationen auf einem höheren Abstraktionslevel angesiedelt werden, im Kontext der Gebäudebaus beispielsweise durch Methoden wie `computeRoof` oder

`addWindows`, die eine Reihe komplexer Bearbeitungsschritte durchführen. Innerhalb des vorliegenden Systems werden solche Aufgaben direkt in Form von Java-Methoden implementiert und stehen dann für unterschiedliche Gebäudetypen zur Verfügung. Die GML ermöglicht durch ihre Sprachkonstrukte ebenfalls die Formulierung solcher Routinen, bei Bedarf können Algorithmen auch direkt innerhalb des C++-Kerns erstellt und anschließend vom GML-Interpreter aufgerufen werden. Somit ist die GML bezüglich ihrer Mächtigkeit dem vorliegenden System überlegen, da die Menge der durch sie erzeugbaren Objekte größer ist. Allerdings ist der große Nachteil, den sich die GML durch ihre größere Ausdrucksstärke erkauft, die Komplexität, die mit dieser einhergeht. Es ist davon auszugehen, dass die Einarbeitung in die GML für einen unerfahrenen Nutzer eine noch größere Hürde darstellt als die Auseinandersetzung mit einer Modellierungsumgebung wie 3ds Max oder SketchUp.

Hierfür spielen zwei Faktoren eine wichtige Rolle. Zum einen muss der Nutzer die Prinzipien einer Programmiersprache verinnerlichen, was ohne vorherige Programmierkenntnisse eine große Schwierigkeit darstellt. Zum anderen steht der Anwender auch hier vor dem Problem, sich die Auswirkungen geometrischer Operationen auf dreidimensionale Körper vorzustellen, die er nicht direkt vor sich sieht. Dies erfordert eine gute räumliche Vorstellung und die Fähigkeit, den geschriebenen Code in Zusammenhang mit dem erzeugten Modell zu betrachten. Ein ähnliches Problem haben auch die erläuterten regelbasierten Verfahren. Auch bei diesen besteht eine große Hürde für den Nutzer darin, sich die Auswirkungen bestimmter Regeln auf das finale Ergebnis vorzustellen, um diese derart zu formulieren, dass sie seiner Zielsetzung entsprechen. Für Nutzer ohne Vorkenntnisse scheint die GML in ihrer aktuellen Struktur somit ungeeignet, da das Erlernen der Sprache und ihrer Konzepte zu schwierig ist. Zwar liegen ihre Vorteile auf der Hand, konkret löst sie das Problem der Wiederverwendbarkeit von Modellierungsprozessen und ermöglicht dadurch erst den Aufbau von Bibliotheken, trotzdem müssen solche Bibliotheken erst erstellt werden. Hier muss sich eine ausreichend große Nutzerschaft finden, die von bewährten Lösungsansätzen beispielsweise im Bereich des CAD abrückt und stattdessen die GML für die Modellsynthese verwendet. Erst, wenn es ausreichend Nutzer eines solchen Systems gibt, ist auch davon auszugehen, dass die Entwicklung größerer Online-Bibliotheken mit GML-Code und –Modellen vorangetrieben wird und sich die Vorteile der Codewiederverwendung positiv auf die Modellentwicklung auswirken.

Dabei ist festzuhalten, dass Code nicht allein durch den Einsatz einer Programmiersprache wie der GML automatisch wiederverwendbar wird. Wie bei der Entwicklung von Anwendungsprogrammen ist eine sorgfältige Planung des Systems erforderlich. Das wichtigste Merkmal wiederverwendbarer Komponenten ist dabei die Verwendung vereinbarter und im Idealfall unveränderlicher Schnittstellen. Diese ermöglicht die Integration externer Softwarebibliotheken in das eigene System, sofern man sich an die Schnittstellenvereinbarungen hält. Für einen Nutzer ohne Erfahrung im Bereich der Softwareentwicklung und des -entwurfs ist die erfolgreiche Planung eines komplexen Systems eine nicht zu bewältigende Aufgabe. Darum ist davon auszugehen, dass GML-Code, der von solchen Nutzern verfasst wird, wenn überhaupt, dann nur in Auszügen wiederverwendbar ist. Dadurch scheint die GML primär für Spezialisten geeignet, die sowohl im Bereich der Softwareentwicklung als auch der 3D-Computergrafik bewandert sind. Hierin sieht auch Havemann einen potentiellen Anwendungsbereich [Ha05]. In einem solchen Kontext könnte sich ein Markt entwickeln, bei dem sowohl Erweiterungen der GML auf der Seite der C++-Implementation in Form von Codebibliotheken als auch in der GML verfasste Modelle angeboten und verkauft werden. Allerdings erfordert auch die Entwicklung eines solchen Marktes zunächst die allgemeine Akzeptanz der Technologie und die Entwicklung benutzerfreundlicher Anwendungssysteme, die den Nutzer bei der Erstellung des GML-Codes unterstützen und dadurch die Arbeit erleichtern. Solange dies nicht der Fall ist, wird sich die GML auch nicht als neues Modellierungsparadigma durchsetzen und primär eine Nischentechnologie bleiben.

Dies wird deutlich, wenn man die aktuelle Situation im Bereich der Modellierungssysteme betrachtet. Modelle, die in einer Modellierungsumgebung wie 3ds Max erstellt wurden, besitzen zwar nicht die nachträgliche Modifizierbarkeit von GML-Modellen, dafür sind die Einstiegshürden speziell für Nutzer ohne Vorkenntnisse im Bereich der Programmierung tendenziell niedriger. Dies soll nicht die vorab genannten Nachteile von Modellierungssoftware relativieren, auch dort ist der Einarbeitungsaufwand nicht zu unterschätzen. Die Erstellung komplexer Modelle erfordert viel Erfahrung und eine gute Kenntnis der zur Verfügung stehenden Werkzeuge. Dies ist allerdings auch in der GML der Fall. Hier stehen die Werkzeuge in Form der zur Verfügung gestellten Operationen bereit. Die von Havemann exemplarisch implementierten Funktionen wie die bereits erwähnten Extrusionsvarianten existieren auch in Modellierungswerkzeugen und können vom Nutzer

direkt eingesetzt werden. In der GML erfolgt der Aufruf allerdings in einer stackbasierten Programmierumgebung, was für viele Nutzer zumindest ungewohnt sein dürfte.

Aufgrund der großen Verbreitung von Modellierungswerkzeugen existieren weltweit unzählige 3D-Modelle, die online verfügbar und für eigene Modelle einsetzbar sind. Der Semantic Building Modeler setzt darum auf einen kombinierten Ansatz. Er implementiert eine Reihe zentraler Schritte bei der Erzeugung eines Gebäudes in Form vorgefertigter Funktionen, bietet dem Nutzer aber gleichzeitig die Möglichkeit, durch Konfigurationsparameter und die Bereitstellung weiterer 3D-Modelle eine größere Gebäudevielfalt zu erzeugen und das System sukzessive zu erweitern, ohne eine Zeile Code schreiben zu müssen. Dieser Ansatz versucht, die Wiederverwendbarkeit auf die Ebene von Gebäudekomponenten zu verschieben. Das Modell eines gotischen Fensters kann durch einen geeigneten Algorithmus in verschiedene gotische Gebäude integriert werden, gleiches gilt für eine Reihe weiterer Gebäudebestandteile. Im Idealfall muss ein Nutzer des Systems weder modellieren noch programmieren. Er greift entweder auf die bereits vorhandene Modellbibliothek zurück oder erweitert diese durch online verfügbare Modelle und steigert dadurch automatisch auch die Vielfalt der Gebäude, die durch das System berechnet werden. Konzeptuell soll hier also eine Synthese der Vorteile beider Varianten erreicht werden, indem wiederkehrende Arbeiten bei der Gebäudeerzeugung wie die Positionierung von Fenstern oder der Bau von Dächern dem Nutzer durch parametrisierte Algorithmen abgenommen werden, er aber durch die Bereitstellung weiterer Modelle die Vielfalt der erzeugten Gebäude steigern kann. Fortgeschrittene Nutzer mit Programmierkenntnissen sind darüber hinaus in der Lage, das System analog zur GML durch die Implementation eigener Algorithmen zu erweitern und dadurch den Funktionsumfang sukzessive zu vergrößern.

Trotz der Tatsache, dass sie für das vorliegende Ziel nicht geeignet erscheint, wurde die GML sehr detailliert erläutert, da sie einen vergleichsweise neuen Ansatz für die Erstellung von 3D-Modellen darstellt. Dieser von Havemann postulierte Paradigmenwechsel weg vom Objekt hin zu den Operationen, die das Objekt erstellen [Ha05], scheint in diesem Zusammenhang ein durchaus fruchtbarer Ansatz zu sein. Speziell die Zerlegung der einzelnen Schritte bei der Erzeugung eines Gebäudes und die Identifikation von Konstruktionsgemeinsamkeiten können hier zielführend sein. Die Implementation wiederkehrender Konstruktionsschritte wie der Positionierung von Fenstern oder der Erzeugung von Erkern stellt im vorliegenden System einen ähnlichen Ansatz dar. Auch die Erweiterbarkeit durch die Implementation weiterer Methoden und Algorithmen ist in beiden

Systemen möglich. Havemann nennt in seiner Arbeit eine Reihe möglicher Anwendungsfelder für die GML, unter anderem als mögliches allgemeines Dateiformat für die nicht-proprietäre Speicherung von 3D-Modellen. In diesem Zusammenhang bezeichnet er die GML als potentiellen Nachfolger der *Virtual Reality Modeling Language* (VRML) und diskutiert die Vorteile der GML gegenüber XML-Dokumenten in Kombination mit *Extensible Stylesheet Language Transformations* (XSLT). Hier sieht er die Stärken der GML in ihrer Fähigkeit, nicht nur Daten, sondern auch Operationen zu spezifizieren, die auf diese Daten angewendet werden, während XML nur für die Datenrepräsentation geeignet sei [Ha05].

Zusammenfassend ist die GML ein sehr fruchtbarer Ansatz. Die Verschiebung der Modellierung weg von Objekten hin zu den erzeugenden Operationen ist aus theoretischer Sicht eine sinnvolle Entwicklung. Sie löst vorab diskutierte Probleme in Bezug auf Wiederverwend- und nachträgliche Modifizierbarkeit einmal erstellter Modelle, allerdings stellt sich die Frage nach der Praxistauglichkeit des Ansatzes, da die großen Vorteile mit einer großen Komplexität erkauft werden. Ob die Vorteile diese Komplexität übersteigen, hängt sowohl vom konkreten Anwendungsfall als auch vom konkreten Nutzer ab. Speziell die Wiederverwendbarkeit von Modellcode durch den Aufbau umfangreicher Bibliotheken wird sich erst dann positiv auswirken, wenn die Nutzergruppe zunimmt und gleichzeitig bereit ist, ihre Ergebnisse mit anderen Nutzer zu teilen und diesen zur Verfügung zu stellen. Mit steigender Verbreitung des Ansatzes werden auch Synergieeffekte immer größer werden, so dass die GML zu einer signifikanten Reduktion der Entwicklungszeiten komplexer 3D-Modelle beitragen kann.

In Bezug auf die vorab eingeführte Unterscheidung zwischen prozeduralen und regelbasierten Ansätzen ist die GML den prozeduralen Verfahren zuzuordnen, da der Nutzer unter Verwendung der stackbasierten Programmiersprache die einzelnen Konstruktionsschritte spezifiziert. Die Gemeinsamkeit zu regelbasierten Verfahren wie sie beispielsweise in der CityEngine eingesetzt werden, besteht in der Tatsache, dass beide Systeme intern auf Verfahren zurückgreifen, die in einer Hochsprache implementiert sind. Während dieser Zugriff bei der CityEngine im Rahmen des Formalismus von Ersetzungssystemen erfolgt, greift die GML auf eine Scriptsprache zurück, in der die einzelnen Konstruktionsschritte festgelegt werden. Durch das Vorhandensein von Kontrollstrukturen, Schleifenkonstrukten und verschiedenen Datentypen können in der GML auch komplexe Algorithmen umgesetzt werden, die intern auf die zur Verfügung

stehenden C++-Methoden abgebildet werden. Wiederum geht mit dieser größeren Freiheit auch eine größere Komplexität einher. Als Beispiel kann man die Umsetzung wiederholt durchzuführender Aktionen wie die Unterteilung von Fassaden oder Positionierung von Modellen in den Gebäuden betrachten. In der GML tut man dies durch die Verwendung von Schleifen, in Ersetzungssystemen spezifiziert man eine Regel, die wiederholte Anwendung wird durch den Formalismus umgesetzt. Dadurch ist die Realisierung in Ersetzungssystemen konzeptuell als einfacher zu bewerten, da man mit der Formulierung einer Regel auskommt, während man in der GML auf Programmierkonzepte zurückgreift.

Die nachfolgend vorgestellte Arbeit stellt ein rein prozedurales Verfahren dar, das in seiner Struktur dem Semantic Building Modeler am ähnlichsten ist. In der Dissertation von Dieter Finkenzeller [Fi08] entwickelt er das System *ProcMod*, das parametrisierte Algorithmen für die Gebäudeerzeugung einsetzt, denen der Nutzer die erforderlichen Parameter auf unterschiedliche Arten zur Verfügung stellt. Basierend auf den Eingaben erzeugt das System anschließend einzelne Gebäude. Das nachfolgende Kapitel stellt die einzelnen Algorithmen und Konzepte vor, mittels derer Finkenzeller Gebäude erzeugt.

5.3.4 **ProcMod [Fi08]**

5.3.4.1 **Definition / Spezifikation der Grundrisse**

In Finkenzellers System werden Grundrisse aus einzelnen Grundrissmodulen zusammengesetzt, die auf zwei unterschiedliche Arten miteinander verbunden sind. Ein Grundrissmodul ist ein konvexes Polygon, das zusätzlich zu seinen geometrischen Daten weitere Informationen speichert. Dazu gehören unter anderem der architektonische Typ des Moduls (beispielsweise Risalit oder Balkon) sowie weitere Angaben wie Höhe, Wanddicke oder Material. Die Verbindung zweier Module kann über *Punkt-Punkt-* (PP-) oder *Kante-Kante-Verbindungen* (KK-Verbindungen) hergestellt werden. Bei einer PP-Verbindung werden zwei Grundrissmodule derart miteinander verbunden, dass eine neue Kante zwischen diesen aufgebaut wird. Diese Kante kann sowohl bei Eckpunkten der Module, als auch auf den vorhandenen Kanten ansetzen, darf allerdings keines der beteiligten Module und keine andere PP-Verbindung schneiden. Da sich der finale Grundriss durch das Umlaufen der Kanten der Grundrissmodule ergibt, erzeugt das Hinzufügen einer neuen PP-Verbindung auch eine neue Kante in der Umrisslinie.

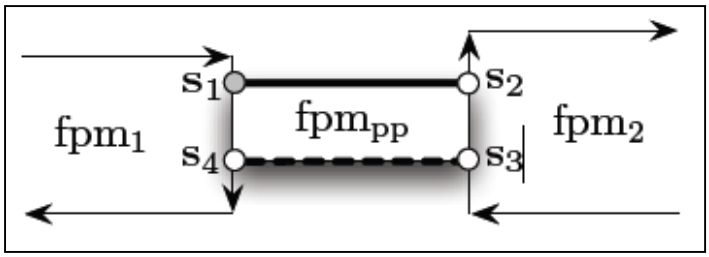

Abbildung 40: Punkt-Punkt-Modul [Fi08]

Abbildung 40 zeigt ein Beispiel für einen solchen Fall. Zu sehen sind zwei Grundrissmodule fpm_1 und fpm_2. Diese werden über zwei PP-Verbindungen (zwischen den Punkten s_1 und s_2 sowie zwischen den Punkten s_3 und s_4) zusammengefasst. Bei der Ermittlung der Umrisslinie entstehen durch diese Verbindungen zwei neue Kanten.

Der zweite verwendete Verbindungstyp sind Kante-Kante-Verbindungen. Dieser Typ verbindet zwei Module über eine gemeinsame Kante. Eines der beteiligten Module wird als dominantes Modul bezeichnet, an das das andere Modul andockt. Kanten eines dominanten Moduls können mehrere KK-Verbindungen zu anderen Modulen aufbauen. Allerdings dürfen sich die Intervalle der andockenden Module bezüglich der Kantenlänge nicht überschneiden. Diese beiden Verbindungstypen können miteinander kombiniert werden. So ist es möglich, die neu eingefügte Kante einer PP-Verbindung als gemeinsame Kante einer KK-Verbindung zu definieren.

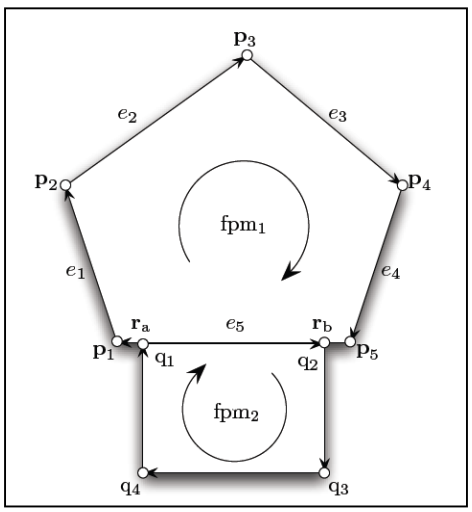

Abbildung 41: Beispiel eines Grundrisses bestehend aus zwei Modulen [Fi08]

Ein Grundriss ist definiert als eine Menge von Grundrissmodulen *M,* die über eine Menge von Verbindungen *C* miteinander verknüpft sind. Abbildung 41 zeigt einen beispielhaften

Grundriss bestehend aus zwei Modulen, die über eine Kante-Kante-Verbindung miteinander verbunden sind. Bei der eigentlichen Erzeugung der Grundrisslinie aus den Modulen und ihren Verbindungen wird der entstehende Polygonzug in einzelne Kantenzüge untergliedert. Wie diese Unterteilungen vorgenommen werden, hängt ab von der Struktur der Module und der Art, wie diese miteinander verbunden sind. Ein einzelner Kantenzug wird als *Feature* bezeichnet. Abbildung 42 zeigt drei über KK-Verbindungen zusammengefasste Module. Die eingefärbten Kanten stellen jeweils einzelne Features des erzeugten Grundrisses dar. Features bestehen nur aus Außenkanten der Grundrissmodule, diese bilden später die Wände des Gebäudes. Die Ermittlung der Features ist abhängig von den Modulen und der Art ihrer Verbindung. Insgesamt existieren neun verschiedene Konfigurationen, aus denen jeweils unterschiedliche Merkmale abgeleitet werden.

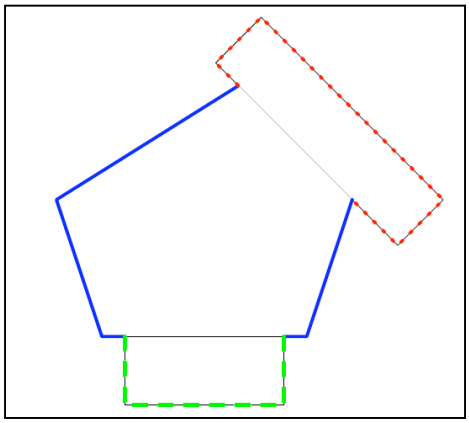

Abbildung 42: Unterteilung des Grundrisses in Kantenzüge [Fi08]

Basierend auf einem derart strukturierten Grundriss ist es möglich, neue Grundrisse für weitere Stockwerke zu erzeugen. Dabei können sowohl neue Module hinzugefügt, als auch alte Module auf höheren Stockwerken entfernt werden. Soll ein Grundriss auf den nachfolgenden Stockwerken wiederverwendet werden, so sind keine weiteren Schritte erforderlich. Auch die Bestimmung der Features muss in einem solchen Fall nicht wiederholt werden. Für Variationen eines Ausgangsgrundrisses auf nachfolgenden Stockwerken wird dieser zunächst unterteilt. Seine „Module bzw. deren Unterteilungen bilden die Grundlage der Module des neuen Stockwerks“ [Fi08]. Die Erzeugung neuer Module erfolgt durch die Unterteilung aller vorhandenen Kanten des Ausgangsgrundrisses. Diese Unterteilung erzeugt eine Menge neuer Punkte, die anschließend durch das Einfügen zusätzlicher Kanten miteinander verbunden werden.

Sofern sich diese Kanten wechselseitig schneiden, werden auch die Schnittpunkte zur Punktemenge hinzugefügt. Dieses Unterteilungsverfahren erzeugt eine Menge konvexer Polygone. Für die Erzeugung neuer Module für das folgende Stockwerk werden allerdings nur solche Polygone verwendet, die durch keine weitere Kante unterteilt werden. Aus diesen Polygonen können dann neue Module für das nachfolgende Stockwerk erzeugt werden, wobei nur solche Kombinationen gültig sind, die konvexe Module erzeugen. Abbildung 43 zeigt ein vereinfachtes Beispiel eines unterteilten Grundrisspolygons. Dieses ist definiert durch die Punkte p_1, p_2, p_3, p_4, p_5. Durch die Unterteilungen entstehen die neuen Punkte q_1, q_2, q_3 und q_4.

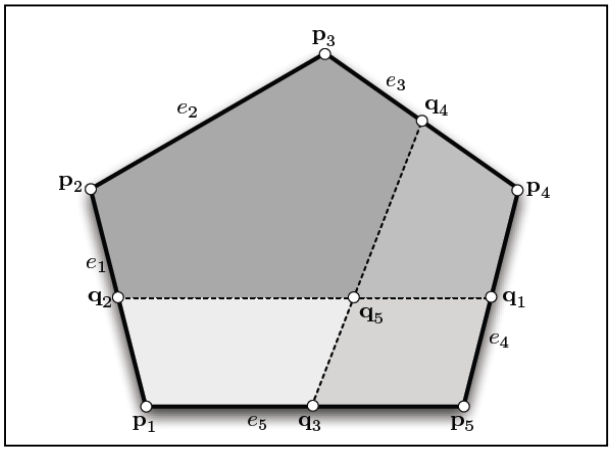

Abbildung 43: Unterteiltes Grundrisspolygon [Fi08]

Diese Unterteilung ermöglicht die Erzeugung von neun unterschiedlichen Modulen durch die Kombination der unterteilten Polygone. Die einzige Einschränkung, die für eine solche Kombination existiert, ist die Überlappungsfreiheit der daraus entstehenden Module.

Auch die derart erzeugten Module verfügen über Verbindungen zu weiteren Modulen. Diese werden in *interne* und *externe* Verbindungen unterteilt. Externe Verbindungen betreffen Kanten, die aus einer Kante des ursprünglichen Moduls hervorgehen. Weitere Module, die über diese Kante angebunden werden, liegen außerhalb des Polygons des Ausgangsmoduls. In Abbildung 43 sind dies die fett gezeichneten Begrenzungskanten. Demgegenüber sind interne Verbindungen solche, die über Kanten erfolgen, die durch die Unterteilung des Moduls entstanden sind, also innerhalb des Ausgangspolygons verlaufen. Diese sind in Abbildung 43 gestrichelt eingezeichnet.

Neben der Unterscheidung der Verbindungen existieren auch für Stockwerke unterschiedliche Arten von Merkmalen. Der erste Merkmalstyp sind freie Außenkanten. Diese definiert Finkenzeller wie folgt:

„Freie Außenkanten verlaufen an der Außenseite eines Stockwerks und gehören freien Flächen an. Freie Flächen sind Umrisslinien auf einem Stockwerk, die nicht mit Modulen für das nächsthöhere Stockwerk belegt sind“ [Fi08]

Innenkanten stellen einen weiteren Merkmalstyp dar, bei dem die Kantenzüge im Inneren eines Stockwerks verlaufen. Polygone, die durch verbundene Kantenzüge des Typs „Innenkante“ und des Kantentyps „freie Außenkante“ definiert sind, begrenzen freie Flächen. Externe Verbindungen sind solche Modulverbindungen, die über Außenkanten des Ursprungsmoduls erfolgen. Solche Verbindungen werden verwendet, wenn neue Module zu einem Grundriss hinzugefügt werden, beispielsweise um einen Erker an einem Stockwerk anzubringen. Der letzte Merkmalstyp sind die sogenannten Nahtstellen. Diese definiert Finkenzeller als „Außenkanten, an welchen zwei Stockwerke bzw. die zugehörigen Module aneinander stoßen“[Fi08].

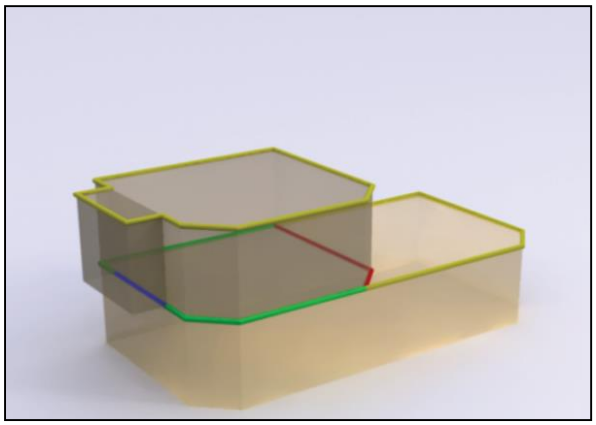

Abbildung 44: Unterschiedliche Arten von Stockwerksmerkmalen [Fi08]

Abbildung 44 zeigt die unterschiedlichen Arten der Stockwerksmerkmale. Der Grundriss des oberen Stockwerks wurde aus dem darunterliegenden Grundriss erzeugt. Dabei wurden sowohl Module des Ausgangsgrundrisses entfernt, als auch ein neues Modul hinzugefügt. Innerhalb der Grafik sind freie Außenkanten gelb gekennzeichnet, Innenkanten rot, externe Verbindungen blau sowie Nahtstellen grün. Man erkennt, wie das Polygon, bestehend aus freien Außenkanten und der einzigen vorhandenen Innenkante auf dem unteren Grundriss eine freie Fläche begrenzt. Weiterhin wird die einzige vorhandene externe Verbindung

genutzt, um ein neues Modul anzuschließen, das zur Erzeugung eines Erkers verwendet wird.

5.3.4.2 **Dachformrepräsentation**

Die Erzeugung von Dächern ist ein wichtiger Aspekt für die automatisierte Erstellung realistischer Gebäude. Finkenzeller wählt einen Ansatz, bei dem Dächer für jedes einzelne Grundrissmodul des obersten Stockwerks ermittelt werden. Dieser Ansatz ermöglicht die Verwendung eines allgemeinen Berechnungsverfahrens, das für die Bestimmung des Dachfirsts eingesetzt wird. Die Idee besteht darin, auf zwei Polygonkanten e_i bzw. e_j zwei Punkte q_{e_i} bzw. q_{e_j} zu berechnen, deren genaue Position auf der Kante durch zwei Parameter c bzw. d bestimmt wird. Diese beiden Punkte werden dann durch eine neue Kante miteinander verbunden. Die Kante wird über zwei Parameter a bzw. b parametrisiert. In Kombination mit der Dachhöhe h berechnet man die Punkte r_a bzw. r_b, die den First des berechneten Dachs definieren. Abbildung 45 zeigt die Konstruktion der einzelnen Punkte über die Ausganskante. Durch Variation der verwendeten Parameter können mit Hilfe dieses Berechnungsverfahrens verschiedene Dachformen erzeugt werden. Setzt man beispielsweise $a = 0$ und $b = 1$, so wird ein Satteldach berechnet, dessen Dachneigung eine Funktion der Höhe ist. Weitere über dieses Verfahren erzeugbare Dachtypen sind das Walmdach, das Pultdach sowie das Zeltdach.

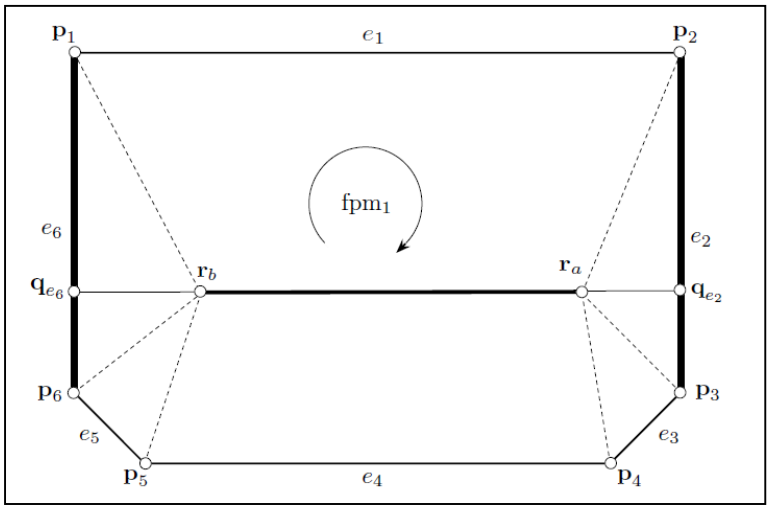

Abbildung 45: Konstruktion eines Daches für ein Grundrissmodul [Fi08]

Dieses Berechnungsverfahren berechnet Dächer unterschiedlicher Typen für jedes Modul eines Grundrisses. Nach Abschluss dieser Berechnung überprüft das System, ob die Dächer der einzelnen Module miteinander verbunden werden können. Voraussetzung hierfür ist das Vorhandensein einer Verbindung zwischen den beiden Modulen. Liegt eine solche Verbindung vor, wird getestet, ob die Dächer miteinander verschmolzen werden können oder nicht. Dies ist genau dann der Fall, wenn die Verbindungskante vom First eines der beiden Dächer geschnitten wird und die Firstkante das Dach des jeweils anderen Daches trifft. In diesem Fall verlängert man den First bis zu seinem Schnittpunkt mit dem anderen Dach und erzeugt ein einzelnes Dach aus den beiden Quelldächern.

5.3.4.3 **Fassadenbeschreibung**

Der erste Schritt nach der Grundrissbeschreibung ist die Erzeugung von Wänden basierend auf den Merkmalen des Stockwerksgrundrisses. Finkenzeller unterscheidet verschiedene Arten von Wänden, Basiswände beispielsweise werden für jede einzelne Merkmalskante erzeugt. Auch Hausecken werden direkt aus den Umrisspolygonen abgeleitet und an allen konvexen Ecken erstellt.

Basiswände können anschließend horizontal in Zwischenwandelemente unterteilt werden. Diese Elemente entstehen durch Unterteilung der Merkmalskante, die die Basiswand definiert. Zusätzlich zu einer solchen horizontalen Unterteilung ist es möglich, sämtliche Wandarten vertikal zu unterteilen und dadurch einzelne Partitionen zu definieren [Fi08].

5.3.4.4 **Erzeugung der Mauerwerke**

Ausgangspunkt von Finkenzellers Verfahren zur Generierung von Mauerwerken ist eine Problemstellung, die häufig im Umgang mit vorgefertigten Texturen auftritt. Die Verwendung solcher Texturen kann für komplexe Wandstrukturen zu unschönen Artefakten führen, die die erzeugte Wand unrealistisch erscheinen lassen. Dies liegt daran, dass die Texturen nicht die zugrunde liegende Wandstruktur berücksichtigen, wodurch es zu hässlichen Brüchen kommen kann. Finkenzellers Ansatz erzeugt Wandtexturen prozedural, wodurch sich der Texturverlauf an der Wandgeometrie orientieren kann. Hierfür wird ein Ansatz verwendet, bei dem alle Komponenten des Mauerwerks zunächst in einer *erweiterten Backus-Naur-Form* (EBNF) beschrieben werden. Anschließend wird basierend auf diesen Beschreibungen algorithmisch die Wandtextur erzeugt.

Als erste Komponente des Mauerwerks definiert man eine Reihe von Ziegeltypen mittels Parametern wie Farbe, Größe etc.. Derart beschriebene Ziegel werden in einer weiteren Beschreibungsdatei zu einem Muster zusammengesetzt, das neben der reinen Abfolge der unterschiedlichen Ziegeltypen auch die Definition eines Versatzes ermöglicht. Die letzte Beschreibungsdatei kombiniert die Musterfestlegung mit Parametern für den verwendeten Mörtel. Damit ist die Struktur der gewünschten Wand vollständig festgelegt.

Der Berechnungsalgorithmus für die Mauertexturen basiert auf der Arbeit von Legarkis et al. [LDG01]. Grundlage des Verfahrens sind Wandpolygone, die Löcher enthalten dürfen. Basierend auf diesen Polygonen wird ein Graustufenbild erzeugt. Löcher und Bereiche, in denen keine Ziegel positioniert werden dürfen, werden mit einem weißen Farbwert markiert, der restliche Bereich ist schwarz eingefärbt. Die eigentliche Positionierung eines Ziegels basiert dann auf der Projektion des Ziegels auf das Grauwertbild. Aus sämtlichen überlagerten Pixeln wird ein mittlerer Farbwert des Ziegelbereichs bestimmt. Liegt dieser Mittelwert nahe bei schwarz, wird der Ziegel vollständig gezeichnet, befindet er sich nah bei weiß, wird er verworfen. Liegt der Farbwert im Grauwertbereich, so muss er an den Wandpolygonen beschnitten werden. Abbildung 46 zeigt ein einfaches Beispiel für einen Mauerziegelverbund, der mit dem beschriebenen Verfahren erzeugt wurde.

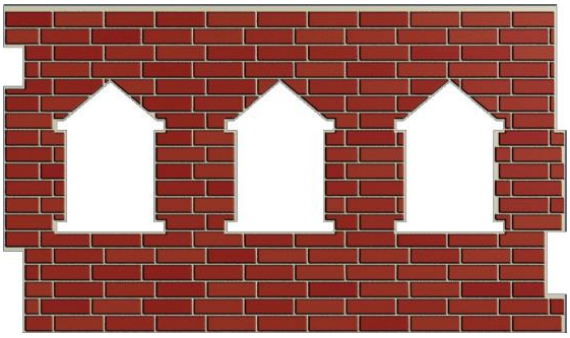

Abbildung 46: Mauerziegelverbund [Fi08]

Für Quadersteinmauerwerke wird dieser Algorithmus leicht abgewandelt. Bei solchen Mauerwerken werden die Steine bereits während des Platzierens in die Zielfläche eingepasst. Im Gegensatz zu Mauerziegelverbänden besitzen die Ziegel bei Quadersteinmauerwerken keine feste Größe, sondern einen Größenbereich. Innerhalb des Größenbereichs wird für einen freien Bereich innerhalb der Textur zufallsbasiert eine Größe ausgewählt. Passt der Stein an die jeweilige Position, wird er eingefügt, ansonsten wird so lange weiter nach passenden Steinen gesucht, bis die Mindestgröße unterschritten wird.

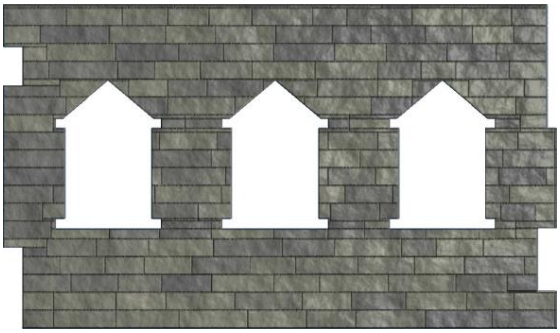

Abbildung 47: Quadersteinmauerwerk [Fi08]

Verbleibende Lücken werden dann mit Quadersteinen aufgefüllt, die im Gegensatz zu den Mauerziegelverbänden auch die Wandpolygone überlappen dürfen. Erst im letzten Schritt werden solche überstehenden Steine gegen die Polygone geschnitten. Abbildung 47 zeigt ein Quadersteinmauerwerk, das über den leicht angepassten Algorithmus erzeugt wurde.

Als weitere Mauerwerksvariante ermöglicht Finkenzellers System [Fi08] die Erzeugung von Zyklopenmauerwerken. Die Berechnung der einzelnen Steine erfolgt dabei durch die Bestimmung eines *Voroni-Diagramms*, die Formen der Steine entsprechen den einzelnen *Voronoi-Regionen*. Die Schwierigkeit besteht nicht in der Berechnung dieser Regionen, sondern in der Positionierung der einzelnen Punkte, für die diese Regionen erzeugt werden. Finkenzeller verwendet in einem ersten Schritt ein Zellrasterverfahren, bei dem er in jeder Ausgangszelle zufallsbasiert eine feste Anzahl von Punkten verteilt. Nach diesem initialen Verteilen der Punkte erfolgt die weitere Berechnung in einem iterativen Verfahren. In jedem Durchlauf bestimmt man für die vorhandene Punktemenge ein Voronoi-Diagramm. Anschließend werden die Schwerpunkte der einzelnen Voronoi-Regionen berechnet. Diese dienen als Eingabe in die jeweils nächste Iteration. Das Verfahren terminiert, sobald der Abstand zwischen den Schwerpunkten einer Voronoi-Region unterhalb eines Grenzwertes liegt. Die Anzahl der Durchläufe bestimmt das Aussehen des erzeugten Mauerwerks. Je früher die iterative Berechnung abgebrochen wird, desto unregelmäßiger ist das erzeugte Muster. Abbildung 48 zeigt ein Zyklopenmauerwerk, das für ein Polygon mit einem Torbogen errechnet wurde.

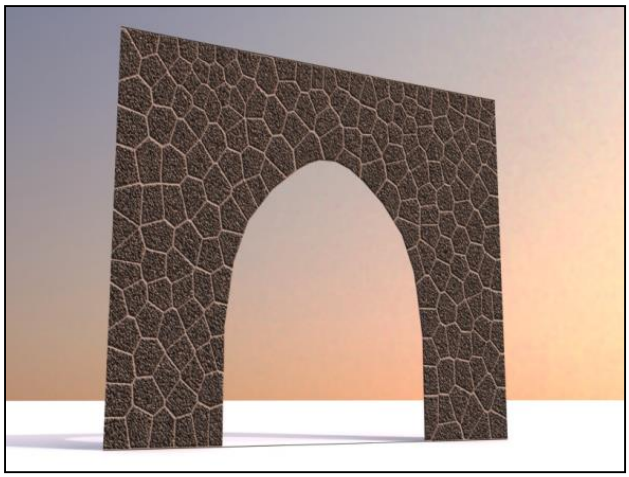

Abbildung 48: Zyklopenmauerwerk [Fi08]

5.3.4.5 **Erzeugung von Gesimsen**

Gesimse werden zur Dekoration von Fassaden eingesetzt und bestehen aus einem aus der Mauer hervortretenden Streifen zur Betonung waagerechter Bauabschnitte [Fi08]. In der Literatur werden unterschiedliche Arten von Gesimsen definiert, die sich unter anderem in Größe und Position unterscheiden. Finkenzeller führt die Erzeugung solcher Gesimse auf einen einheitlichen Ansatz zurück, bei dem zunächst das Profil des Gesimses definiert und dieses anschließend an einer Wandpartition appliziert wird. Die Beschreibung erfolgt in einer an die Programmiersprache Logo angelegten Beschreibungssprache und verwendet Konzepte der Turtle-Grafik. Diese wurde bereits im Zusammenhang mit L-Systemen im Abschnitt „Graphische Interpretation der Zeichenketten“ erörtert. Die Sprache verwendet nur zwei Kommandos. Das Kommando `line` zeichnet eine gerade Linie, das Kommando `arc` einen Bogen. Für beide Kommandos gibt man Parameter an, die den Zeichenprozess steuern, für `line` ist dies beispielsweise die Länge der Linie, bei `arc` legt man durch die Übergabe des Winkels und des Radius fest, wie der erzeugte Kreisbogen bestimmt wird. Zentral für die erzeugten Profile ist die Ebene, in der die Zeichenschritte ausgeführt werden. Diese Ebene wird als Profilbasis bezeichnet und ist definiert durch zwei orthogonale Vektoren, den Up- und den Normal-Vektor. Der Up-Vektor legt die Startrichtung eines Kommandos fest. Der Winkelparameter beider Kommandos führt zu einer Drehung in Richtung des Normal-Vektors um einen übergebenen Winkel. Die Ausführung von Kommandos ändert die Profilbasis, wobei die Art der Änderung kommandoabhängig ist. Gleichzeitig definiert der Endpunkt einer gezeichneten Kante den Startpunkt für das jeweils nachfolgende Kommando.

```
Modifications
{
  line, rel,  90, 0.3;
  line, rel, -90, 10.0;
  line, rel,  90, 0.6;
  line, rel, -90, 11.0;
  line, rel,  90, 0.6;
  line, rel, -90, 0.6;
  line, rel,  90, 0.6;
  line, rel, -90, 2.0;
  line, rel,  90, 2.0;
  arc, rel, 0, -90, 4.0;
  line, rel,  90, 5.0;
  line, rel, -90, 5.0;
  line, rel,  90, 0.6;
  line, rel, -90, 0.6;
  line, rel,  45, 7.0;
  line, rel, -45, 0.6;
  line, rel,  90, 0.6;
  line, rel, -90, 2.0;
  line, rel, -84, 22.0;
}
```

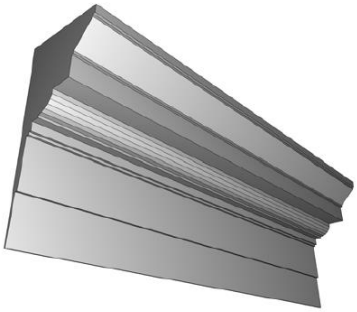

Abbildung 49: Gesimsbeschreibung und daraus erzeugtes Gesims [Fi08]

Bei der Erzeugung eines Profils basierend auf dessen textueller Beschreibung werden die einzelnen Schritte sequentiell durchlaufen. Dadurch, dass der Endpunkt jedes Kommandos gleichzeitig Startpunkt des nachfolgenden Kommandos ist, entsteht ein geschlossener Linienzug. Abbildung 49 zeigt die Beschreibung eines Gesimses in der von Finkenzeller definierten Sprache sowie das Gesimse, dessen Profil durch das Codefragment erzeugt wird.

Im Zusammenhang mit Gesimsen spielen auch Friese eine wichtige Rolle. Friese werden nicht als eigenständige Objekte, sondern als texturierte Gesimse erzeugt. Bei einer solchen Texturierung können Artefakte entstehen, wenn die Gesimse Stoßpunkte besitzen. Solche Stoßpunkte treten beispielsweise an Hausecken auf. Je nach Art des Gesimses kann es dabei zu Verzerrungen oder Beschneidungen der verwendeten Texturen kommen. Diese Problematik schwächt Finkenzeller durch einen einfachen Ansatz ab. Hierzu definiert man nicht eine einzelne Textur, die auf das Gesimse aufgetragen wird, sondern unterteilt diese in einen Anfangs-, einen Mittel- und einen Endteil. Der Mittelteil sollte dabei so gewählt werden, dass er kachelbar ist, also ohne Artefaktbildung mehrfach hintereinander eingefügt werden kann. Umläuft ein zu texturierendes Gesimse ein Gebäude, so wird am Anfang und am Ende des Gesimses für eine Wandkante zusätzliche Geometrie erzeugt, die dann mittels des Anfangs- bzw. Endteils texturiert wird. Zwar können dadurch nicht alle auftretenden Fehler vermieden werden, ihre Effekte werden aber zumindest abgeschwächt.

5.3.4.6 **Erzeugung von Ornamenten**

Ornamente werden in vielen Bereichen zur Ausschmückung und Verzierung verwendet. Die verwendeten Motive sind vielfältig und reichen von rein geometrischen Formen wie Aneinanderreihungen von Punkten und Linien über pflanzenähnliche bis hin zu künstlerischen Motiven. Neben den Motiven stellt die Form ein weiteres Unterscheidungsmerkmal dar. Finkenzeller beschränkt sich in seiner Arbeit auf die Erzeugung begrenzter Flachornamente [Fi08]. Hierbei handelt es sich um Ornamente, die auf beliebig geformten Flächen appliziert und exakt in diese eingepasst werden. Als Motive verwendet Finkenzeller pflanzenähnliche Grundelemente. Die Erzeugung der Ornamente gliedert sich in zwei Hauptschritte. Zunächst muss innerhalb der Zielfläche ein freier Bereich gefunden werden. Anschließend werden in diesem Bereich Elemente erzeugt.

Nachfolgend wird der Ansatz vorgestellt, mittels dessen in ProcMod Ornamente prozedural erzeugt werden. Eingabe in dieses Verfahren ist ein Binärbild, das freie Bereiche durch schwarze, belegte Bereiche durch weiße Pixel kodiert. Ein solcher Ansatz wurde auch bei der Erzeugung der Mauerwerkstexturen eingesetzt. Innerhalb des Verfahrens werden die Bereiche über ein Grauwertbild repräsentiert. Dies ist erforderlich, da durch das Verfahren belegte Bereiche durch graue Pixel repräsentiert werden. Ein naives Verfahren zur Bestimmung freier Bereiche basierend auf einem Grauwertbild könnte jedes schwarze Pixel durchlaufen und anschließend sämtliche Nachbarpixel dahingehend prüfen, ob diese ebenfalls unbelegt sind. Ist dies der Fall, so wird der Suchbereich schrittweise erweitert, bis ein belegtes Pixel gefunden wird. Dieser Berechnungsansatz ist allerdings sehr zeitaufwendig, weshalb Finkenzeller einen Algorithmus einsetzt, der auf der *Mittelachsentransformation* (MAT) basiert.

Die MAT ähnelt in ihrem Ansatz dem Straight-Skeleton-Algorithmus, der im Semantic Building Modeler in einer modifizierten Variante für die Berechnung der Hausdächer eingesetzt wird. Auch die MAT bestimmt für ein beliebiges Eingabepolygon dessen topologisches Skelett. Anschaulich repräsentiert die MAT ein Polygon durch eine Menge von Kreismittelpunkten. Für jeden zu einem solchen Mittelpunkt gehörenden Kreis gilt, dass dieser den Randbereich des Polygons an mindestens zwei Punkten tangential berührt. Die aus diesen Mittelpunkten erzeugten Kanten zeichnen sich somit dadurch aus, dass jeder Punkt auf dieser Kante zu mindestens zwei Kanten gleich weit entfernt ist.

Abbildung 50 zeigt die einzelnen Schritte des Verfahrens anhand eines einfachen, im ersten Bild vollständig schwarz eingefärbten Eingabepolygons. Das mittlere Bild illustriert, wie die

bitangentialen Kreise in dieses Polygon eingepasst werden. Berechnet man diese für alle Mittelpunkte, so erhält man eine Repräsentation des Polygons durch sein topologisches Skelett, welches im rechten Bild zu sehen ist. Für die Berechnung der MAT existieren verschiedene Algorithmen, Finkenzeller verwendet einen Ansatz von [GW87].

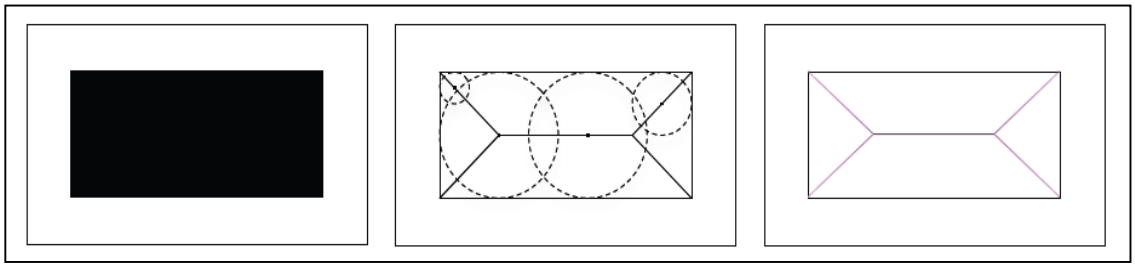

Abbildung 50: Mittelachsentransformation [Fi08]

Das Ergebnis dieser Berechnung wird anschließend nicht als Grauwertbild, sondern als Liste mit sämtlichen berechneten Kreisen gespeichert, die aufgrund ihres Radius absteigend sortiert sind. Dadurch ist es leicht, jederzeit auf den größten freien Bereich zuzugreifen. Nachdem freie Bereiche ermittelt wurden, werden im nächsten Schritt Ornamentelemente innerhalb dieser Bereiche erzeugt. Die Elemente werden durch eine beliebig große Menge geometrischer Grundformen beschrieben. Ein Beispiel für ein solches Element ist eine Blüte, die durch Kreisbögen und Bézier-Kurven dargestellt werden kann. Neben einer exakten Repräsentation der Elemente werden zusätzlich auch vereinfachte Darstellungen (beispielsweise Kreise) verwendet, die zur Aktualisierung des Grauwertbildes für die Verwaltung der freien Bereiche eingesetzt werden. Die Elemente selber sind konzeptionell angelehnt an den Objektbegriff der objektorientierten Programmierung, da sie sowohl ihre geometrische Form, als auch die Logik ihres Wachstums im Sinne eines Objektes zusammenfassen. Auch die Entscheidung, ob das Element positioniert wird, obliegt den Elementen selber. Dazu bekommt die Wachstumsmethode unter anderem einen freien Bereich in Form eines Kreises übergeben und entscheidet anschließend, ob ein Wachstum erfolgt. Anschließend werden sowohl die MAT als auch die Liste mit allen positionierten Elementen aktualisiert. Diese iterative Berechnung endet, sobald keine freien Bereiche mehr vorhanden sind. Das Ergebnis der Berechnung wird anschließend als Vektorgrafik exportiert und kann verwendet werden, um Texturen zu erzeugen [Fi08].

5.3.4.7 **Erzeugung von Türen und Fenstern**

Die Erzeugung von Fenstern und Türen basiert auf der Verwendung der vorab vorgestellten Wandpartitionen. Finkenzellers Ansatz verwendet als Eingabe rechteckige Polygone, die durch die Wandunterteilungen erzeugt wurden. Diese Eingaberechtecke können dann im weiteren Verlauf verfeinert werden. Die Verfeinerung erfolgt auf der Ebene einzelner Kanten des Rechtecks. Diese können unterteilt und anschließend durch beliebige Kantenzüge ersetzt werden, deren Verlauf in der Beschreibungssprache angegeben wird, mittels derer auch Gesimsprofile (s. Abschnitt „Erzeugung von Gesimsen“) definiert werden. Hierfür legt man zunächst die Länge der unterteilten Kante durch Abstandsparameter relativ zu ihrem Ursprung fest. Ein zusätzlicher Parameter `size` kann verwendet werden, um die Höhe eines Rechtecks basierend auf dem neuen Kantenabschnitt festzulegen. In ein solches Rechteck können anschließend verfeinerte Kantenzüge eingepasst werden.

Nachdem die Kantenzüge für die Fenster- und Türöffnungen erzeugt wurden, ist es möglich, Fenster- und Türstöcke zu berechnen. Hierbei handelt es sich um die feststehenden Teile der Tür oder des Fensters, in denen die beweglichen Komponenten verankert sind. Diese besitzen eine vorgegebene Dicke und können aus der Wand heraus ragen.

Weiterhin ist es möglich, Kanten des Kantenzuges einen rechteckigen Bereich zuzuweisen, innerhalb dessen Ornamente generiert werden können. Einem solchen Rechteck kann anschließend ein Rahmen hinzugefügt werden. Für die Erzeugung des Rahmens stehen unterschiedliche Stile zur Verfügung. Komplexe Rahmenformen können analog zur Erzeugung von Gesimsen durch die Angabe eines Rahmenprofils in der Gesimsbeschreibungssprache durch das System errechnet werden.

Im letzten Schritt werden Fenster- und Türrahmen festgelegt. Finkenzeller unterscheidet Fenster- bzw. Türstöcke von Fenster- bzw. Türrahmen dahingehend, dass Erstere als zur Wand gehörend aufgefasst werden, während Rahmen als Bestandteile der Fenster bzw. Türen betrachtet werden. Fensterrahmen verlaufen entlang des Fensterstocks. Dessen Polygon wird durch zwei Streben unterteilt und für die Erzeugung der Fensterscheiben verwendet. Auch bei Türen entspricht der Verlauf des Rahmens dem Kantenzug des Türstocks, die Tür selber wird durch ein Türpolygon repräsentiert, das bei Bedarf weiter unterteilt werden kann. Auch bei Fenster- und Türrahmen ist es möglich, die Rahmenform durch Rahmenprofile zu definieren, wie dies auch bei den Fenster- bzw. Türstöcken der Fall ist.

Abbildung 51: Verschiedene Fenstertypen [Fi08]

5.3.4.8 **Repräsentation und Modellierung der Gebäudefassaden**

In den vorhergehenden Abschnitten wurden die grundlegenden Verfahren und Algorithmen erläutert, mittels derer die Bestandteile eines Gebäudes innerhalb Finkenzellers System modelliert werden. Im nun folgenden Abschnitt wird die Repräsentationsform vorgestellt, die verwendet wird, um die einzelnen Komponenten miteinander zu verknüpfen und Gebäude zu beschreiben. Ein zentraler Aspekt bei dieser Repräsentation ist die Trennung der groben Gebäudestruktur vom verwendeten Stil. Dadurch lassen sich Stil und Struktur austauschen und wiederverwenden. Weiterhin unterteilt Finkenzeller das verwendete Modell in zwei unterschiedliche Teile. Das *Konzeptmodell* enthält eine abstrakte hierarchische Beschreibung der Gebäudekomponenten und repräsentiert die zwischen den Komponenten bestehenden Beziehungen. Demgegenüber umfasst das *Weltmodell* konkrete Instanzen der Konzeptmodellinstanzen. Dadurch modelliert es eine konkrete Ausprägung einer zunächst abstrakt beschriebenen Fassade.

Als Datenstruktur verwendet Finkenzeller *typisierte Graphen* (TGraphen). Dabei handelt es sich um einen gerichteten Graphen, bei dem sowohl Knoten als auch Kanten typisiert sind und über Eigenschaften verfügen können. Innerhalb des Systems stellen Knoten einzelne Elemente der Gebäudefassade dar, die über typisierte Beziehungen mit anderen Knoten verbunden sind. Um eine Trennung von Gebäudestruktur und Stil zu erreichen, werden auch die vorhandenen Komponenten und Relationen anhand ihres Anwendungsbereiches unterteilt. So beschränken sich die Konzepte und Relationen der Gebäudestruktur auf solche, die für die Erstellung eines groben Gebäudemodells mit einer Reihe von Stockwerken

erforderlich sind. Dazu gehören unter anderem Komponenten für Grundrissmodule und Dächer. Relationen werden verwendet, um die Konzepte miteinander zu verbinden.

Dies sei am Beispiel der Grundrissmodule verdeutlicht, die über PP- oder KK-Verbindungen miteinander verbunden werden. Innerhalb des Konzeptmodells existieren Relationstypen für beide Verbindungsarten. Für ein konkretes Gebäude enthält der Graph dann Instanzen des Grundrissmodultyps, die über Instanzen der Relationentyps verbunden sind. Ein ähnlicher Ansatz wird verwendet, um ein Grundrissmodulkonzept mit einem Dachkonzept zu verbinden. Auch hierfür existiert ein Relationstyp mit Namen `hat_Dach`.

Die Definition des Gebäudestils basiert analog zur Gebäudestruktur auf der Verwendung von Konzeptknoten, die über Instanzen unterschiedlicher Relationstypen innerhalb des Graphen miteinander verbunden werden. Dabei umfasst der Stil nicht nur Konzepte zur Definition von Gesimsen oder Tür- / Fensterverfeinerungen, sondern betrifft auch die Unterteilungen der Basiswände in Zwischenwandelemente und Wandpartitionen. Dadurch wird die Unterstrukturierung einer Fassade zu einer Eigenschaft des Stils und ermöglicht dadurch die Anwendung unterschiedlicher Stile auf unterschiedliche Bereiche der Wandfassade.

Abbildung 52 zeigt das Modell des Fassadenstils als TGraph. Knoten stellen Konzepte dar, die einzelne Elemente der Fassade betreffen. Diese Konzepte sind über typisierte Relationen miteinander verbunden. Die Typen der Relationen sind in der Grafik nicht eingezeichnet. Außerdem nicht dargestellt sind die Eigenschaften der Konzepte und Relationen. Über diese Eigenschaften werden die in den vorherigen Abschnitten vorgestellten Ansätze zur Modellierung einzelner Gebäudekomponenten umgesetzt. So besitzt das Konzept *Mauerziegelverband* als Eigenschaft die Beschreibung der Mauerwerkskomponenten. Gleiches gilt für das *Gesimskonzept*, das unter anderem eine Eigenschaft *Gesimsbeschreibung* enthält, mittels derer das Gesims erzeugt wird.

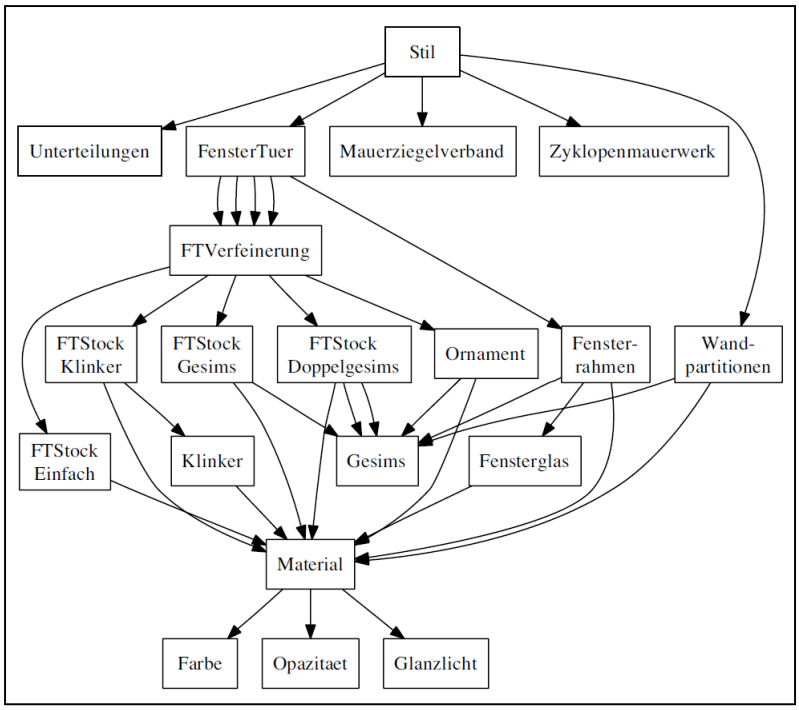

Abbildung 52: Semantisches Modell des Fassadenstils [Fi08]

Unter Verwendung dieser Konzepte und Relationen ist es möglich, beliebige Arten von Stilen zu definieren. Auch die abschließende Verbindung von Stil und Gebäudestruktur erfolgt über typisierte Relationen. Hier stehen zwei Arten von Relationen zur Verfügung. Die erste Relation weist einem ganzen Gebäude oder einzelnen Stockwerken einen bestimmten Stil zu. Bei der zweiten Relation kann diese Festlegung auf der Ebene einzelner Grundrissmodule erfolgen.

Das Konzeptmodell umfasst somit eine Beschreibung des Stils und der Gebäudestruktur. Basierend auf der Zuweisung der Stile zu unterschiedlichen Elementen der Struktur erzeugt das System dann in einem ersten Schritt das *Weltmodell*. Dieses beinhaltet konkrete Ausprägungen der im Konzeptmodell abstrakt definierten Gebäudekomponenten. Allerdings enthält das Weltmodell noch keinerlei Unterteilungen von Wänden oder Hausecken. Dies begründet Finkenzeller damit, dass die Umsetzung der Unterstrukturierung auf der Ebene des Weltmodells im Falle des Austauschens eines Stiles potentiell umfangreiche Neustrukturierungen des Graphen erfordern würde. Außerdem würde dadurch die klare Trennung zwischen Struktur und Stil aufgeweicht [Fi08]. Aus diesem Grund wird zunächst ein Zwischenmodell erzeugt, das Wände, Zwischenwandelemente und Hausecken enthält. Jede Komponente innerhalb dieses Modells besitzt räumliche Informationen über seine Nachbarn. Neben den Nachbarn speichern die Komponenten auch Verweise auf

Kindkomponenten, bei Wänden können dies beispielsweise erzeugte Wandpartitionen oder Verweise auf Tür- und Fensterstile sein. Aus dem derart berechneten Zwischenmodell wird erst im letzten Schritt das eigentliche 3D-Modell des Gebäudes erzeugt. Dazu gehört sowohl die Geometrieerzeugung als auch die prozedurale Texturberechnung.

Finkenzeller nennt eine Reihe von Gründen für die gewählte Unterteilung in unterschiedliche Modelle. So bietet das Konzeptmodell eine Beschreibungsmöglichkeit auf einem hohen Abstraktionsniveau, wodurch es übersichtlich und gut änderbar ist. Das Weltmodell wiederum befindet sich bezüglich seines Abstraktionsniveaus unterhalb des Konzeptmodells, ermöglicht aber trotzdem eine kompakte Beschreibung der Geometrie. Diese Beschreibung speichert Finkenzeller in XML-Dateien. Eine solche Repräsentation hat den Vorteil, dass sie gegenüber der Speicherung des 3D-Modells deutlich weniger Speicher benötigt als das fertig berechnete Modell. Dieser Vorteil wurde auch bereits im Kontext der GML ausführlich diskutiert, auch Finkenzeller sieht hier ein mögliches Anwendungsszenario darin, aus den Weltmodellbeschreibungen in Echtzeitanwendungen dynamisch 3D-Modelle errechnen zu lassen. Dadurch ließe sich zumindest auf Seiten der verwendeten 3D-Modelle der Speicherbedarf deutlich reduzieren [Fi08]. Auf der anderen Seite müssen solche Anwendungen dann auch sämtliche von Finkenzeller verwendeten Algorithmen implementieren und berechnen, was speziell in laufzeitkritischen Anwendungen wie Computerspielen eine zu große Belastung darstellt.

Abbildung 53: Erstelltes Gebäude [Fi08]

Basierend auf dem erstellten Zwischenmodell werden im letzten Schritt Geometrie und Texturen errechnet. Das Berechnungsergebnis kann in unterschiedliche Ausgabeformate exportiert und anschließend beispielsweise in Modellierungsumgebungen weiterverarbeitet

werden. Abbildung 53 zeigt ein gerendertes Gebäude, das mit den beschriebenen Techniken erstellt wurde. An diesem Gebäude erkennt man den hohen Detailgrad, den das System erzeugen kann. Durch die Verwendung der Beschreibungssprache für die Definition von Gesimsen und weiteren an der Fassade applizierten Strukturen können feine Details erzeugt werden, die einen guten visuellen Eindruck erzeugen.

5.3.4.9 **Diskussion des ProcMod-Systems**

Im Kontext der hier vorliegenden Arbeit soll Finkenzellers System nun aufgrund der eigenen Zielsetzungen betrachtet werden. Das Hauptziel besteht in der Einfachheit der Bedienung, die es Nutzern ohne große Erfahrungen im Bereich der 3D-Modellierung möglichst leicht machen soll, Gebäude automatisch erzeugen zu lassen.

Ausgehend vom Ablauf der Gebäudeerstellung soll zunächst die Grundrissdefinition diskutiert werden. Leider ist aus der Arbeit von Finkenzeller nicht zu erkennen, wie der Nutzer die Polygone angibt, aus denen die Module bestehen, die anschließend über die verschiedenen Verbindungstypen zu einem Grundriss zusammengefasst werden. Dabei ist davon auszugehen, dass es einem unerfahrenen Nutzer schwer fallen wird, die Koordinaten der Polygonpunkte manuell anzugeben. Auch die Wahl geeigneter Verbindungen erfordert ein grundlegendes Verständnis des Systems und der dahinter stehenden Konzepte. Auf der anderen Seite ist die automatisch durchgeführte Unterteilung der Grundrisse zur Erzeugung neuer Stockwerke ein interessanter Ansatz, der vielfältige Gebäudestrukturen erzeugen kann. Allerdings ist auch dort ein Eingreifen des Nutzers erforderlich. Die Definition der Gebäudestruktur ist somit zwar durch die Typisierung der Grundrissmodule und deren Variation zwischen Stockwerken ein mächtiger Ansatz, allerdings handelt es sich in der Umsetzung um einen stark durch Nutzerintervention geprägten Erzeugungsprozess. Das System ist nicht in der Lage, selber Variationen zu erzeugen, beispielsweise, indem Grundrissmodule randomisiert erzeugt oder modifiziert werden. Jedes Gebäude muss demnach manuell durch den Nutzer festgelegt werden und kann anschließend aufgrund der vollständigen Beschreibung automatisiert errechnet werden. Für die Erzeugung einer Vielzahl unterschiedlicher Häuser eines bestimmten Typs ist die Software aus Sicht der Gebäudestruktur dagegen nicht geeignet.

Die Beschreibungssprache, die Finkenzeller für die Definition der Gesimsstrukturen und Fenster- bzw. Türrahmen verwendet, ist analog zu den Grundrissmodulen ein Ansatz, der

zwar vergleichsweise mächtig, allerdings für unerfahrene Nutzer schwer zu begreifen ist. Die Ausdrucksstärke des Turtle-Grafikkonzepts wurde bereits ausführlich im Zusammenhang mit der prozeduralen Erzeugung von Pflanzen durch L-Systeme diskutiert. Dort wurde gezeigt, wie vielfältig die Formen sein können, die mit Hilfe dieses Formalismus erzeugt werden können. Allerdings stellt die Beschreibung eines Profils durch geometrische Kommandos hohe Anforderungen an den Nutzer. Der Vorteil des Ansatzes besteht in der Schlankheit der Beschreibung, da diese aufgrund der textuellen Repräsentation in der Lage ist, auch komplexe Profile mit geringem Speicherbedarf zu verwalten. Ein großer Nachteil ist dagegen die Tatsache, dass der Einsatzbereich auf das System beschränkt ist, Profile, die beispielsweise in Modellierungsanwendungen erzeugt wurden, müssen zunächst in die Beschreibungssprache übersetzt werden.

Eine große Stärke von Finkenzellers System ist die prozedurale Erzeugung von Texturen und Ornamenten. Speziell die Berücksichtigung der Gebäudegeometrie bei der Berechnung ist ein Ansatz, der in der Lage ist, häufig auftretende Unschönheiten bei der Texturierung zu vermeiden.

Zusammenfassend ist Finkenzellers Ansatz für die Modellierung einzelner Gebäude sehr gut geeignet. Er erzeugt komplexe Gebäude, bei denen sich die Komplexität allerdings auch im Erstellungsaufwand widerspiegelt. Die Erzeugung komplexer Geometrie, beispielsweise in Bezug auf Fenster oder Türen erfordert die Beschreibung des Geometrieverlaufs in der Logo-basierten Beschreibungssprache und ist dadurch zwar sehr flexibel, erfordert aber sowohl ein gutes Verständnis als auch eine hohe räumliche Vorstellungskraft des Nutzers, um die Profile und Unterteilungen so anzugeben, dass das gewünschte Ergebnis erreicht wird. Der Einsatz bereits vorhandener 3D-Modelle von Türen, Fenstern oder anderen Komponenten des Gebäudes ist nicht vorgesehen. Auch hier gilt somit, dass die Wiederverwendung solcher Komponenten nicht möglich ist und diese zunächst in der Beschreibungssprache nachgebaut werden müssen. Weiterhin ähnelt Finkenzellers Konzept in Bezug auf die Fassadenstrukturierung dem Vorgehen regelbasierter Systeme wie der CityEngine. Auch hier muss der Nutzer zunächst horizontale und vertikale Unterteilungen vornehmen, die dann bei der Erzeugung des Zwischenmodells aus dem Weltmodell umgesetzt werden. Dies entspricht der Verwendung der Unterteilungsregeln, die [Mü06] bei der Regeldefinition für die Fassadenerzeugung verwenden. Auch bei Finkenzellers System ergeben sich dadurch Nachteile für den Nutzer, da dieser Fassaden aufgrund geometrischer Operationen strukturieren muss, wodurch die Gebäudeerzeugung in diesem Punkt auf

vergleichsweise niedrigem Abstraktionsniveau erfolgt. Für Nutzer ohne mathematischen Hintergrund wäre es intuitiver, solche Operationen durch Kommandos auf einem höheren Abstraktionsniveau zu kapseln und dadurch ihre Komplexität zu reduzieren.

5.4 **Weitere Arbeiten**

Der folgende Abschnitt stellt Arbeiten vor, die sich mit der Lösung bestimmter Probleme im Bereich der prozeduralen Modellierung befassen oder neue Techniken vorstellen, die mit bestehenden Ansätzen kombiniert werden können.

5.4.1 **Interactive Architectural Modeling with Procedural Extrusion [KW11]**

Eine aktuelle Arbeit, die sich mit dem Bereich der interaktiven prozeduralen Gebäudegenerierung befasst, stammt von Peter Wonka und Tom Kelly [KW11]. Der Ansatz basiert auf dem Straight-Skeleton-Algorithmus [Ai95] und implementiert diesen in einer Variante, die auf der Verwendung einer *Sweep-Plane* basiert und unterschiedliche Gewichte gestattet. Die Idee unterschiedlicher Kantengewichte geht auf eine Arbeit von Eppstein et al. zurück [EE98]. Der Straight-Skeleton-Algorithmus und seine verschiedenen Erweiterungen werden im Abschnitt „Dacherzeugung – der Weighted-Straight-Skeleton-Algorithmus“ ausführlich diskutiert, hier sei nur auf den Ansatz von Kelly und Wonka eingegangen und die Fragestellung, wie sie das Verfahren für die Erzeugung von Häusern einsetzen.

Die Autoren verwenden einen Ansatz, der das Konzept der Pfad-Extrusion mit den Möglichkeiten eines gewichteten Straight-Skeleton-Algorithmus verbindet. Bei der Pfad-Extrusion definiert man zunächst einen Querschnitt und eine Profilkurve, an der der Querschnitt während der Extrusion entlanggeführt wird. Auch im System von Wonka und Kelly legt man zunächst einen Grundriss fest und definiert anschließend für jede Kante innerhalb dieses Grundrisses eine Profilkurve. Hierfür stehen dem Nutzer verschiedene Werkzeuge zur Verfügung, mittels derer die Kurven und Polygone erstellt werden können. Die Erzeugung der Geometrie erfolgt dann durch die Berechnungsschritte des Straight-Skeleton-Algorithmus, der von den Autoren an verschiedenen Punkten modifiziert wurde, um unterschiedliche Kantengewichte zuzulassen und in der Methodik dem in dieser Arbeit verwendeten Ansatz ähnelt. Das Verfahren weist jeder Kante im Eingabegrundriss eine Ebene zu, die die Kante vollständig enthält. Die Steigung der Ebene wird kontrolliert durch die Profilkurve, die der jeweiligen Kante zugewiesen wird. An Kontrollpunkten innerhalb

der Kurve ändert sich deren Steigung und somit auch die Steigung der zugehörigen Ebene im Straight-Skeleton-Algorithmus. Dies beeinflusst die Geschwindigkeit des Schrumpfungsprozesses der betroffenen Kante. Dadurch drückt die Ebenensteigung das Kantengewicht aus, wie es von Eppstein et al. [EE98] erstmals im Kontext gewichteter Straight-Skeleton-Algorithmen erwähnt wurde. Den Kantenprofilen kommt somit eine besondere Bedeutung für die Geometrieerzeugung zu.

Innerhalb des Systems können die Profile verwendet werden, um eine Vielzahl unterschiedlicher architektonischer Elemente zu erzeugen und dienen darüber hinaus als Anknüpfungspunkt für prozedurale Techniken, wie sie aus regelbasierten Systemen bekannt sind. Neben der Festlegung von Kantensteigungen ist es möglich, den einzelnen Profilen *Ankerpunkte* zuzuweisen. Im Verlaufe der Extrusion können an solchen Ankerpunkten beliebige Elemente positioniert werden, bsw. geladene Fenster- und Türmodelle. Die Ankerpunkte können parametrisiert und anschließend eingesetzt werden, um Elemente wiederholt auf Fassaden zu positionieren. Eine solche Wiederholung kann sowohl in Bezug auf Spalten als auch auf Reihen erfolgen, so dass sich beispielsweise auf unterschiedlichen Stockwerken Fensterreihen positionieren lassen. Neben der Modifikation der Kantenprofile kann der Nutzer auch den Querschnitt auf unterschiedlichen Höhen anpassen. Dadurch lässt sich beispielsweise ein spezieller Dachquerschnitt verwenden, anhand dessen anschließend das eigentliche Dach erzeugt werden kann. Auf die Implementation des Verfahrens wird im Kontext der Diskussion des Straight-Skeleton-Algorithmus eingegangen, konzeptuell basiert das Verfahren auf der Berechnung von Schnittpunkten der Kantenebenen und die Behandlung daraus entstehender Events durch das Hinzufügen oder Löschen von Kanten aus dem aktuellen Querschnitt. Zusätzlich zu diesen durch den Algorithmus produzierten Events behandeln die Autoren weitere Events, die sich aus der Verwendung der Profilkurve zur Steigungssteuerung ergeben. Die drei zusätzlich behandelten Eventarten sind *Edge Direction Events*, *Profile Offset Events* und *Anchor Events*. Edge Direction Events sind Ereignisse, die durch die Änderung der Steigung eines Segments in der Profilkurve entstehen. Profile Offset Events sind definiert durch nutzerdefinierte Änderungen der Querschnitte auf bestimmten Höhen der Straight-Skeleton-Extrusion, während Anchor Events durch das Setzen von Ankerpunkten in den Profilkurven ausgelöst werden. Die jeweiligen Eventarten werden auf unterschiedliche Weise behandelt. Für eine detaillierte Diskussion der Implementation sei auf [KW11] verwiesen.

Neben einer Reihe von Beispielen, wie sich diese Technik für die Konstruktion komplexer Gebäude einsetzen lässt, verwenden sie den Algorithmus auch für die prozedurale Erzeugung von Fenstern und Gesimsen. Dieser Ansatz ähnelt Konzepten, die auch Havemann in der GML [Ha05] einsetzt. Auch er verwendet seine Straight-Skeleton-Implementation als eine spezielle Form der Extrusion, greift allerdings nicht auf Profilkurven für die Steigungssteuerung zurück, wie dies bei Wonka und Kelly der Fall ist. Abbildung 54 zeigt verschiedene Beispiele für die Verwendung des Verfahrens zur prozeduralen Gestaltung von Fassadenelementen.

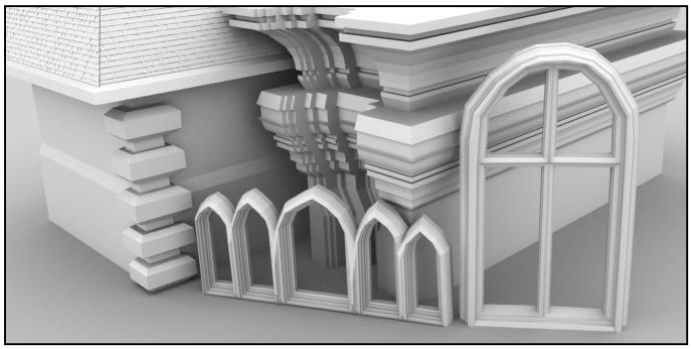

Abbildung 54: Verwendung der prozeduralen Extrusion für die Erzeugung architektonischer Elemente [KW11]

Das System von Wonka und Kelly ist in der Lage, durch den Einsatz der prozeduralen Extrusion mittels Straight-Skeleton-Algorithmus komplexe Gebäude zu erstellen. Dabei betonen die Autoren besonders die Möglichkeiten zur Generierung komplexer Dachformen und Fassadenelementen, die ihr Ansatz bietet. Abbildung 55 gibt einen Eindruck von der Vielfalt, die durch das Verfahren erzeugt werden kann.

Das Verfahren stellt einen interessanten Ansatz zur Nutzung des Straight-Skeleton-Verfahrens nicht nur zur Erzeugung von Dachflächen, sondern als allgemeines Werkzeug zur Erzeugung komplexer Geometrien dar. Die Steuerung der Kantengewichte durch die Verwendung von Profilkurven ist eine sinnvolle Erweiterung der gewichteten Straight-Skeleton-Implementation von Eppstein [EE98].

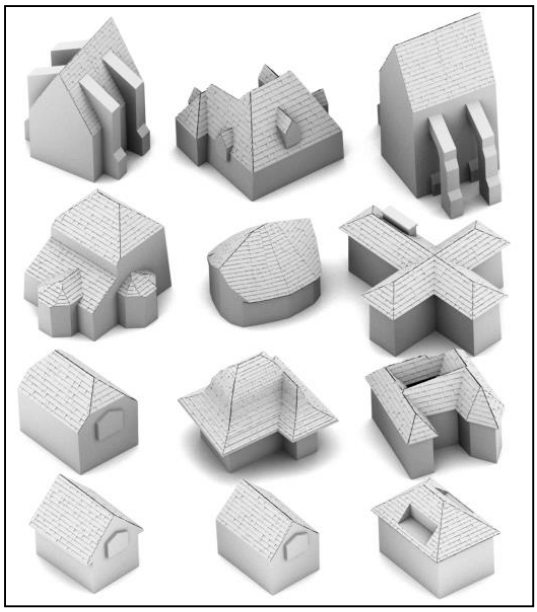

Abbildung 55: Häuser, die mittels prozeduraler Extrusion erstellt wurden [KW11]

In Bezug auf die hier vorliegende Arbeit stellt sich dabei die Frage, ob dieses Verfahren auch eingesetzt werden kann, um ganze Städte zu modellieren. Dazu geben die Autoren ein Beispiel, bei dem sie ein virtuelles Atlanta durch ihr System erzeugen ließen. Eingabe waren 6000 Grundrisse, die aus einer GIS-Datenbank importiert wurden. Die für die prozedurale Extrusion erforderlichen Profile wurden vorab erzeugt und anhand eines Klassifikationsverfahren den unterschiedlichen Kanten zugewiesen. Die Klassifikation der Kanten basiert auf verschiedenen Kriterien, beispielsweise ihrer Ausrichtung (beispielsweise Richtung Straße) oder ihrer Länge. Darüber hinaus spielt auch der Gebäudetyp eine wichtige Rolle für die Profilauswahl. Dieser muss durch die Quelldaten zur Verfügung gestellt werden und kann dann für die Profilzuweisung eingesetzt werden.

5.4.2 **Interactive Visual Editing of Grammars for Procedural Architecture [LWW08]**

Ein zentrales Problem regelbasierter Ansätze wurde bereits im Kontext der CityEngine thematisiert. Speziell für Nutzer ohne Kenntnisse im Bereich der Programmierung ist die textbasierte Entwicklung umfangreicher Regelsätze ein schwieriges Problem. GUI-Elemente, die den Nutzer beim Verfassen solcher Regelsysteme unterstützen, sind somit von großer Bedeutung, damit auch unerfahrene Anwender mit Systemen dieser Art arbeiten können. Ein wichtiger Aspekt, der im Rahmen der CityEngine-Diskussion noch nicht erwähnt wurde, ist die Frage nach der Feinheit der Steuerung der Geometrieerzeugung. Genau dieser Frage widmen sich Lipp et al. in ihrer Arbeit [LWW08] und entwickeln

Konzepte, die es ermöglichen, gezielt einzelne Elemente eines Gebäudes zu editieren. Diese implementieren sie in einer GUI-Anwendung, die die Erstellung von Regeln basierend auf CGA-Shape ermöglicht.

Ein Problem regelbasierter Systeme resultiert aus deren Struktur und dem Formalismus der fortlaufenden Ersetzung von Elementen durch andere Elemente. Die Ersetzung erzeugt dabei für einen Eingabeshape einen bestimmten Ausgabeshape. Kommen in einer Fassade beispielsweise mehrere unterteilte Fassadenelemente vor, so werden diese alle durch ein Fenster fester Größe und Form ersetzt, sofern eine entsprechende Regel existiert. Soll nun ein bestimmtes Fenster aus dieser Ersetzungsmenge mit anderen Parametern erzeugt werden, müssen hierfür neue Regeln eingeführt werden, die unter bestimmten Vorbedingungen ausgewählt werden und das gewünschte Ergebnis erzeugen. Solche Ausnahmen führen dabei leicht zu einer deutlichen Zunahme der Regelanzahl im verwendeten Regelsystem und dadurch automatisch auch zu einer höheren Komplexität. Mit diesem Problem befassen sich Lipp et al., indem sie unterschiedliche Selektionsmechanismen in ihr GUI-System integrieren. Das Ziel besteht darin, lokale Modifikationen an Elementen zuzulassen, ohne die gesamte Regelbasis modifizieren zu müssen.

Das erste Problem ist die Frage, wie die zu modifizierenden Elemente ausgewählt werden. Innerhalb des Systems stehen drei verschiedene Auswahlverfahren zur Verfügung. Ausgangspunkt dieser Verfahren ist die Repräsentation sämtlicher durch den Ersetzungsprozess erzeugten Shapes in Form einer hierarchischen Baumstruktur. Jeder Shape ist ein Knoten im Baum, die Anwendung einer Regel auf einen Shape ersetzt diesen und erzeugt seine Kindknoten. Das einfachste Auswahlverfahren wählt einen einzelnen Shape innerhalb des Baumes aus. Bei der Auswahl erzeugt das System einen *exact instance locator*, der aus dem Pfad durch den Baum bis hin zu dem Knoten besteht, dem der gewählte Shape entspricht. Demgegenüber werden bei der *hierarchischen Auswahl* alle Elemente ausgewählt, die sich unterhalb eines bestimmten Knotens befinden. Eine solche Auswahl wird analog zum exact instance locator umgesetzt. Allerdings sind die ausgewählten Knoten bei diesem Blätter, während es sich bei der hierarchischen Auswahl um innere Knoten handelt. Die *semantische Auswahl* ist das letzte zur Verfügung stehende Auswahlverfahren und adressiert ein bereits diskutiertes Problem, unter dem regelbasierte Ansätze im Allgemeinen leiden. Durch den Ersetzungsformalismus existiert innerhalb der Hierarchie keinerlei Semantik, wodurch Anfragen wie „Gesucht sind alle Shapes in der zweiten Spalte der dritten Reihe“ [LWW08] nicht möglich sind. Dieses Problem lösen die Autoren durch

die Integration semantischer Tags in die Regelbasis. Diese werden durch den Nutzer festgelegt und den Ergebnissen von Split- oder Repeat-Rules zugewiesen. Dadurch können semantische Auswahlsets unter Rückgriff auf diese Tags erzeugt und für die Modifikation ausgewählt werden.

Neben der Auswahl bestimmter Shapes spielt auch die Frage der Modifikation eine wichtige Rolle. Solche Modifikationen bestehen in der Auswahl einer bestimmten Variablen und der Zuweisung eines Wertes zu dieser. Wird ein Shape über eine Auswahl spezifiziert, so kann diesem eine Variable und ein Wert zugewiesen werden. Besitzt der Shape Kindknoten innerhalb der Hierarchie, so werden auch bei diesen die Variablenzuweisungen umgesetzt, wodurch eine Zuweisung bis zu den Blattknoten durch den Baum propagiert. Aufgrund des *Locator-Konzepts* ist es möglich, dass Variablen auf niedrigeren Stufen des Baumes überschrieben werden, sofern mehrere Zuweisungen auf den Elternknoten oder dem Knoten selber erfolgen. Die Speicherung der Zuweisungen erfolgt nicht innerhalb der Hierarchie selber, sondern wird extern vorgenommen. Bei der Erstellung der Geometrie werden diese extern gespeicherten Zuweisungen ausgewertet und umgesetzt. Dadurch können lokale Modifikationen realisiert werden, ohne dass die eigentliche Regelbasis angepasst werden muss.

Das entwickelte GUI-System unterstützt den Nutzer sowohl bei der Auswahl der Shapes, als auch bei der Zuweisung von Variablen-Werten und ermöglicht dadurch die vorgenannten Operationen und lokalen Anpassungen.

5.4.3 **Continuous Model Synthesis [MM08]**

Einen interessanten Ansatz verfolgen Merrell et al. in ihrer Arbeit [MM08]. Die Autoren entwickeln ein System, bei dem Variationen von Gebäuden basierend auf einem einzigen Eingabegebäude automatisch ohne jegliche Nutzerintervention erzeugt werden. Dieser Algorithmus ist die Grundlage des für den Semantic Building Modeler entwickelten Verfahrens für die ähnlichkeitsbasierte Grundrisserzeugung, das im Abschnitt „Ähnlichkeitsbasierte Grundrisserzeugung“ ausführlich vorgestellt wird. Dieser Abschnitt widmet sich darum dem Ausgangsalgorithmus, den die Autoren unter anderem für die ähnlichkeitsbasierte Gebäudegenerierung einsetzen.

Die Grundidee des Verfahrens sei für den 2D-Fall erörtert, bei dem Variationen eines Eingabepolygons erzeugt werden. Für sämtliche Kanten des Eingabepolygons wird eine

Menge von zu diesen parallelen Kanten erzeugt und in festgelegten Abständen in der Ebene positioniert. Die erzeugten Kanten zerteilen die Ebene in eine Menge von Faces, Kanten und Vertices. Zu jedem Element dieser Menge wird anschließend eine Menge von Konfigurationen zugewiesen, für die gilt, dass sie die *Adjazenz-Bedingung* (engl. *adjacency constraint*) erfüllen. Das Adjazenz-Kriterium und das Finden gültiger Konfiguration ist die zentrale Komponente des Algorithmus, den die Autoren vorstellen. Wird dieses Kriterium für sämtliche Elemente erfüllt, so gilt für das neu errechnete Modell M, dass für jede lokale Nachbarschaftskonfiguration eine analoge lokale Konfiguration im Originalmodell E existiert. Für jeden Punkt in M gilt darüber hinaus, dass es einen Punkt in E gibt, der bezüglich seiner lokalen Konfiguration zu diesem identisch ist.

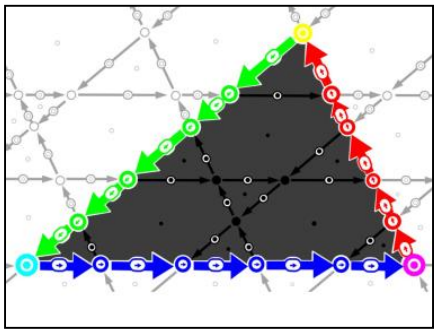

Abbildung 56: Eingabepolygon mit einer Menge paralleler Kanten [MM08]

Das Prinzip sei anhand eines einfachen Beispiels erläutert. Abbildung 56 zeigt ein einfaches, dreieckiges Eingabepolygon, dessen Kanten durch farbige Pfeile markiert sind. Weiterhin eingezeichnet ist bereits die Menge paralleler Kanten, die für die Eingabekanten automatisch erzeugt wurden. Die Kanten des Polygons teilen die Ebene, in der es liegt, in einen inneren und einen äußeren Bereich, der innere Bereich ist dunkelgrau dargestellt, der äußere weiß. Durch die Schnitte der Kanten entstehen neue, kleinere Flächen, die die Grundkomponenten für die Erzeugung von Variationen darstellen. Zentral für die Auswahl dieser Komponenten sind *Zuweisungen* (engl. *assignments*) zu diesen. Eine Zuweisung A besteht aus einem *Status* $s \in \{0,1\}$ und einem Oberflächenelement f und hat die Form $A = (f, s)$. Der Status eines Flächenelements ist 0, sofern sich dieses außerhalb des Bereiches befindet, der im Inneren des Eingabepolygons liegt, sonst 1. Der Status einer Kante ist nun definiert durch die Zuweisungen der Oberflächenelemente, die sich diese Kante teilen. Ein Status $s_e = \{(f_1, 0), (f_2, 0)\}$ beschreibt beispielsweise eine Kante, deren adjazente Oberflächenelemente alle außerhalb des Eingabebereichs liegen. Einfache, gültige Konfigurationen für eine Kante,

die das Adjazenz-Kriterium erfüllen, liegen genau dann vor, wenn die Status der adjazenten Oberflächenelemente identisch sind, diese also entweder beide innerhalb oder außerhalb des Eingabebereichs liegen. Der Umkehrschluss ist dagegen nicht möglich, haben die Elemente einen unterschiedlichen Status, muss nicht zwingend eine Verletzung des Kriteriums vorliegen. In solchen Fällen ist die Struktur der Eingabe entscheidend.

Das Adjazenz-Kriterium gilt nicht nur für Kanten, sondern auch für Vertices. Als Beispiel sei ein Vertex betrachtet, das den Schnittpunkt zweier Kanten darstellt und diese in je zwei Segmente teilt. Der Status eines Vertex besteht dann aus der Menge der Zuweisungen der vier Kantensegmente, zu denen es gehört. Die Kantenzuweisungen erfolgen analog zum vorher erörterten Fall. Wiederum ist es erforderlich, für jedes Vertex eine Liste von Status zu finden, die das Adjazenz-Kriterium erfüllen. Auch hier gilt, dass dieses erfüllt ist, sofern die Zuweisungen sämtlicher Kanten den gleichen Status besitzen. Das Auffinden weiterer gültiger Status erfordert die Untersuchung des Eingabepolygons. Bei Kanten durchsucht man das Eingabepolygon nach Kanten mit gleicher Steigung und verwendet deren Status. Entspricht dieser dem Status der gerade untersuchten Kante, so handelt es sich um einen weiteren gültigen Status. Um weitere gültige Status für Vertices zu finden, durchsucht man die Vertices des Eingabepolygons nach solchen Vertices, deren adjazente Kanten die gleiche Steigung besitzen.

Der grundlegende Algorithmus ist ein iterativer Berechnungsansatz, der zunächst die Menge paralleler Kanten für das Eingabepolygon erzeugt. Anschließend wird eine Menge möglicher Status $C(m)$ ermittelt, die den erzeugten Kanten und Vertices zugewiesen werden können. Zu Beginn der Berechnung existieren noch keine Zuweisungen zu den Elementen. Nun wird für jede Kante und jedes Vertex ein Status zufällig aus $C(m)$ ausgewählt und diesem Element zugewiesen. Diese zufällige Auswahl ist Basis der Erzeugung automatischer Variationen im Ausgabepolygon. Nachdem der zufällig ausgewählte Status dem Element zugewiesen wurde, werden alle Status aus $C(m)$ entfernt, die mit dieser Zuweisung nicht kompatibel sind. Dadurch reduzieren sich die möglichen Statuszuweisungen für alle nachfolgenden Elemente. Das Verfahren terminiert, sobald sämtliche Kanten und Vertices über einen Status verfügen. Die Ergebnisstruktur wird dann durch die Auswertung der Status sämtlicher Komponenten erzeugt.

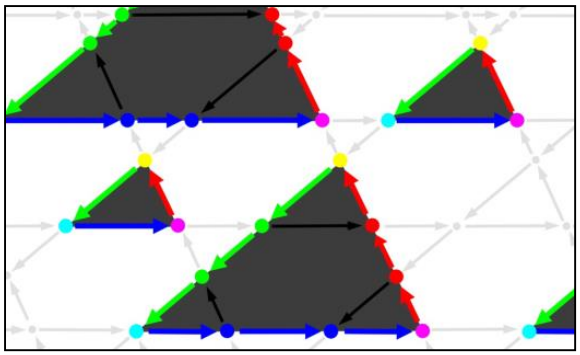

Abbildung 57: Automatische Variationen des Eingabepolygons [MM08]

Abbildung 57 zeigt einen möglichen Output des Algorithmus, der für das in Abbildung 56 gezeigte Eingabepolygon erzeugt wurde.

Das Grundprinzip des Ansatzes übertragen die Autoren auf 3D-Eingabepolyeder. Hier werden keine parallelen Kanten, sondern zu den Eingabepolygonen parallele Ebenen erzeugt. Diese unterteilen den Raum in Regionen, die analog zu den Kanten als *innen* oder *außen* klassifiziert werden und die Grundlage der Statuszuweisungen bilden. Die Berechnung und Zuweisung gültiger Status ist für den 3D-Fall komplexer, vom Prinzip her aber mit dem Vorgehen für die 2D-Variante vergleichbar. Für eine detaillierte Beschreibung des Algorithmus sei auf die Arbeit [MM08] verwiesen. Abbildung 58 zeigt ein Beispiel für den Einsatz des Verfahrens zur automatisierten Gebäudekonstruktion basierend auf dem links dargestellten Eingabegebäude.

Der Ansatz von Merrell et al. unterscheidet sich grundlegend von den vorab vorgestellten Verfahren. Bezüglich seiner Einordnung handelt es sich eindeutig um ein prozedurales Verfahren, das vollständig ohne Nutzerintervention auskommt. Der Einsatzbereich des Algorithmus ist dabei nicht auf die Erstellung von Gebäuden beschränkt, sondern kann auch für beliebige andere Polyeder eingesetzt werden. Diese fehlende Spezialisierung manifestiert sich allerdings auch in Bezug auf die Struktur der erzeugten Gebäude. Konzeptuell erzeugt das Verfahren ausschließlich Massemodelle. Systeme wie die CityEngine oder der Semantic Building Modeler verwenden solche Modelle als Ausgangspunkt für die Unterteilung in Stockwerke und das Hinzufügen architektonischer Komponenten wie Fenster oder Türen. Solche Details werden im System von Merrell et al. über Texturen simuliert, wodurch die Ergebnisse des Konstruktionsprozesses vergleichbar mit denen fotogrammetrischer Technologien sind. Für die Erzeugung von Gebäude- und Stadtmodellen mit hoher geometrischer Komplexität ist dieses Verfahren nicht geeignet, da das Ergebnis nur eine

erste Stufe auf dem Weg zu komplexen Gebäuden darstellt und nicht an die Komplexität anderer vorgestellter Systeme heranreicht. Trotzdem handelt es sich um einen potentiell fruchtbaren Ansatz, eine Kombination und Integration des Verfahrens mit prozeduralen oder regelbasierten Verfahren zur Erzeugung komplexer Fassaden könnte die Gebäudekomplexität deutlich erhöhen.

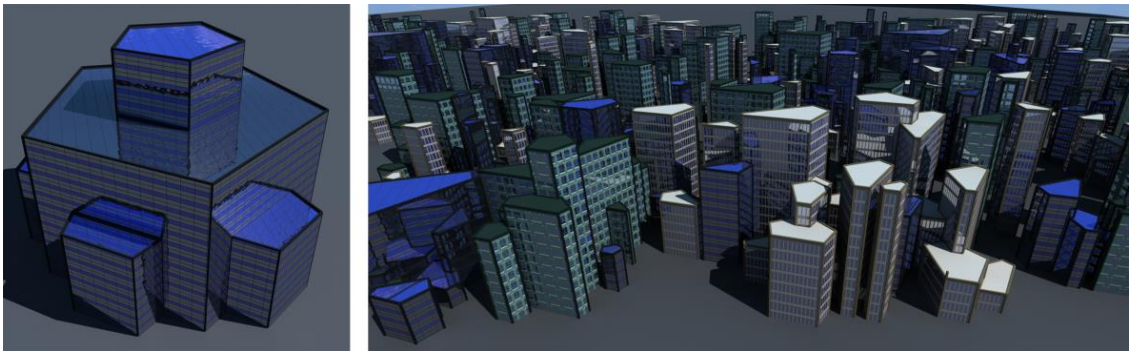

Abbildung 58: Beispiel für die Anwendung des Algorithmus für ein Eingabegebäude [MM08]

6 Vergleichende Diskussion der Ansätze zur prozeduralen Gebäudegenerierung

Nachdem die Stärken und Schwächen der verschiedenen vorgestellten Arbeiten diskutiert wurden, soll nun ein zusammenfassender Vergleich anhand unterschiedlicher Kriterien erfolgen. Hierzu werden die wichtigsten Zielsetzungen der hier vorgestellten Arbeit als Ausgangspunkt genommen.

6.1 **Anzahl der modellierten Häuser**

Ein erstes Unterscheidungsmerkmal ist die Frage, ob die Systeme darauf ausgelegt sind, einzelne Häuser zu erzeugen, oder ob mehrere Gebäude auf einmal berechnet werden können.

3D-Modellierungssysteme sind hier auf die Erstellung einzelner Gebäude beschränkt. Der Nutzer modelliert jedes Gebäude nach seinen eigenen Vorstellungen. Aufgrund der fehlenden Wiederverwendbarkeit des Modellierungsprozesses ist es nur sehr eingeschränkt möglich, Ergebnisse für andere Gebäude erneut einzusetzen.

Auch die vorgestellten fotogrammetrischen Technologien sind zunächst auf die Erzeugung einzelner Gebäude ausgelegt, speziell das Façade-System von Debevec [DTM96] kann nur Gebäude rekonstruieren, für die Fotografien zur Verfügung stehen. Das Build-by-Number-System von Bekins und Aliaga [BA05] schwächt diese Problematik durch die Extraktion von Strukturmustern zwar ab, trotzdem müssen die Massemodelle, auf die die abgeleiteten Strukturunterteilungen angewendet werden, zunächst manuell erzeugt werden. Erst anschließend kann auf die Strukturmuster zurückgegriffen werden.

Die CityEngine ist als regelbasiertes System potentiell in der Lage ganze Städte zu gestalten. Durch die vorgestellten Verfahren zur automatisierten Berechnung von Straßennetzen und der Ableitung von Grundrissen aus diesen können beliebig viele Gebäude automatisch berechnet und positioniert werden. Durch die Integration mit dem ArcGIS-System können darüber hinaus GIS-Daten existierender Städte als Ausgangspunkt für die Berechnungen eingesetzt werden.

Von den vorgestellten prozeduralen Ansätzen ist die GML aufgrund der Mächtigkeit der stackbasierten Sprache potentiell ebenfalls in der Lage, eine Menge von Gebäuden zu erzeugen, die Varianten aufweisen und somit nicht identisch sind.

Das ProcMod-System von Finkenzeller [Fi08] erreicht dies nicht, die Nutzereingaben werden immer zur Verwendung eines einzelnen Gebäudes eingesetzt, welches anschließend vollständig errechnet wird.

Der Modell-Synthese-Ansatz von Merrell et al. [MM08] ist ausschließlich darauf ausgelegt, diese Zielsetzung zu erfüllen und in der Lage, basierend auf einem Beispielgebäude eine beliebige Anzahl von Gebäuden zu errechnen.

System	Mehrere Gebäude?
3D-Modellierungswerkzeuge bsw. SketchUp	×
Fotogrammetrische Technologien bsw. Façade	×
Build-by-Number	×
CityEngine	✓
GML	×
ProcMod	×
Continuous Model Synthesis	✓
Semantic Building Modeler	✓

Tabelle 1: Fähigkeit, mehrere Gebäude auf einmal zu erstellen

6.2 **Erforderlicher Aufwand für die Gebäudegenerierung**

Das zweite zentrale Kriterium betrifft den Vergleich der Techniken bezüglich des Aufwandes, den der Nutzer betreiben muss, um ein Gebäude zu erzeugen. Außerdem wird die Fragestellung untersucht, inwiefern der Aufwand bei der Erstellung weiterer Gebäude abnimmt.

3D-Modellierungswerkzeuge benötigen aufgrund der fehlenden Automatisierungsmöglichkeiten den größten Aufwand. Der Nutzer muss die Geometrie vollständig manuell modellieren, je komplexer diese ist, desto größer ist der Zeitbedarf. Außerdem reduziert sich der Zeitaufwand für die Erstellung weiterer Gebäude nicht, sieht man einmal von der zunehmenden Expertise des Nutzers ab.

Bei fotogrammetrischen Technologien reduziert sich der Aufwand bereits beträchtlich. Je nach verwendetem System muss der Nutzer Vorarbeiten leisten, die das System bei der

anschließenden automatischen Extraktion der Geometrie aus den Quellfotografien unterstützen. Beim Façade-System und dem darauf aufbauenden Build-by-Number-Ansatz muss die grobe Form des Modells zunächst durch volumetrische Grundkörper angenähert werden, bevor anschließend Referenzpunkte und –kanten in den Fotografien markiert und den Grundkörpern zugeordnet werden. Rhinophoto löst das Übereinstimmungsproblem durch das Anbringen von Markern direkt auf dem Modell, was die Einsetzbarkeit dieses Ansatzes speziell für große Gebäude fraglich macht. Diesen Ansätzen ist gemein, dass keine Wiederverwendbarkeit existiert, sämtliche Arbeitsschritte müssen für jedes Gebäude von neuem durchgeführt werden. Build-by-Number verwendet den gleichen Ansatz wie Façade für die Erzeugung des Gebäudemodells, erfordert dann aber eine manuelle Unterteilung der Fassadenelemente. Somit ist der Aufwand höher als bei den anderen Verfahren, dafür sind die abgeleiteten Strukturmuster für die Erzeugung weiterer Gebäude erneut einsetzbar.

Die CityEngine benötigt für die Gebäudeerzeugung die Erstellung eines Regelsystems. Die Komplexität eines solchen Regelsatzes wächst mit der Komplexität der Gebäude, die durch diesen modelliert werden. Sollen darüber hinaus stochastische Elemente verwendet werden, um Variationen zu erzeugen, kann dies für einen unerfahrenen Nutzer einen vergleichsweise hohen Aufwand bedeuten. Allerdings unterstützt das Softwaresystem den Anwender bei der Erstellung. Ist ein Regelsatz für die Fassadengestaltung erstellt, so kann er für die Erstellung unterschiedlicher Gebäude eingesetzt werden. Regelbasierte Systeme wie die CityEngine zeichnen sich darum durch eine hohe Wiederverwendbarkeit aus.

Bei prozeduralen Systemen wie der GML, ProcMod oder dem Semantic Building Modeler ist die Definition des Aufwandes und der Wiederverwendbarkeit schwieriger. Hier ist eine Trennung zwischen Entwickler und Nutzer vonnöten. Entwickler implementieren in solchen Systemen die parametrisierten Algorithmen. Dies ist mit sehr großem Aufwand verbunden, da ein vollständiges Softwaresystem implementiert werden muss. Dies ist allerdings bei 3D-Modellierungswerkzeugen und fotogrammetrischen Werkzeugen ebenso der Fall, so dass diese Aufwandsabschätzung zu vernachlässigen ist. Bei ProcMod und dem Semantic Building Modeler beschränkt sich der Aufwand des Nutzers auf die Eingabe von Parametern. Bei beiden Ansätzen gilt, dass komplexere Gebäude mehr Parameter erfordern. Im Vergleich zu den anderen Systemen ist die notwendige Nutzerintervention allerdings immer noch gering.

Dies ist bei der GML anders, da diese einen anderen Ansatz verfolgt und ihr Einsatzbereich deutlich weiter gefasst ist. Bei dieser muss man die Konstruktionsbeschreibung

programmieren und kann diese anschließend wiederverwenden, um Varianten des beschriebenen Objekts zu erzeugen.

Die Wiederverwendbarkeit ist bei ProcMod und dem Semantic Building Modeler ebenfalls schwerer zu beurteilen als dies bei Regelsystemen oder Modellierungsanwendungen der Fall ist. Auf der Ebene der Algorithmen sind diese natürlich für jedes neue Gebäude wiederverwendbar und werden nur neu parametrisiert. Die Parameter selber sind bezüglich ihrer Werte nicht wiederverwendbar, da sie für die Erstellung anderer Gebäudetypen modifiziert werden müssen. Somit ist die Wiederverwendbarkeit gering, dafür aber der Erstellungsaufwand selber sehr niedrig.

Bei der Continuous Model Synthesis besteht die Hauptaufgabe des Nutzers in der Erstellung eines Beispielmodells, das als Eingabe in den Algorithmus fungiert. Je nach Komplexität des Gebäudes variiert auch der Erstellungsaufwand. Inwiefern das System auch mit hochkomplexen Eingabemodellen umgehen kann, ist unklar. Einfache Massemodelle ohne architektonische Fassadenelemente lassen sich in Modellierungssoftware vergleichsweise schnell realisieren, da die zugrunde liegenden Körper sehr einfach sind. Insofern ist davon auszugehen, dass für solche Modelle auch der Nutzeraufwand als gering einzuschätzen ist.

System	Aufwand	Wiederverwendbarkeit
3D-Modellierungswerkzeuge bsw. SketchUp	sehr hoch	nicht vorhanden
Fotogrammetrische Technologien bsw. Façade	niedrig-mittel	nicht vorhanden
Build-by-Number	hoch	mittel
CityEngine	hoch	hoch
GML	hoch	mittel
ProcMod	niedrig	mittel
Continuous Model Synthesis	mittel	hoch
Semantic Building Modeler	niedrig	mittel

Tabelle 2: Aufwand und Wiederverwendbarkeit für die Erstellung komplexer Gebäudemodelle

Bezüglich der Wiederverwendbarkeit gilt, dass einmal erstellte Modelle beliebig oft als Eingabe in das Verfahren verwendet werden können. Aufgrund des randomisierten Algorithmus sollten auch bei wiederholten Eingaben des gleichen Gebäudes immer wieder neue Variationen erzeugt werden können.

6.3 **Geometrische Komplexität der erstellten Modelle**

Das nächste Unterscheidungskriterium zur Beurteilung der verschiedenen Systeme ist die geometrische Komplexität, die durch die Technologien erzeugt werden kann.

Diese ist bei Modellierungswerkzeugen potentiell unbegrenzt und hängt ausschließlich von der Expertise des Nutzers im Umgang mit dem System ab.

Fotogrammetrische Ansätze sind prinzipiell nur sehr eingeschränkt in der Lage, komplexe Geometrien abzubilden. Dies hängt damit zusammen, dass mit steigender Komplexität die Anzahl der erkannten und in die Geometrie integrierten Details zunehmen muss. Verdeckungsprobleme bei verschachtelten Strukturen wie Gesimsen führen dazu, dass solche Strukturen nicht oder nur unvollständig aus Fotografien ermittelt werden können. Um solche Details extrahieren zu können, benötigt man eine deutlich größere Abtastrate, wie sie beispielsweise durch den Einsatz von Laserscannern erreicht werden kann. Allerdings haben auch solche Verfahren große Probleme mit Strukturen, die teilweise verdeckt sind. Dies ist der Grund, warum viele fotogrammetrische Ansätze mit Technologien des Image Based Renderings arbeiten und blickwinkelabhängiges Texturmapping verwenden. Diese Verfahren täuschen geometrische Komplexität nur vor, Details werden ausschließlich durch das Mapping der Fotografien erzeugt.

Die CityEngine und der Semantic Building Modeler verwenden einen hybriden Ansatz, bei dem Komponenten mit hoher geometrischer Komplexität, die prozedural nicht oder nur schwer beschreibbar sind, durch das Laden vorhandener 3D-Modelle in die Gebäude integriert werden. Während dies bei der CityEngine durch die Formulierung von Regeln erfolgt, ist das Laden von Modellen beim Semantic Building Modeler abhängig von den Algorithmen. So lädt die Methode zur Positionierung von Fenstern nur Fenstermodelle, bei der CityEngine muss der Nutzer dagegen explizit angeben, welches Modell geladen werden soll, da die Regeln keine Semantik besitzen. ProcMod verfolgt einen rein prozeduralen Ansatz, der der Technologie zur Erzeugung von Gesimsen im Semantic Building Modeler ähnelt. Finkenzellers System produziert Kantenverläufe von Fenstern, Türen oder Gesimsen

vollständig basierend auf formalen Beschreibungen. Im Semantic Building Modeler werden Gesimse oder Fensterbänke ebenfalls nicht als 3D-Modelle geladen, sondern basieren gleichfalls auf der Angabe von Profilquerschnitten. Deren Form wird allerdings nicht in einer Beschreibungssprache definiert, sondern direkt als Polygonzug in das System geladen.

Die GML ist in Bezug auf die geometrische Komplexität ebenfalls sehr mächtig, da sie sämtliche Operationen für die Modifikation von 3D-Modellen zur Verfügung stellt, wenn auch auf einer sehr niedrigen Abstraktionsebene. Somit ist die Komplexität vergleichbar mit der, die durch 3D-Modellierungssysteme erreicht werden kann, wenn auch auf einem anderen Wege.

System	Geometrische Komplexität
3D-Modellierungswerkzeuge bsw. SketchUp	unbegrenzt
Fotogrammetrische Technologien bsw. Façade	gering
Build-by-Number	gering
CityEngine	sehr hoch[1]
GML	unbegrenzt
ProcMod	hoch
Continuous Model Synthesis	mittel
Semantic Building Modeler	sehr hoch[1]

Tabelle 3: Geometrische Komplexität

[1]Komplexität bezieht sich sowohl auf die geometrische Komplexität, als auch auf die Komplexität der geladenen Komponentenmodelle (Fenster etc.), die vorab in Modellierungsanwendungen erstellt wurden.

Bezüglich des Modell-Synthese-Ansatzes beschränkt sich die geometrische Komplexität auf die Erstellung potentiell komplexer Massemodelle. Inwieweit das System in der Lage ist, komplexere Details wie beispielsweise Gesimse zu erzeugen, ist unklar. Allerdings ist davon auszugehen, dass sich der Algorithmus nicht dafür eignet, Elemente wie Türen oder Fenster in der Fassade zu variieren dieser hinzuzufügen.

Nachdem die vorherigen Kriterien vergleichsweise allgemein gehalten waren, werden die unterschiedlichen Systeme nun anhand spezifischer Merkmale verglichen, die für das Erreichen der Zielsetzungen des Semantic Building Modelers relevant sind. Zunächst werden darum die unterschiedlichen Ansätze besprochen, mittels derer die Systeme

Grundrisse erzeugen. Dabei spielen mehrere Fragen eine Rolle. Zunächst wird verglichen, wie Grundrisse in das System eingegeben werden und ob eine Möglichkeit besteht, Grundrisse automatisiert zu erstellen. Anschließend wird verglichen, inwiefern Grundrisse über verschiedene Stockwerke hinweg variiert werden können und ob auch hier automatische Variationen möglich sind.

6.4 **Grundrisseingabe und -typen**

Bei Modellierungswerkzeugen gilt allgemein, dass die Grundrissstruktur beliebig ist, da sie durch den Nutzer vollständig festgelegt wird, sowohl für das Erdgeschoss, als auch für alle anderen Stockwerke. Dadurch sind beliebige Variationen möglich. Allerdings gibt es keinerlei Möglichkeiten, Grundrisse automatisiert zu erstellen oder zu modifizieren.

Bei rein fotogrammetrischen Ansätzen besteht die Eingabe in einer Menge von Fotografien des zu rekonstruierenden Gebäudes. Dadurch ist auch der Grundriss dieser Gebäude bereits vorgegeben. Dieser wird durch den Nutzer unter Verwendung der volumetrischen Grundkörper nachgebaut und kann somit auch auf unterschiedlichen Stockwerken variiert werden. Automatische Variationen ermöglichen diese Ansätze dagegen nicht.

Die GML ähnelt auch in diesem Punkt stärker den Modellierungswerkzeugen als den anderen vorgestellten Systemen, da sie in der Lage ist, beliebige Grundrisse zu erzeugen und zu verwenden. Konzeptuell ist sie aufgrund ihres Modellierungsparadigmas in der Lage, Variationen zu generieren, allerdings müssen die hierfür erforderlichen Algorithmen zunächst implementiert werden. Verwendet man die GML als reines Modellierungswerkzeug und beschreibt durch prozedurale Techniken die Konstruktion eines Gebäudes, so sind automatische Variationen nicht möglich, dafür aber beliebige Grundrisse.

Die CityEngine bietet mehrere Ansätze zur Festlegung von Grundrissen, sei es durch die Verwendung der L-Systeme zur Straßennetzerzeugung oder durch den GIS-Import. Weiterhin können auch einzelne Gebäude durch CGA-Shape modelliert werden, bei denen man den Grundriss durch Ersetzungsregeln aus einem initialen Massemodell erzeugt. Durch solche Modifikationen ist es möglich, Stockwerke mit unterschiedlichen Grundrissen zu verwenden. Automatisierte Variationen sind dabei im System nicht vorgesehen.

Auch ProcMod [Fi08] gestattet die Gestaltung beliebiger Grundrisse durch das Grundrissmodulkonzept. Obwohl sich das System konzeptuell dafür eignen würde, müssen allerdings auch hier die Grundrisse aller Stockwerke durch den Nutzer manuell vorgegeben werden.

Im Semantic Building Modeler existieren verschiedene Möglichkeiten für die Grundrissfestlegung. Es ist möglich, Grundrisse als Polygone zu laden und für unterschiedliche Stockwerke unterschiedliche Grundrisse zu erzeugen. Weiterhin können programmierte Grundrissklassen erzeugt und verwendet werden, die auch über Innenraumstrukturen verfügen.

System	**Grundrisseingabemöglichkeiten**	**Unterschiedliche Grundrisse auf unterschiedlichen Stockwerken**
3D-Modellierungswerkzeuge bsw. SketchUp	Manuelle Gestaltung	✓
Fotogrammetrische Technologien bsw. Façade	Aus Quellfotografien abgeleitete Grundrisse	✓
Build-by-Number	Durch Quellfotografien abgeleitete Grundrisse	✓
CityEngine	L-System, GIS-Daten, Manuelle Spezifikation durch CGA-Shape-Regeln	✓
GML	Manuelle Gestaltung	✓
ProcMod	Manuelle Gestaltung	✓
Continuous Model Synthesis	Grundrisseingabe durch Bereitstellung eines Beispielgebäudes	✓
Semantic Building Modeler	Eingabe als Grundrisspolygon, Automatische Erzeugung	✓

Tabelle 4: Grundrisseingabe und -typen

Außerdem ist es möglich, Grundrisse durch zufallsbasierte Modifikationen vorheriger Grundrisse zu erzeugen und auf anderen Stockwerken zu verwenden. Als letzte Möglichkeit

zur Grundrissgenerierung implementiert der Semantic Building Modeler eine ähnlichkeitsbasierte Grundrisssynthese, bei der basierend auf einem Eingabegrundriss zu diesem ähnliche Grundrisse errechnet werden.

Bei der Continuous Model Synthesis [MM08] erfolgt die Eingabe des Grundrisses durch das Bereitstellen eines Beispielgebäudes. Dessen Struktur definiert den Ausgangspunkt für die Erzeugung von Grundrissvarianten. Somit ist es sowohl möglich, unterschiedliche Grundrisse für unterschiedliche Stockwerke einzusetzen, als auch Grundrisse vollständig automatisch basierend auf der Eingabe zu erzeugen. Im Gegensatz zu den anderen vorgestellten Systemen kann der Nutzer allerdings nicht festlegen, wie die verwendeten Grundrisse aussehen, da sie vollständig prozedural erzeugt werden und es keinerlei Möglichkeiten gibt, die Grundrisserzeugung zu beeinflussen.

Als weiteres Kriterium wird die Frage betrachtet, ob die Systeme die Möglichkeit bieten, Grundrisse auch für Innenräume festzulegen und dadurch Innenraumstrukturen abzubilden. Dies ist sowohl bei der GML als auch bei den Modellierungswerkzeugen der Fall.

System	Automatische Variation	Möglichkeit, Grundrisse für Innenräume festzulegen
3D-Modellierungswerkzeuge bsw. SketchUp	×	✓
Fotogrammetrische Technologien bsw. Façade	×	×
Build-by-Number	×	×
CityEngine	×	×
GML	×	✓
ProcMod	×	×
Continuous Model Synthesis	✓	×
Semantic Building Modeler	✓	✓

Tabelle 5: Automatische Grundrisserzeugung und Innenräume

Der Semantic Building Modeler ist bezüglich der verbleibenden Systeme das Einzige, das in der Lage ist, Innenraumstrukturen zu erzeugen. Allerdings können diese im Gegensatz zu Grundrissen nicht automatisch generiert, sondern müssen vorgegeben werden.

6.5 **Technologien zur Erzeugung von Gebäudekomponenten**

Nachdem die verschiedenen Ansätze bezüglich der Grundrisserzeugung verglichen wurden, wird als nächstes Kriterium betrachtet, wie Gebäudekomponenten, also bsw. Fenster, Türen und Gesimse, in den unterschiedlichen Systemen erzeugt werden und ob Möglichkeiten vorgesehen sind, automatische Variationen bezüglich der integrierten Komponenten zu realisieren.

3D-Modellierungswerkzeuge und die GML erfordern eine manuelle Erzeugung der jeweiligen Komponenten. Diese Komponenten können anschließend kopiert und wiederholt in den Gebäuden positioniert werden. Bei 3D-Modellierungssystemen sind solche automatischen Variationen nicht möglich. Die GML ist dagegen in der Lage, für Komponententypen, deren Konstruktion prozedural beschreibbar ist, Variationen zu erstellen. Dies zeigt Havemann am Beispiel gotischer Fenster, die sich aufgrund ihrer hohen Selbstähnlichkeit und der vergleichsweise einfachen geometrischen Grundobjekte gut prozedural konstruieren lassen [Ha05]. Inwiefern solche Varianten auch für Komponententypen erzeugt werden können, bei denen diese Vorbedingungen nicht gegeben sind, ist unklar.

Fotogrammetrische Technologien und das Build-by-Number-Verfahren erzeugen Komponenten nicht in der konkreten Geometrie, sondern setzen auf blickwinkelabhängige Texturierung. Somit bieten diese Verfahren keinerlei Möglichkeiten für die Integration komplexer Komponenten [BA05].

Der Semantic Building Modeler und ProcMod [Fi08] setzen auf ein hybrides Vorgehen, bei dem Elemente prozedural erzeugt oder direkt aus 3D-Modellen geladen und integriert werden. Die Systeme unterscheiden sich in der Häufigkeit, mit der die verschiedenen Ansätze verwendet werden und darin, inwiefern zufällige Variationen möglich sind. Der Semantic Building Modeler lädt die meisten Komponenten aus 3D-Modelldateien, die vorab in Modellierungswerkzeugen erstellt wurden. Dazu gehören beispielsweise Fenster und Türen. Bei Gesimsen wird das Profil geladen und anschließend an der Fassade appliziert. Durch die Gruppierung der geladenen 3D-Modelle und Profile nach Stilrichtungen können

zufallsbasiert Modelle geladen und positioniert werden. Dadurch entstehen Variationen in den erzeugten Gebäuden.

System	Verfahren zur Erzeugung von Komponenten	Automatische Auswahl/Variation möglich?
3D-Modellierungswerkzeuge bsw. SketchUp	Manuelle Erzeugung der Komponenten	×
Fotogrammetrische Technologien bsw. Façade	keine Möglichkeit zur Erzeugung von Komponenten	×
Build-by-Number	keine Möglichkeit zur Erzeugung von Komponenten	×
CityEngine	Laden von Komponenten in Form von 3D-Modellen	×
GML	Manuelle Erzeugung der Komponenten	Abhängig von der Art der jeweiligen Komponente
ProcMod	Laden von Komponenten in Form von 3D-Modellen, prozedurale Komponentenerzeugung	×
Continuous Model Synthesis	×	×
Semantic Building Modeler	Laden von Komponenten in Form von 3D-Modellen, prozedurale Komponentenerzeugung	Variation basierend auf Komponentengruppierung in Modellbibliotheken

Tabelle 6: Technologien zur Erzeugung von Gebäudekomponenten (beispielsweise Fenster, Gesimse)

ProcMod lädt nur Konsolen als 3D-Objekte, alle anderen Komponenten werden prozedural basierend auf festen Beschreibungen erzeugt. Dies entspricht dem grundsätzlichen Konzept

von ProcMod, das darauf basiert, einen exakten Bauplan für ein bestimmtes Gebäude anzugeben, automatische Variationen sind in diesem Paradigma nicht vorgesehen.

Die CityEngine ermöglicht ebenfalls das Laden von 3D-Modellen und deren Integration in die Fassade. Die Auswahl erfolgt über CGA-Shape-Regeln. Dabei müssen die Modelle fest angegeben werden, eine zufallsbasierte Modellwahl ist nicht möglich. Der Continuous Model Synthesis-Ansatz bietet keine Lösung für das Problem der Erzeugung und Positionierung komplexer Fassadenelemente, da er Texturen für die Darstellung von Fassaden verwendet [MM08].

6.6 **Dachgenerierung**

Die Dacherzeugung ist ein weiteres komplexes Problem im Bereich der Gebäudemodellierung. Dabei bestehen zwei zentrale Forderungen an das eingesetzte Verfahren. Es muss sowohl für beliebig komplexe Grundrisse einsetzbar sein als auch unterschiedliche Dachformen, vom Giebel- bis zum Mansardendach, erzeugen können.

Auch in Bezug auf Dächer und Dachformen sind Modellierungswerkzeuge und die GML uneingeschränkt in der Lage, beliebige Dachformen für beliebige Grundrisse zu generieren.

Bei den fotogrammetrischen Ansätzen unterscheidet sich die Dacherzeugung nicht von der Erzeugung der anderen Komponenten des Gebäudes. Façade und Build-by-Number verwenden volumetrische Grundkörper und weisen die Kanten der Grundkörper den Dachkanten innerhalb der Fotografien zu. Somit sollten die Systeme in der Lage sein, beliebige Dachformen zu erzeugen. Allerdings ist davon auszugehen, dass auch im Bereich der Dächer mit steigender Komplexität von Dach und Grundriss der Aufwand der Geometrieextraktion zunimmt.

Die CityEngine greift auf ein hybrides Vorgehen zurück, das für einfache Grundrisse in der Lage ist, beliebige Dachformen zu generieren. Lassen sich die Grundrisskomponenten nicht in die vorhandenen Basistypen klassifizieren, wird eine Implementation des Straight-Skeleton-Algorithmus eingesetzt, die in der Basisversion nur Walmdächer erzeugen kann. Somit hängt die Fähigkeit zur Dachgenerierung von der Komplexität der zugrundeliegenden Grundrisse ab.

ProcMod setzt auf einen ähnlichen Algorithmus, bei dem Dächer unterschiedlichen Typs für die einzelnen Grundrissmodule berechnet und nachfolgend bei Bedarf miteinander

verschmolzen werden [Fi08]. Unterschiedliche Dachtypen können über unterschiedliche Parameterwerte erzeugt werden.

System	Dacherzeugungsverfahren	Beliebige Grundrisse?
3D-Modellierungswerkzeuge bsw. SketchUp	Manuelle Dacherzeugung	✓
Fotogrammetrische Technologien bsw. Façade	Dacherzeugung durch Geometrieextraktion	✓
Build-by-Number	Dacherzeugung durch Geometrieextraktion	✓
CityEngine	Dacherzeugung durch Grundrissklassifikation + Verschmelzung / Straight-Skeleton	✓
GML	Manuelle Dacherzeugung	✓
ProcMod	Dacherzeugung für konvexe Grundrissmodule + Verschmelzung	✓
Continuous Model Synthesis	Dacherzeugung wird analog zur Erzeugung aller anderen Komponenten durchgeführt	unklar
Semantic Building Modeler	Erweiterte, gewichtete Straight- Skeleton-Implementation	✓

Tabelle 7: Verfahren zur Dacherzeugung / Unterstützung beliebiger Grundrisse

Die Dacherzeugung im Semantic Building Modeler erfolgt durch eine modifizierte Version des Straight- Skeleton-Algorithmus, die es erlaubt, den einzelnen Kanten des Dachgrundrisses unterschiedliche Gewichte zuzuweisen. Diese Gewichte beeinflussen die

Dachneigung, wodurch sich verschiedene Dächer für beliebig komplexe, also sowohl konvexe als auch konkave Grundrisse erzeugen lassen.

Die Continuous Model Synthesis behandelt Dächer wie alle anderen Komponenten eines Gebäudes. Es existiert kein spezielles Verfahren, mittels dessen sich explizit Dächer erstellen lassen, die Form der automatisch erzeugten Dächer hängt somit stark vom Eingabemodell ab [MM08]. Ob das System auch für Beispielgebäude mit komplexen Dachstrukturen in der Lage ist, Variationen dieser Gebäude zu berechnen, ist unklar und aus der Arbeit nicht zu erkennen.

System	**Beliebige Dachformen?**
3D-Modellierungswerkzeuge bsw. SketchUp	✓
Fotogrammetrische Technologien bsw. Façade	eingeschränkt
Build-by-Number	eingeschränkt
CityEngine	Beliebige Dachformen bei klassifizierten Grundrissen, sonst Straight-Skeleton Walmdach
GML	✓
ProcMod	✓ Dacherzeugung für konvexe Grundrissmodule + Verschmelzung
Continuous Model Synthesis	unklar
Semantic Building Modeler	✓ Erweiterte, gewichtete Straight-Skeleton-Implementation

Tabelle 8: Erzeugung beliebiger Dachformen?

7 Systemarchitektur

7.1 **Die Komponenten des Semantic Building Modelers**

Nachdem im ersten Teil der vorliegenden Arbeit wichtige Kerntechnologien für die prozedurale Gebäudegenerierung vorgestellt wurden, widmet sich der zweite Teil dem Semantic Building Modeler und den darin umgesetzten Konzepten und Algorithmen. Bevor die einzelnen Bestandteile des Systems detailliert in ihrer Funktion erläutert werden, soll in diesem Abschnitt ein Überblick über die Gesamtarchitektur des Systems gegeben werden, in dem sämtliche verwendeten Submodule des Hauptsystems in ihrer Funktion kurz erläutert werden.

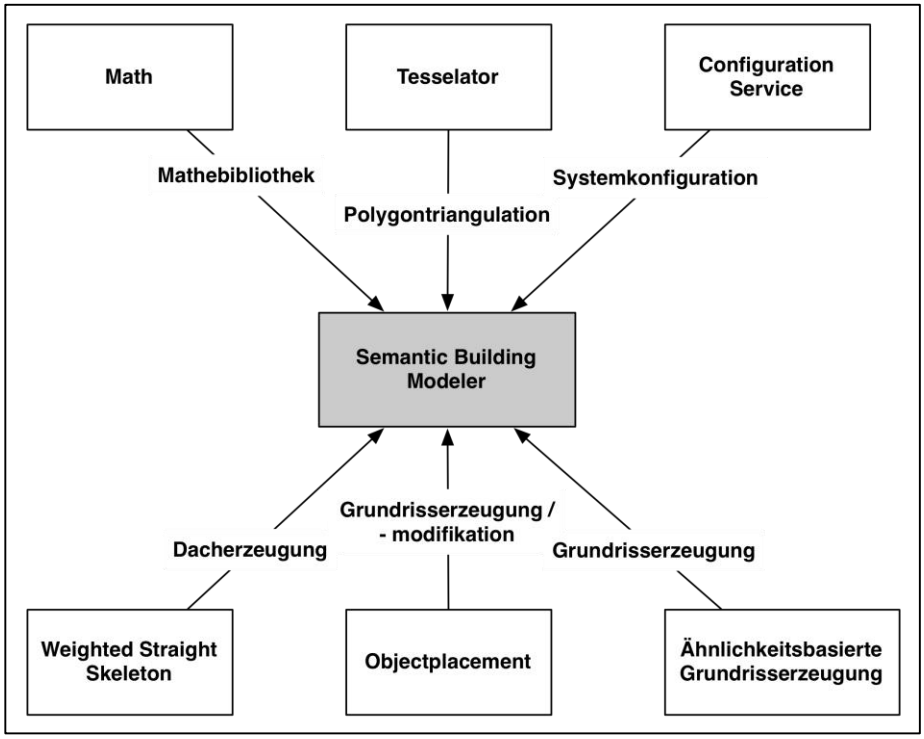

Abbildung 59: Module des Semantic Building Modelers

Abbildung 59 zeigt eine schematische Darstellung der verschiedenen Module, die durch den Semantic Building Modeler verwendet werden. Diese Submodule stellen dem Hauptmodul zum Teil umfangreiche Funktionalitäten zur Verfügung oder implementieren Verfahren losgelöst von den Strukturen des Hauptsystems. Speziell diese Auslagerung umfangreicher Verfahren ist im Kontext großer Softwaresysteme wünschenswert, da sie es ermöglicht, die einzelnen Komponenten getrennt voneinander zu entwickeln und zu testen. Bis auf die Mathe-Bibliothek sind sämtliche anderen Module lauffähig und können losgelöst vom Hauptprogramm ausgeführt werden. Die Kommunikation des Hauptmoduls mit den

Submodulen erfolgt über klar festgelegte Schnittstellen, an denen die Module ihre Funktionalität zur Verfügung stellen. Dies kann die Abfrage eines Konfigurationsparameters oder auch die vollständige Berechnung eines Daches sein, je nachdem, welche Aufgabe durch ein spezifisches Submodul realisiert wird.

7.1.1 **Das Math-Submodul**

Das Math-Submodul ist das einzige Modul, das nicht ausgeführt werden kann. Es handelt sich um ein reines Bibliotheksmodul, das sowohl Funktionen als auch Datenstrukturen für die mathematischen Berechnungen innerhalb des Hauptmoduls zur Verfügung stellt. Dabei greift nicht nur das Hauptmodul auf die bereitgestellten Methoden und Strukturen zurück. Bis auf das Configuration Service-Modul verwenden sämtliche in Abbildung 59 dargestellten Module das Math-Submodul.

7.1.2 **Das Tesselator-Submodul**

Die Tesselator-Komponente stellt innerhalb des Semantic Building Modelers die notwendige Funktionalität bereit, um beliebige Polygone in Dreiecke zu unterteilen. Dazu gehören sowohl konvexe als auch konkave Polygone sowie solche, die Löcher enthalten. Dabei greift das System auf die Funktionen zurück, die die *OpenGL Utility Library* (GLU) bereitstellt und integriert diese in das Tesselator-Modul. Die GLU bietet die erforderlichen Methoden, um beliebige Polygone derart zu unterteilen, dass sie durch eine dreiecksbasierte Geometrieverwaltung verarbeitet werden können, liefert dabei aber verschiedene Formen der Dreieckslisten zurück, die durch den Semantic Building Modeler nicht unterstützt werden.

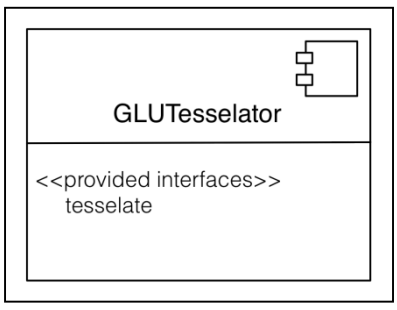

Abbildung 60: UML-Komponentendiagramm des GLUTesselator-Submoduls

Dazu gehören beispielsweise Triangle Fans oder Triangle Meshes [Sh08]. Das Tesselator-Submodul fungiert in diesem Zusammenhang als *Fassade* [Ga12] im Sinne eines Entwurfsmuster, die die OpenGL-spezifische Funktionalität vor dem Aufrufer versteckt und die Rückgabe der GLU in das Format umwandelt, welches vom Semantic Building Modeler verwendet wird. Abbildung 60 zeigt das Komponentendiagramm des GLUTesselator-Submoduls inklusive der vom Modul durch die Methode `tesselate` bereitgestellte Funktionalität.

7.1.3 **Das Configuration-Service-Submodul**

Das Configuration-Service-Submodul ähnelt der Mathe-Bibliothek darin, dass es sich um ein Modul handelt, welches nicht nur vom Hauptmodul verwendet wird. Vielmehr stellt es auch anderen Submodulen seine Funktionen für die XML-basierte Service-Konfiguration zur Verfügung. Die Aufgabe des Configuration-Service besteht in der Verarbeitung von XML-Konfigurationen für unterschiedliche Dienste. Dies können Konfigurationen für die Dachberechnung im Weighted-Straight-Skeleton-Modul sein ebenso wie Systemkonfigurationsparameter, die für die Initialisierung des Hauptmoduls verwendet werden. Allen diesen Konfigurationen ist gemein, dass der Nutzer sie in Form von XML-Dateien bereitstellt. Außerdem existiert für jeden Konfigurationstyp ein XML Schema-Dokument, mittels dessen die Konfigurationen validiert werden können.

Die XML Schema-basierte Validierung ist die zweite große Aufgabe des Configuration-Service-Submoduls. Diese Gültigkeitsprüfung garantiert die Verwendbarkeit der Konfigurationen innerhalb der jeweiligen Submodule.

Abbildung 61 zeigt die schematische Darstellung der Configuration-Service-Komponente. Die innerhalb des Diagramms eingezeichneten Methoden `processSystemConfiguration` und `processCityConfiguration` werden vom Hauptmodul verwendet.

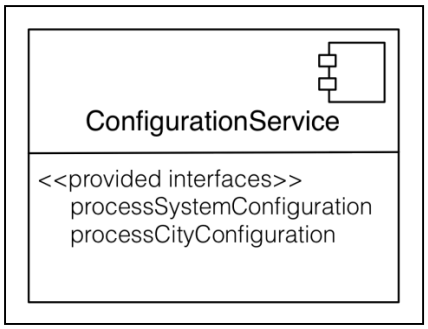

Abbildung 61: UML-Komponentendiagramm des Configuration-Service-Submoduls

Für die anderen Submodule stehen weitere modulspezifische Methoden zur Verfügung, auf deren Darstellung aus Gründen der Übersichtlichkeit verzichtet wurde. Die erste Methode `processSystemConfiguration` verarbeitet sämtliche Parameter, die für die Konfiguration des Hauptmoduls selber relevant sind. Dazu gehören Parameter wie die Ausdehnungen des Anwendungsfensters oder Pfade zu Texturen und Gebäudekonfigurationen. Die Parameter für die Erzeugung von Städten werden dagegen durch `processCityConfiguration` verarbeitet. Auf die vorhandenen Strukturen und Elemente, mittels derer der Nutzer Stadtkonfigurationen festlegt, wird im Kapitel „Das Konfigurationsmodul“ detailliert eingegangen.

7.1.4 **Das Weighted-Straight-Skeleton-Submodul**

Das Weighted-Straight-Skeleton-Submodul implementiert eine erweiterte Variante des ursprünglich von Aichholzer et al. vorgestellten Straight-Skeleton-Algorithmus [Ai95]. Dieses Verfahren wird verwendet, um für beliebige Grundrisse Dächer zu konstruieren. Im Gegensatz zu der ursprünglichen Variante gestattet die vorliegende Implementation die Vergabe unterschiedlicher Gewichte für unterschiedliche Kanten des Eingabegrundrisses. Dadurch ist es möglich, die Neigung der einzelnen Dachflächen zu variieren. Dies erlaubt die Berechnung unterschiedlicher Dachformen mittels eines einheitlichen Verfahrens für beliebige Grundrisse. Die Implementation selber ist vergleichsweise umfangreich und komplex. Sie umfasst neben den algorithmischen Berechnungen auch die Extraktion der Dachkonstruktion aus den Ergebnissen des Algorithmus und deren Überführung in eine Struktur, die möglichst einfach vom Hauptmodul verarbeitet werden kann. Hier ähnelt das Modul der vorab im Zusammenhang mit dem GLUTesselator-Modul angesprochenen Fassadenstruktur, die die Komplexität der eigentlichen Berechnungen vor dem Aufrufer versteckt. Das Modul selber wird vollständig über eine *Controller-Klasse* gesteuert, der bei

der Initialisierung ein Konfigurationsobjekt übergeben wird, welches sämtliche für die Berechnung erforderlichen Parameter enthält. Dazu gehören unter anderem der Grundriss, für den das Dach bestimmt werden soll, sowie die Gewichtungen der einzelnen Grundrisskanten. Auf der Grundlage dieser Parameter ermittelt das Modul ein Dach, welches dem Hauptmodul durch die Methode `getResultComplex` zur Verfügung gestellt wird. Abbildung 62 zeigt das Komponentendiagramm des Weighted-Straight-Skeleton-Moduls

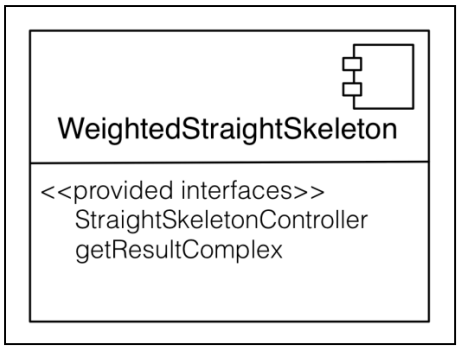

Abbildung 62: UML-Komponentendiagramm des Weighted-Straight-Skeleton-Submoduls

7.1.5 **Das Objectplacement-Submodul**

Das Objectplacement-Submodul dient der Erzeugung neuer und der Modifikation bereits vorhandener Grundrisspolygone. Das hierfür entwickelte Verfahren basiert auf der Verwendung einfacher zweidimensionaler Grundformen wie Rechtecke oder Zylinder, die nach einem Baukastenprinzip miteinander kombiniert werden, um komplexe Grundrissformen zu erzeugen. Der Nutzer kann das Verfahren durch eine Reihe von Konfigurationsparametern steuern. Anhand dieser Steuerparameter berechnet der Algorithmus zunächst eine Menge von Komponenten, aus denen er anschließend ein Polygon extrahiert, welches als Stockwerksgrundriss eingesetzt werden kann. Das Objectplacement-Submodul ist das erste Modul für die Grundrisserzeugung, über das der Nutzer in der Lage ist, Grundrissstrukturen algorithmisch zu erzeugen. Seine Struktur ist in Abbildung 63 dargestellt. Analog zum Weighted-Straight-Skeleton-Submodul wird auch diesem Modul bei der Initialisierung ein Konfigurationsobjekt übergeben, das die notwendigen Parameter und Daten enthält. Basierend auf diesen Informationen erzeugt der Algorithmus eine Menge von Komponenten, die anschließend durch das Hauptmodul abgerufen werden. Abschließend wird innerhalb des Hauptmoduls entweder durch den an späterer Stelle vorgestellten *Footprint-Merger-Algorithmus* (s. Kapitel „Der Footprint-

Merging-Algorithmus – Verschmelzung beliebiger Gebäudekomponenten“) oder durch eine *Convex-Hull-Implementation* (s. Kapitel „Grundrissextraktion als Convex-Hull-Problem“) das Ergebnispolygon extrahiert.

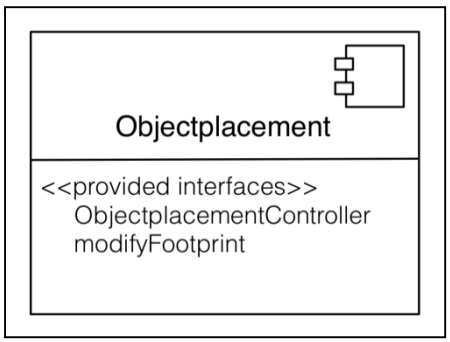

Abbildung 63: UML-Komponentendiagramm des Objectplacement-Submoduls

7.1.6 **Das Submodul für die ähnlichkeitsbasierte Grundrisserzeugung**

Das Submodul für die ähnlichkeitsbasierte Grundrisserzeugung basiert auf dem ursprünglich von Merrell et al. vorgestellten Continuous Model Synthesis-Algorithmus [MM08], der im Kapitel „Continuous Model Synthesis [MM08]“ vorgestellt wurde. Für den Semantic Building Modeler wurde dieser Algorithmus erweitert und angepasst, um für die Grundrisssynthese eingesetzt zu werden. Im Gegensatz zum vorab vorgestellten Objectplacement-Submodul eignet sich dieses Verfahren nicht zur Modifikation bereits vorhandener Grundrisse, sondern wird für die Erzeugung neuer Grundrisspolygone eingesetzt.

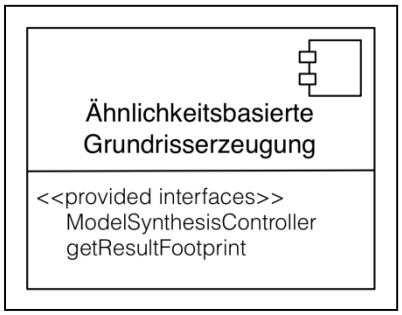

Abbildung 64: UML-Komponentendiagramm des Moduls für die ähnlichkeitsbasierte Grundrisserzeugung

Auch das Modul für die ähnlichkeitsbasierte Grundrisserzeugung, dessen Struktur in Abbildung 64 dargestellt ist, erhält für die Berechnungsschritte zunächst ein Konfigurationsobjekt, das sämtliche für das Verfahren erforderlichen Parameter enthält.

Aufgrund dieser Parameter erzeugt der Algorithmus ein Ergebnispolygon. Dieses kann durch das Hauptprogramm über die Methode `getResultFootprint` abgerufen und weiterverarbeitet werden.

7.1.7 **Der Semantic Building Modeler – das Hauptmodul**

Nachdem vorab die verschiedenen Submodule vorgestellt wurden, auf deren Funktionalität der Semantic Building Modeler zurückgreift, soll abschließend kurz auf die Aufgaben des Hauptmoduls eingegangen werden. Zunächst steuert es den eigentlichen Konstruktionsprozess für Gebäude basierend auf den nutzergenerierten XML-Konfigurationsdateien. Zu diesem Zweck benötigt es einerseits Kontrollstrukturen, die für die Wahl der geeigneten Konstruktionsalgorithmen eingesetzt werden, sowie Datenstrukturen, die die Ergebnisse der Berechnung vorhalten. Darüber hinaus implementiert das System die notwendige Funktionalität für das Rendering der Geometriedaten und setzt dabei die Kernfunktionen einer rudimentären 3D-Grafik-Engine um. Dadurch ist es dem Nutzer beispielsweise möglich, die Kamera zu rotieren und die Gebäude von unterschiedlichen Blickwinkeln zu betrachten. Weiterhin enthält das Hauptmodul sämtliche Funktionen zum Laden und zur prozeduralen Generierung von 3D-Komponentenmodellen sowie die Routinen für deren Applikation an bestehende Gebäude. Nach Abschluss sämtlicher Berechnungen können die Modelle aus dem Programm exportiert werden, um sie anschließend in beliebigen 3D-Modellierungsumgebungen weiterzuverarbeiten.

Nachdem in diesem Abschnitt ein kurzer Überblick über die innerhalb des Semantic Building Modelers vorhandenen Komponenten und deren bereitgestellte Methoden gegeben wurde, widmet sich der nächste Abschnitt unter anderem den innerhalb des Hauptmoduls verwendeten Verwaltungsstrukturen für die während der Verarbeitung erzeugten Geometriedaten.

7.2 **Datenstrukturen zur Gebäudeverwaltung**

7.2.1 **Verwaltung der logischen Gebäudekomponenten**

Die Verwaltung der Gebäudekomponenten erfolgt innerhalb des Systems in einer hierarchischen Struktur, deren Organisation in Form des *Composite-Design-Patterns* [Ga12] erfolgt. Die Grundidee dieses Designmusters ist es, die Handhabung von zusammengesetzten (`Composites`) und einzelnen Objekten (`Leafs`) einheitlich zu gestalten.

Dafür implementieren die `Leafs` und `Composites` Methoden einer gemeinsamen Schnittstelle. Abbildung 65 zeigt das UML-Diagramm des Composite-Patterns in der von Gamma et al. [Ga12] vorgestellten Version.

Hauptanwendungsbereich des Composite-Patterns ist die Modellierung von *Teil-Ganzes-Hierarchien*. Solche Hierarchien zeichnen sich dadurch aus, dass a-priori keinerlei Aussagen über die Hierarchiestruktur gemacht werden können. Dies gilt insbesondere für die Tiefe der erzeugten Hierarchie. So können `Composite`-Objekte beliebig viele weitere `Composite`-Objekte enthalten, diese wiederum Mischungen von `Composite`- und `Leaf`-Instanzen usw. Für den Zugriff auf solche Hierarchien ist es aufgrund der beliebigen Struktur wünschenswert, diesen derart zu gestalten, dass der Nutzer kein Wissen darüber besitzen muss, ob er aktuell mit einem `Composite`- oder einem `Leaf`-Objekt kommuniziert. In Abbildung 65 ist das Prinzip des Patterns gut zu erkennen.

Kernidee ist die Ableitung sowohl von `Composite`- als auch von `Leaf`-Klassen von einer gemeinsamen abstrakten Basisklasse `Component`. Die abgeleiteten Klassen implementieren dabei nur die Operationen, die sie tatsächlich benötigen. Bei `Leaf`-Klassen ist dies die Verarbeitungslogik, im Beispiel mit `Operation` bezeichnet. `Composite`-Klassen müssen dagegen auch die Verwaltungsoperationen zum Hinzufügen und Entfernen von `Leaf`-Elementen implementieren, im Diagramm mit `AddComponent`, `Remove` und `GetChild` bezeichnet. Hierbei besteht die typische Umsetzung der `Operation` im `Composite`-Objekt in der Weiterleitung des Methodenaufrufs an alle Komponenten. Dadurch ist es für den Client, der nur die Struktur der abstrakten Basisklasse `Component` kennt, vollkommen transparent, ob er mit einer `Leaf`- oder einer `Composite`-Instanz kommuniziert. Meist geht man bei der Implementation so vor, dass die `Component`-Klasse abstrakt ist und kein reines Interface darstellt. Die Verwaltungsoperationen werden als leere Operationen in der Basisklasse implementiert und nur von der `Composite`-Klasse überschrieben. Dadurch ist es möglich, die `Leaf`-Ableitungen vollkommen frei von jeglichem Verwaltungscode für Mengen von `Leaf`-Instanzen zu halten.

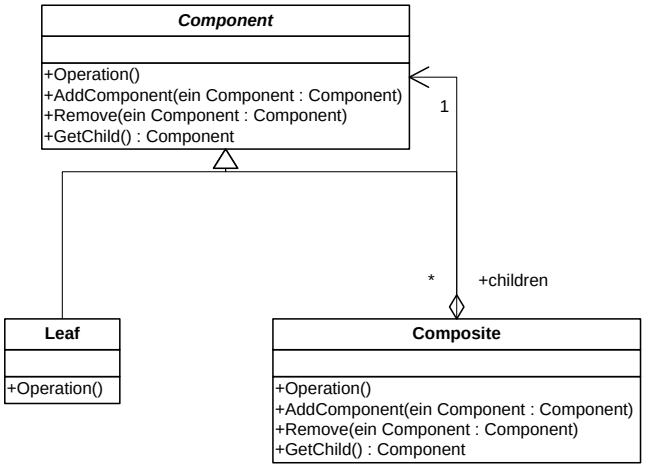

Abbildung 65: Das Composite-Pattern nach Gamma et al.

Im Kontext des Semantic Building Modelers werden die erzeugten Gebäude in Form einer `Composite`-Struktur verwaltet. Gebäude werden als `Composites` umgesetzt, die sowohl weitere Untergebäudestrukturen repräsentiert durch weitere `Composites` aber auch `Leafs` enthalten können. `Leafs` sind in der vorliegenden Implementation beispielsweise Stockwerke oder beliebige andere 3D-Objekte, die entweder prozedural erzeugt oder in das System geladen wurden. Auch die Gebäude selber werden in einem `Composite`-Objekt zusammengefasst. Dieses bildet die Wurzel des Szenegraphen, der sämtliche berechneten Geometriekomponenten enthält. Die durch das Pattern entstehende hierarchische Struktur ist der Grund für die Bezeichnung `Leaf` bei nicht-zusammengesetzten Objekten. Sie bilden die Blätter einer Baumstruktur, deren innere Knoten aus `Composites` bestehen. Durch eine solche Organisation ist der Methodenaufruf für Knoten des Baumes sehr einfach handhabbar. Gute Beispiele für solche Methoden sind Operationen wie das Zeichnen, die Texturierung, die Dachberechnung oder geometrische Berechnungen.

Konzeptuell kann man Gebäude innerhalb dieser Struktur als Container für unterschiedliche Arten von 3D-Daten auffassen. Ein Gebäude kann weitere Gebäudeteile enthalten, aber ebenso komplexe 3D-Objekte. Soll ein Gebäude gerendert werden, so wird der Zeichenbefehl ausschließlich an das Gebäude gesendet. Dieses ist dann dafür zuständig, die Zeichenbefehle so lange durch die Hierarchie zu reichen, bis sie innerhalb eines Blattes ankommen, welches anschließend die konkreten Operationen vornimmt.

7.2.2 **Verwaltung der Geometriedaten**

Nachdem vorab auf die Verwaltung der Gebäudekomponenten eingegangen wurde, befasst sich dieser Abschnitt mit der Organisation der geometrischen Daten. Abbildung 66 zeigt die Repräsentationsstruktur der relevanten Datenstrukturen. Ein Gebäude kann eine beliebige Anzahl von Stockwerken und Komponenten enthalten, die als komplexe Grafikobjekte direkt durch das System gerendert werden. Wie bereits im Kontext der verschiedenen Repräsentationsformen für Polygonnetzmodelle thematisiert, erfolgt die Repräsentation der Polygondaten als hybrider Ansatz, einer Mischung aus punktbasiertem und kantenbasiertem Umrissmodell. Das Gebäude, die Stockwerke und geladene Komponenten sind komplexe Objekte, die das gleiche Interface implementieren wie das Gebäude. Jedes komplexe Objekt besitzt einen eigenen Vertexbuffer sowie eine Menge von primitiven Objekten, aus denen es zusammengesetzt ist.

Solche primitiven Objekte sind ebenfalls hierarchisch organisiert, auf der obersten Ebene steht das Polygon, das aus einer Menge von Dreiecken besteht. Jedes Dreieck referenziert drei Kanten, jede Kante verweist auf zwei Vertices, die ihren Anfangs- und Endpunkt definieren. Diese Hierarchie ist in Abbildung 66 dargestellt. Zwischen den Ebenen sind die Verbindungen bidirektional, somit kennt jedes Dreieck das Polygon, zu dem es gehört. Analog besitzen Polygone Zeiger auf die komplexen Objekte, deren Teile sie sind. Dies gilt nicht für Kanten und die in der Hierarchie darunter liegenden primitiven Objekttypen, da diese in verschiedenen Objekten referenziert werden können, beispielsweise dann, wenn sich Polygone eine Kante und somit auch Vertices teilen.

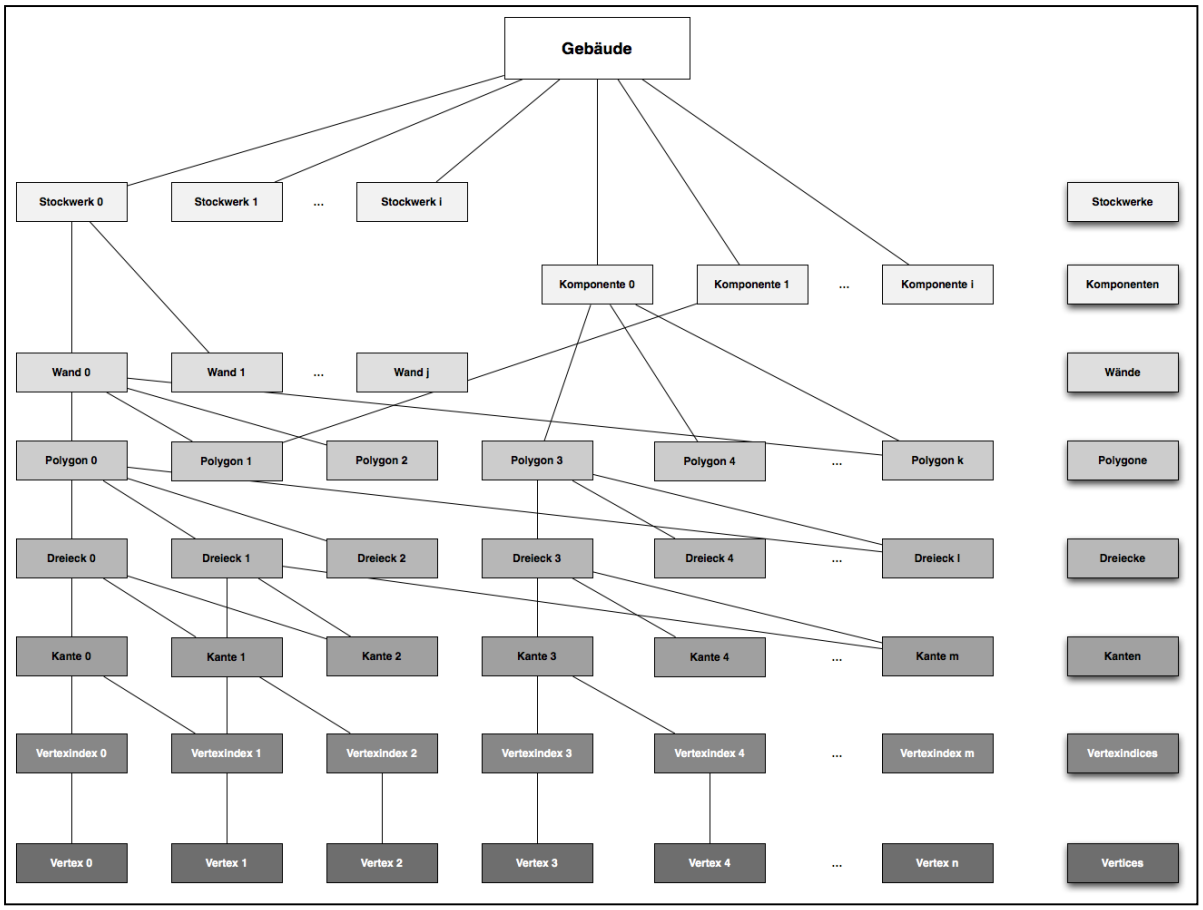

Abbildung 66: Hierarchische Repräsentation eines Gebäudes

Wird ein komplexes Objekt erzeugt, beispielsweise durch Laden des Objekts aus einer Datei, so wird es zunächst als eigenständiges Objekt verwaltet. Es bekommt eine eindeutige Identifikationsnummer und meldet sich bei der Objektverwaltung an. Zu diesem Zeitpunkt besitzt es einen eigenen Vertexbuffer und eine beliebige Anzahl primitiver Subobjekte. Die Subobjekte selber referenzieren die Vertices über Indices, wobei die Indexanzahl von der Art der Subobjekte abhängt. Wird ein solches Objekt nun einem `Composite`-Objekt, beispielsweise einem Gebäude, hinzugefügt, so ist es nicht mehr länger eigenständig, sondern übergibt die Kontrolle an sein neues Elternobjekt in der Strukturhierarchie. Das Objekt wird dann nicht mehr direkt gezeichnet, sondern indirekt über sein Elternobjekt, das alle Zeichenaufrufe an seine Subkomponenten weiterleitet. Außerdem werden die Vertexbuffer des neu hinzugefügten und des `Composite`-Objektes miteinander verschmolzen, damit alle zu einem Gebäude gehörenden Vertices zentral verfüg- und zugreifbar sind. Weiterhin verfolgt dieses Merging das Ziel, gemeinsame Vertices zu erkennen und Duplikate in der Vertexverwaltung zu vermeiden. Das Zusammenführen erfordert allerdings eine Aktualisierung sämtlicher Indices im neu hinzugefügten Objekt, da

sich die Indices innerhalb des nunmehr verwendeten Vertexbuffers geändert haben können. Nachdem das Zusammenführen abgeschlossen wurde, verwaltet das hinzugefügte Objekt keinen eigenen Vertexbuffer mehr, sondern besitzt nur noch einen Zeiger auf den objektglobalen Vertexbuffer seines `Composite`-Objekts. Die Verwendung eines derart zusammengeführten Buffers hat neben der potentiellen Wiederverwendung gemeinsamer Vertices auch den Vorteil, dass Transformationen, die das gesamte Gebäude betreffen, nur einmal auf dem globalen Buffer ausgeführt werden, anstatt sie an alle Komponentenobjekte weiterzureichen. Außerdem wird ein solcher Buffer für verschiedene Berechnungsschritte benötigt, beispielsweise für die Berechnung durchgängiger Texturkoordinaten der Wände über Stockwerke hinweg.

Die vorgestellte Struktur entspricht konzeptuell den Ansätzen punktbasierter Umrissmodelle, bei denen die einzelnen geometrischen Primitive hierarchisch organisiert sind und die Art der Relation festlegt, wie die Kommunikation zwischen den einzelnen Ebenen abläuft. Durch die Verwendung globaler Vertexbuffer, die von den Subkomponenten über Indices referenziert werden, werden Vertexduplikate vermieden und der Speicherverbrauch reduziert. Die Schwäche dieser Struktur ist allerdings, dass sie keinerlei Informationen über Nachbarschaften speichert und somit keine topologischen Anfragen unterstützt. Möchte man für ein Polygon potentielle Nachbarpolygone ermitteln, erfordert dies aufwendige Suchprozesse basierend auf den gemeinsamen Vertices der Nachbarn. Um diese Schwäche auszugleichen, besitzt jedes komplexe Objekt eine Kantenverwaltungsstruktur, die mit der Kantentabelle in der Winged-Edge-Datenstruktur von Baumgart [Ba72] vergleichbar ist. Diese Verwaltungsstruktur wird innerhalb des Systems als *Edge-Manager* bezeichnet und dient zunächst der Wiederverwendung von bereits existierenden Kanten innerhalb von Dreiecken. Werden durch die Tesselation eines Polygons neue Dreiecke erzeugt, so wird für jede Dreieckskante beim Edge-Manager des komplexen Elternobjekts angefragt, ob bereits eine Kante zwischen den Vertices existiert. Sofern dies der Fall ist, reicht der Manager die Kante zurück, ansonsten wird eine neue Kante erzeugt und in die Verwaltungsstrukturen aufgenommen. Der Edge-Manager zählt dabei für jede Kante, von wie vielen Dreiecken diese referenziert wird. Da Kanten von mehr als zwei Primitiven geteilt werden können, ist die Verwendung eines solchen Managers erforderlich, da ansonsten die Komplexität der eigentlichen Winged-Edge-Strukturen deutlich zunehmen würde. Wird eine Kante zurückgereicht, wird der Referenzzähler für diese Kante inkrementiert. Sobald ein Dreieck zerstört wird, meldet es dies beim Edge-

Manager, der daraufhin den Zähler dekrementiert. Wird eine Kante nicht mehr referenziert, wird sie aus dem Edge-Manager entfernt.

Analog zur Verschmelzung der Vertexbuffer beim Hinzufügen eines komplexen Objekts zu einem Composite-Objekt müssen auch die Edge-Manager verschmolzen werden, um die topologischen Daten innerhalb des Composite-Objekts zu aktualisieren. Auch für die Edge-Manager gilt dann, dass das hinzugefügte Objekt nicht mehr länger einen eigenen Manager verwaltet, sondern stattdessen den globalen `Composite`-Edge-Manager referenziert.

Aufgrund der Speicherung der Dreiecke, die eine Kante referenzieren, können topologische Anfragen nach benachbarten Dreiecken direkt ohne aufwendige Suchoperationen beantwortet werden, wodurch ein großer Vorteil kantenbasierter Umrissmodelle in das System integriert wird.

Nachdem vorab auf die Bestandteile des Semantic Building Modelers eingegangen wurde, widmen sich die folgenden Abschnitte den verschiedenen Technologien zur prozeduralen Generierung und Modifikation von Gebäude- und Stockwerksgrundrissen. Zunächst wird auf den Objectplacement-Algorithmus eingegangen, bevor an späterer Stelle ein Verfahren zur ähnlichkeitsbasierten Grundrisserzeugung vorgestellt wird.

8 Prozedurale Grundrissgenerierung und -modifikation

8.1 Das Objectplacement-Verfahren – zufallsbasierte Erzeugung nicht-trivialer Grundrisse

8.1.1 Motivation und Zielsetzung

Das hier vorgestellte Verfahren verfolgt das Ziel, einen zweidimensionalen Eingabebereich automatisiert mit zweidimensionalen Bausteinen zu füllen, die nach Abschluss der Berechnungen durch dreidimensionale Gebäudekomponenten ersetzt werden. Die Art, Ausrichtung, Position und Größe soll dabei zufallsbasiert innerhalb konfigurierbarer Parameterbereiche bestimmt werden, so dass sich die Berechnungsergebnisse auch für identische Eingabebereiche nicht gleichen. Neben der Problematik der algorithmischen Umsetzung und internen Verwaltung der erzeugten Datenstrukturen spielt darum auch die geeignete Wahl der Konfigurationsparameter eine wichtige Rolle. Durch die Änderung dieser Parameter soll es möglich sein, ohne Eingriffe in den Programmcode Einfluss auf die Art und Weise der Objektplatzierung zu nehmen. Dadurch soll der Nutzer sowohl die Möglichkeit bekommen, sich bei der Objekterzeugung vollständig auf das System zu verlassen, aber auch durch Modifikation der Konfigurationsparameter direkt in das System und seine Berechnungen einzugreifen und innerhalb eingeschränkten Rahmens den Berechnungsablauf und das Ergebnis zu beeinflussen.

Bevor auf die algorithmischen Ansätze eingegangen wird, werden zunächst die verwendeten Datenstrukturen und Grundelemente erörtert.

8.1.2 Basisstrukturen der Objektplatzierung

8.1.2.1 Verwaltung des Eingabebereichs

Wie auch andere Komponenten wird die Objektplatzierung als weitgehend unabhängiges Bibliotheksmodul in das Hauptsystem integriert. Um die Kopplung der Komponenten zu minimieren, müssen sowohl die Eingabe- als auch die Rückgabestrukturen möglichst einfach sein. Auf die Art der Ergebnisrückgabe soll dabei an späterer Stelle eingegangen werden, zunächst wird erläutert, wie Eingaben in das System aussehen und intern verwaltet werden.

Die Objektplatzierung befasst sich immer mit rechteckigen Eingabebereichen, die als Liste von 3D-Koordinaten an das System übergeben werden. Aufgabe der Berechnungen ist es dann, innerhalb des definierten Eingabebereichs eine konfigurierbare Anzahl von 2d-

Objekten zu positionieren, die nach der Rückgabe durch das Hauptsystem in 3D-Objekte umgewandelt werden. Die Berechnung lässt sich dabei in zwei Hauptschritte unterteilen.

Zunächst wird eine Hauptkomponente positioniert, auf deren Kanten und Ecken anschließend weitere Subkomponenten erzeugt werden. Um den Eingabebereich effizient verwalten zu können, wird eine Datenstruktur zur Verwaltung zweidimensionaler räumlicher Bereiche benötigt. Innerhalb der Computergrafik existieren verschiedene Strukturen, die effizient einsetzbar sind, beispielsweise *Binary Space Partition-Bäume* (BSP-Baum) oder *Quadtree-Strukturen*. Hier wird ein Quadtree-Ansatz verfolgt, dessen Funktionsweise kurz erläutert wird.

Quadtrees und ihr dreidimensionales Pendant, die *Octrees*, spielen in der Computergrafik eine wichtige Rolle unter anderem im Bereich der räumlichen Szeneunterteilung für 3D-Grafik-Engines. Sie werden als Datenstruktur zur Implementation von Szenegraphen verwendet und beispielsweise eingesetzt, um effizient entscheiden zu können, welche Teile der Szenegeometrie von einem bestimmten Betrachtungspunkt aus sichtbar sind und gerendert werden müssen [CKM04]. Quadtrees sind Baumstrukturen mit einem Grad von vier, jeder Knoten, der nicht Blatt ist, besitzt genau vier Kindknoten. Octrees besitzen einen Grad von acht und eignen sich somit auch für die Unterteilung dreidimensionaler Räume. Aufgebaut werden Quadtrees durch die rekursive Unterteilung eines Eingabebereichs in quadratische oder rechteckige Unterbereiche, bei Octrees sind die Unterbereiche dreidimensional und bestehen bsw. aus Würfeln oder Quadern. Die Einsatzmöglichkeiten von Quad- bzw. Octrees sind umfangreich. So ist es möglich, Octrees zur Verwaltung vollständiger Mesh-Instanzen oder Terrain-Regionen zu verwenden oder bis auf die Ebene einzelner Dreiecke für komplexe statische Meshes herunterzugehen und diese abzubilden. Gemein ist den Ansätzen dabei, dass sie Abbruchkriterien für die rekursive Unterteilung verwenden, die sich an der Belegung der Eingabebereiche orientieren. Je nach Verwendungszweck kann man beispielsweise festlegen, dass die Unterteilung stoppen soll, sobald sich nur noch ein Mesh oder Dreieck innerhalb eines Bereiches befindet. Um nun beispielsweise entscheiden zu können, welche Objekte innerhalb einer Szene von der Kamera aus sichtbar sind, durchläuft man den Baum ausgehend von der Wurzel und testet jeden Knoten auf Schnitte mit dem *View-Frustrum*. Liegt ein Knoten nicht innerhalb des sichtbaren Bereichs, können alle Objekte, die als Blätter innerhalb des Teilbaums vorkommen, vom Rendering ausgeschlossen werden. [Gr09].

Die hier vorliegende Implementation unterscheidet sich in verschiedenen Punkten von der vorab beschriebenen. Zunächst wird der Eingabebereich nicht belegungsabhängig unterteilt, sondern größenabhängig. Die Rekursion terminiert, sobald die Unterteilungsbereiche eine Maximalgröße unterschreiten. Dies hat zur Folge, dass man immer mit einem *vollständigen Baum* arbeitet, bei dem alle Blattknoten die gleiche Tiefe haben. Dieser Unterschied gegenüber der vorher beschriebenen Variante hängt mit der Anwendung zusammen. Die Datenstruktur erlaubt die gleichmäßige Unterteilung eines rechteckigen Eingabebereichs. Auf der Blattebene ist es möglich, die Bereiche aufgrund ihrer Belegungsarten zu markieren, beispielsweise ob sie eine Ecke, Teile einer Kante oder auch nichts enthalten. Diese Informationen sind hilfreich, wenn entschieden werden muss, wo eine Komponente positioniert wird und wie viel Platz für die Komponente zur Verfügung steht.

Die Markierungsoperationen selber basieren dabei auf *Punkt-In-Polygon-Tests*, die über den *Winding-Number-Algorithmus* implementiert sind [He94]. Durch die Organisation der Unterbereiche in einem Quadtree ist die Anzahl der erforderlichen Tests zur Ermittlung eines Bereiches, der einen bestimmten Punkt enthält, konstant und entspricht der Höhe des gesamten Baumes. Mit steigender Anzahl an Blattknoten amortisieren sich die initialen Berechnungskosten, da die Anzahl der erforderlichen Berechnungen deutlich geringer ist, als bei einem naiven Vorgehen, das alle Blattknoten testen würde.

8.1.2.2 **Positionierbare Komponenten**

Die Problemstellung der Positionierung von Komponenten wird innerhalb des Moduls vollständig als zweidimensionales Problem aufgefasst. Alle Berechnungen erfolgen innerhalb der Ebene, die durch den Eingabebereich definiert wird, wobei diese in einer beliebigen Ausrichtung vorliegen kann und nicht zwingend einer Koordinatenebene entsprechen muss. Darum besitzen auch die positionierten Komponenten keinerlei Höhenparameter, sondern sind ebenfalls zweidimensional. Man kann sich das Ergebnis der Berechnungen als zweidimensionalen Bauplan vorstellen, der den Grundriss eines beliebig komplexen Gebäudes beschreibt. Das Hauptprogramm wiederum interpretiert diesen Grundriss und weist den einzelnen positionierten Komponenten eine Höhe zu, überführt sie also von der zweidimensionalen in die dreidimensionale Welt. Damit die Kommunikation des Hauptprogramms mit der Platzierungskomponente einwandfrei funktioniert, muss gewährleistet sein, dass das Hauptprogramm die Komponenten kennt, die die Platzierungskomponente verwendet. Somit muss für jedes komplexe Objekt, das kein

zusammengesetztes `Composite`-Objekt ist, eine zweidimensionale Repräsentation innerhalb der Platzierungskomponente vorhanden sein. Die Verwendung des Bauplans reduziert sich dann schematisch auf eine Extrusion der zweidimensionalen Komponenten entlang der Höhenachse des Weltkoordinatensystems. Da der aktuelle Implementationsstand des Hauptprogramms Quader sowie zylinderförmige Polygonzüge beliebiger Vertexanzahl zur Verfügung stellt, existieren auch in der Platzierungskomponente zweidimensionale Komponenten nur für diese komplexen Objekte. Allerdings gilt sowohl für die Platzierungs- als auch für die Hauptkomponente, dass die Integration weiterer Objekte durch die Definition klarer Schnittstellen einfach ist, wodurch das System jederzeit erweiterbar bleibt.

8.1.3 **Der zweistufige Positionierungsalgorithmus**

Die Positionierung der Komponenten lässt sich unterteilen in einen zweistufigen Prozess. In der ersten Phase positioniert man eine Hauptkomponente, in der zweiten Phase erzeugt man Subkomponenten an den Kanten und Eckpunkten dieser Hauptkomponente. In beiden Phasen spielt der Zufall eine entscheidende Rolle für die Bestimmung von Position, Ausrichtung, Größe und Art der instanziierten Komponenten.

8.1.3.1 **Die erste Phase – Positionierung der Hauptkomponente**

Die Positionierung der Hauptkomponente ist vergleichsweise einfach, da sie im Gegensatz zur Subkomponentenpositionierung keinerlei Rücksicht auf bereits vorhandene Objekte nehmen muss. Die einzigen relevanten Parameter sind die Ausdehnungen des Eingabebereichs sowie die Konfigurationsparameter, die definieren, in welchem Bereich Höhe und Breite der Komponente schwanken dürfen. Die Hauptkomponente stellt den Haupttrakt des Gebäudes dar, für das der Grundriss erstellt wird. Aus diesem Grund beschränkt sich die Art des Objekts, das als Hauptkomponente verwendet werden darf, aktuell auf Rechtecke, während bezüglich der Subkomponenten a-priori keinerlei Einschränkungen vorgenommen werden. Nachdem das System die Ausdehnungen zufallsbasiert berechnet hat, positioniert es die Komponente derart, dass der Mittelpunkt des neuen Objekts mit dem Mittelpunkt des Eingabebereichs zusammenfällt. Nachdem die Positionierung abgeschlossen wurde, markiert das System Kanten, Eckpunkte sowie solche Blattknoten innerhalb des Quadtrees, die sich im Inneren der Komponente befinden. Hierbei wird so vorgegangen, dass die Markierungen nur auf der Ebene der Blätter vorgenommen

werden und dann in Richtung der Wurzel propagieren. Kern dieses Weiterreichens der Markierung ist die Zuweisung von Prioritäten zu den unterschiedlichen Belegungsarten. Die geringste Priorität besitzt der Zustand `Empty`, da dieser die Standardbelegung der Blattknoten ist, anschließend folgen Zustände wie `Edge`, `Corner` oder `Full` mit aufsteigender Bedeutung. Wird eine Belegung von Kindknoten ausgehend in Richtung Wurzel des Baumes durchgereicht, bekommt der Elternknoten immer die Belegung des Kindknotens zugewiesen, die die höchste Priorität innerhalb der Liste besitzt. Die Motivation dahinter ist, dass man Blattknoten hoher Priorität häufiger sucht und schneller finden will, als solche mit niedriger Priorität. Dadurch ist es beispielsweise sehr schnell und einfach möglich, die Blattknoten anzusteuern, die Eckpunkte von Komponenten enthalten. Die so vorgenommenen Markierungen werden sowohl für die Hauptkomponente als auch für die anschließend zu positionierenden Subkomponenten durchgeführt.

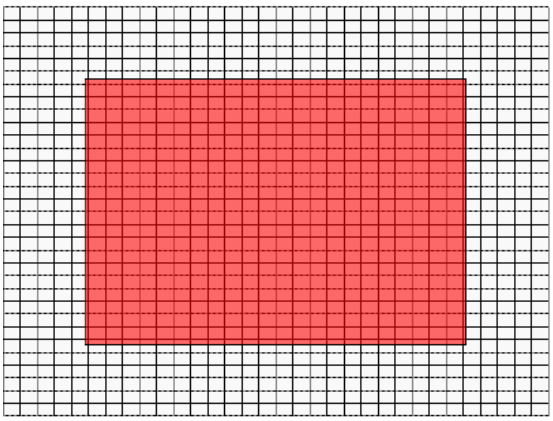

Abbildung 67: Positionierte Hauptkomponente

Abbildung 67 zeigt das Ergebnis der zufälligen Positionierung der Hauptkomponente. Diese ist rötlich eingefärbt. Das Hintergrundraster visualisiert sämtliche Blattknoten des Eingabebereichs. Der im Beispiel verwendete Quadtree besteht aus insgesamt 1365 Knoten bei einer Baumhöhe von 5. Die nachfolgende Abbildung zeigt nun das Ergebnis der Markierung der Blattknoten für die gezeigte Komponente. Die positionierte Komponente selber ist ausgeblendet.

Grün dargestellte Blattknoten befinden sich im Inneren der Hauptkomponente, rote Blätter enthalten Kanten, blaue Blätter Eckpunkte. Für die nachfolgende zweite Phase des Verfahrens sind die Kanten- und Eckpunkte von vorrangiger Bedeutung.

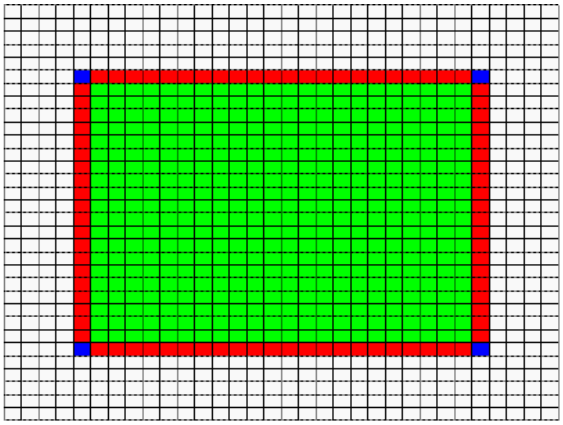

Abbildung 68: Markierungen der Hauptkomponente im Quadtree

8.1.3.2 **Die zweite Phase – Positionierung der Subkomponenten**

Die zweite Phase unterteilt sich in weitere zwei Phasen. Ob und wie diese Phasen berechnet werden, ist abhängig von Zufallsentscheidungen. In der ersten Phase werden Subkomponenten auf den Eckpunkten positioniert, in der zweiten Phase auf den Kanten der Hauptkomponente.

Sofern Subkomponenten erzeugt werden, erfolgt dies derart, dass die Komponenten nicht direkt als Mengen von Vertices berechnet werden. Vielmehr erzeugt man eine Datenstruktur zur Beschreibung der zu erstellenden Komponente, die Parameter für alle möglichen Subkomponenten enthält. Die so erzeugte Datenstruktur fungiert dann als Fabrik für die eigentlichen Komponenten und gibt diese an den Aufrufer zurück. Mittels Reflection ist man so in der Lage, Komponenten auf einem hohen Abstraktionsniveau zu beschreiben und anschließend zu erzeugen. Diese Baupläne sind am Ende der Berechnungen Kern der Rückgabe an das Hauptprogramm. Sie enthalten sowohl komponententypunabhängige Informationen wie Position oder Ausdehnungen, aber ebenfalls Informationen, die nur für bestimmte Subkomponentenarten relevant sind. Ein gutes Beispiel hierfür ist die Anzahl der Segmente, die nur für Zylinder-Objekte benötigt wird, für Quader dagegen vollkommen irrelevant ist. Die Verwendung solcher abstrakter Beschreibungen als Ergebnisstrukturen ist aus verschiedenen Gründen vorteilhaft. Sie überlässt beispielsweise dem Hauptprogramm sämtliche Entscheidungen darüber, wie es die 3D-Objekte aus den zweidimensionalen Strukturen erstellt. Außerdem erspart dieser Ansatz Berechnungen bezüglich der Maße der jeweiligen Objekte, da diese direkt in den Strukturen enthalten sind. Weiterhin ist es innerhalb des Hauptprogramms ersichtlich, was Haupt- und was Subkomponente ist und welchen Typ diese Komponenten haben.

Die Erstellung der Komponenten ist über verschiedene Parameter steuerbar. Dazu gehört zunächst die Fragestellung, ob überhaupt Subkomponenten für Eckpunkte oder Kanten erzeugt werden. Sofern dies der Fall ist, muss geprüft werden, ob ausreichend Platz für eine solche Komponente zur Verfügung steht. Je nachdem, ob eine Eck- oder eine Kantenkomponente erzeugt wird, erfolgen sowohl die Positions- als auch die Ausdehnungsberechnungen unterschiedlich.

Da Eck- vor Kantenkomponenten positioniert werden, sind für diese nur die Abstände zu den Außenkanten des Eingabebereichs relevant. Das System berechnet diese Abstände, skaliert sie nachfolgend zufällig innerhalb eines konfigurierbaren Bereichs und prüft, ob die Maße die ebenfalls einstellbaren Unter- und Obergrenzen nicht überschreiten. Befinden sich alle Parameter innerhalb der gültigen Bereiche, wird die Komponente auf der Ecke positioniert. Für Eckkomponenten werden diese Berechnungen allerdings nur ein einziges Mal durchgeführt, so dass alle positionierten Komponenten den gleichen Typ und die gleichen Ausdehnungen besitzen, sie unterscheiden sich nur in ihrer Position. Eine derart durchgeführte Erzeugung führt zu realistischer wirkenden Grundrissen als Komponenten, die zwar auf den Ecken eines Haupttrakts positioniert werden, sich sonst allerdings vollständig unterscheiden.

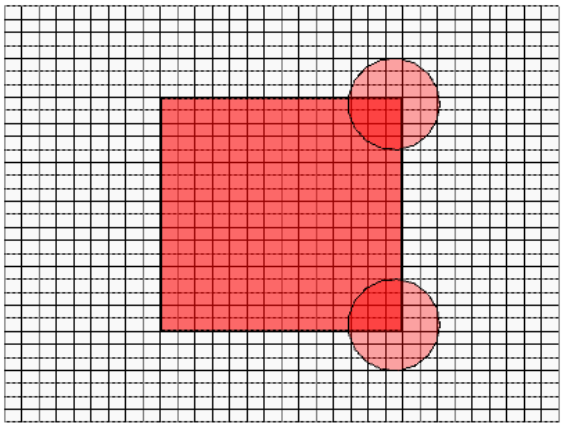

Abbildung 69: Hauptkomponente mit Eckpunktsubkomponenten

Abbildung 69 zeigt einen Screenshot aus der Objectplacement-Komponente, bei der das Verfahren zwei zylindrische Subkomponenten auf zwei Eckpunkte positioniert hat.

Für die Positionierung von Objekten auf den Kanten der Hauptkomponente ist die Berechnung aufwendiger. Die Bestimmung erfolgt als iterativer Prozess kantenweise, wobei für jede Kante zufallsgesteuert entschieden wird, ob diese überhaupt Komponenten erhalten

soll. Wird eine Kante gewählt, so positioniert das System so lange Subkomponenten auf dieser, bis eines der Abbruchkriterien erfüllt ist. Abbruchkriterien sind unter anderem die maximale Anzahl an Komponenten pro Kante, die minimalen Ausdehnungen der Komponenten und die Anzahl freier Blöcke zwischen einzelnen Komponenten. Alle diese Kriterien sind über Parameter steuerbar. Neben den Abständen zu den Außenkanten spielen für diese Positionierungsberechnungen auch andere Komponenten auf der Kante oder den adjazenten Ecken eine Rolle, da diese die Ausdehnungen einschränken.

Die Ermittlung der relevanten Parameter basiert auf der Auswertung der belegten Blattknoten innerhalb des Quadtrees. Das System ermittelt zunächst alle Blattknoten, die die jeweilige Kante enthalten. Die Berechnung wird aus Performancegründen nur ein einziges Mal bei der initialen Markierung der Kanten durchgeführt. Anschließend wird das Ergebnis für nachfolgende Berechnungen vorgehalten. Die Bestimmung der Knoten selber folgt einem *Sampling-Ansatz*, bei dem man ausgehend von einem Punkt auf dem Strahl eine zufallsbasierte Distanz in Richtung der Kante geht und prüft, ob sich der derart bestimmte Punkt noch innerhalb des gleichen Knotens befindet. Ist dies nicht der Fall, durchsucht man den Quadtree nach dem Blattknoten, der den so gesampelten Punkt enthält. Dadurch erhält man eine Menge von Blattknoten, die alle von der Kante geschnitten werden. Für die Subkomponentenpositionierung berechnet man dann einfache Datenstrukturen, die freie Bereiche auf der Kante speichern. Hierfür ist es ausreichend, den Startindex und die Länge des freien Bereichs zu ermitteln. Wurde weder eine Eck- noch eine Kantenkomponente positioniert, so erzeugt die Berechnung nur eine einzige Instanz dieser Strukturen, deren Index das Startblatt beschreibt und dessen Länge der Anzahl der geschnittenen Blattknoten entspricht. Nun sortiert man die erstellten Strukturen aufgrund der Anzahl freier Knoten. Für die Positionierung wählt man die Struktur mit den meisten aufeinanderfolgenden freien Knoten und verwendet diese Anzahl als weiteren Begrenzungsparameter für die Ausdehnungs- und Positionsbestimmung.

Neben der vollständigen Neuberechnung der Kantenbelegung bietet das System noch eine weitere Variante an, nämlich die symmetrische Positionierung von Komponenten. Auch die Entscheidung für symmetrische Komponenten auf gegenüberliegenden Kanten ist dabei abhängig vom Zufall. Sofern eine Kante symmetrisch zu einer anderen Kante belegt werden soll, erzeugt das System Kopien der Bauplan-Instanzen der Komponenten auf der Quellkante und ändert ausschließlich die gespeicherte Position. Basierend auf diesen Plänen werden dann wiederum neue Komponenten erzeugt. Hierin zeigt sich ein weiterer großer

Vorteil der vorab erläuterten Bauplan-Strukturen, die ihren Nutzen nicht nur als Ergebniselemente haben, sondern auch innerhalb der Verarbeitung Berechnungsschritte deutlich vereinfachen.

Abbildung 70 zeigt einen Screenshot, bei dem die Subkomponenten auf den horizontalen Kanten basierend auf den Symmetrieberechnungen positioniert wurden. Dabei sind die durch die Subkomponenten belegten Quadtree-Blätter violett eingefärbt, die weiteren Einfärbungen entsprechen dem vorab erläuterten Farbschema für die Markierung der Hauptkomponente.

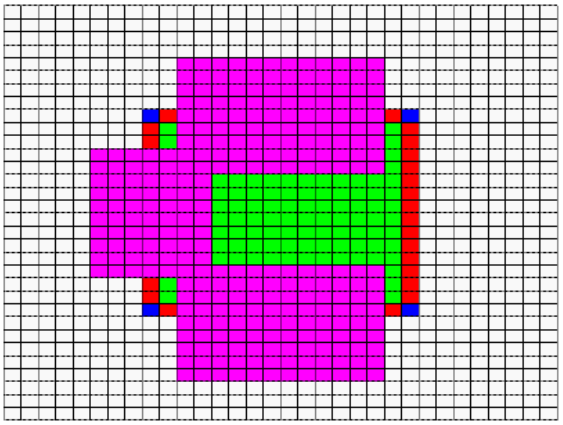

Abbildung 70: Symmetrische Subkomponentenpositionierung

8.1.4 **Iterative Anwendung der Komponentenpositionierung**

Nachdem auf die unterschiedlichen Mechanismen zur Positionierung von Haupt- und Subkomponenten und auf die verschiedenen Parameter für die Steuerung des Positionierungsprozesses eingegangen wurde, soll nun die Möglichkeit eines rekursiven Aufrufs des Verfahrens vorgestellt werden. Die Kernidee dieses Ansatzes ist es, dem Nutzer die Möglichkeit zu bieten, den Berechnungsprozess nicht nach einer Iteration abbrechen zu lassen, sondern beliebig viele Iterationen des Verfahrens zu durchlaufen. Das grundsätzliche Verfahren ändert sich dabei nicht, konzeptuell verwaltet man bei einer Iteration, die Komponenten auf vorab positionierten Subkomponenten erzeugt, nicht mehr länger eine einzige Hauptkomponente. Hierfür hält man sämtliche, während einer Iteration erzeugten Subkomponenten vor und speichert diese in einer Mapstruktur, die über den Zielstrahl der Komponenten indiziert wird. Nach Abschluss einer Iteration holt man nur die Komponenten, die in der vorherigen Iteration ermittelt wurden. Im nächsten Schritt durchläuft man diese Menge und macht jedes Element temporär zur Hauptkomponente der aktuellen Iteration.

Sämtliche Positionierungsberechnungen beziehen sich dann auf die aktuelle Hauptkomponente, so dass der Algorithmus nur Subkomponenten für das jeweilige Element berechnet. Je mehr Iterationen das Verfahren auf diese Art und Weise durchläuft, desto mehr Subkomponenten werden erzeugt und desto komplexer werden die aus dem Verfahren extrahierten Grundrisspolygone.

Dabei ist festzuhalten, dass die Anzahl der Iterationen nicht zu hoch gewählt werden sollte, da das Verfahren dazu tendiert, bei jeder Iteration kleinere Komponenten zu erzeugen. Wählt man einen zu hohen Wert für die Iterationsanzahl, können sehr stark verschachtelte Strukturen entstehen, die für die Verwendung als Grundriss ungeeignet sind. Abbildung 71 zeigt drei Beispiele für das Ergebnis der Objectplacementberechnungen mit zwei Iterationen.

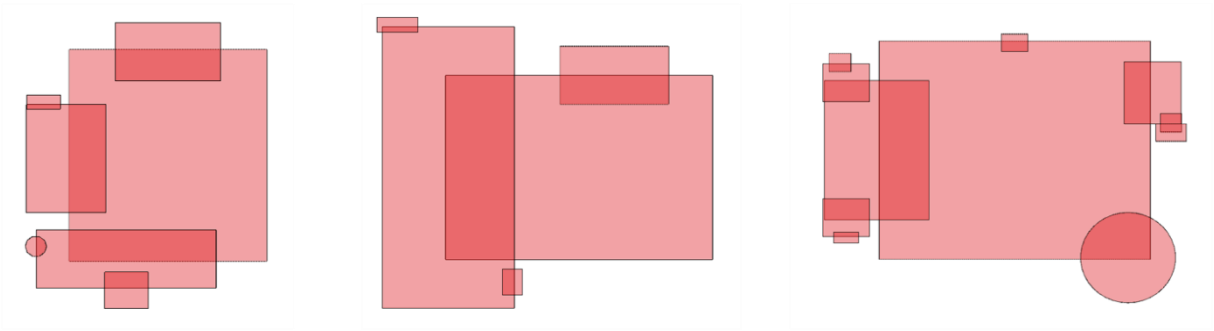

Abbildung 71: Objectplacement mit 2 Iterationen

8.2 Organisation der Bausteine – Verwaltung und Zusammenfassung komplexer Komponentenhierarchien

8.2.1 Struktur und Erweiterbarkeit der Placementkomponente

Der vorab vorgestellte Algorithmus folgt der Idee eines Baukastens. Durch die Kombination einfacher Grundbausteine lassen sich komplexe Gebilde erschaffen. Im vorliegenden System entstehen aus einfachen rechteckigen und zylinderförmigen Grundelementen komplexe Grundrisse. Die Grundbausteine werden in Form, Größe und Ausrichtung variiert, miteinander kombiniert und nach Abschluss der Positionsberechnungen zusammengefasst. Der Reiz dieses Ansatzes besteht in der Einfachheit der zugrundeliegenden Elemente und der überraschenden Vielfalt an Strukturen, die man durch deren Komposition realisieren kann. Je mehr Grundbausteine zur Verfügung stehen, desto komplexer werden die Formen, die sich durch deren Kombination realisieren lassen. Durch die Möglichkeit der Integration

weiterer Grundelemente in das System ist eine optimale Erweiterbarkeit gegeben, die es mit wenigen Eingriffen gestattet, die Modellierungsmächtigkeit schrittweise zu steigern.

Diese Erweiterbarkeit wird innerhalb des Objectplacement-Moduls durch den Einsatz geeigneter Schnittstellen erreicht. Eine Integration neuer Basiskomponenten erfordert ausschließlich die Implementation von typspezifischen Grundoperationen beispielsweise zur Erzeugung oder Unterteilung der jeweiligen Komponente. Neben der Einfachheit der Integration weiterer Bestandteile, muss die Möglichkeit gegeben sein, Bausteine effizient zu verwalten und zu gruppieren, um komplexere Gebilde erschaffen zu können. Auch solche komplexen Objekte sollen wiederum beliebig gruppierbar sein, um dadurch eine Hierarchie von Bausteinen erschaffen zu können. Dabei soll auf jeder Ebene die Transparenz gewahrt werden, ob aktuell mit einem gruppierten oder einem atomaren Baustein gearbeitet wird. Weiterhin soll die Möglichkeit gegeben sein, Gruppen von Objekten zu einem einzelnen Objekt zusammenzufassen. In Abgrenzung zu einer Gruppe ist es nach einer solchen Zusammenfassung nicht mehr möglich, die Gruppenmitglieder einzeln anzusprechen oder sie aus der Gruppe zu entfernen. Die Vorteile einer solchen Zusammenfassung zu einem einzelnen Objekt bestehen in der Reduktion der Komplexität, die durch diese erreicht wird. Während eine Gruppe individuelle Verwaltungsstrukturen für jede Komponente benötigt, beispielsweise für die Verwaltung von Kanten, Texturen oder Vertexstrukturen, benötigt eine zusammengefasste Gruppe nur Verwaltungsstrukturen für ein einziges Objekt. Dies gestattet einen effizienteren Umgang mit den zur Verfügung stehenden Ressourcen.

8.2.2 **Der Footprint-Merging-Algorithmus – Verschmelzung beliebiger Gebäudekomponenten**

8.2.2.1 **Motivation und Zielsetzung**

Nachdem im vorherigen Abschnitt die interne Verwaltung und Organisation der Bausteine erläutert wurde, befasst sich dieser Abschnitt mit dem Vorgehen zur Reduktion komplexer Objekte bestehend aus einer potentiell großen Anzahl komplexer oder atomarer Bausteine auf ein einzelnes Objekt. Eine solche Reduktion stellt einen irreversiblen Schritt in der Objektverarbeitung dar, da die Grundelemente während der Berechnung des neu entstehenden Objekts zugeschnitten und modifiziert werden. Dabei gehen die ursprünglichen Komponenten verloren. Aus diesem Grund sollte eine solche Reduktion erst nach Abschluss aller geometrischen Operationen wie Unterteilungen oder Verformungen durchgeführt werden. Solche Operationen sind deutlich einfacher auf der Ebene der Grundbausteine

durchzuführen. Je komplexer die Form der zusammengesetzten Strukturen, desto schwieriger wird die Berechnung von Unterteilungen in beliebigen Raumachsen. Aus diesem Grund ist es sinnvoller, solche Operationen durchzuführen so lange die betroffenen Objekte noch in ihrer einfachen Grundform vorliegen.

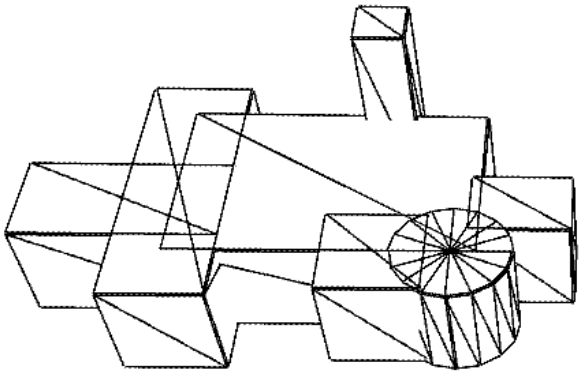

Abbildung 72: Beispielkonfiguration verschiedener atomarer Grundbausteine

Was ist dann aber der Nutzen, den man durch eine solche Reduktion erreicht? Dies sei kurz anhand eines Beispiels erläutert. Abbildung 72 zeigt eine Konfiguration mehrerer einfacher Bausteine, die intern zu einem `Composite`-Objekt zusammengefasst wurden. Man erkennt an den Linienverläufen, dass sich die einzelnen Bausteine überlappen und gegenseitig schneiden. Durch das `Composite`-Pattern sind zu diesem Zeitpunkt sämtliche Grundbausteine weiterhin verfügbar und können modifiziert oder aus der Gruppe entfernt werden.

Problematisch wird eine solche Konfiguration an verschiedenen Punkten der Verarbeitung. Ein solcher Schritt ist beispielsweise die Berechnung eines Daches durch den Straight-Skeleton-Algorithmus. Dieser benötigt als Eingabe einen Polygonkantenzug definiert durch eine Folge von Vertices, die im Uhrzeigersinn angegeben werden. Ein `Composite`-Objekt, wie es im Screenshot sichtbar ist, kennt diesen Kantenzug a-priori nicht.

Abbildung 73: Texturierungsfehler durch nicht reduzierte Bausteinkomponenten

Es kennt nur die einzelnen atomaren Bausteine, im Beispiel Zylinder und Quader, und rendert diese einzeln, so dass das dargestellte Bild entsteht. Jedes atomare Objekt verfügt über eine eigene Kanten- und Vertexverwaltung, die zunächst nicht im Composite-Objekt zusammengeführt werden. Die Erzeugung eines Daches für die gezeigte Konfiguration erfordert darum zunächst die Extraktion des Linienzuges, der den gemergten Grundriss der Komponenten beschreibt. Ein solcher Grundriss kann nicht nur als Eingabe für ein Dachprimitiv fungieren, sondern auch als Basis für die Erstellung des zusammengefassten Objektes.

Neben der Dachformberechnung ist die Konfiguration noch an anderer Stelle problematisch. Im `Composite`-Zustand besteht die obige Konfiguration für das System ausschließlich aus einer Menge einfacher Basisbausteine. Möchte man diese nun mit einer Textur belegen, so geht das System so vor, dass es Texturkoordinaten für jede Fläche der Grundobjekte berechnet. Die einzelnen Bausteine wissen dabei nichts von den anderen Bausteinen, mit denen sie kombiniert werden und berechnen die Texturraumkoordinaten unabhängig voneinander. Kommt es nun zu Überschneidungen zwischen Bausteinen, wie sie in der Abbildung gut erkennbar sind, so führt dies zu Texturierungsfehlern, da die Texturen weiterlaufen und abrupt unterbrochen werden. Diese Problematik erkennt man gut in Abbildung 73, ebenfalls ein Screenshot aus dem laufenden System, der die obige Konfiguration ohne vorherige Zusammenfassung der Grundbausteine mit Texturen darstellt. Die roten Kästen zeigen Übergänge zwischen Grundbausteinen, bei denen die Texturierung aufgrund der erläuterten Problematik zu unschönen Fehlern führt, die eine kontinuierliche und störungsfreie Texturierung verhindern.

Abbildung 74 zeigt das Ergebnis der Anwendung des Footprint-Merger-Verfahrens, das nachfolgend erläutert wird. Hierbei ist zu beachten, dass das Verfahren nur solche Flächen berücksichtigt, die sich auf der gleichen Höhe bezüglich des zugrunde liegenden Weltkoordinatensystems befinden. Aus diesem Grund taucht auch der Turm, der in der Objektkonfiguration erkennbar ist, im Grundriss nicht auf. Für diesen wird ein eigener Grundriss nach dem gleichen Verfahren extrahiert. Der derart ermittelte Polygonzug wird im Rahmen der nachfolgenden Verarbeitung sowohl für die Bestimmung eines zusammengefassten Grundobjekts als auch für die Dachberechnung verwendet.

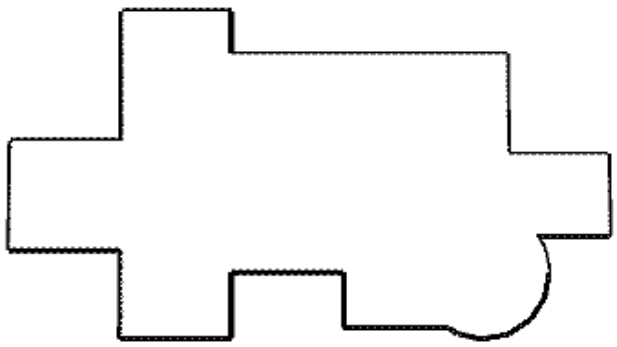

Abbildung 74: Ergebnisgrundriss nach Anwendung des Footprint-Merger-Ansatzes

8.2.2.2 **Das Verfahren**

Nachdem die Motivation und die Anwendungsmöglichkeiten für das Verfahren erläutert wurden, befasst sich der folgende Abschnitt mit der Funktionsweise des Merging-Verfahrens und beschreibt die Kernschritte, die erforderlich sind, um aus einer Menge von sich überschneidenden Grundbausteinen einen Grundriss zu extrahieren.

Das Verfahren selber ist ein zweistufiger Prozess. Im ersten Schritt berechnet man eine Menge von *Eimern*, in die alle Grundrisse „geworfen“ werden, die sich schneiden. Im zweiten Schritt berechnet man für jeden dieser Eimer einen gemergten Grundriss basierend auf den atomaren Grundrissen, die sich im Eimer befinden.

8.2.2.2.1 Erste Phase – Berechnung der Eimer

Die erste Stufe des Verfahrens ermittelt alle Grundrisse innerhalb der Eingabe, die einen Schnitt aufweisen. Grundrisse, die sich schneiden, werden in einem Eimer

zusammengefasst. Der Berechnungsansatz ist aus laufzeittechnischer Sicht optimierbar, führt allerdings zu korrekten Ergebnissen. Das naive Vorgehen durchläuft alle Eingabegrundrisse und testet für jeden einzelnen Grundriss zunächst, ob dieser sich bereits in einem Eimer befindet. Ist dies nicht der Fall, wird ein neuer Eimer erzeugt und der aktuell untersuchte Grundriss hinzugefügt. Anschließend durchläuft man in einer inneren Schleife erneut alle Grundrisse und prüft auf Schnitte. Findet man für den aktuellen einen geschnittenen Grundriss, so fügt man diesen ebenfalls in den aktuellen Eimer ein. Dadurch kann es während der Verarbeitung dazu kommen, dass sich ein Grundriss bereits in mehreren Eimern befindet, obwohl die Hauptschleife für diesen noch gar nicht berechnet wurde. Ein solches Vorgehen ist aus dem Grund erforderlich, da keinerlei Vorannahmen über die Anordnung der Grundrisse in der Eingabe gemacht werden. Es kann dazu führen, dass ein Grundriss in mehreren Eimern auftaucht, nachdem die beschriebenen Rechenschritte durchgeführt wurden. Dies wiederum bedeutet aber, dass alle Grundrisse in den jeweiligen Eimern Bestandteile eines gemergten Grundrisses sind. Liegen beispielsweise in Eimer 1 die Grundrisse 1 und 2, in Eimer 2 dagegen die Grundrisse 2 und 3, so verbindet Grundriss 2 die Grundrisse 1 und 3, so dass diese in einem einzelnen Eimer zusammengeführt werden können. Ein solches Zusammenführen stellt den letzten Berechnungsschritt in der ersten Phase des Merging-Algorithmus dar. Auch dieser Schritt ist vergleichsweise rechenintensiv, da er das Durchlaufen sämtlicher erzeugter Eimer erfordert. In jedem Eimer vergleicht man die Elemente miteinander, wird ein Element gefunden, das in beiden Eimern vorkommt, füllt man den Inhalt des einen Eimers in den anderen um und löscht den nun geleerten Eimer. Dies wiederholt man so lange, bis entweder nur noch ein Eimer vorhanden ist oder alle Eimer paarweise gegeneinander getestet wurden. Ist zu diesem Zeitpunkt mehr als ein Eimer vorhanden, so handelt es sich bei der Eingabe um Grundrisse, die sich auf unterschiedlichen Höhen befinden.

Offensichtlich ist die Laufzeit des Ansatzes nicht optimal. Durch die Notwendigkeit paarweiser Vergleiche der Grundrisse liegt die Berechnungszeit für die erste Phase der Eimerstellung für n Eingabegrundrisse bei $O(n^2)$. Das Zusammenführen der Eimer erfordert im Worst-Case das erneute Vergleichen der Elemente in den Eimern, so dass man auch hier auf eine Laufzeit von $O(n^2)$ kommt. Der zweite Schritt ist allerdings nicht immer zwingend erforderlich, beispielsweise dann, wenn nur ein Eimer als Ergebnis der Zuordnung erzeugt wurde. Außerdem reduziert sich die Anzahl der Eimer nach jeder Zusammenführung, so dass auch hier eine günstigere Laufzeit angenommen werden darf. Zusammengefasst bewegt

man sich trotzdem im Bereich von $O(n^2)$ für die gesamte erste Stufe. Günstig ist dabei, dass die Anzahl der Eingabegrundrisse typischerweise vergleichsweise klein ist, man also mit kleinen n arbeitet. Im vorab mehrfach erwähnten Beispiel wird die Berechnung auf sieben Eingabegrundrissen durchgeführt. Das daraus gebildete Gebäude ist bereits vergleichsweise komplex, so dass davon ausgegangen werden kann, dass die Anzahl der Eingabeobjekte typischerweise relativ gering ist.

8.2.2.2.2 Zweite Phase – Berechnung der Grundrisse

Eingabe in die zweite Phase der Grundrissberechnung ist eine Menge von Eimern, die jeweils eine Menge von Grundrissen enthalten. Innerhalb jedes Eimers gilt, dass sich alle enthaltenen Grundrisse direkt oder indirekt über dazwischenliegende Grundrisse schneiden. Grundrisse, die in verschiedenen Eimern liegen, besitzen dagegen keinerlei Schnitte, weder direkt noch indirekt. Dies kann der Fall sein, wenn sich die Grundrisse auf unterschiedlichen Höhen befinden oder wenn freie Flächen zwischen den Grundbausteinen existieren.

In diesem Abschnitt werden die Berechnungen erläutert, die für jeden Eimer durchgeführt werden. Ergebnis der Berechnung ist ein gemergter Grundriss, wie er in Abbildung 74 gezeigt wird.

Vereinfacht dargestellt, verfolgt das Verfahren ausgehend von einem Startpunkt die Kantenzüge der Eingabegrundrisse. Für jede Kante testet man auf Schnitte mit allen anderen Kanten innerhalb des aktuellen Eimers. An jedem Schnittpunkt wechselt man die verfolgte Kante und folgt nun der neu gewählten Kante. Das Verfahren terminiert, sobald die Suche wieder am Startpunkt ankommt.

Offensichtlich muss das Verfahren darauf achten, sich nur auf den Außenkanten des späteren Ergebnisgrundrisses zu bewegen. Aus diesem Grund kommt auch der Wahl des Startpunktes eine besondere Bedeutung zu. Für diesen muss garantiert sein, dass er sich auf der Außenkante des Ergebnispolygons befindet. Um diese Voraussetzung zu erfüllen, ist es erforderlich, einen Startpunkt zu wählen, der definitiv nicht im Inneren eines der Eingabegrundrisse liegt. Dazu wählt man einen Punkt, der bezüglich einer Koordinatenachse den kleinsten vorkommenden Wert aufweist. Ein solcher Punkt kann sich nicht innerhalb eines Grundrisses befinden, sondern nur als Eckpunkt vorkommen. Allerdings muss zunächst ermittelt werden, auf welche Komponente man testet, da das Verfahren unabhängig von der Ebene sein soll, in der die Grundrisse liegen. Hierfür durchläuft man sämtliche

Grundrisse innerhalb des Eimers und analysiert die Normalenvektoren der Ebenen, in denen sie enthalten sind. Dabei greift das System auf ein Verfahren von Snyder und Barr [SB87] zurück, das diese als Teil eines schnellen Schnitttests zwischen Strahlen und Dreiecken verwendeten. Die Autoren analysierten die Komponenten der Normalenvektoren von Dreiecken, um die Komponente zu ermitteln, die den größten Absolutwert aufweist. Die Projektion des Dreiecks auf eine Koordinateneben erfolgte anschließend derart, dass die vorab ermittelte Komponente weggelassen wurde. Dadurch erhält man diejenige Projektion des Dreiecks mit maximalem Flächeninhalt. Diese einfache Berechnung wird auch für die Ermittlung derjenigen Komponente verwendet, die als Ausgangspunkt für die Startpunktermittlung eingesetzt wird.

Nachdem ein Startpunkt ermittelt wurde, berechnet man Schnittpunkte der Kante, die diesen Punkt als Startpunkt besitzt, mit allen anderen Kanten innerhalb des Eimers. Für jeden Schnitt mit einer Kante erzeugt man eine Datenstruktur, die verschiedene Informationen über den Treffer speichert. Dazu gehören der getroffene Strahl und der Schnittpunkt, sowie die Distanz des Treffers vom Startpunkt der Testkante. Weiterhin muss für die nachfolgende Auswahl eines Schnittes gespeichert werden, ob der getroffene Punkt Startpunkt einer anderen Kante ist oder sich irgendwo auf der getroffenen Kante befindet. Der erste Fall tritt beispielsweise dann auf, wenn die untersuchte Kante nicht von einem anderen Grundriss geschnitten und somit eine Kante des gleichen Grundrisses getroffen wurde. Bei der Ermittlung möglicher Schnittstrahlen muss berücksichtigt werden, dass die Schnittpunktberechnung keinen Schnittpunkt zurückgibt, wenn die Strahlen identische Richtungen haben. Zusätzlich zu den regulären Schnittpunkttests ist es darum erforderlich, auch solche Strahlen zu ermitteln, die eine Verlängerung des Teststrahls darstellen, allerdings nicht zur gleichen Kante gehören, weil sie beispielsweise Teil eines anderen Grundrisses sind. Eine solche Situation ist in Abbildung 75 für den Fall der grünen und violetten Kante zu sehen. Da die beiden Strahlen die gleiche Richtung haben, verläuft die Suche nach einem Schnittpunkt erfolgreich, so dass zusätzliche Berechnungen erforderlich sind, um auch diesen Fall abzudecken. Wurden solche Strahlen gefunden, so werden auch für diese Treffer-Instanzen erzeugt. Nachdem sämtliche möglichen Treffer ermittelt wurden, sortiert man die Instanzen aufgrund ihrer Entfernung zum Startpunkt aufsteigend. Nun wählt man alle Treffer aus, die die minimale Distanz besitzen. Es gibt immer dann mehrere Treffer, die dieses Kriterium erfüllen, wenn eine Ecke der Linienzüge getroffen wird, an der mehrere Kanten starten. Tritt dieser Fall nicht auf, so gibt es nur einen einzigen

Ergebnisstrahl, der zum Ausgangspunkt der nachfolgenden Iteration wird. Sonst ist es erforderlich, sich für einen der potentiellen Ergebnisstrahlen zu entscheiden. Diese Entscheidung wiederum wird durch einen *Voting-Ansatz* realisiert, bei dem für jeden Strahl mehrere Tests gerechnet werden. Je nach Ausgang der Tests wird die Stimmanzahl für den jeweiligen Strahl um unterschiedliche Werte erhöht. Am Ende wählt man den Strahl mit den meisten Stimmen und verwendet diesen als Ausgangspunkt für den nächsten Durchlauf. Die Votingberechnungen werden erforderlich, wenn sich Grundrisse im Eimer befinden, deren Kanten sich nicht überschneiden, sondern zusammenfallen. Abbildung 75 soll die Notwendigkeit und Funktionsanweise des Voting-Ansatzes verdeutlichen indem sie einen potentiell problematischen Sonderfall illustriert, bei dem die im vorherigen Durchlauf ermittelte rote Kante den Startpunkt dreier unterschiedlicher Kanten trifft. Hierbei teilen sich außerdem die beiden unteren Grundrisse ein Segment der grünen Kante, was die Entscheidung zusätzlich erschwert.

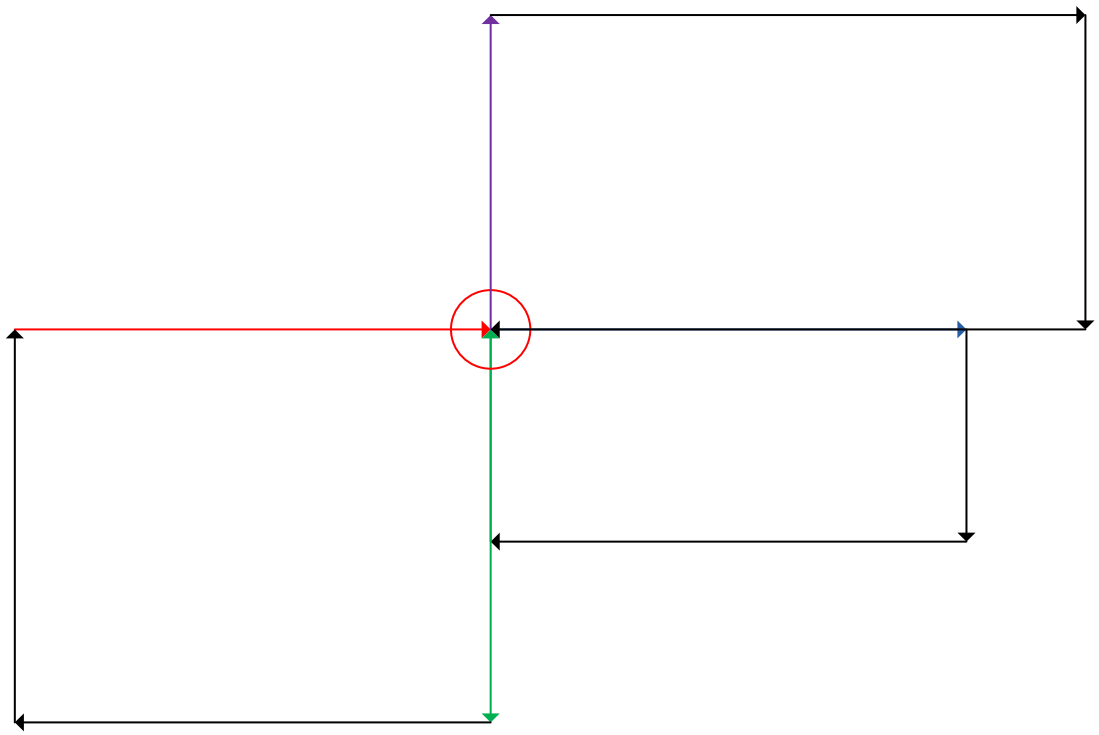

Abbildung 75: Problematische Grundrissanordnung mit Voting-Bedarf

Nach der Berechnung möglicher Schnittpunkte existieren drei mögliche Kandidaten für die nachfolgende Iteration. Dies sind die blaue, die grüne und die violette Kante. Für alle drei Kanten ist der Startpunkt offensichtlich gleich weit vom Startpunkt der letzten Iteration entfernt, da sie alle im gleichen Punkt beginnen. Hier ist es erforderlich, den nächsten

Testkandidaten über den Voting-Ansatz zu ermitteln. Dieser Ansatz verteilt Stimmen für zwei unterschiedliche Kriterien. Er bevorzugt solche Kanten, die gegenüber der aktuellen Testkante ihre Richtung ändern. Hierfür bekommen Kanten, die dieses Kriterium erfüllen, eine Stimme. Im Beispiel würden also die grüne und die violette Kante eine Stimme erhalten, die blaue Kante geht dagegen leer aus. Weiterhin bevorzugt der Ansatz solche Kanten, die nicht zum gleichen Grundriss gehören wie die vorherige Testkante. Hierfür werden zwei Stimmen vergeben, konkret für die blaue und die violette Kante. Nach Abschluss der Tests liegt die violette Kante also mit drei Stimmen vor der blauen und der grünen Kante und wird für die nächste Iteration ausgewählt.

Tatsächlich ist das Verfahren sehr robust in seinen Berechnungen und liefert auch für komplexe Grundrisse mit einer großen Anzahl an Grundbausteinen korrekte Ergebnisse. Rückgabe der Berechnung ist eine Menge von Vertices, die als Grundrisse im Hauptprogramm weiterverwendet werden. Da die Eingabegrundrisse bezüglich ihrer Vertices alle im Uhrzeigersinn definiert sind, liegen auch die Ergebnisvertices nach der Berechnung automatisch im Uhrzeigersinn vor. Dies hängt mit der Ausrichtung und Verfolgung der Strahlen zusammen. Umsortierungen oder Nachverarbeitungen der Ergebnisvertices sind nicht erforderlich.

Als Alternative zum vorab vorgestellten Footprint-Merger-Algorithmus kann man auf ein Verfahren zurückgreifen, das in verschiedenen Bereichen der Informatik eine wichtige Rolle spielt. Auf dieses wird im folgenden Abschnitt eingegangen.

8.3 **Grundrissextraktion als Convex-Hull-Problem**

Betrachtet man die Rückgabe des Objectplacement-Algorithmus auf der Ebene der einzelnen Vertices, aus denen die berechneten Komponenten bestehen, so kann man deren Reduktion auf ein einzelnes Polygon als *Convex-Hull-Problem* auffassen. Im Unterschied zum Footprint-Merger-Algorithmus, bei dem durch Kantenverfolgung im Regelfall konkave Polygone erzeugt werden, bieten Algorithmen zur Berechnung der konvexen Hülle für eine Punktmenge eine Variante, die konvexe Polygone ermittelt.

Bevor auf den implementierten Algorithmus eingegangen wird, soll zunächst eine formale Definition des Begriffs der *konvexen Hülle* gegeben werden. „Die konvexe Hülle einer Menge Q von Punkten, die wir mit $CH(Q)$ bezeichnen, ist das kleinste konvexe Polygon P, für das sich jeder Punkt aus Q entweder auf dem Rand von P oder in seinem Inneren

befindet.“ [Co10]. Es gibt verschiedene Algorithmen für die Berechnung konvexer Hüllen, eine erste Unterscheidung besteht darin, ob die Verfahren für Punktemengen im zwei- oder dreidimensionalen Raum eingesetzt werden. Der Semantic Building Modeler implementiert Algorithmen für beide Varianten, den *Graham-Scan-Algorithmus* für zweidimensionale Punktemengen und den *Quickhull-Algorithmus*, sofern sich die Punkte im dreidimensionalen Raum befinden. Das Quickhull-Verfahren wird innerhalb des Semantic Building Modelers in einer Vorstufe eines Verfahrens eingesetzt, das *objektausgerichtete Bounding Boxen* (OBB) ermittelt. Da es sich bei dieser Anwendung nicht um eine konkret für die Gebäudekonstruktion relevante Technik handelt, wird auf die Quickhull-Implementation nicht weiter eingegangen. Stattdessen soll das Graham-Scan-Verfahren in seinen Grundzügen erläutert werden, um einen besseren Eindruck von der Funktionsweise und Einsetzbarkeit im Kontext der Grundrissextraktion zur vermitteln.

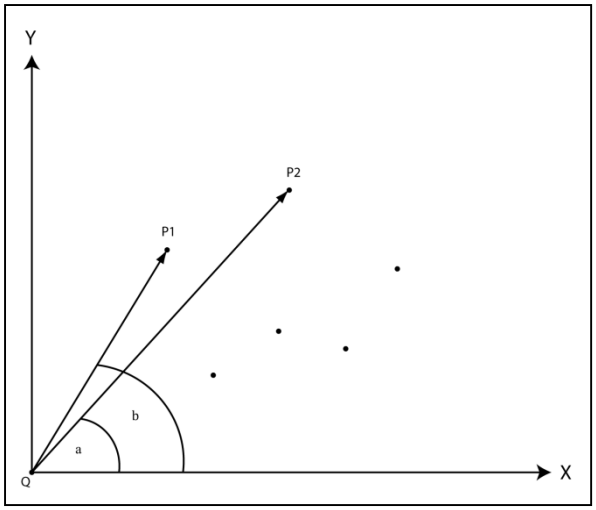

Abbildung 76: Illustration des Graham-Scan-Algorithmus

Abbildung 76 zeigt eine Menge von Punkten in der xy-Ebene. Für diese Punktemenge soll der Algorithmus ein konvexes Polygon errechnen, das die vorab genannten Anforderungen erfüllt. Der erste Schritt des Verfahrens besteht in der Ermittlung eines Punktes, der mit Sicherheit auf der konvexen Hülle liegt. Hierfür wählt man den Punkt Q mit der minimalen y-Koordinate aus der Punktemenge. Sofern es mehrere Punkte gibt, die den gleichen minimalen y-Wert besitzen, so wählt man aus dieser Untermenge denjenigen mit der kleinsten x-Koordinate [La12]. Unter Verwendung des Startpunktes Q sortiert man nun sämtliche verbleibenden Punkte P aus der Punktemenge anhand des Winkels zwischen $\overrightarrow{QP}$ und der x-Achse. Dies ist in Abbildung 76 zu erkennen. Der eingezeichnete Punkt Q ist der

im ersten Schritt ermittelte Startpunkt des Verfahrens. Exemplarisch wurden innerhalb der Abbildung die Winkel b und a für $\overrightarrow{QP1}$ beziehungsweise $\overrightarrow{QP2}$ eingezeichnet. Nachdem sämtliche Winkel für alle Punkte berechnet und diese sortiert wurden, iteriert man über die Punktemenge. Intern wird ein Stack verwendet, auf dem sich vor jeder weiteren Iteration des Verfahrens sämtliche bis zu diesem Zeitpunkt berechneten Punkte der konvexen Hülle befinden. Zu Beginn legt man die beiden anfänglich bestimmten Punkte ab, im Beispiel wären dies der Startpunkt Q und der Punkt $P1$. In jeder folgenden Iteration entnimmt man der sortierten Liste einen weiteren Punkt. Im Beispiel wäre dies der Punkt $P2$. Falls mehrere Punkte mit dem gleichen Winkel innerhalb der Liste vorkommen, so wählt man denjenigen, der am weitesten vom Startpunkt Q entfernt ist [La12]. Durch das Hinzufügen neuer Punkte zum Stack kann der Fall auftreten, dass vorab vorhandene Punkte nicht mehr länger auf der konvexen Hülle liegen. Für einen beliebigen Punkt P, der in der aktuellen Iteration zum Stack hinzugefügt wurde, muss darum für die vorherigen Punkte getestet werden, ob sie sich links oder rechts des Vektors $\overrightarrow{QP}$ befinden. Dabei beginnt man mit dem vorletzten Punkt auf dem Stack. Befindet sich dieser nicht links vom Testvektor $\overrightarrow{QP}$, so wird er vom Stack entfernt. Es ist typischerweise nicht erforderlich, den gesamten Stack zu durchlaufen. Sobald ein Punkt gefunden wurde, der links vom Testvektor liegt, kann die Überprüfung beendet werden. Das Verfahren terminiert, sobald sämtliche Punkte in der initialen Punktemenge verarbeitet wurden.

Das Verfahren besitzt eine Laufzeit von $O(n \log n)$ [La12]. Diese resultiert aus den Kosten der Sortierung der Punktemenge basierend auf den vorab berechneten Winkeln. Schnelle Sortierverfahren wie *Mergesort* oder *Quicksort* erreichen Laufzeiten von $O(n \log n)$, das nachfolgende Durchlaufen der Punktmenge liegt bezüglich der Laufzeit in $O(n)$, somit gilt für die Gesamtlaufzeit $O(n) + O(n \log n) = O(n \log n)$.

8.4 **Ähnlichkeitsbasierte Grundrisserzeugung**

8.4.1 **Einleitung**

Die automatische Erzeugung von Grundrissen ist ein Kernbestandteil des hier vorgestellten Systems. Neben dem bereits vorgestellten Objectplacement-Verfahren bietet das System eine weitere Technologie, die in der Lage ist, realistische Grundrisse zu erzeugen. Während das Objectplacement-Verfahren darauf basiert, innerhalb vorgegebener Parameterbereiche Grundelemente zu positionieren und aus den derart entstehenden Kombinationen Grundrisse

abzuleiten, ist die Grundlage der ähnlichkeitsbasierten Grundrisserzeugung die Eingabe eines Beispielgrundrisses. Das Verfahren analysiert diesen Beispielgrundriss und ist anschließend in der Lage, neue Grundrisse zu erzeugen, die diesem ähneln. Der Algorithmus ist inspiriert durch das Continuous Model Synthesis-Verfahren, das Merrell et al. 2008 vorgestellt haben [MM08].

8.4.2 **Verfahren**

Nachdem die Hauptverfahrensschritte der Continuous Model Synthesis bereits im Kapitel „Continuous Model Synthesis [MM08]“ erläutert wurden, wird nun auf deren konkrete Umsetzung im Semantic Building Modeler eingegangen. Anschließend werden die Anpassungen diskutiert, die für den Einsatz des Verfahrens zur Grundrisserzeugung erforderlich sind.

Auf der obersten Ebene lässt sich der Algorithmus in zwei Hauptschritte gliedern. In einem ersten Schritt wird der Eingabegrundriss analysiert und das Ergebnis dieser Analyse in Form einer Menge von Regeln gespeichert. Der zweite Schritt besteht in der Anwendung dieser Regeln mit dem Ziel, einen neuen Grundriss zu synthetisieren. Zunächst wird auf die gemeinsamen Berechnungen eingegangen, die in beiden Hauptschritten erforderlich sind, bevor anschließend die spezifischen Berechnungen thematisiert werden.

8.4.2.1 **Ermittlung der Grundkomponenten des Verfahrens**

Die geometrischen Grundkomponenten des Algorithmus sind Faces, Kanten und Vertices. Sowohl im Zuge der Regelberechnung als auch bei der Model-Synthese ist es zunächst erforderlich, diese Komponenten zu bestimmen. Grundsätzlich startet die Berechnung mit einer Menge von Strahlen. Diese werden paarweise auf Schnitte getestet. An Schnittpunkten werden Vertices erzeugt, die ihre ein- und ausgehenden Strahlen speichern. Die Schnittberechnung erzeugt ein Raster bestehend aus einer Menge von Vertices, die durch Kanten miteinander verbunden sind. Intern wird das Ergebnis durch eine gerichtete Graphenstruktur repräsentiert. Jedes Vertex besitzt dabei maximal zwei ausgehende und zwei eingehende Kanten. Für die weiteren Berechnungen ist es erforderlich, zusätzlich zu Kanten und Vertices auch die Faces zu berechnen, die durch die Kantenzüge eingeschlossen werden. Dabei ist jede Kante adjazent zu maximal zwei Faces. Die Berechnung der Polygone, die durch die Kantenzüge beschrieben werden, ist kein triviales Problem.

Innerhalb des vorliegenden Systems wurde die Berechnung als *Kürzeste-Wege-Problem* modelliert. Die Grundidee sei am Beispiel einer Kante mit zwei adjazenten Faces beschrieben, die zwei Vertices A und B miteinander verbindet. Jedes dieser Faces entspricht innerhalb der Graphenstruktur einem kürzesten Weg zwischen den beiden Vertices, der nicht über die direkte Kante verläuft, die die Vertices miteinander verbindet. Für die Berechnung dieser kürzesten Wege verwendet das System den *Dijkstra-Algorithmus*, der bereits 1959 von Edsger W. Dijkstra veröffentlicht wurde [Di59].

8.4.2.1.1 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus gehört zur Klasse der *Greedy-Algorithmen*. Diese zeichnen sich durch eine iterative Problemlösungsvorschrift aus, bei der in jedem Iterationsschritt die Teillösung gewählt wird, die zum Zeitpunkt des Schrittes „die Beste" ist. Im Falle des Dijkstra-Algorithmus ist es „das Beste", den Streckenabschnitt zu wählen, der die kürzeste Distanz von einem Startknoten zu einem bestimmten Knoten im Graph realisiert. Konzeptuell berechnet das Verfahren alle kürzesten Wege vom Startknoten zu jedem anderen Knoten im Graph. In der vorliegenden Implementation terminiert das Verfahren, sobald es am Zielknoten angekommen ist. Aufgrund der Struktur des Algorithmus gilt, dass zum Zeitpunkt des Erreichens eines Knotens der kürzeste Weg zu diesem bestimmt wurde. Diese Kernvoraussetzung trifft allerdings nur für Graphen mit positiven Kantengewichten zu. Sobald negative Kantengewichte im Graphen vorkommen, müssen andere Algorithmen, beispielsweise das *Bellman-Ford-Verfahren*, für das Kürzeste-Wege-Problem eingesetzt werden.

In jeder Iteration des Verfahrens startet man bei einem Ausgangsknoten X. Dessen Distanz zum Startknoten sei $Dist(X)$. Falls X der Startknoten ist, so ist $Dist(X) = 0$, sonst sei der vorher bestimmte Abstand $Dist(X) = a$. Für jeden noch nicht besuchten Nachbarknoten von X ist die Distanz zum Startknoten über den Knoten X gleich der Distanz vom Startknoten zu X plus der Länge der Verbindung von X zu seinem Nachbarn. Ist dieser Weg zum Nachbarknoten kürzer als die vorab gespeicherte Distanz für den Nachbarknoten, wird dessen Distanz aktualisiert und der Weg über X als kürzester Weg gespeichert. Dieser Schritt wird als *Update* bezeichnet, da die Datenstrukturen zur Verwaltung der Distanz und des Weges für eine Menge von Knoten aktualisiert werden. In der nächsten Iteration wählt man nun genau den Knoten als Ausgangsknoten, dessen Abstand zum Startknoten minimal ist. Aufgrund der Verfahrensstruktur ist es nicht möglich, dass ein kürzerer Weg zu diesem Knoten gefunden wird, da jeder weitere potentielle Ausgangsknoten in der Liste mindestens

genauso weit entfernt ist. Das Verfahren terminiert, sobald der Zielknoten erreicht wurde. Die Laufzeit des Verfahrens für n Knoten beträgt bei einer naiven, listenbasierten Implementation $O(n^2)$. Im schlimmsten Fall muss n-mal der Knoten gesucht werden, der aktuell den geringsten Abstand zum Startknoten besitzt. Dieser fungiert als Ausgangsknoten für die nächste Iteration. Eine naive Implementation benötigt für die Suche nach diesem Ausgangsknoten eine Laufzeit von $O(n)$, somit ergibt sich die Laufzeit $O(n * n)$. Diese lässt sich durch den Einsatz geeigneter Datenstrukturen wie eines *Fibonacci-Heaps* auf $O(m + n \log n)$ reduzieren, da das Auffinden des Knotens mit minimaler Entfernung zum Startknoten effizienter umsetzbar ist. Dabei ist m die Anzahl der Kanten im gesamten Graph und muss in der Laufzeitabschätzung berücksichtigt werden, da sie für die Anzahl der Updates relevant ist.

8.4.2.1.2 Einsatz des Dijkstra-Algorithmus zur Polygonberechnung

Nachdem vorab das Prinzip des Dijkstra-Verfahrens erläutert wurde, folgt die Diskussion nun der Frage, wie dieser Algorithmus zur Bestimmung von Polygonen in einem Graphen eingesetzt werden kann. Ausgangspunkt ist eine Graphenstruktur, die für die Dijkstra-Berechnung als ungerichtet interpretiert wird. Um nun mittels des Dijkstra-Verfahrens adjazente Faces zu einer Kante zu bestimmen, wählt man diese Kante als Ausgangspunkt des Verfahrens. Sämtliche Kanten werden zunächst auf ein Kantengewicht von 1 initialisiert. Anschließend setzt man die Kante, für die adjazente Faces bestimmt werden sollen, auf einen beliebig hohen Wert, mindestens aber auf die Summe aller Kantengewichte im Graphen. Dadurch wird diese Kante von vornherein aus der Wegberechnung ausgeschlossen. Der direkte Weg wird für den Algorithmus somit niemals der Kürzeste sein. Das Ergebnis der Berechnung ist nun genau das Polygon, welches das erste adjazente Face der Kante beschreibt. Dieses Face wird sämtlichen Kantensegmenten des Polygonzugs hinzugefügt. Außerdem wird jede Kante innerhalb dieses Kantenzugs markiert. Wurde eine Kante auf diese Art und Weise zweimal markiert, so wurden für diese beide adjazenten Faces berechnet. In diesem Fall muss das Verfahren für diese Kante nicht gestartet werden. Neben diesem Ausschluss vollständig berechneter Kanten dient die Markierung auch der Gewichtung des Dijkstra-Algorithmus für den nächsten Durchlauf mit der gleichen Startkante. Da die Ausgangskante adjazent zu zwei Faces ist, muss auch das Verfahren zweimal berechnet werden. Um zu garantieren, dass nicht wiederum das gleiche Polygon berechnet wird, werden die markierten Kanten im nächsten Durchlauf höher gewichtet. Dadurch vermeidet der Algorithmus den erneuten Besuch einer bereits vorab durchlaufenen

Kante. Das Ergebnis des zweiten Durchlaufs ist dann genau das zweite adjazente Face. Auch für dieses markiert man besuchte Kanten und entfernt solche aus der Liste der noch ausstehenden Berechnungen, für die bereits die adjazenten Faces ermittelt wurden. Die Berechnungen terminieren, sobald für alle Kanten ihre adjazenten Faces berechnet wurden.

Der Einsatz des Dijkstra-Algorithmus für die Faceberechnung benötigt für einen Graphen mit n Knoten und m Kanten eine Laufzeit von $O(2m(m + n\,logn))$. In der Praxis ist das Verfahren allerdings aus mehreren Gründen deutlich schneller. Zum einen ist die Menge der Knoten, die in jedem Durchlauf bis zur Terminierung des Verfahrens durchlaufen werden müssen, deutlich kleiner als n. Dies liegt in der Struktur des Graphen begründet, innerhalb dessen man sich bewegt. Es muss einen relativ kurzen Kantenzug geben, der die beiden Ausgangsvertices miteinander verbindet, somit ist auch die Anzahl der erforderlichen Iterationen meist deutlich kleiner als n. Zum anderen muss das gesamte Verfahren nicht für sämtliche m Kanten berechnet werden, da in jedem Durchlauf der gefundene Kantenzug für alle enthaltenen Kanten als Polygon hinzugefügt wird. Sobald auf diese Art für eine Kante zwei adjazente Polygone bestimmt wurden, wird diese aus der Hauptschleife entfernt. In der Praxis ist das Gesamtverfahren demnach deutlich schneller als die vorab genannte Worst-Case-Laufzeitschranke.

8.4.2.2 **Analyse des Eingabegrundrisses / Berechnung der Regelmenge**

Bevor innerhalb des Algorithmus neue Grundrisse synthetisiert werden können, ist es zunächst erforderlich, die Struktur des Eingabegrundrisses zu analysieren und innerhalb des Systems abzubilden. Das Ergebnis dieser Analysephase besteht wie bei Merrell et al. aus einer Menge von Regeln, die in der Synthese-Phase als Ausgangspunkt für die Grundrisserzeugung eingesetzt werden. Diese Regeln enthalten eine Menge von Festlegungen für die adjazenten Faces, die angeben, ob das jeweilige Face innerhalb oder außerhalb des Eingabegrundrisses liegt. Die Regelberechnung erfolgt darum in zwei Schritten. Zunächst ermittelt man mittels des oben beschriebenen Verfahrens die Grundelemente der Graphenstruktur und erhält dadurch eine Menge von Vertices, Kanten und Faces. Basierend auf diesen Basiskomponenten ist es anschließend möglich, die Regelmenge zu bestimmen.

In einem ersten Schritt ist es erforderlich, Labels für Kanten und Vertices zu vergeben. Diese Labels werden anhand der Komponenten des Eingabepolygons bestimmt. Bei Kanten

wird die Steigung für die Labelvergabe verwendet. Diese wird berechnet als Winkel der Kante zur x-Achse, wobei davon ausgegangen wird, dass sich der Eingabegrundriss in der xz-Ebene befindet. Dadurch werden antiparallele Kanten als unterschiedlich angesehen, was für die Regelberechnung von großer Bedeutung ist. Besitzen zwei Kanten die gleiche Steigung, erhalten sie das gleiche Label. Die Kantenlabels werden durchnummeriert und mit dem Präfix *r* versehen. Anhand der Kantenlabels können im nächsten Schritt Labels für die Vertices innerhalb der Eingabe vergeben werden. Ein Vertexlabel besteht immer aus den Labels der Kanten, die sich in dem Vertex schneiden. Liegt ein Vertex *a* beispielsweise am Schnittpunkt der beiden Kanten *r1* und *r2*, so lautet das Vertexlabel *r1_r2*. Nachdem die Labels für Kanten und Strahlen bestimmt wurden, verfügt man über eine interne Repräsentation der innerhalb der Eingabe vorkommenden Strukturen, die für die weitere Berechnung eingesetzt werden kann.

Abbildung 77 zeigt ein Beispiel, das das Benennungsschema illustriert. Die hellgrüne Fläche zeigt das Eingabepolygon, dessen Vertices im Uhrzeigersinn definiert sind. Die Richtung der Polygonkanten wird durch Pfeile verdeutlicht. Neben jeder Kante und jedem Vertex findet sich das Label der jeweiligen Komponente. Bei Vertices ist anzumerken, dass die Labels in einer normalisierten Form vergeben werden, bei der immer das alphabetisch kleinere Kantenlabel zuerst steht. In der Abbildung ist außerdem zu erkennen, dass antiparallele Kanten unterschiedliche Labels zugewiesen bekommen.

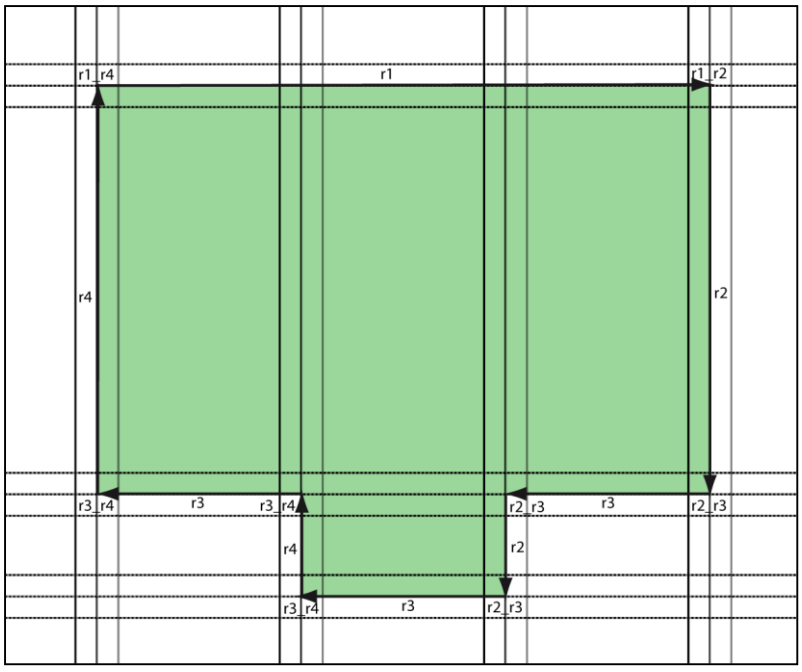

Abbildung 77: Labelberechnung für Kanten und Vertices

Neben der Labelvergabe zeigt die Abbildung zusätzlich auch die Strukturen, die für die Regelberechnung eingesetzt werden. Zusätzlich zu den Kanten und Vertices des Eingabepolygons sind in der Abbildung weitere Kanten zu sehen. Für jede Kante existieren jeweils zwei parallele Kanten, eine aus Sicht des Polygonmittelpunkts weiter entfernte und eine nähere Kante. Diese Kanten werden als Ausgangspunkt für die Schnittpunkt- und Faceberechnungen verwendet. Nachdem mittels des oben beschriebenen Verfahrens diese Grundstrukturen ermittelt wurden, kann man diese verwenden, um Regeln automatisiert abzuleiten.

Dabei muss zunächst zwischen Regeln für Kanten und solchen für Vertices unterschieden werden. Kanten sind adjazent zu genau zwei Faces, Vertices zu vier. Dementsprechend enthält eine *Kantenregel* auch Zuweisungen für zwei, eine *Vertexregel* Zuweisungen für vier Faces. Da diese Zuweisungen an Positionen in Bezug auf die Komponenten gebunden sind, ist es zentral, dass die Positionen für die jeweiligen Faces nach einem einheitlichen Schema ermittelt werden. Bei Vertices unterscheidet das System zwischen *links_oben*, *links_unten*, *rechts_oben* und *rechts_unten*, bei Kanten wird nur zwischen *oben* und *unten* differenziert. Darauf basierend erfolgt anschließend die Regelberechnung, beginnend mit der Bestimmung der Regeln für Strahlen, anschließend werden Regeln für Vertices ermittelt.

Die Berechnung der Strahlenregeln durchläuft sämtliche berechneten Kanten, dazu gehören sowohl die Eingabekanten als auch die zu diesen parallelen Kanten. Durch Schnittpunktberechnung mit den anderen Kanten innerhalb des Systems entstehen Vertices, die die Ausgangsstrahlen in Kantensegmente unterteilen. Die Kantensegmente erhalten das gleiche Label wie die Kanten, aus denen sie hervorgehen. Vertices, die an Schnittpunkten erstellt wurden, erhalten ein Label, das aus den Kanten bestimmt wird, die sich in diesem Punkt schneiden. Bei den Kantenregeln durchläuft man sämtliche Kanten und erzeugt für jedes Kantenlabel eine Menge von Regeln, die für dieses gültig sind. Die Bestimmung dieser Regeln testet, ob sich die zu der Kante adjazenten Faces innerhalb oder außerhalb des Eingabepolygons befind en. Diese Berechnung erfolgt mittels Punkt-in-Polygon-Tests, die den Mittelpunkt des adjazenten Faces dahingehend untersuchen, ob er sich innerhalb des Eingabebereiches befindet. Für die Bestimmung der Vertexregeln geht man analog vor, testet aber sämtliche vier adjazenten Faces in Bezug auf das Eingabepolygon. Das Ergebnis dieser Berechnung ist eine Menge von Regeln je Label, die die Struktur des Eingabepolygons repräsentieren.

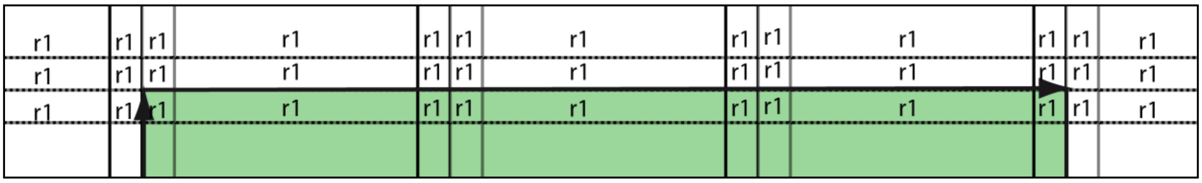

Abbildung 78: Bestimmung gültiger Strahlenregeln

Abbildung 78 zeigt einen Auszug aus Abbildung 77, wobei nur die obere Kante mit Label *r1* betrachtet wird. Durch die Schnittpunktberechnung der Strahlen mit allen anderen Strahlen innerhalb des Systems entsteht eine Menge von Kantensegmenten, die ebenfalls alle das Label *r1* zugewiesen bekommen. Bei der Schnittpunktberechnung werden nicht nur Kanten, sondern Strahlen als Ausgangspunkt verwendet, wodurch auch Schnittpunkte entstehen, die nicht innerhalb der Liniensegmente der Eingabekanten liegen. Nun durchläuft das Verfahren alle Kantensegmente und ermittelt Regeln für das Label *r1*. Tabelle 9 listet die drei unterschiedlichen Regeln auf, die für die Kante mit Label *r1* durch das System ermittelt werden. Werden Regeln mehrfach berechnet, werden diese nur einmal verwendet.

Regel	**Oberes Face**	**Unteres Face**
#1	außen	außen
#2	innen	innen
#3	außen	innen

Tabelle 9: Regelberechnung für Kanten

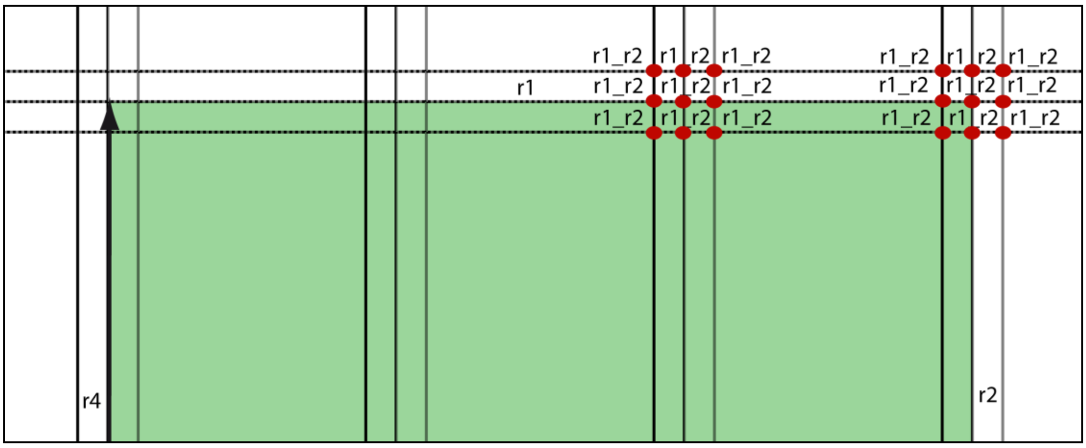

Abbildung 79: Bestimmung gültiger Vertexregeln

Abbildung 79 illustriert die Berechnung von Vertexregeln. Innerhalb des Beispiels existieren insgesamt 18 Vertices mit dem Label *r1_r2*, die durch Schnitte von Strahlen mit den Labels *r1* und *r2* erzeugt wurden. Diese Vertices sind in der Abbildung durch rote Kreise markiert und mit ihrem Label beschriftet. Anhand dieser Vertices und den zu ihnen adjazenten Faces ist es möglich, Vertexregeln abzuleiten.

Tabelle 10 zeigt die Ergebnisse der Regelberechnung für das Vertexlabel *r1_r2*. Auch hierbei gilt, dass mehrfach vorkommende Regeln immer nur einmal zur Regelmenge hinzugefügt werden. Das beschriebene Verfahren zur Regelberechnung ähnelt konzeptuell dem Vorgehen von Merrell et al., zum exakten Verfahren für die Regelbestimmung äußern sich die Autoren nicht. Auch ihr Verfahren für die Bestimmung der erforderlichen Grundelemente bleibt unklar. Aus diesem Grund wurde das hier gewählte Vorgehen entwickelt und detailliert beschrieben.

Regel	**Face oben links**	**Face unten links**	**Face oben rechts**	**Face unten rechts**
#1	innen	innen	innen	innen
#2	außen	außen	außen	außen
#3	außen	innen	außen	innen
#4	außen	innen	außen	außen
#5	innen	innen	außen	außen

Tabelle 10: Regelberechnung für Vertices

8.4.2.2.1 Regelklassifizierung

Eine Erweiterung der Regeln im Vergleich zu Merrell et al., die in der hier vorliegenden Variante verwendet wird, besteht in der Klassifizierung der Regeln. Diese dient im Kontext der Grundrisssynthese zur Steuerung des Verfahrens. Kantenregeln werden in drei verschiedene Kategorien eingeteilt. *INDOOR*-Regeln sind Kantenregeln, bei denen beide adjazenten Faces auf innen gesetzt werden, also im Inneren des neu erzeugten Grundrisses liegen. *OUTDOOR*-Regeln sind das Gegenteil von *INDOOR*-Regeln und erzeugen Strukturen, die vollständig außerhalb des Ergebnisgrundrisses liegen. Bei *EDGE*-Regeln befindet sich eines der adjazenten Faces innerhalb und eines außerhalb des neuen Grundrisses. Solche Regeln erzeugen demnach Kanten in der neuen Struktur. Bei

Vertexregeln unterscheidet man neben den drei bereits erläuterten Regeltypen auch noch zwischen Regeln, die Eckpunkte erzeugen. Eckpunkte selber werden wiederum in zwei Typen unterschieden. *CORNER*-Regeln erstellen Eckpunkte, bei denen der Innenwinkel zwischen den Kanten des Eckpunktes einen Betrag kleiner 180° aufweist. *REFLEX_CORNER*-Regeln erzeugen konkave Vertices, bei denen der Innenwinkelbetrag größer als 180° ist. Diese Regelklassifizierung dient an späterer Stelle als eine der zentralen Möglichkeiten zur Steuerung des Verfahrens.

Nachdem die Regelmenge vollständig berechnet wurde, wird diese im zweiten Hauptschritt des Verfahrens eingesetzt, um neue Grundrisse zu berechnen, die dem Eingabegrundriss ähneln.

8.4.2.3 **Synthesephase**

Die Synthesephase stellt die zweite Hauptphase innerhalb des gesamten Verfahrens dar. Sie verwendet als Ausgangspunkt die Label- und Regeldefinitionen, die in der ersten Hauptphase berechnet wurden. Außerdem wird für jedes Strahlenlabel dessen zugehöriger Richtungsvektor weiterverwendet. Alle weiteren vorab berechneten Strukturen werden geleert.

8.4.2.3.1 Vorbereitung der Syntheseberechnung

Auch die Synthesephase beginnt mit der Berechnung der Grundstrukturen, also der Vertices, Kanten und Faces basierend auf dem oben beschriebenen Verfahren. Ausgangspunkt sind nun nicht mehr die Kanten des Eingabepolygons, sondern nur noch deren Richtungsvektoren. Für jeden Richtungsvektor und somit jedes Strahlenlabel erzeugt das Verfahren eine Menge paralleler Strahlen, die in einem festen Abstand zueinander positioniert werden. Anschließend berechnet man sämtliche Schnitte der Strahlen untereinander, erzeugt an den Schnittpunkten Vertices und weist diesen nach dem obigen Schema Labels zu. Die Berechnung der parallelen Strahlen unterscheidet sich in einem wichtigen Aspekt von diesem Verfahrensschritt im Rahmen der Regelbestimmung. Während bei den Regeln der exakte Verlauf der Eingabekanten, also sowohl Richtungs- als auch Stützvektor als Ausgangspunkt verwendet werden, ist dies bei der Synthesephase nicht mehr der Fall. Hier wird nur noch der Richtungsvektor, nicht aber der Stützvektor verwendet. Als Stützvektor benutzt das Verfahren einen Basisvektor, der für jedes Strahlenlabel

zufallsbasiert leicht verschoben wird. Diese randomisierte Verschiebung soll verhindern, dass antiparallele Strahlen durch das Verfahren übereinander gelegt werden. Dies führt in der weiteren Berechnung, speziell in Bezug auf die Bestimmung und Zuordnung von Faces, zu großen Problemen. Konkret entstehen genau dann Schwierigkeiten, wenn sich mehr als zwei Strahlen in einem Punkt schneiden. In diesem Fall ist das Vertex nicht mehr adjazent zu vier, sondern zu sechs oder mehr Faces. Eine solche Konfiguration ist aus verschiedenen Gründen problematisch und führt zum Scheitern der Berechnung. Ursächlich dafür sind mehrere Aspekte. Zum einen ist dies die Frage, was für ein Label ein Vertex zugewiesen bekommt, das im Schnittpunkt von mehr als zwei Strahlen liegt. Nicht nur die Labelzuweisung ist in diesem Fall schwierig, auch die Auswahl und Anwendung gültiger Regeln ist nicht mehr eindeutig möglich. Durch die zufallsbasierte Verschiebung von Strahlen über ihren Stützvektor kann die Auftretenswahrscheinlichkeit dieses Falls reduziert werden, ganz auszuschließen ist eine solche Konfiguration allerdings nicht. Aus diesem Grund testet die Software, ob es zu einem solchen Fall gekommen ist und startet bei Bedraf die Berechnung erneut. Durch die zufallsbasierte Verschiebung verlaufen die Strahlen in der nächsten Iteration leicht verschoben, wodurch die Wahrscheinlichkeit eines erneuten Auftretens des Fehlers verringert wird. Leider ist diese einfache Form der Fehlerbehandlung nur bei der Synthese, nicht aber bei der Analyse möglich. In dieser Berechnungsphase ist es von entscheidender Bedeutung, dass der Strahlenverlauf dem Eingabepolygon exakt gleicht. Nur so kann garantiert werden, dass die ermittelte Regelmenge eine korrekte Repräsentation des Eingabepolygons darstellt. Dies gilt insbesondere für Vertexregeln, da eine Verschiebung der Vertices zu anderen Regeln und somit inkorrekten Vorbedingungen für die Synthesephase führt. Leider ist es darum nicht möglich, den gleichen Ansatz zur Fehlervermeidung zu nutzen, der in der Synthesephase verwendet wird. Eine solche Konfiguration kann als potentielle Fehlerursache nicht vollständig ausgeschlossen werden, allerdings ist ihr Auftreten auch im Vergleich zur Synthesephase weniger wahrscheinlich. Dies liegt zum einen daran, dass weniger Strahlen auf gegenseitige Schnitte getestet werden, um die Regeln zu berechnen. Zum anderen zeichnen sich diese Strahlen bereits dadurch aus, dass sie sich an bestimmten Punkten innerhalb des Polygons treffen. In einem gültigen Eingabepolygon kann die problematische Konfiguration nur dann auftreten, wenn es an Verlängerungen der vorhandenen Strahlen zu einem Schnittpunkt von mehr als zwei Strahlen kommt. Da dieser Fall äußerst selten auftritt, wird er nicht behandelt, kann aber zum Scheitern der Berechnung führen, wenn das Eingabepolygon ungünstig strukturiert ist.

Nachdem die Grundstrukturen mittels der vorab beschriebenen Verfahren berechnet wurden, startet die eigentliche Synthese. Diese beginnt mit der Erzeugung eines Katalogs. Dieser Katalog enthält für sämtliche Kanten und Vertices alle Regeln, die auf diese anwendbar sind. Daraus ergibt sich automatisch das Vorgehen für die Katalogberechnung. Basierend auf der Regelmenge, die in der Regelberechnung erstellt wurde, wird für jedes Element eine Liste innerhalb einer Mapstruktur erzeugt, die über das Element indiziert wird. Diese Liste enthält sämtliche Regeln, die für das Label des Elements errechnet wurden.

8.4.2.3.2 Synthesealgorithmus

Die auf diesem Katalog basierende Synthese ist ein iterativer Prozess, der sich in mehrere Schritte gliedern lässt.

1. Auswahl einer Komponente (Kante oder Vertex)
2. Auswahl einer möglichen Regel für die ausgewählte Komponente aus dem Katalog
3. Überprüfung der Anwendbarkeit der Regel für die Komponente (sofern die Validierung erfolgreich war, geht das Verfahren weiter mit Schritt 4, sonst beginnt die Berechnung wieder mit Schritt 1)
4. Zuweisung der Regel zur Komponente
5. Aktualisierung des Katalogs

Die Schritte 1 und 2 können auf verschiedene Arten umgesetzt werden. Die Auswahl kann beispielsweise deterministisch oder randomisiert erfolgen. Die Frage, welche Regel- beziehungsweise Komponentenmenge für diese Auswahl verwendet wird, wird an späterer Stelle diskutiert, wenn die vorgenommenen Anpassungen des Basisverfahrens vorgestellt werden. Zunächst wird aber auf die Schritte 3-5 eingegangen.

8.4.2.3.2.1 Schritt 5 – Aktualisierung des Katalogs

In der Basisvariante von Merrell et al. wählt das Verfahren eine Komponente aus dem Katalog. Für diese Komponente wird anschließend eine Regel aus der Liste möglicher Regeln bestimmt. Welche Regeln für eine Komponente zuweisbar sind, hängt von den bisherigen Berechnungen ab. Jede Zuweisung einer Regel zu einer Komponente und somit das Setzen der zu dieser Komponente adjazenten Faces führt in Schritt 5 zu einer Aktualisierung des Katalogs während derer nicht länger zuweisbare Regeln entfernt werden. Das Verständnis der Notwendigkeit von Schritt 3 erfordert zunächst ein grundlegendes

Verständnis der Katalogaktualisierung und der Schritte, die bei dieser durchgeführt werden. Darum wird nun zunächst auf den eigentlich letzten Schritt jeder Iteration eingegangen, bevor die vorhergehenden Berechnungen detailliert erläutert werden.

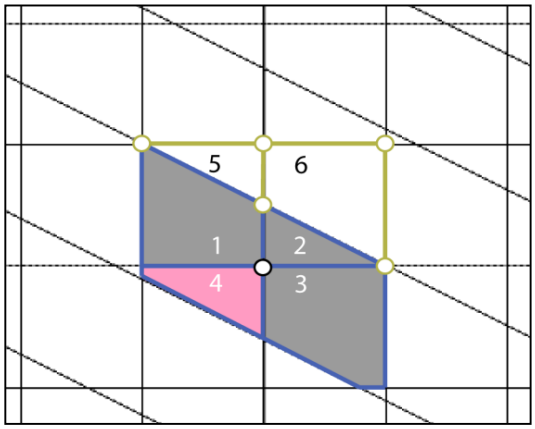

Abbildung 80: Regelzuweisung und Katalogupdates

Abbildung 80 zeigt einen Auszug aus dem laufenden Programm nach einem einzelnen Iterationsschritt. Während dieses Berechnungsschrittes wurde eine Regel für das mittlere Vertex (markiert mit weißem Kreis mit schwarzer Kontur) ausgewählt und diesem zugewiesen. Diese Regel hat den von 1-4 nummerierten, adjazenten Faces dreimal den Status *außen* (grau hinterlegte Faces) und einmal den Status *innen* (hellrot hinterlegtes Face) zugewiesen. Im letzten Teil dieser Iteration muss nun der Katalog basierend auf der durchgeführten Zuweisung aktualisiert werden. Die erforderlichen Berechnungen werden anhand der Faces 5 und 6 sowie der gelb gezeichneten Kanten in Abbildung 80 skizziert. Die offensichtlichste Form des Updates betrifft alle Komponenten, die direkt an die neu gesetzten Faces angrenzen. Dies sind sowohl Vertices als auch Kanten. Das System muss also alle blau hinterlegten Kanten durchlaufen und für jede Kante sämtliche Regeln aus dem Katalog löschen, die mit der gerade durchgeführten Zuweisung nicht kompatibel sind. Betrachtet man beispielsweise die Kante zwischen den Faces 2 und 6, so müssen aufgrund der Facefestlegung für Face 2 sämtliche Regeln aus dem Katalog entfernt werden, bei denen Face 2 nicht auf *außen* gesetzt wird. Kanten, bei denen beide adjazenten Faces gesetzt wurden, können vollständig aus dem Katalog entfernt werden, da für diese keine Zuweisung mehr existiert. Gleiches gilt für Vertexkomponenten. Wurden sämtliche adjazenten Faces eines Vertex durch Statuszuweisungen gesetzt, können auch die Vertices aus den Verwaltungsstrukturen entfernt werden. Diese Berechnungen bilden den Abschluss jeder Iteration.

Trotz des Entfernens nicht mehr zuweisbarer Regeln im Rahmen des Katalogupdates kann es innerhalb des Katalogs immer noch zu Inkonsistenzen kommen. Das Testen auf solche Inkonsistenzen kann entweder im Rahmen des Katalogupdates direkt erfolgen, oder erst im nächsten Iterationsschritt durchgeführt werden, wenn eine neue Zuweisung erfolgen soll. Die hier vorgestellte Implementation verschiebt solche Tests auf spätere Berechnungsdurchläufe, da der Berechnungsaufwand deutlich reduziert wird.

8.4.2.3.2.2 Schritt 3 – Überprüfung der Anwendbarkeit der gewählten Regel auf die gewählte Komponente

Nachdem vorab auf die Katalogaktualisierung eingegangen wurde, sollte nachfolgend der Ablauf der Gültigkeitsprüfung für Komponente und Regel leichter nachvollziehbar sein. Dabei gilt prinzipiell, dass eine Komponente-Regel-Kombination, die aus dem Katalog gewählt wurde, lokal, also in Bezug auf die Komponente, immer gültig ist. Es kann keine Regel im Katalog vorkommen, die Zuweisungen zu den adjazenten Faces der Komponente enthält, die mit dem aktuellen Zustand inkompatibel sind. Eine Zuweisung kann nur dann ungültig sein, wenn sie gegen Zuweisungen benachbarter Komponenten verstößt. Dies sei wiederum an Abbildung 80 verdeutlicht. Nachdem im vorherigen Durchlauf die eingefärbten Faces erfolgreich zugewiesen und der Katalog aktualisiert wurde, soll im nächsten Iterationsschritt der Kante zwischen den Faces 2 und 6 ein Status zugewiesen werden. Da die Kanten zwischen den gesetzten Faces alle vollständig definiert sind, befinden sie sich nicht mehr im Katalog. Somit müssen nur die gelb eingefärbten Kanten und Vertices untersucht werden, die adjazent zu Face 6 sind. Ohne Einschränkung der Allgemeinheit sei die Zuweisung zu Face 6 *innen*. Diese Zuweisung ist nur dann gültig, wenn sämtliche adjazenten Komponenten auf der Faceposition von Face 6 mindestens eine Regel besitzen, die Face 6 auf *innen* setzt. Ist dies nicht der Fall, würde die Zuweisung des Faces mit dem Status *innen* zu einem inkonsistenten Zustand führen. Ein solcher Fall kann beispielsweise dann eintreten, wenn das über Face 6 liegende Face bereits eine Zuweisung erhalten hätte. Da beim Katalogupdate nur die direkten Komponenten dahingehend getestet werden, ob sie inkonsistente Zuweisungen erhalten, fällt diese Inkonsistenz zunächst nicht auf. Prinzipiell wäre eine solche Berechnung auch im Rahmen des Katalogupdates durchführbar, indem man ausgehend von einer Zuweisung sämtliche Komponenten innerhalb des Katalogs auf Konsistenz testet. Dieses Vorgehen wäre allerdings deutlich zeitaufwendiger als der lokale Test, sobald eine konkrete Zuweisung erfolgen soll. Wird

dabei festgestellt, dass eine Regel-Komponente-Kombination nicht zuweisbar ist, wird sie aus den Katalogen entfernt und die Iteration beginnt erneut mit der Auswahl einer neuen Komponente und einer neuen Regel.

8.4.2.3.2.3 Schritt 4 – Zuweisung der Regel zur Komponente

Nachdem erfolgreich getestet wurde, ob die ausgewählte Regel für die Komponente zuweisbar ist, wird diese angewendet. Eine Regel besteht aus einer Festlegung für sämtliche adjazenten Faces einer Komponente, ob sich diese inner- oder außerhalb des synthetisierten Grundrisses befinden. Konzeptuell weist dieser Schritt jedem Face einen vorgegebenen Status zu, weitere Berechnungen sind nicht erforderlich.

8.4.2.3.2.4 Schritt 1 – Auswahl einer Komponente

In jeder Iteration stellt die Auswahl derjenigen Komponente, für die eine Zuweisung vorgenommen werden soll, den ersten Schritt dar. In der Basisvariante von Merrell et al. erfolgt die Auswahl randomisiert aus der Menge der noch nicht zugewiesenen Komponenten. Ein solcher Ansatz erzeugt hoch variable Strukturen, die Ergebnisse jedes Durchlaufs unterscheiden sich deutlich voneinander. Für die Grundrisserzeugung ist ein solcher Ansatz zu willkürlich, die erzeugten Strukturen sind typischerweise klein und nicht miteinander verbunden. Als Grundrisse sind solche Strukturen ungeeignet, dort benötigt man möglichst große, zusammenhängende Strukturen.

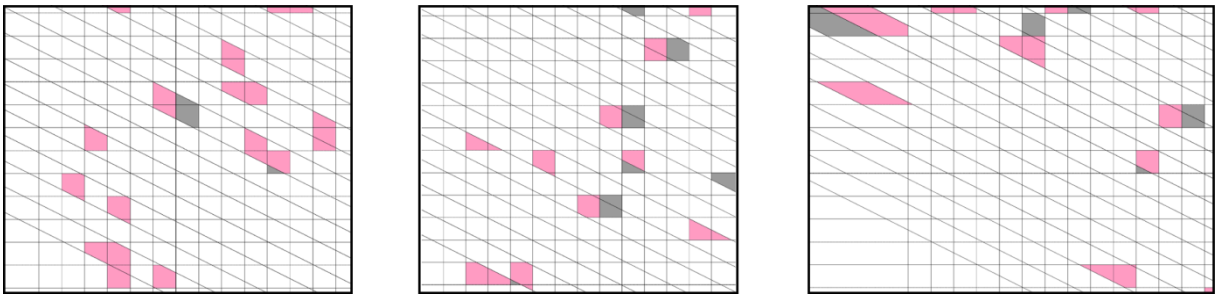

Abbildung 81: Zufallsbasierte Komponentenauswahl

Abbildung 81 zeigt ein Beispiel dafür, wie sich die zufallsbasierte Komponentenauswahl selbst bei deterministischer Regelwahl massiv auf das Ergebnis des Syntheseprozesses auswirkt. Als Eingabe wurde für das Verfahren ein rechtwinkliges Dreieck gewählt, das zwar ein eher untypisches Beispiel für einen Grundriss darstellt, dafür aber für die

Verdeutlichung der Verfahrensunterschiede gut geeignet ist. Wie auch bei vorherigen Abbildungen sind Faces mit Status *innen* hellrot, Faces mit Status *außen* hellgrau dargestellt.

Offensichtlich ist die zufallsbasierte Auswahl einzelner Komponenten für die hier verfolgte Zielsetzung nicht der richtige Weg. Für die Grundrisssynthese muss das Verfahren geordneter sein, zentral ist dabei die Erzeugung geschlossener, möglichst großer Strukturen. Diese Notwendigkeit erkannten auch Merrell und Manocha in einer späteren Arbeit [MM09]. Als Anwendungsbeispiel nennen die Autoren ein Straßennetz, das automatisch durch den Synthesealgorithmus anhand eines Beispielnetzes erzeugt werden soll. Würde die Auswahl der Zuweisungsregeln vollständig zufallsbasiert erfolgen, so würde dies mit hoher Wahrscheinlichkeit zu einem nicht geschlossenen Straßennetz führen, sondern viele vereinzelte, unverbundene Straßenstrukturen generieren. Als Lösung für diese Problematik nennen sie *Connectivity Constraints*, bei denen nur solche Vertices als Straßen markiert werden können, die bereits direkte Nachbarn von Straßenstrukturen sind. Das hier vorgestellte System verwendet mehrere Ansätze, die ein ähnliches Ziel verfolgen. Dabei ist zu unterscheiden zwischen einem Ansatz zur Komponenten- und einem Ansatz zur Regelauswahl. Der Ansatz zur Regelauswahl ist inspiriert durch die Idee der Connectivity Constraints von Merrell und Manocha. Darüber hinaus verwendet das System auch eine weitere Anpassung für die Komponentenauswahl. Bevor auf die Umsetzung der Regelauswahl eingegangen wird, soll zunächst auf das Konzept für die Komponentenbestimmung eingegangen werden.

Abbildung 81 zeigt drei unterschiedliche Berechnungszustände nach jeweils 15 Durchläufen durch die oben erläuterte Schrittfolge. Deutlich zu erkennen ist der Einfluss der randomisierten Komponentenauswahl, die gesetzten Komponenten sind über den gesamten Syntheseraum verteilt. Bei jedem Durchlauf der Software ist der Zustand nach 15 Iterationen unterschiedlich. Für die Grundrisserzeugung ist diese Berechnung zu willkürlich, die Möglichkeit, beliebige Komponenten auszuwählen führt sehr schnell zu sehr kleinen und weit verteilten Strukturen. Dies hängt unter anderem damit zusammen, dass das System bei der Regelauswahl typischerweise nicht auf Informationen aus der lokalen Nachbarschaft zurückgreifen kann, da diese zumindest in den ersten Iterationen nicht definiert ist. Erst nach einer potentiell großen Menge an Iterationen wachsen die Strukturen langsam zusammen, wodurch die zur Verfügung stehende Regelmenge verkleinert wird und somit den vorhergehenden Schritten Rechnung trägt.

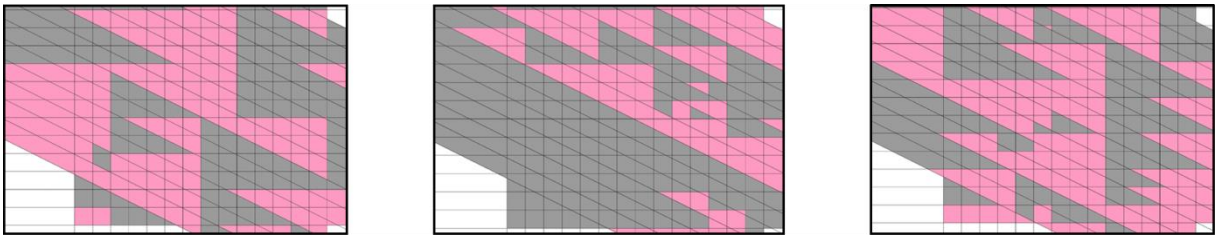

Abbildung 82: Syntheseergebnisse bei zufallsbasierter Komponentenauswahl

Abbildung 82 zeigt drei unterschiedliche Ergebnisse des Syntheseverfahrens mit zufallsbasierter Komponentenauswahl. Die erzeugten Strukturen sind bis auf einige Ausnahmen meist sehr klein und nicht zusammenhängend.

Ein erster Ansatz, der versucht, diese große Streuung zu reduzieren, ist eine *kontrollierte Komponentenauswahl*, bei der die Komponenten nicht aus dem gesamten Katalog, sondern nur aus einer Teilmenge ausgewählt werden. Diese Teilmenge ist zu Beginn des Verfahrens leer, wird aber mit Elementen gefüllt, sobald die ersten Zuweisungen erfolgt sind und Nachbarschaftsaktualisierungen berechnet wurden. Das Ziel des Einsatzes dieser Teilmenge ist eine größere Kontrolle über das Wachstum der Strukturen, unabhängig davon, welche Regeln durch das Verfahren konkret ausgewählt und angewendet werden. Ein kontrolliertes Wachstum wird dabei als ein solches aufgefasst, bei dem nur solche Komponenten zur Auswahl stehen, die adjazent zu bereits gesetzten Faces sind. Die Umsetzung eines solchen gesteuerten Wachstumsprozesses basiert auf der Verwaltung einer Menge *angefasster* Komponenten. In jeder Iteration werden Elemente nicht mehr aus dem Katalog selber, sondern nur noch aus dieser Teilmenge gewählt. Eine Komponente wird dieser Menge hinzugefügt, wenn sie adjazent zu einem Face ist, das in einer vorherigen Iteration durch eine Regelzuweisung gesetzt wurde. Wählt man nun in der nächsten Iteration eine Komponente aus dieser Menge, so werden automatisch Faces durch Regelzuweisungen gesetzt, die direkte Nachbarn bereits vorab gesetzter Faces sind.

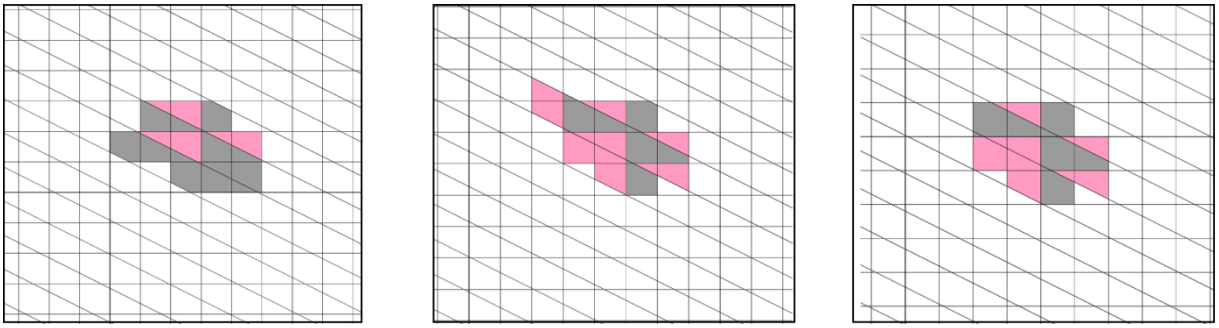

Abbildung 83: Komponentenauswahl bei kontrolliertem Wachstum

Abbildung 83 illustriert sehr deutlich, wie sich die berechneten Strukturen ändern, sobald die Komponentenauswahl nur noch über die Teilmenge der bereits angefassten Komponenten erfolgt. Wie in Abbildung 81 wurden jeweils 15 Iterationen des Gesamtverfahrens berechnet, die alle von der gleichen Startkomponente ausgehen. Innerhalb der Teilmenge erfolgte die Auswahl wiederum randomisiert, wodurch die unterschiedlichen Strukturen zustande kommen.

Eine weitere Möglichkeit, mittels derer man die Komponentenauswahl beeinflussen kann, ist die Beschränkung der Auswahl auf einen bestimmten *Komponententyp*. Unabhängig davon, ob die Komponenten aus der Liste der bereits angefassten Komponenten oder aus dem Gesamtkatalog stammen, kann die Komponentenauswahl ausschließlich Vertices oder ausschließlich Kanten selektieren. Die Beschränkung auf einen bestimmten Komponententyp kann großen Einfluss auf die Ergebnisstrukturen haben. Typischerweise entstehen bei Berechnungen, die ausschließlich auf Kanten zurückgreifen, kleinere, stärker vereinzelte Strukturen. Beschränkungen auf Vertexkomponenten tendieren dagegen dazu, größere Strukturen zu erzeugen. Allerdings hängt die Form des Berechnungsergebnisses stark von der Wahl des Verfahrens zur Komponentenauswahl in Kombination mit den nachfolgend erörterten Möglichkeiten zur Regelauswahl ab.

8.4.2.3.2.5 Schritt 2 – Regelauswahl

Neben der Verwendung der vorab erläuterten Ansätze zur Komponentenauswahl steht auch ein zusätzliches Verfahren für die Regelauswahl zur Verfügung. Dieses ähnelt den Consistency Constraints, da es versucht, gezielt Regeln aus einer Regelmenge auszuwählen. Auch hier sei zunächst auf das Kernverfahren von Merrell und Manocha verwiesen, bei dem neben der Komponenten- auch die Regelauswahl zufallsbasiert erfolgt. Nachdem im 1. Schritt eine Komponente gewählt wurde, greift der Algorithmus auf den Katalog zu und holt

sämtliche Regeln, die für die gewählte Komponente noch zur Auswahl stehen und somit gültige Zuweisungen darstellen. Innerhalb dieser Regelmenge kann die Auswahl deterministisch oder randomisiert erfolgen. Bei einer randomisierten Auswahl ist die Wahrscheinlichkeit, dass eine bestimmte Regel gewählt wird, für alle Regeln gleich hoch. Die regelabhängigen Ergebnisstrukturen, die aus der Regelanwendung hervorgehen, werden bei der Regelauswahl nicht berücksichtigt. Die Möglichkeit, gezielt Einfluss auf diesen Auswahlprozess zu nehmen und Regeln unterschiedliche Wahrscheinlichkeiten zuzuweisen, ist neben der kontrollierten Komponentenauswahl die zweite Variante, das Syntheseverhalten des Algorithmus stärker kontrollierbar und dadurch seine Ergebnisse vorhersagbarer zu machen

Der erste Mechanismus, der hierfür eingesetzt werden kann, wird als *Kontinuitätskriterium* bezeichnet. Dieses basiert auf der Überlegung, dass die Ergebnisse der Synthese besonders dann für die Verwendung als Grundriss geeignet sind, wenn sie relativ groß und zusammenhängend sind. Kleine Strukturen, wie sie durch zufallsbasierte Komponenten- und Regelauswahl entstehen, sind dagegen für die Weiterverwendung ungeeignet. Das Verfahren ist darum in der Lage, bei der Regelauswahl die direkte Nachbarschaft einer Komponente zu analysieren und als Ergebnis eine Regel zu bestimmen, die „gut passende“ Strukturen erzeugt, sobald sie auf die jeweilige Komponente angewendet wird. Die Definition „gut passender“ Strukturen ist dabei durch den Anwendungsfall der Grundrisserzeugung festgelegt, deren vorrangiges Ziel es ist, Strukturen zu generieren, die eine hohe Kontinuität aufweisen. Beispielsweise sollen bestehende Kanten eher verlängert als durch Ecken gestört werden. Gleiches gilt für Bereiche, die vollständig innen oder außen liegen. Auch hier soll das Verfahren solche Regeln stärker gewichten, die die bestehenden Strukturen fortsetzen, anstatt sie zu unterbrechen. Die Umsetzung dieser einfachen, allerdings recht effektiven Vorgabe basiert auf der Auswahl solcher Regeln, die bestehende Facezustände fortsetzen. Dabei werden für alle Regeln, die potentiell einer Komponente zugewiesen werden können, die Ergebnisse der Zuweisung dahingehend untersucht, inwiefern Zustände fortgesetzt werden, also beispielsweise ein Face mit Status innen nach der Zuweisung ebenfalls an ein solches Face angrenzt. Je mehr Kontinuität durch eine Regel erzeugt wird, desto höher ist ihre Auswahlwahrscheinlichkeit.

Neben dem vorab erläuterten Kontinuitätskriterium bietet die Implementation eine weitere Möglichkeit für den Nutzer, gezielt in den Regelauswahlprozess einzugreifen. Hierbei wird auf die vorab erläuterte Regelklassifizierung zurückgegriffen. Jede Regel, unabhängig

davon, ob sie auf Kanten oder Vertices angewendet werden kann, wurde im Rahmen der Regelbestimmung einer von mehreren Kategorien zugeordnet. Die Kategorien unterscheiden sich in den Strukturen, die sie erzeugen. Regeln in der Kategorie *INDOOR* generieren Innenflächen, Regeln in der Kategorie *CORNER* Eckpunkte. Offensichtlich sind nicht alle Regeln gleich gut geeignet, um zu einem Eingabegrundriss ähnliche Strukturen zu generieren. *CORNER*- und *REFLEX_CORNER*-Regeln sind in Bezug auf die Abbildung der Charakteristika eines Eingabegrundrisses relevanter, da sie seltener sind. Für jedes Kanten- und jedes Vertex-Label existieren aufgrund der Struktur der Regelberechnung automatisch eine *INDOOR*-, eine *OUTDOOR*- und eine *EDGE*-Regel. Darum sind diese Regeln für die Generierung neuer Grundrisse weniger informationstragend als die Vertexregeln, die in der Lage sind, Eckpunkte zu erzeugen. Die Bedeutung der Regelklassifizierung wird nachfolgend an einem Beispiel verdeutlicht.

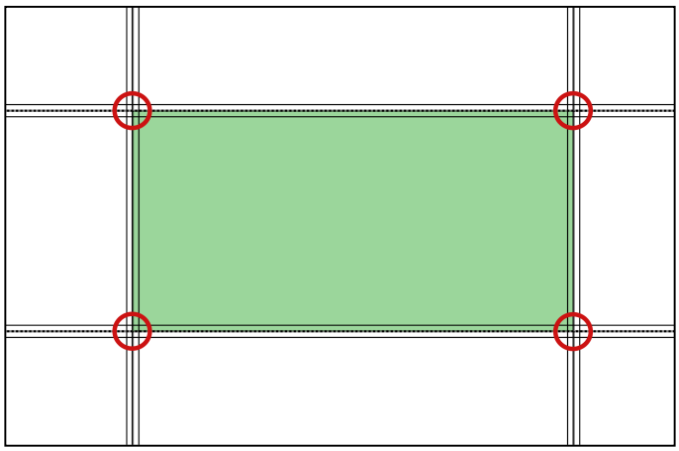

Abbildung 84: Rechteckiger Eingabegrundriss

Abbildung 84 zeigt einen einfachen rechteckigen Eingabegrundriss, bei dem die parallelen Kanten für die Regelbestimmung eingezeichnet sind. Insgesamt erzeugt das System vier Kanten- und vier Vertexlabels.

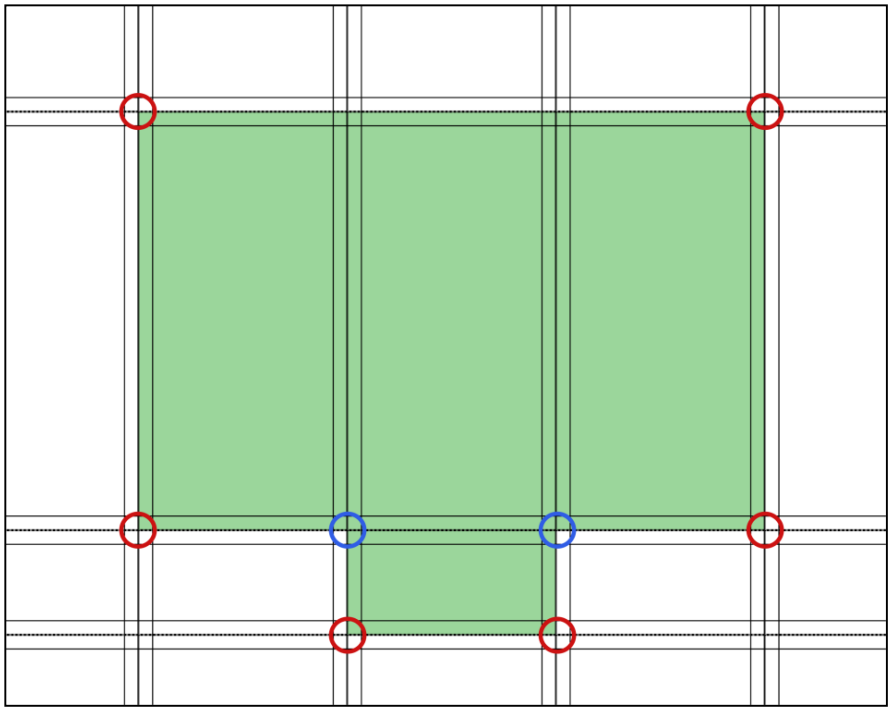

Abbildung 85: Rechteckiger Grundriss mit Erker

Abbildung 85 besteht aus einem ebenfalls rechteckigen Basisgrundriss, der an der unteren Kante eine Erkerkomponente enthält. Auch bei diesem Beispiel sind die parallelen Kanten in der Grafik dargestellt. Lässt man den an der Unterseite angebrachten Erker außen vor, so generiert das System auch hier Kanten- und Vertexlabels für die vier Basiskanten. Dabei sind die für die rot markierten Vertices generierten Regeln identisch mit denen, die für den rechteckigen Grundriss in Abbildung 84 erzeugt wurden. Die Vertices, die den ersten Grundriss von dem vorherigen Grundriss unterscheiden, sind in der Abbildung hervorgehoben. Die Regelberechnung erstellt für diese Vertices keine neuen Labels, da sie durch Kantenschnitte entstehen, für die bereits Labels existieren. Der Unterschied, der durch die Berücksichtigung dieser Vertices entsteht, ist die Ableitung von *REFLEX_CORNER*-Regeln. In der internen Regelrepräsentation unterscheiden sich die Grundrisse aus Abbildung 84 und Abbildung 85 nur aufgrund dieser beiden Vertex-Regeln. Während für den rechteckigen Grundriss insgesamt 32 verschiedene Regeln berechnet werden, sind es für den rechteckigen Grundriss mit Erker 34. An diesem Beispiel kann man sehr gut erkennen, dass die Bedeutung der verschiedenen Regeltypen für die Repräsentation der Charakteristika verschiedener Eingabegrundrisse unterschiedlich hoch ist. Aus diesem Grund ist es für die Erzeugung ähnlicher Grundrisse zentral, charakteristische Regeln mit einer größeren Wahrscheinlichkeit auszuwählen als weniger charakteristische. Die Umsetzung erfolgt durch

die Verwendung eines *Boost-Faktors*, den der Nutzer für jeden Regeltyp angeben kann. Sollen beispielsweise bevorzugt *CORNER*-Regeln verwendet werden, so erhöht man den Boost-Faktor für ebendiesen Regeltyp. Dabei wird der Faktor mit den vorher beschriebenen Auswahlkriterien für die Regelbestimmung kombiniert und ist somit nicht das alleinige Auswahlkriterium.

Im Kontext der Regelklassifizierung muss ein weiterer Aspekt berücksichtigt werden. Wie am vorherigen Beispiel gut zu erkennen, sind bestimmte Regeltypen zentraler für die Grundrissgenerierung, da sie in der Lage sind, charakteristische Strukturen zu erzeugen. Allerdings muss auch die Erzeugung solcher Strukturen in kontrollierter Art und Weise erfolgen. Der Grundriss in Abbildung 85 enthält zwei Vertices, die *REFLEX_CORNER*-Regeln erzeugen, ein ähnlicher Nachbau dieses Grundrisses sollte demnach diese Regeln nicht mehr als zweimal anwenden. Die Kontrolle der Häufigkeit der Regelanwendung erfolgt sowohl für charakteristische Regeln wie *CORNER*- und *REFLEX_CORNER*- als auch für "Standard"-Regeln wie *EDGE*, *INSIDE* und *OUTSIDE* durch die Angabe einer Obergrenze. Sobald für eine bestimmte Regel diese Grenze erreicht wurde, wird diese aus sämtlichen Katalogen entfernt. Weitere Anwendungen sind dann nicht mehr möglich.

8.4.2.4 **Extraktion des Grundrisspolygons**

Nachdem in den vorherigen Abschnitten auf die Verfahren der Analyse- und Synthesephase eingegangen wurde, befasst sich dieser Abschnitt mit der Extraktion des synthetisierten Grundrisspolygons aus dem Ergebnis der Synthesephase. Das Berechnungsergebnis besteht aus einer Menge von Faces, die durch das Verfahren entweder als *innen* oder *außen* markiert wurden. Im letzten Schritt müssen diese Faces zu Polygonen zusammengefasst werden, die in den weiteren Komponenten des Systems als Grundrisse weiterverarbeitet werden können. Hierfür greift das Verfahren auf den vorab vorgestellten Footprint-Merger-Algorithmus zurück. Die erneute Verwendung dieser Implementation bietet sich aus dem Grunde an, da die Vorbedingungen sowohl im Kontext der ähnlichkeitsbasierten Grundrisssynthese als auch im Rahmen des Objectplacement-Verfahrens sehr ähnlich sind. In beiden Fällen wird durch einen vorab berechneten Algorithmus eine Menge Polygone erzeugt, die zu einem einzelnen Polygon verschmolzen werden sollen. Dieses Polygon stellt das Ergebnis der Berechnungen dar. In beiden Verfahren ist nicht garantiert, dass sich ein einzelnes, zusammenhängendes Polygon erzeugen lässt, speziell bei der ähnlichkeitsbasierten Grundrisserzeugung können mehrere, nicht miteinander verbundene

Polygone existieren. In diesem Fall erzeugt der Footprint-Merger-Algorithmus für jede Menge miteinander verbundener Polygone ein einzelnes Ergebnispolygon.

8.4.3 **Vergleich der Verfahrensergebnisse bei unterschiedlicher Parametrisierung**

Der nun folgende Abschnitt widmet sich dem Vergleich des Einflusses unterschiedlicher Parameterkonfigurationen auf das Syntheseergebnis und versucht dadurch, zu einer Konfiguration zu gelangen, die für den speziellen Kontext der Grundrisssynthese als geeignet angesehen werden kann. Speziell die vielfältigen Möglichkeiten zur Steuerung und Parametrisierung des Verfahrens stellen dabei eine sinnvolle Erweiterung des Basisverfahrens von Merrell und Manocha dar. Erst diese Erweiterungen und Modifikationen des grundlegenden Berechnungsansatzes sorgen dafür, dass das Verfahren überhaupt für die automatisierte Grundrissberechnung eingesetzt werden kann. Nachdem in den vorherigen Abschnitten intensiv auf die einzelnen Parameter und ihre Auswirkungen auf das Berechnungsergebnis eingegangen wurde, sollen nun noch einmal einzelne Konfigurationsparameter hervorgehoben und ihre Auswirkungen auf die Syntheseberechnung anhand von Beispielen erläutert werden.

8.4.3.1 **Parameter zur Steuerung der Komponentenauswahl**

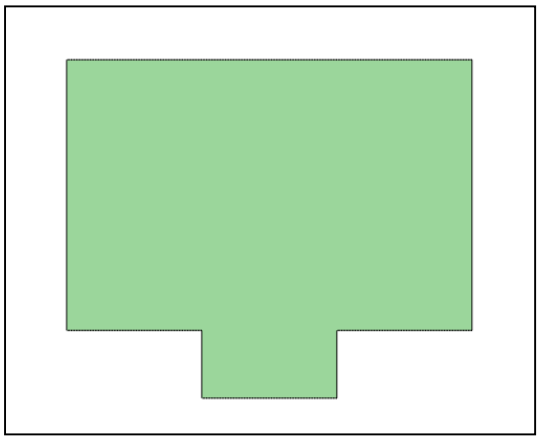

Abbildung 86: Rechteckiger Grundriss mit Erker

Die nachfolgenden Beispiele basieren auf dem in Abbildung 86 dargestellten Eingabegrundriss, der bereits an früherer Stelle verwendet wurde.

8.4.3.1.1 Einschränkung auf bestimmte Komponententypen

Ein Parameter, der im Kontext der Komponentenauswahl bereits erwähnt wurde, beeinflusst die Art der Komponenten, die das Verfahren für die Zuweisung auswählen darf. Der Nutzer hat dabei die Wahl zwischen der Beschränkung auf Kanten- oder Vertexkomponenten.

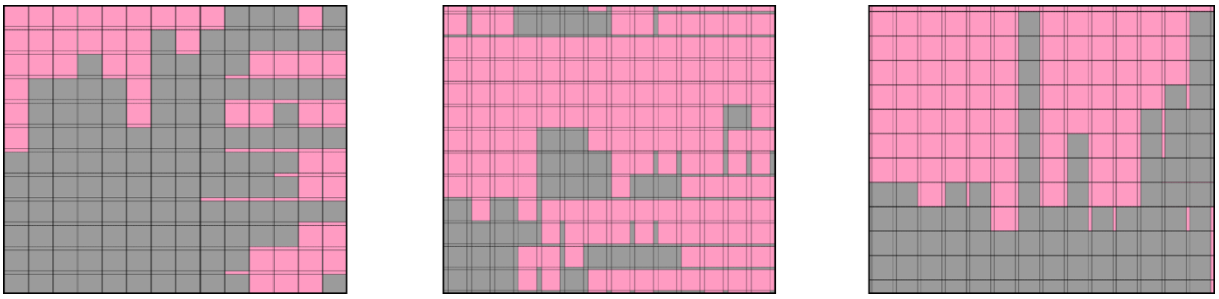

Abbildung 87: Syntheseergebnisse bei Auswahlbeschränkung auf Kantenkomponenten

Abbildung 87 zeigt verschiedene Ergebnisse der Syntheseberechnung bei einer Einschränkung auf Kanten als ausschließlich verfügbare Synthesekomponenten. Weiterhin wird bei der Regelauswahl auf die Datenstrukturen für die Verwendung bereits angefasster Komponenten zurückgegriffen, um ausgehend von einer Startkomponente möglichst gleichmäßig zu wachsen. Aus diesem Beispiel lassen sich mehrere Punkte ableiten. Zum einen tendieren Beschränkungen auf Kantenkomponenten dazu, stark verschachtelte Strukturen zu erzeugen. Dies liegt im Vergleich zu Vertexkomponenten daran, dass die Zuweisung einer Regel zu einer solchen Komponente immer nur zwei Faces setzt. Dies kann zu ausgefransten Strukturen führen. Speziell bei Grundrissen wie dem Eingabegrundriss aus Abbildung 87, die sich dadurch auszeichnen, dass zum einen die Anzahl der Kantenlabels gering ist und zum anderen antiparallele Kantenrichtungen auftreten, führt die Kantenbeschränkung schnell zu den in Abbildung 87 sichtbaren Strukturen. Darüber hinaus kann bei diesem Parameterwert sehr leicht der Fall auftreten, dass keine Elemente mehr in der Liste der angefassten Komponenten vorkommen. In diesem Fall greift das System als Fallback-Lösung auf den gesamten Katalog zurück und wählt zufallsbasiert aus diesem aus. Dadurch verstärkt sich die Vereinzelung und Ausfransung. Dieses Phänomen, das durch die rein zufallsbasierte Auswahl entsteht, wurde bereits im Kontext der Komponentenauswahl diskutiert. Ursächlich hierfür ist die Tatsache, dass Kantenkomponenten bei der Regelzuweisung deutlich weniger Komponenten zur Liste der angefassten Komponenten hinzufügen, als dies bei Vertexkomponenten der Fall ist. Dies liegt darin begründet, dass Kanten nur adjazent zu jeweils zwei Faces, Vertices dagegen adjazent zu vier Faces sind.

Dadurch erhöht sich bei Vertices automatisch auch die Anzahl der hinzugefügten Komponenten.

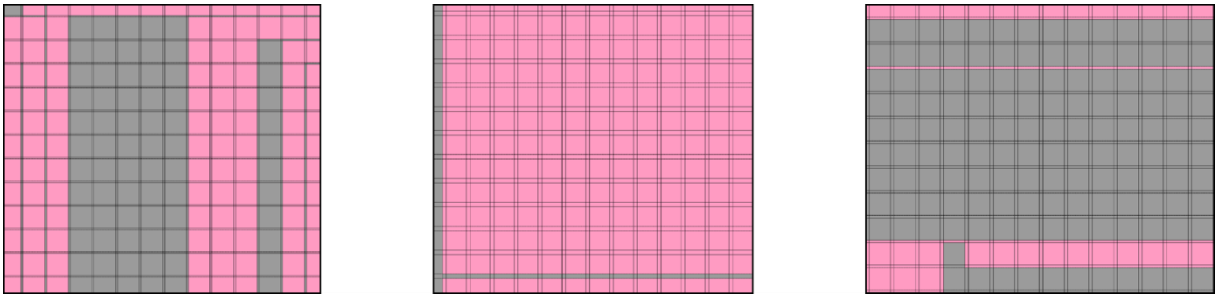

Abbildung 88: Syntheseergebnisse bei Auswahlbeschränkung auf Vertexkomponenten

Abbildung 88 zeigt Beispiele von Berechnungsergebnissen, bei denen ausschließlich Vertexkomponenten ausgewählt werden durften. Im Vergleich mit Abbildung 87 existieren zwar immer noch einzelne Komponenten und die entstehenden Strukturen sind häufig unterbrochen, insgesamt sind sie aber deutlich regelmäßiger und weniger ausgefranst. Dies liegt daran, dass bei Vertexregeln doppelt so viele Faces bezüglich ihres Status festgelegt werden als bei Kantenregeln. Werden beispielsweise *INSIDE*- oder *OUTSIDE*-Regeln angewendet, so sind mit einer Regelzuweisung vier benachbarte Faces einheitlich markiert. Dadurch erhöht sich die Regelmäßigkeit im Berechnungsergebnis, was sich auch in den Beispielen gut erkennen lässt. Da Grundrisse typischerweise wenig Ausfransungen aufweisen, eignet sich die Beschränkung auf Vertexkomponenten deutlich besser für die Grundrisssynthese.

8.4.3.1.2 Vertexkomponenteneinschränkung und Kontinuitätskriterium

Die Steuerung der Komponentenauswahl durch die Beschränkung auf Vertexkomponenten und die Menge bereits angefasster Komponenten führt zu deutlich regelmäßigeren Strukturen. Allerdings zeigt sich in Abbildung 88 die Auswirkung einer rein zufallsbasierten Regelauswahl. Da alle Regeln mit gleicher Wahrscheinlichkeit gewählt werden, spielt die lokale Umgebung einer Komponente keine Rolle. Im Kontext der Regelauswahl wurde bereits auf die Bedeutung des Kontinuitätskriteriums eingegangen, dessen Ziel darin besteht, möglichst geschlossene Strukturen ohne plötzliche Unterbrechungen zu erzeugen. Ohne Anwendung dieses Kriteriums werden geschlossene Flächen plötzlich durch Kanten unterbrochen, es entstehen voneinander getrennte Strukturen, die für die Verwendung als

Grundriss ungeeignet sind. Neben einer kontrollierten Komponentenauswahl spielt darum auch die Kontrolle über die Wahl der verwendeten Regeln eine wichtige Rolle.

Nachfolgend werden einige Beispiele für Syntheseergebnisse vorgestellt und diskutiert, für welche Arten von Eingabepolygonen das Verfahren gut oder weniger gut geeignet ist.

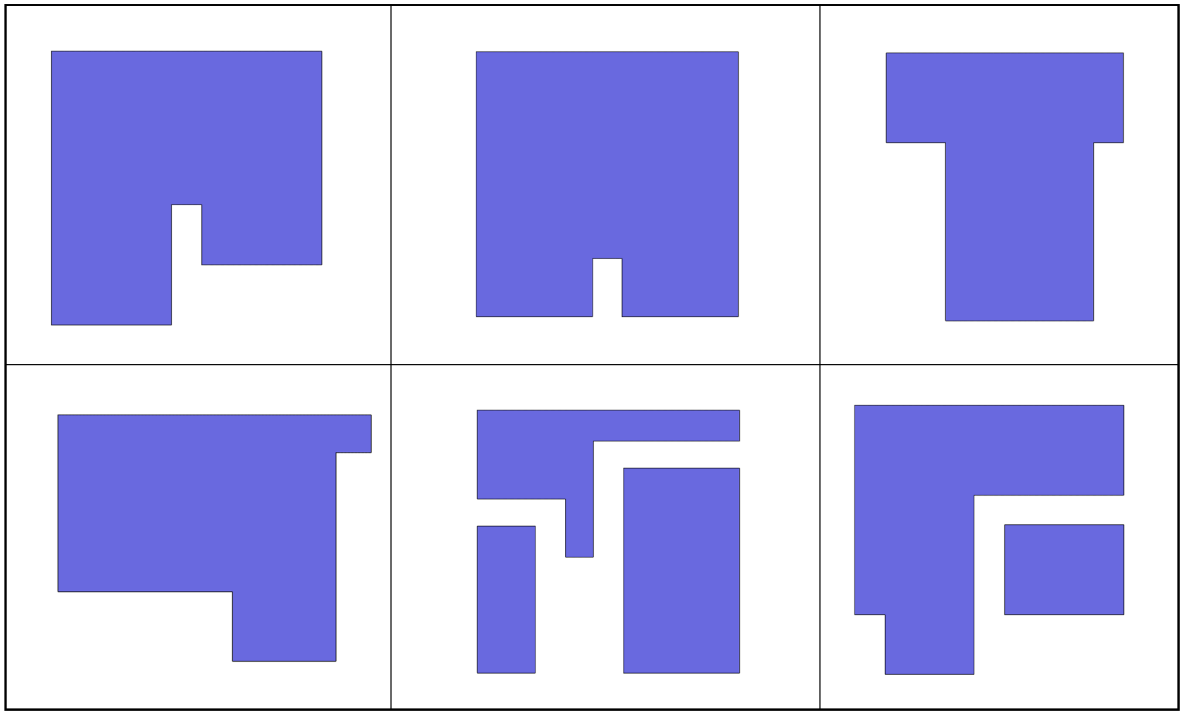

Abbildung 89: Syntheseergebnisse

Abbildung 89 zeigt eine Menge von Ergebnissen der Syntheseberechnung, die alle als Eingabe den in Abbildung 86 gezeigten Erkergrundriss erhalten haben. Bei der Synthese verwendet das System die Einschränkung auf Vertexkomponenten und setzt das Kontinuitätskriterium um. Weiterhin ist die Anzahl der Anwendungen von *CORNER*- und *REFLEX_CORNER*-Regeln auf jeweils eine Anwendung pro Regel beschränkt. Dies ist in den Beispielen gut zu erkennen. Kommen in einer Abbildung mehrere Polygone vor, so berechnet auch das Verfahren eine Reihe unterschiedlicher Grundrisse, die vom Hauptsystem eingesetzt werden können.

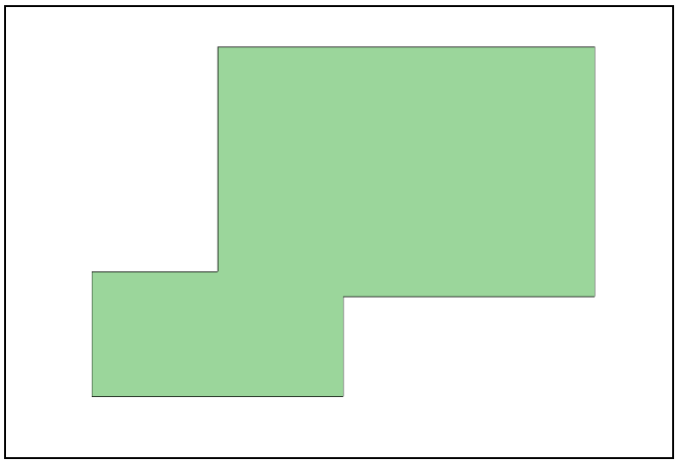

Abbildung 90: Beispielgrundriss

Abbildung 90 zeigt einen anderen Basisgrundriss, der als Eingabe in das System gegeben und für die Synthese neuer Grundrisse eingesetzt wurde. Die in Abbildung 91 gezeigten Ergebnisgrundrisse wurden unter Verwendung identischer Einstellungen basierend auf dem Eingabegrundriss errechnet.

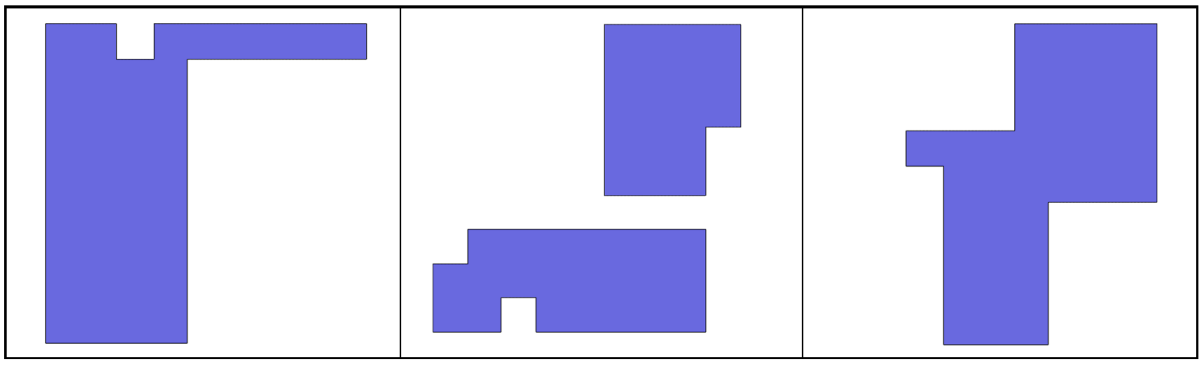

Abbildung 91: Syntheseergebnisse

8.4.4 **Diskussion der ähnlichkeitsbasierten Modellsynthese**

Für Grundrisse dieser Art ist das Verfahren gut geeignet und erzeugt realistische Variationen, die für die Gebäudekonstruktion eingesetzt werden können. Dabei ist die Auswahl geeigneter Parameter von entscheidender Bedeutung, ein geeigneter Eingabegrundriss führt nicht automatisch auch zu geeigneten Ausgaben. Die zur Verfügung stehenden Konfigurationsmöglichkeiten erlauben eine gute Steuerung des Syntheseprozesses, die Festlegung der Parameter ist allerdings vergleichsweise komplex. Hierfür ist eine Reihe von Ursachen zu nennen. Die Wichtigste ist die Tatsache, dass die Konfiguration geeigneter Werte immer die Struktur des Eingabegrundrisses berücksichtigen

muss. Parameterkonfigurationen, die für eine Art von Grundrissen zu sehr guten Ergebnissen führen, können bei einer anderen Grundrissart zu unrealistischen Resultaten oder dem Scheitern der Syntheseberechnung führen. Neben der Grundrissart spielt auch die Zielsetzung des Verfahrenseinsatzes eine große Rolle. Je größer die Ähnlichkeit der berechneten Grundrisse zum Eingabegrundriss sein soll, desto weniger sollte das Verfahren durch Zufallsoperationen gesteuert werden. Schaltet man den Zufall auf der anderen Seite vollständig aus, so sind die berechneten Strukturen uninteressant und die Ergebnisse unbefriedigend. Aufgrund der Abhängigkeit von Eingabe und Zielsetzung ist es schwierig, Parametereinstellungen zu nennen, die immer zu guten Ergebnissen führen. Als zielführend hat sich der Einsatz des Kontinuitätskriteriums erwiesen, das zu geschlossenen Strukturen führt. In Kombination mit der Einschränkung auf Vertexkomponenten erzielte das Verfahren zumindest bei Grundrissen wie den vorab diskutierten gute Ergebnisse. Schwierig ist die Wahl der Boost-Faktoren für die bevorzugte Wahl bestimmter Regeltypen. Dabei sollte als Daumenregel berücksichtigt werden, dass charakteristischere Regeln einen höheren Boost-Faktor erhalten und dadurch mit höherer Wahrscheinlichkeit ausgewählt werden. Trotzdem sollten die Faktoren nicht derart gesetzt werden, dass ausschließlich ein bestimmter Regeltyp angewendet wird, da auch dies zu unrealistischen Strukturen führt.

Nachdem vorab über geeignete Konfigurationen und Grundrisse diskutiert wurde, soll nun abschließend betrachtet werden, für welche Eingaben das Verfahren Grundrisse erzeugt, die für die Weiterverarbeitung nicht oder nur schlecht verwendbar sind. Allgemein ist festzuhalten, dass mit steigender Anzahl unterschiedlicher Labels automatisch auch die Regelmenge anwächst. Weiterhin wird auch der Sampleraum, in dem die zweite Phase des Verfahrens abläuft, zunehmend chaotischer und komplexer. Dies hängt mit dem grundsätzlichen Berechnungsvorgehen zusammen. Für jede Kantensteigung erzeugt das Verfahren eine Menge paralleler Strahlen. Je mehr unterschiedliche Kantensteigungen vorhanden sind, desto mehr parallele Kanten, Vertices und Faces werden erzeugt. Dadurch sinkt auch die Größe der einzelnen Faces, was zu potentiell stärker ausgefransten Strukturen führen kann.

Abbildung 92 zeigt einen Eingabegrundriss mit insgesamt 12 unterschiedlichen Kantensteigungen, die hauptsächlich durch den halbkreisförmigen Turm auf der rechten Grundrissseite zustande kommen.

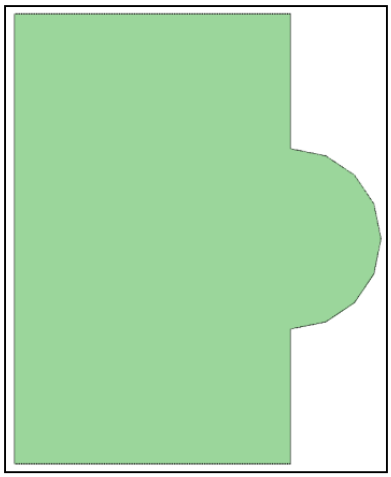

Abbildung 92: Problematischer Eingabegrundriss

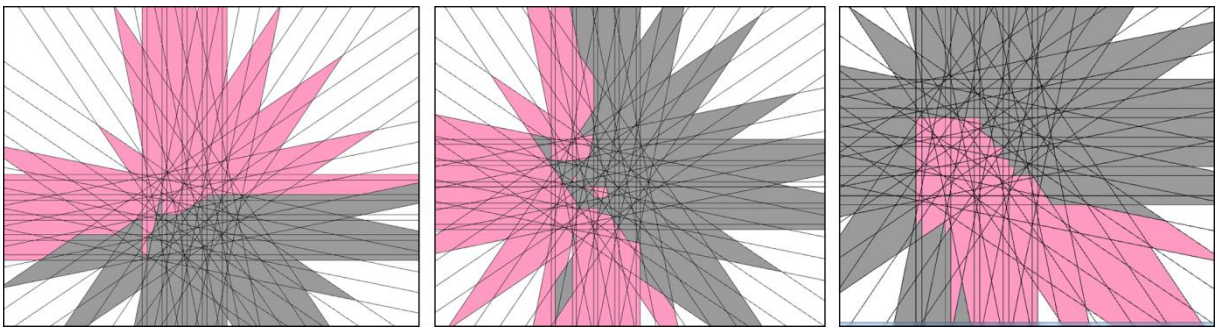

Abbildung 93: Ergebnis der Modelsynthese basierend auf dem komplexen Eingabegrundriss

Abbildung 93 enthält drei Beispiele für Syntheseergebnisse, die das Verfahren aufgrund des in Abbildung 92 gezeigten Eingabegrundrisses erzeugt. Gut zu erkennen ist zunächst die Tatsache, dass die errechneten Polygone, die die Grundlage der Syntheseberechnung darstellen, teilweise sehr klein und stark unterschiedlich geformt sind. Darüber hinaus erhöht sich mit der Anzahl unterschiedlicher Labels automatisch auch die Anzahl dieser Polygone. Die flächenmäßig kleineren Strukturen begünstigen die Ausfransung und erhöhen die Unregelmäßigkeit in der Berechnung. Zwar greift auch hier noch der Mechanismus des Kontinuitätskriteriums, trotzdem sind die Strukturen zu stark dem Zufall unterworfen. Strukturen, wie der halbkreisförmige Turm im Eingabegrundriss sind unter diesen Voraussetzungen nur schwer realisierbar, da sie einerseits geeignet geformte Polygone voraussetzen und andererseits darauf angewiesen sind, dass einer Menge aufeinanderfolgender Komponenten immer der jeweils passende Status zugewiesen wird. Hier reichen die genannten Mechanismen nicht aus, um das Ausgangschaos im Sampleraum

in eine kontrollierte Synthese zu überführen, die auch unter solchen Vorbedingungen in der Lage ist, sinnvolle Strukturen zu erzeugen.

9 Innenraumkonstruktion und Wanderzeugung

9.1 **Besonderheiten der Innenraumkonstruktion**

Ein System zur Erzeugung realistischer Gebäude muss die Möglichkeit bieten, Innenraumstrukturen zu beschreiben und diese zu berechnen. Für Anwendungsbereiche, in denen die äußere Fassade des Modells die zentrale Rolle spielt und die derart erstellten Gebäude nicht durch den Nutzer *virtuell betreten* werden, erscheint dies zunächst von untergeordneter Relevanz zu sein. Betrachtet man allerdings spezielle Gebäudetypen, beispielsweise griechische oder römische Tempelkonstruktionen, so stellt man fest, dass die Trennung zwischen Fassade und Innenraumstrukturen teilweise verschwimmt, zumindest für die Implementation eines Systems, das solche Gebäude konstruiert. Um auch bei der Konstruktion eines griechischen Tempels auf bestehende Komponenten zurückgreifen zu können, benötigt man Abstraktionen, die es ermöglichen, besondere Grundformen und –strukturen auf die Variation bereits bestehender Komponenten zurückzuführen.

Am Beispiel eines Tempels dorischer Ordnung sei diese Problematik kurz erörtert. Ein solcher Tempel wird in drei unterschiedliche Bereiche unterteilt, nämlich *Sockel*, *Säule* und *Gebälk*. Der Sockel besteht aus dem Fundament und den Treppenstufen, die hinauf zum Tempelbereich führen. Der eigentliche Tempelbereich ist umgeben von mindestens einer Säulenreihe und wird in verschiedene Unterbereiche unterteilt. Der Bereich vor dem Eingang wird als *Pronaos* bezeichnet. *Cella* benennt den eigentlichen Tempelbereich, der durch den Eingang betreten wird [SP04]. Der rückwärtige Bereich, als Gegenstück zum *Pronaos,* wird *Opisthodom* genannt. Für das System problematisch ist die Modellierung des Abschnitts zwischen Sockel und Gebälk. Räumlich handelt es sich um einen Quader, bei dem die Seitenflächen entfernt wurden. An ihrer Stelle befinden sich stattdessen Säulenreihen, die das Gebälk des Tempels tragen. Eine solche Konstruktion hat den Reiz, dass sie es aus Sicht des Semantic Building Modelers ermöglicht, einen Tempel basierend auf bestehenden Strukturen zu errechnen. Wie auch bei anderen Gebäuden arbeitet man zunächst mit einem rechteckigen Grundriss, der extrudiert wird, um dann das räumliche Modell zu bilden. Anschließend entfernt man die Seitenflächen des so erstellten Objekts und berechnet die Säulenpositionen an den Außenkanten.

Problematisch ist nun aber der Bereich der Cella. Prinzipiell gibt es hier zwei Möglichkeiten. In der ersten Variante modelliert man einen solchen Bereich als eigenständige Komponente, die über einen Grundriss beschrieben und als eigenständiges

Stockwerk in ein Gebäude eingefügt werden kann. Das Problem dieses Ansatzes lässt sich gut am Tempelbeispiel aufzeigen.

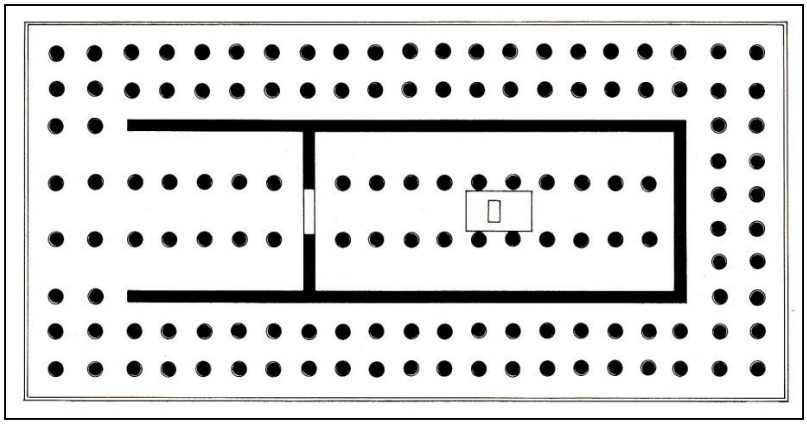

Abbildung 94: Grundriss eines Dipteros-Tempels [SP04]

Abbildung 94 zeigt den Grundriss eines *Dipteros*-Tempels. Die dicken schwarzen Linien beschreiben den Grundriss der Cella, die Kreise zeigen die Positionen von Säulen an. Problematisch am Cella-Grundriss ist die Tatsache, dass er nicht mehr durch einen geschlossenen Linienzug modelliert werden kann. Weiterhin enthält er Durchgänge zwischen einzelnen Abschnitten, die als Türen oder Portale modelliert werden. Diese Eigenschaften erschweren die Verwendung bestehender, extrusionsbasierter Strukturen, da mehrere Probleme gelöst werden müssen. Das größte Problem ist das bereits erwähnte Fehlen geschlossener Linienzüge für die Modellierung der Grundrissstrukturen. Man bräuchte demnach eine erweiterte Polygondatenstruktur, die in der Lage ist, solche Strukturen auszudrücken. Die hier gezeigten Schwierigkeiten gelten nicht nur für die Besonderheiten eines Dipteros-Tempels, sondern finden sich analog bei der Modellierung von Innenräumen im Allgemeinen wieder. Auch dort ist es nicht möglich, Wandverläufe durch geschlossene Linienzüge zu modellieren, es müssen Durchgänge etc. integrierbar sein. Aus diesem Grund wird die Cella eines Tempels analog zu Innenräumen durch eine neu eingeführte Komponente realisiert.

9.2 **Innenraumkonstruktion**

Um die vorab beschriebenen Schwierigkeiten zu lösen, teilt man die Grundrisse in mehrere Teile auf, die man auf unterschiedliche Arten verbinden kann. Dies erlaubt die Verwendung einfacher Grundbestandteile, wie beispielsweise rechteckiger Polygone, die beliebig

miteinander kombiniert werden können. Der in Abbildung 94 dargestellte Cella-Grundriss lässt sich auf diese Art durch die Kombination zweier Rechtecke darstellen, wobei das linke Rechteck auf der linken Seite offen und auf der rechten Seite durch einen Durchgang mit dem rechten Rechteck verbunden ist. Hierin finden sich bereits zwei Arten erforderlicher Verbindungselemente. Zum einen der Durchgang zwischen benachbarten Grundrisskomponenten, zum anderen eine offene Fläche. Die Konstruktion eines 3D-Modells für einen solchen Innenbereich erfolgt dann als zweistufiger Prozess. Zunächst extrudiert man jede Grundrisskomponente mit einer frei wählbaren Höhe. Anschließend verarbeitet man die Verbindungsstrukturen. Eine offene Fläche erfordert dabei nur das Entfernen einer Wandfläche, die bei der Extrusion erzeugt wurde, und stellt somit eine einfache Operation dar. Aufwendigere Berechnungen erfordert die Verbindung zweier Grundrisskomponenten. Hier stellt das System zwei unterschiedliche Arten der Verbindung zur Verfügung, die sich allerdings in den ersten Berechnungsschritten gleichen. Die erste Variante schneidet ein einfaches Loch in die Durchgangswand und verbindet darüber die benachbarten Räume. Bei der zweiten Variante verwendet man ein 3D-Modell, dessen Art bei der Definition des Verbindungstyps als zusätzliche Information festlegt wird. Bei beiden Ansätzen existiert zunächst das gleiche Grundproblem. Prinzipiell ist es möglich, dass die angrenzenden Räume unterschiedliche Ausdehnungen haben, was die Ausmaße der gemeinsamen Wand angeht. Dies gilt sowohl für die Länge als auch für die Höhe. Es ist darum erforderlich, die Wandpolygone gegeneinander zu beschneiden, um einen Durchgang zu erzeugen, der die Strukturen der Quellwände widerspiegelt. Dieser Schritt muss bei beiden Formen der Durchgangsberechnung durchgeführt werden, also sowohl für Durchgänge, bei denen Wände entfernt werden, um einen Durchgang zu formen als auch bei solchen, bei denen das 3D-Modell einer Tür oder eines Portals als Durchgang fungiert. Bei einem 3D-Modell verwendet man als Schnittschablone die konvexe Hülle des Modells nach einer Parallelprojektion auf die Zielwand und verwendet diese Schablone zur Beschneidung der anliegenden Wände. Dieses Verfahren erweist sich als robust und kann für alle Formen von Innenräumen eingesetzt werden, bei denen man anliegende Wandstrukturen umsetzen möchte. Dies gilt sowohl für griechische Tempel, als auch für Innenräume in modernen Gebäuden.

Wenn die Berechnung von Innenräumen und ihre Definition nun ein vergleichsweise aufwendiger Prozess ist, warum ist sie erforderlich? Beim Beispiel des Tempels ist dies offensichtlich. Da es keine Fassade im klassischen Sinne gibt, sondern der Innenraum von

Säulenzügen umgeben ist, muss ein Innenraum modelliert werden, da man sonst einen Tempel ohne Tempelbereich erstellen würde. Bei Gebäuden, die dagegen über eine Außenfassade verfügen, scheint dies nicht der Fall zu sein. Sofern der Nutzer nicht in der Lage ist, das Gebäude aus einer Innenansicht zu durchwandern, könnte man sich die aufwendige Berechnung von Innenräumen sparen. Betrachtet man allerdings ein erstelltes Gebäude ohne Innenstrukturen, so wirkt es vergleichsweise unrealistisch, selbst wenn die Außenstrukturen aufwendig konstruiert wurden. Diese Wahrnehmung entsteht dann, wenn der Nutzer über Fenster in der Fassade in den Innenraum sehen kann. Je nach Position des Betrachters und der Fenster ist dies nicht weiter relevant, schaut er beispielsweise in ein höher gelegenes Fenster, so sieht er von seinem Standpunkt nur die Decke des Raumes. Befindet sich das Fenster aber auf seiner Blickhöhe, so kann er durch das ganze Stockwerk hindurchblicken und sieht Fenster auf der ihm abgewandten Gebäudeseite. Dies widerspricht der typischen Wahrnehmung eines Gebäudes, da man normalerweise nicht von einer Seite durch das gesamte Gebäude schauen kann. Aus diesem Grund spielt die Modellierung von Innenräumen auch dann eine wichtige Rolle, wenn diese gar nicht betreten werden können. Allerdings kann die Berechnung in diesem Fall einfacher sein, da sie dem außenstehenden Betrachter nur die Existenz von Innenstrukturen suggerieren muss, ob diese dabei realistische Räume modellieren, ist für ihn in diesem Fall nicht entscheidbar.

9.3 **Erzeugung realistischer Wände**

Durch die Erstellung von Innenräumen mit beliebigen Grundrissformen entsteht ein weiteres Problem, das keine Rolle spielt, sofern man nur mit Gebäuden arbeitet, von denen ausschließlich die Außenfassade sichtbar ist. Sobald aber Innenraumstrukturen auftreten, wird deutlich, dass die bisherigen Gebäudekomponenten durch eine einfache Grundrissextrusion erstellt wurden. Basis der Extrusionsberechnung ist ein Grundrisspolygon, das in Richtung der Polygonnormalen bis auf seine Zielhöhe extrudiert wird. Die Seitenflächen des Körpers, die durch die Erzeugung der dritten Dimension entstehen, werden ebenfalls durch einfache Polygonzüge dargestellt, die nachfolgend weiterverarbeitet werden können. Das Problem dabei ist nun, dass solche Polygone per Definition keine Tiefe besitzen. Alle Punkte, die ein solches Polygon beschreiben, liegen in einer Ebene, die mathematisch betrachtet keine Dicke hat. Schaut man von einer Betrachtungsposition auf das Polygon, von der aus die Blickrichtung parallel zur Ebene verläuft, sieht man diese nicht. Dies ist so lange nicht relevant, wie man Hausfassaden durch

geschlossene Linienzüge modelliert. Sitzt man in einem geschlossenen Raum, so erkennt man ebenfalls nicht, dass die Wände des Raumes eine Dicke haben, dies fällt erst auf, wenn man von der Seite auf eine solche Wand schaut. Wiederum kann man die Problematik gut am Beispiel von Abbildung 94 erläutern. Modelliert man die linke Rechteckkomponente und verarbeitet anschließend die Verbindungsinformationen, so entfernt das System die linke Wand der linken Komponente. Dadurch ist der Linienzug nicht mehr länger geschlossen, sondern an einer Seite geöffnet, die fehlende Wanddicke wird sichtbar. Aus diesem Grund ist es erforderlich, Wände mit einer konfigurierbaren Dicke basierend auf den extrudierten Grundrissen zu berechnen.

Die Berechnung ist vergleichsweise einfach und erfolgt als zweistufiger Prozess. Zunächst arbeitet man auf dem Grundriss für die aktuelle Komponente. Bei Innenräumen werden alle Komponenten nacheinander verarbeitet, bei Außenfassaden, die auf einem einzigen Grundriss basieren, wird nur dieser in die Berechnung einbezogen. Nun durchläuft man den Kantenzug, der durch das jeweilige Polygon beschrieben wird. An jedem Eckpunkt berechnet man dessen Winkelhalbierende basierend auf den adjazenten Kanten. Skaliert mit der Wandbreite beschreibt die Winkelhalbierende dann einen Translationsvektor, mittels dessen man die Vertices im extrudierten Körper an der aktuellen Position verschieben kann. Da die Polygone selber geschlossene Linienzüge beschreiben, auch wenn die extrudierten Wände nach Abschluss der Verbindungsberechnungen eventuell Öffnungen enthalten können, kann man aufgrund dieser a priori nicht feststellen, ob man an eine *offene* Stelle im ansonsten geschlossenen Raum gelangt ist. Die Ermittlung muss darum auf interne Verwaltungsstrukturen zurückgreifen, um zu bestimmen, ob für die aktuell untersuchte Kante tatsächlich eine Wand im finalen Objekt existiert. Hierfür fragt man die Kantenverwaltung des Systems an und testet, ob die jeweilige Kante innerhalb dieser vorkommt und von anderen primitiven Objekten referenziert wird. Da der Innenraum in der Verarbeitungskette als eigenständiges Objekt verarbeitet wird und als solches eine eigene Kantenverwaltung besitzt, kann man über die Anzahl der Referenzierungen der Kante ermitteln, ob diese Teil eines Wandelements ist. Stellt man fest, dass die Wand in der Verbindungsverarbeitungsphase entfernt wurde, wird der Translationsvektor nicht mehr länger durch die Winkelhalbierende an diesem Punkt beschrieben, sondern verläuft senkrecht zur Quellwand. Würde man diese Unterscheidung nicht treffen, so würde die berechnete Wand nicht bündig mit der Quellwand verlaufen. Nachdem auf diesem Wege für alle Vertices im Komponentenpolygon Translationsvektoren bestimmt wurden, kann man

die zugehörigen Vertices in der eigentlichen Wand mittels dieser Vektoren verschieben und dadurch eine Innenwand erzeugen.

An offenen Wandstellen entsteht durch diese Berechnung die Situation, dass zwar eine Innenwand ermittelt wird, diese aber nicht mit der Quellwand durch ein Polygon verbunden wurde. Somit schaut man in den Zwischenraum zwischen zwei parallelen Wänden. Abbildung 95 zeigt diese Problematik als Darstellung eines Grundrisses mit erzeugten Wänden. An den offenen Wandseiten an der linken und rechten Grundrisskomponente erkennt man, dass zwar die Wände korrekt berechnet wurden, allerdings offene Stellen im resultierenden Grundriss entstehen.

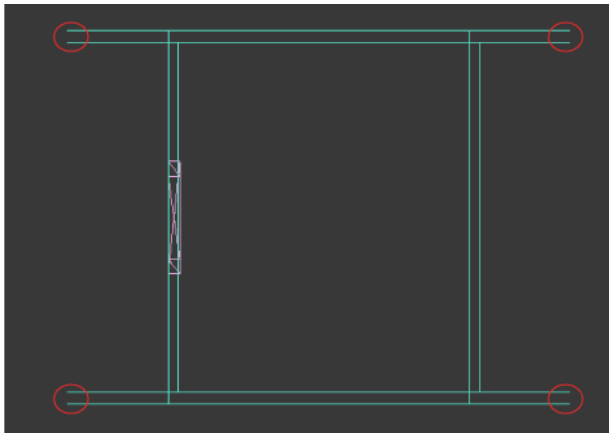

Abbildung 95: Problem der Innenwandberechnung bei offenen Wänden

Die Lösung dieses Problems basiert wiederum auf einem Algorithmus, der sich die Kantenverwaltungsstrukturen des Systems zunutze macht. Die Schwierigkeit besteht nämlich zunächst in der Ermittlung, bei welchen Wänden bzw. Eckpunkten dieser Fall auftritt. Auch hier sind solche Strukturen dadurch erkennbar, dass die berechneten Wandpolygone Kanten enthalten, die von nur einer einzigen Komponente referenziert werden. Bei geschlossenen Wandzügen ist dies dagegen nicht der Fall. Für solche Strukturen gilt, dass jede Kante einer Wand von mindestens zwei Komponenten referenziert wird, nämlich den beiden benachbarten Wandstrukturen. Somit bietet wiederum die Anzahl der Referenzierungen der Kanten ein eindeutiges Indiz, über das man in der Lage ist, potentiell problematische Strukturen zu identifizieren. Die eigentliche Berechnung der erforderlichen neuen Wandelemente ist dann vergleichsweise einfach, da man nur ein Polygon berechnen muss, das die Eckpunkte der parallelen Wandsegmente enthält. Das Ergebnis dieser Berechnungen ist in Abbildung 96 dargestellt.

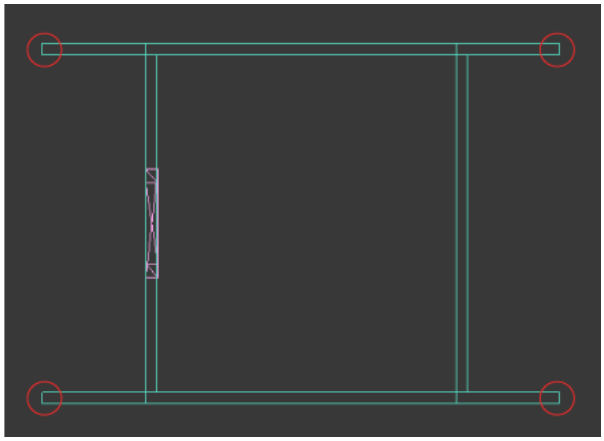

Abbildung 96: Lösung des Innenwandproblems

10 Verwaltung von externen 3D-Modellen

Die Verwaltung, das Laden und Positionieren von 3D-Modellen stellt eine zentrale Aufgabe des Semantic Building Modelers dar. 3D-Modelle sind ein wichtiger Faktor für Vielfalt und Abwechslung in den Modellen, die durch das System erzeugt werden. Weiterhin stellen sie eine Möglichkeit dar, das System durch die Bereitstellung neuer 3D-Modelle zu erweitern. Eingriffe in den Programmcode sind nicht erforderlich. Der Semantic Building Modeler ist in der Lage, aus den vorhandenen Komponentenmodellen zufallsbasiert zu wählen und diese anschließend in die Gebäudestrukturen zu integrieren. Der Umgang mit 3D-Modellen bezieht sich dabei aber nicht nur auf den Import einzelner Komponenten, sondern genauso auf den Export berechneter Gebäude. Ohne solche Exportstrukturen könnten die errechneten Modelle nicht weiterverarbeitet werden, was einer der zentralen Zielsetzungen bei der Entwicklung des Systems widersprechen würde. In den nachfolgenden Abschnitten werden darum die vorhandenen Strukturen für den Import und Export von 3D-Modellen thematisiert und erörtert.

10.1 **Das verwendete Dateiformat**

Das Dateiformat, das für die Entwicklung des Semantic Building Modelers zum Einsatz kommt, sollte verschiedene Anforderungen erfüllen. Zum einen sollte es lesbar sein, also nicht binär kodiert vorliegen. Dadurch lassen sich solche Dateien einfach mit gängigen Texteditoren öffnen und überprüfen. Dies spielt speziell für die Fehlersuche und –behebung eine wichtige Rolle. Weiterhin sollte es von möglichst vielen unterschiedlichen Modellierungsanwendungen unterstützt werden. Dies ist zum einen relevant, da die Importmodelle in solchen Softwareumgebungen erstellt werden, zum anderen, weil diese Werkzeuge aufgrund ihrer umfangreichen Funktionalität große Hilfen für das Debugging der erstellten Modelle darstellen. Dazu gehören umfangreiche Möglichkeiten zur visuellen Inspektion der exportierten Modelle, die es erlauben, fehlerhafte Strukturen innerhalb der Modelle zu entdecken. Darüber hinaus bieten diese Softwaresysteme auch Analyseverfahren an, die Modelle auf häufig auftretende Fehler untersuchen. Beispiele für solche Fehler sind falsch ausgerichtete Normalenvektoren, fehlerhafte Texturkoordinaten, mehrfach am gleichen Ort auftretende Vertices oder überlagerte Polygonflächen mit gleichen Tiefenwerten, die zu Effekten wie dem *z-Fighting* [BAH09] während des Renderings führen. Solche Softwareumgebungen stellen eine umfangreiche Werkzeugpalette zur Verfügung, die den Nutzer sowohl bei der Erzeugung von Modellen innerhalb der Systeme

unterstützt, aber auch die Gültigkeit geladener Modelle überprüfen und auf potentielle Fehler hinweisen kann. Typischerweise verwenden solche Modellierungsumgebungen proprietäre Dateiformate, die binär kodiert relativ kleine Ausgabedateien erzeugen. Für Autodesk 3ds Max wäre dies beispielsweise das `*.max`-Dateiformat. Neben der geringeren Dateigröße verbessert die Verwendung binärer Dateiformate typischerweise auch die Ladezeit der Modelle, was speziell für Echtzeitanwendungen von großer Bedeutung ist. Allerdings sind bei solchen Formaten die Spezifikationen meist nicht offen verfügbar, was die Entwicklung eigener *Loader* für Modelle deutlich erschwert. Abhilfe können hier Formate wie *Collada* schaffen. Hierbei handelt es sich um ein offenes, XML-basiertes Format zur Beschreibung von 3D-Modellen, das auf der *SIGGRAPH*-Konferenz 2004 vorgestellt wurde. Das Besondere an diesem neuen Format ist die Tatsache, dass seine Spezifikation aus der Zusammenarbeit einer Vielzahl von Firmen resultierte, darunter u.a. Entwickler von 3D-Modellierungsanwendungen wie *Alias* oder *Discreet*, aber auch große Spieleentwickler wie *Ubisoft* oder *Electronic Arts*. Diese Kooperation zeigt, dass das Dateiformat die Bedürfnisse von 3D-Spielen in besonderem Maße berücksichtigen soll. Das Format wurde in den letzten Jahren weiterentwickelt, auf der *SIGGRAPH* 2008 wurde die aktuelle Spezifikation 1.5 vorgestellt, die neue Features aus dem CAD- und GIS-Bereich in den Standard integriert. Dazu gehören unter anderem Möglichkeiten zur Unterstützung von Berechnungen im Bereich der inversen Kinematik, die Integration geographischer Informationen und vieles mehr [Le11].

Offensichtlich ist das Collada-Format ein sehr mächtiges Format, das eine Vielzahl unterschiedlicher Features im Bereich der 3D-Modellverwaltung unterstützt. Weiterhin ist es nicht binär kodiert, sondern in XML, also auch für den menschlichen Nutzer lesbar. Es scheint also der ideale Kandidat zu sein, leider erfüllte es aber eine Voraussetzung nicht. Um den Aufwand bei der Implementation des ersten Prototypen möglichst gering zu halten, wird bevorzugt auf vorhandene Bibliotheken zurückgegriffen, die für gängige Arbeitsschritte eingesetzt werden können. Ein klassischer Anwendungsbereich ist dabei auch das Laden von 3D-Modellen. Leider stellte sich heraus, dass für die Java-3D-Umgebung *Processing*[14], die für das Rendering innerhalb des Semantic Building Modelers eingesetzt wird, zwar ein Collada-Loader zur Verfügung steht, dieser allerdings zwei große Nachteile mit sich bringt. Zum einen ist er nur in der Lage, Collada-Modelle zu laden, die mit SketchUp erstellt wurden [En10]. Somit müssen Assets, die in das System importiert werden sollen, entweder

[14] Processing: http://processing.org/

direkt in SketchUp erstellt oder zunächst in dieses importiert und anschließend von dort wieder exportiert werden. Beide Varianten erscheinen wenig zielführend. Zum einen ist die Festlegung auf ein einzelnes Modellierungsprogramm unerwünscht, gleiches gilt für die Verwendung einer Modellierungssoftware als reines Konvertierungswerkzeug zwischen unterschiedlichen Dateiformaten. Das zweite große Problem ist die Tatsache, dass die vorhandene Bibliothek nur als Importwerkzeug arbeiten kann, aber nicht in der Lage ist, innerhalb des Systems erstellte Modelle in das Collada-Format zu exportieren. Zwar ist es möglich, einen Exporter für das Format zu entwickeln, da die Spezifikationen frei zugänglich sind, allerdings sollte der hierfür notwendige Zeitaufwand vermieden werden.

Aus diesem Grund fiel die Wahl auf das *Wavefront Object Dateiformat*. Dieses ist ein „UNIX-basiertes 3D-Vektor-Dateiformat der Software *Wavefront Advanced Visualizer*“ [Ku10]. Das *Object-Format* wird als ASCII-Text gespeichert und besitzt eine vergleichsweise einfache Struktur. Es ermöglicht „das Speichern von polygonalen Objekten, die durch Koordinatenpunkte, Kanten und Oberflächen repräsentiert werden sowie von frei geformten Objekten wie Kurven und konvexen oder konkaven Flächen“ [Ku10]. In der einfachsten Variante schreibt man alle Vertices eines erstellten 3D-Modells in eine Textdatei, gleiches gilt für die Normalen und Texturkoordinaten. Anschließend definiert man Polygone, die die Punkte, Normalen und Texturkoordinaten über Indices in der Datei referenzieren. Diese simple Form der Speicherung reicht aus, um die Geometrie statischer 3D-Modelle zu beschreiben. Ein Nachteil des Formats, der allerdings generell für nicht-binäre Formate gilt, ist die Dateigröße, die speziell bei sehr komplexen Objekten sehr umfangreich werden kann und um ein Vielfaches höher liegt, als bei vergleichbaren binärkodierten Dateien. Dieser Nachteil wird allerdings durch verschiedene Vorteile aufgewogen. So handelt es sich bei dem Format um einen gängigen Industriestandard, der von einer Vielzahl von Modellierungswerkzeugen, Game-Engines und anderen Systemen unterstützt wird. Er ist aufgrund der ASCII-Kodierung menschenlesbar, was die Fehlersuche und –behebung deutlich vereinfacht. Außerdem existieren verschiedene Bibliotheken, die für Import und Export in die Processing-Umgebung eingesetzt werden können. Zukünftige Versionen des Semantic Building Modelers können die vorhandenen Schnittstellen des Systems problemlos um die Unterstützung für weitere Dateiformate erweitern. Für die erste Version reichen die Möglichkeiten des Object-Formates allerdings vollkommen aus.

10.2 **Organisation der 3D-Modelle in Kategorien**

Die zentrale Aufgabe des vorliegenden Systems ist die Erzeugung realistischer Gebäude basierend auf nutzerkonfigurierbaren Parametern. Dabei soll das System derart strukturiert sein, dass es keine fest verdrahteten Gebäude erstellt, also rein deterministisch Gebäudepläne abarbeitet und als Ergebnis ein fertiges Haus abliefert. Dies würde dazu führen, dass bei gleichen Parametern immer das gleiche Gebäude entsteht. Um eine höhere Vielfalt zu erzeugen, müssten dann viele verschiedene Konfigurationen durch den Nutzer erstellt werden, dies soll aber nicht der Fall sein, um den Nutzeraufwand auf ein Minimum zu reduzieren. Somit muss das System im Bereich der vorgegebenen Parameter zufallsbasierte Schwankungen einbauen, durch die eine größere Vielfalt erzeugt werden kann. Dies gilt insbesondere auch für die verwendeten 3D-Modelle, die importiert und in den Gebäuden positioniert werden. Sofern der Semantic Building Modeler in der Lage ist, aus einem Pool von Modellen einer bestimmten Kategorie zufallsbasiert auszuwählen, entstehen Variationen in den fertigen Gebäuden durch unterschiedliche Modelle für Türen, Fenster und andere Komponenten. Erweitert man den Pool der vorhandenen Modelle, so besteht auch für die Berechnung eine größere Auswahl, die sich wiederum in einer größeren Vielfalt manifestiert.

Die Einteilung aller vorhandenen 3D-Modelle in Kategorien und die zufallsbasierte Auswahl von Modellen aus diesen ist somit ein wichtiges Feature des Systems. Offensichtlich ist die Definition der Kategorien für die Modelle ein wichtiger Faktor für die Realitätsnähe der erstellten Gebäude. Beispielsweise würde man für ein Jugendstilgebäude andere Fenstermodelle verwenden, als für ein mittelalterliches Gebäude oder ein modernes Hochhaus. Die Kategorien müssen also derart strukturiert sein, dass sie bestimmte architektonische Stile abbilden und Modelle enthalten, die zu diesen Stilen passen. Weiterhin muss das Kategoriensystem erweiterbar sein, um auch nachträglich neue Modelle integrieren zu können.

Die Implementation eines solchen Ansatzes ist einfach, die Zuordnung von Modellen zu Modellkategorien erfolgt über den Dateinamen des Modells. Ändert man diesen, so wird sich automatisch auch die Zuordnung ändern und das Modell wird künftig für andere Gebäudetypen eingesetzt. Zu Beginn der Verarbeitung geht der Semantic Building Modeler so vor, dass er das Modellverzeichnis analysiert und Statistiken über die vorhandenen Modelle in den jeweiligen Kategorien anfertigt. Fordert nun eine Komponente des Systems ein Modell einer bestimmten Kategorie an, so ermittelt die Modellverwaltungskomponente

zufallsbasiert ein Modell innerhalb der Kategorie, lädt dieses und liefert das 3D-Modell an den Aufrufer zurück. Das Laden der Modelle variiert dabei in Abhängigkeit vom Dateiformat des Modells, je nach Dateityp wird ein anderer Loader genutzt. Wie vorab erläutert, unterstützt der Semantic Building Modeler in seiner aktuellen Ausbaustufe nur 3D-Modelle im Object-Format.

Das Laden der Modelle selber ist dabei ein Prozess, bei dem die Geometriedaten der vorhandenen Objekte zunächst aus den Quelldateien extrahiert und anschließend in das interne Format des Prototypen überführt werden. Dann erfolgt bei Bedarf eine erneute Tesselation der geladenen Strukturen. Nach Abschluss dieser Berechnungen steht das Modell innerhalb des Systems zur Verfügung und kann weiterverarbeitet werden.

10.3 **Integration externer 3D-Modelle**

10.3.1 **Anforderungen an Importmodelle**

Die Auswahl der 3D-Modelle basierend auf den Modellkategorien und das Laden der Modelle stellen nur einen ersten Schritt in einer Folge von Berechnungen dar, die zur Integration durchgeführt werden müssen. Die schwierigste Operation besteht in der Positionsberechnung. Dazu gehören verschiedene geometrische Operationen auf den Modelldaten wie die Größenanpassung durch Skalierung und die Ausrichtungsberechnung durch Rotationen der Vertices und Polygone. Damit diese Berechnungen durchgeführt werden können, müssen die Modelle bezüglich ihrer Ausrichtung immer derart erstellt werden, dass der Normalenvektor der Vorderseite in Richtung der positiven z-Achse zeigt. Ursächlich für diese Festlegung ist die Natur der 3D-Geometriedaten. Innerhalb des Systems werden diese als Menge von Vertices verwaltet, die über Kanten verbundene Oberflächenstrukturen bilden. Das Problem besteht darin, dass ohne Vorwissen über die Modelle keinerlei semantische Informationen über die Objekte existiert, die durch die Polygonnetzmodelle beschrieben werden. Zentral für die Positionierung eines Fensters oder einer Tür ist aber beispielsweise das Wissen darüber, welche Seite *vorne* ist. Ohne diese Kenntnis kann das System nicht entscheiden, wie ein Objekt an einer Gebäudefassade ausgerichtet werden muss. Weiterhin ist die Festlegung einer Standardausrichtung notwendig, um die Ausmaße des Modells ermitteln zu können und korrekte Skalierungen durchzuführen. Dafür muss sich das System darauf verlassen können, dass beispielsweise die Länge eines Objekts in seinem lokalen Koordinatensystem immer der Ausdehnung des Objekts in Bezug auf die x-Achse entspricht, gleiches gilt für Höhe und Breite. Leider ist es

nicht möglich, diese Informationen beim Export der Modelle aus den Modellierungsumgebungen direkt mit in die Exportdateien zu schreiben. Alternativ könnte man für jedes Modell eine kleine Konfigurationsdatei mitliefern, die die relevanten Dimensionen und Ausrichtungen enthält, dies erscheint aber impraktikabel, speziell im Hinblick auf potentiell große Modellbibliotheken. Aus diesem Grund müssen alle Objekte, die in das System geladen und von diesem verwendet werden, eine Ausrichtung besitzen, bei der die Frontseite des Objekts eine Normale besitzt, die in Richtung der positiven z-Achse zeigt, also einen Normalenvektor von $\vec{n} = (0,0,1)$ besitzt. Sofern diese Forderung erfüllt ist, kann das System die Modelle korrekt skalieren, rotieren und positionieren.

Die Positionsberechnungen selber variieren in Abhängigkeit vom Typ des geladenen Objekts, für Türen benötigt man naturgemäß andere Steuerparameter als für Fenster. Um die Anzahl der speziellen Positionierungsroutinen möglichst klein zu halten, implementiert das System vergleichsweise allgemeine Berechnungsmethoden, die über Parameterlisten gesteuert werden. Gängige Parameter betreffen dabei beispielsweise Ausdehnungsverhältnisse zwischen positioniertem und Zielobjekt. Weiterhin spielen Abstandsverhältnisse sowohl zwischen Objekten als auch Wandkanten und –ecken eine wichtige Rolle bei den Positionsberechnungen.

Um auch bei der Positionierung selber eine größere Variabilität zu erreichen, werden viele Positionsparameter als Bereiche definiert, innerhalb derer das System zufallsbasiert einen Zielwert bestimmt. Außerdem wird meist mit relativen statt absoluten Größenangaben gearbeitet, um die Flexibilität der Konfigurationen zu erhalten. Dadurch ist es möglich, die gleiche Konfiguration für eine Wand mit einer Länge von 100 Metern zu verwenden wie für eine Wand, die nur 10 Meter lang ist.

10.3.2 **Überschneidungen mit anderen Objekten**

Nachdem die Skalierungen, Rotationen und Positionierungen basierend auf den Konfigurationsparametern durchgeführt wurden, erfolgt im letzten Schritt die Entscheidung, ob das Objekt tatsächlich zum Gebäude hinzugefügt oder verworfen wird. Das System muss feststellen, ob an der jeweiligen Position bereits eine andere Komponente positioniert wurde. Diese Berechnung ist nicht trivial, speziell aus Performancesicht kann dies sehr rechenintensiv werden. Prinzipiell kann man verschiedene Wege gehen, um dieses Problem zu lösen. Ein *Brute-Force-Ansatz* könnte dabei so aussehen, dass man das aktuell zu

positionierende Objekt als Referenzobjekt verwendet. Nun durchläuft man alle Kanten dieses Objektes und testet, ob diese die Oberflächen eines beliebigen anderen Objekts innerhalb des Gebäudes durchstoßen. Bei n Objekten mit durchschnittlich k Kanten und durchschnittlich m Oberflächen pro Objekt käme man somit auf $O(k * n * m)$ Schnittpunkttests. Aufgrund der bei komplexen Modellen potentiell großen Werte für k und m ist diese Berechnung sehr zeitaufwendig. Aus diesem Grund wird hier ein anderer Algorithmus verwendet, der nicht auf Basis der Objekte arbeitet. Zunächst geht man so vor, dass man sämtliche Vertices des positionierten Objekts mittels einer Parallelprojektion auf die Wand projiziert, an der sich das Objekt nach Abschluss der Berechnungen befinden soll. Für die derart entstandene Punktwolke berechnet man nun mittels des Graham-Scan-Algorithmus ihre konvexe Hülle. Diese beschreibt genau die Umrisslinie des Objekts in Bezug auf die Zielwand. Das derart berechnete Polygon enthält typischerweise deutlich weniger Punkte als das ursprüngliche 3D-Objekt. Ein solches Polygon wird für alle an der jeweiligen Wand positionierten Objekte berechnet und als Loch gespeichert. Dadurch kann man Überschneidungen zwischen dem aktuellen Polygon und den Polygonen vorab positionierter Objekte berechnen. Sobald ein Schnitt gefunden wird, weiß man, dass an der Zielposition bereits ein anderes Objekt vorhanden ist, so dass das aktuelle Objekt verworfen werden muss.

10.3.3 **Beschneidung der Wandfläche**

Das Umrisspolygon des projizierten Objekts dient dabei nicht nur der Überprüfung auf Überschneidungen mit bereits gesetzten Objekten, sondern wird auch für die Beschneidung des Wandpolygons verwendet. Ziel dieses Vorgangs ist das Entfernen von Wandflächen in solchen Bereichen, in denen ein 3D-Objekt positioniert wurde und die somit nicht mehr benötigt werden. Die Beschneidung selber wird durch die Tesselator-Komponente durchgeführt, die bereits im Abschnitt „Das Tesselator-Submodul“ als vorgestellt wurde Als Eingabe erwartet der Tesselator beliebige Polygone, über die sowohl der Polygonumriss als auch eventuell vorhandene Löcher angegeben werden. Die Unterscheidung, ob ein Eingabepolygon ein Loch oder das eigentliche Umrisspolygon beschreibt, wird dabei anhand der Anordnung der Vertices durch die Komponente durchgeführt. Sind die Vertices im Uhrzeigersinn angeordnet, so definieren sie eine Umrisskante, sind sie entgegen dem Uhrzeigersinn orientiert, so beschreiben sie ein Loch innerhalb eines sie umgebenden Linienzuges. Die Ergebnisse der Tesselationsalgorithmen der GLUT-Bibliothek werden

über bereitgestellte *Callback-Methoden* ausgewertet und in die systeminterne Darstellung umgewandelt. Bei dieser Berechnung werden beispielsweise aus Triangle-Strip- oder Triangle-Fan-Strukturen automatisch einfache Dreiecke erzeugt. Die Berechnung fügt die derart erzeugten Dreieckskanten automatisch zu den Kantenverwaltungsstrukturen des Elternobjekts hinzu, aktualisiert Referenzzähler und -objekte und sorgt dadurch für eine valide Geometrie des erzeugten Objekts. Durch die Integration des Tesselators ist das System in der Lage, beliebige Eingabestrukturen zu unterteilen und diese intern zu repräsentieren. Dies gilt sowohl für geladene 3D-Modelle, als auch für prozedural erzeugte Objekte, wie beispielsweise Gebäudeumrisse.

10.3.4 **Probleme des Beschneidungsverfahrens**

Die Tesselator-basierten Beschneidungsberechnungen sind dabei grundsätzlich robust, speziell bei gleichmäßig geformten Objekten wie Fenstern oder Türen funktionieren sie fehlerfrei. Allerdings sind aufgrund des projektionsbasierten Beschneidungsalgorithmus Szenarien denkbar, bei denen Flächen aus den Wänden ausgeschnitten werden, die größer sind, als das eingefügte 3D-Objekt an der jeweiligen Position. Problematisch wäre beispielsweise ein Kegel, dessen Höhenachse dem Normalenvektor der Zielwand entspricht und der zu einem Teil in die Wand verschoben wird. Konkret könnte man sich die Wand dann als zum Kegel senkrechte Schnittebene vorstellen, die den Kegel in zwei Teile schneidet. Projiziert man nun alle Vertices des Kegels auf diese Zielwand, so bildet die breite Bodenfläche die konvexe Hülle der projizierten Vertices. Somit schneidet das System ein Polygon aus der Wandfläche aus, das mit dem Bodenpolygon des Kegels identisch ist. Da dieser aber nicht ganz in die Wand verschoben wird, ist die ausgeschnittene Fläche an der Wand größer als der Kegel an dieser Stelle. Es entsteht ein Loch, das durch das 3D-Objekt nicht gefüllt wird, was zu Fehlern in der berechneten Geometrie führt.

Allerdings muss festgehalten werden, dass es sich hierbei um ein hypothetisches Beispiel handelt, da Formen dieser Art nicht in Häuserwänden vorkommen, egal aus welcher Epoche diese stammen. Die tatsächlich verwendeten Objekte zeichnen sich typischerweise dadurch aus, dass die Projektion ihrer Vorder- und Rückseite auf Umrisspolygone ähnlichen Ausmaßes abbildet. Dadurch entsprechen auch die berechneten konvexen Hüllen den Ausdehnungen der positionierten Objekte, so dass das beschriebene Problem in der Praxis keine Rolle spielt.

10.4 **Erzeugung von Gesimsstrukturen**

Gesimse stellen in diesem Abschnitt eine Sonderform geladener 3D-Objekte dar und sollen darum in ihrer Verwendung detaillierter thematisiert werden. Ihre Sonderrolle resultiert in der Verwaltung durch das System daraus, dass sie im Gegensatz zu anderen 3D-Objekten noch nicht „fertig“ sind, wenn sie geladen werden. Das Modell einer Tür oder eines Fensters ist beim Laden in das System bereits vollständig modelliert. Modelle dieser Art müssen skaliert, rotiert und positioniert werden, die geometrischen Strukturen sind vollständig vorgegeben. Gesimse unterscheiden sich von solchen Modellen, da ihre Ausdehnung mit den Strukturen variiert, an denen sie positioniert werden. Betrachtet man beispielsweise ein Gurt- oder Sockelgesims, so handelt es sich um Strukturen, die eine Fassade oder ein ganzes Gebäude vollständig umgeben. Somit hängt die Ausdehnung des Objekts vollständig von den Strukturen ab, an denen das Gesims angebracht werden soll. Dies führt dazu, dass der einfache Ansatz basierend auf dem Laden fertiger 3D-Modelle mit anschließender uniformer Skalierung im Gesimskontext nicht anwendbar ist. Speziell bei sehr langen Fassaden würde ein uniformer Skalierungsansatz dazu führen, dass das 3D-Modell nicht länger die Proportionen eines Gesimses besitzt, sondern mit wachsender Länge auch in den anderen Dimensionen größer wird. Für Gesimse braucht man aber ein Verhalten, bei dem nur die Längendimension variabel ist, die verbleibenden Dimensionen dagegen konstant. Ein ähnlicher Ansatz wird bereits innerhalb des Systems verwendet, um Gebäudestrukturen selber zu erzeugen. Man verwendet eine Extrusion, bei der man ein initiales Grundrisspolygon in Richtung seines Normalenvektors verschiebt und dabei Seitenstrukturen erzeugt. Konzeptuell gleicht der hier verfolgte Gesimsansatz diesem Vorgehen, nur dass die Extrusionsachse nicht senkrecht zum Grundriss des Gebäudes verläuft, sondern entlang der Fassade, an der das Gesims positioniert wird. Durch eine Änderung des Bezugskoordinatensystems ist man in der Lage, die Extrusionsberechnungen auch für die Erzeugung von Gesimsstrukturen zu verwenden.

10.4.1 **Anforderungen an Gesimsprofile**

Gesimsprofile werden wie alle anderen 3D-Modelle importiert, sie können demnach in einer beliebigen Modellierungsumgebung erstellt und anschließend in ein Format exportiert werden, das vom Semantic Building Modeler unterstützt wird. Solche Profile besitzen im Unterschied zu anderen Modellen keine Tiefe, sie sind also zunächst zweidimensional. Erst

durch die Extrusionsberechnungen entstehen dreidimensionale Strukturen. Für Profile dieser Art gilt ebenfalls die vorab erwähnte Voraussetzung, dass der Normalenvektor ihrer Vorderseite in Richtung der positiven z-Achse zeigen muss. Zusätzlich müssen Profile aber noch eine weitere Anforderung erfüllen, um an Wänden angebracht werden zu können. Bei Fenstern oder anderen Objekten richtet man die Objekte derart aus, dass der Normalenvektor der Vorderseite parallel zum Normalenvektor der Wand verläuft. Da man aufgrund der Festlegungen weiß, dass die Vorderseite immer in Richtung der positiven z-Achse verläuft, kann man den Winkel zwischen diesem Vektor und der Wandnormalen errechnen. Anschließend rotiert man das Objekt um diesen Winkel, wodurch dieses mit der Wand zusammenfällt. Dies ist bei Profilen nicht der Fall, da diese in einem 90°-Winkel zur Wand angebracht werden müssen, um anschließend in Richtung des Wandverlaufs extrudiert werden zu können. Wiederum weiß man, dass der Vektor der Vorderseite in Richtung der z-Achse verläuft und kann dadurch einen Rotationswinkel bestimmen. Allerdings muss man in diese Berechnung den 90°-Winkel integrieren, damit das Profil senkrecht zur Wand steht. Bei der nachfolgenden Rotation kann nun der Fall auftreten, dass die Normalenvektoren von Wand und Profil zwar einen 90°-Winkel bilden, das Profil aber mit der falschen Kante an der Wand anliegt. Um solche Ausrichtungsfehler erkennen zu können, benötigt man zusätzliche Festlegungen für die Struktur des Profils. Diese Festlegung besteht darin, dass die Kante des Profils, die später an der Wand anliegt, die Vertices mit den größten x-Koordinaten verbindet. In Kombination mit der Forderung der standardisierten Ausrichtung der Profilvorderseite resultiert daraus, dass man das Profil in der xy-Ebene modelliert und dabei darauf achtet, die anliegende Kante auf der rechten Seite anzubringen. Abbildung 97 zeigt ein korrekt modelliertes Profil innerhalb seines lokalen Koordinatensystems. Dabei ist die rechte Kante, die durch die beiden Eckpunkte mit maximalen x-Koordinaten gebildet wird, diejenige, die innerhalb des Semantic Building Modelers an der Wand ausgerichtet wird. Durch die Berücksichtigung dieser Vorbedingungen ist es möglich, beliebige Profilarten an Gebäudewänden auszurichten und anschließend entlang der Wandkante zu extrudieren.

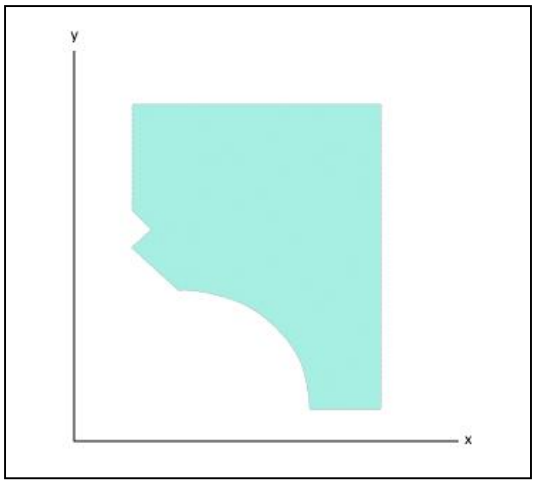

Abbildung 97: Profil in seinem lokalen Koordinatensystem

Das Ergebnis der Extrusion des in Abbildung 97 dargestellten Profils ist in Abbildung 98 zu sehen. Extrusionsberechnungen dieser Art stellen die einfachere Variante der Gesimsberechnung dar. Solche Strukturen werden verwendet, wenn das Gesims bündig mit Hauswänden abschließt und somit nicht über die Ecken des Hauses hinausläuft. Dieser Fall tritt beispielsweise dann auf, wenn ein Gesims nur an der Frontfassade angebracht wird. Wird es dagegen an allen Wänden des Hauses angebracht, so muss die Berechnung angepasst werden, um an den Ecken korrekte Verbindungen zu erzeugen.

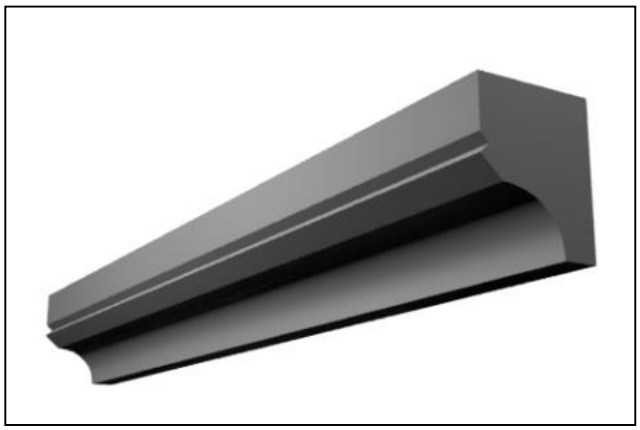

Abbildung 98: Extrudiertes Gesimsprofil

10.4.2 **Berechnung von Eckelementen für Gesimse**

Der erste Ansatz zum Lösen des *Eckproblems* könnte in einer Vergrößerung der Extrusionslänge über die Ecken hinaus bestehen. Dieser sehr einfache Weg führt aber zu Fehlern, da sich die Gesimse an den Ecken überlappen. Dies ist in Abbildung 99 deutlich zu

erkennen. Lässt man alternativ nur an einer Wand das Gesims über die Ecke hinauslaufen, so entstehen wiederum ungültige Strukturen. Man benötigt demnach ein geschickteres Verfahren, um die Gesimse an den Ecken derart zu modifizieren, dass sie bündig aneinander anliegen und dadurch realistische Eckverbindungen entstehen. Die Berechnung solcher Eckverbindungen ist ein mehrstufiger Prozess. Zunächst erzeugt man ein Gesims entlang der Wandkante, dessen Länge der Wand entspricht. Die Verbindung wird dann in einem zweiten Schritt erstellt. Dafür verwendet man das Abschlusspolygon des berechneten Gesimses und extrudiert dieses erneut in Richtung der Ecke. Die Extrusionslänge errechnet sich dabei aus der Länge des importierten Profils und dem Winkel zwischen den Normalenvektoren der Wände, die an der jeweiligen Ecke zusammentreffen. Dieser Winkel wird auch verwendet, um eine gedrehte Winkelhalbierende an der Hausecke zu berechnen.

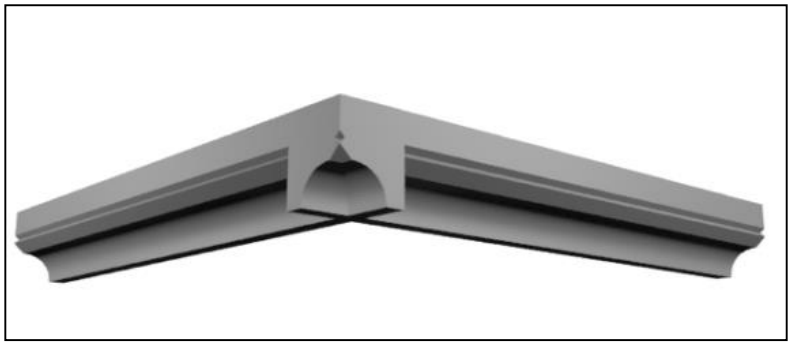

Abbildung 99: Fehlerhafte Gesimsstrukturen

Diese Winkelhalbierende ist definiert als der Vektor, der von der Ecke nach außen zeigt. Nun bestimmt man zunächst die Seite des neu erzeugten Eckelements, die in der Ebene liegt, die die Zielwand vollständig enthält und berechnet eine gemeinsame Kante der anliegenden Seite mit dem Abschlusspolygon des Eckelements. Diese Kante verschiebt man in Richtung der Wandnormalen, also senkrecht vom Gebäude weg um die Breite des Gesimsprofils. Dies bildet eine Ebene, die die vorab berechnete Winkelhalbierende vollständig enthält und an der nach Abschluss der Berechnungen die adjazenten Gesimse bündig aneinander anliegen. An diesem Punkt ist bereits die Zielebene erzeugt, allerdings sind bisher nur die beiden Vertices verschoben worden, die die gemeinsame Kante zwischen Wand- und Abschlusspolygon definiert haben. Alle weiteren Vertices, die die Form des Abschlusspolygons definieren, liegen weiterhin in dessen Quellebene, so dass die Verbindungskanten mit dem gegenüberliegenden Abschlusspolygon die neu erzeugte Ebene durchstoßen. Der letzte Schritt besteht darum darin, sämtliche verbleibenden Vertices des

Abschlusspolygons auf die neu erzeugte Ebene zu projizieren. Dabei verwendet man eine Projektion, bei der die Vertices senkrecht zur Quellebene auf die Zielebene projiziert werden. Dadurch entstehen perspektivische Verzerrungen, die für das realistische Aussehen der entstehenden Gesimse erforderlich sind. Der Grund hierfür besteht darin, dass die Länge der gedrehten Winkelhalbierenden größer ist als die Breite des Profils. Wäre dies nicht der Fall, so würde die Gesimskante zur Ecke hin einen Knick enthalten.

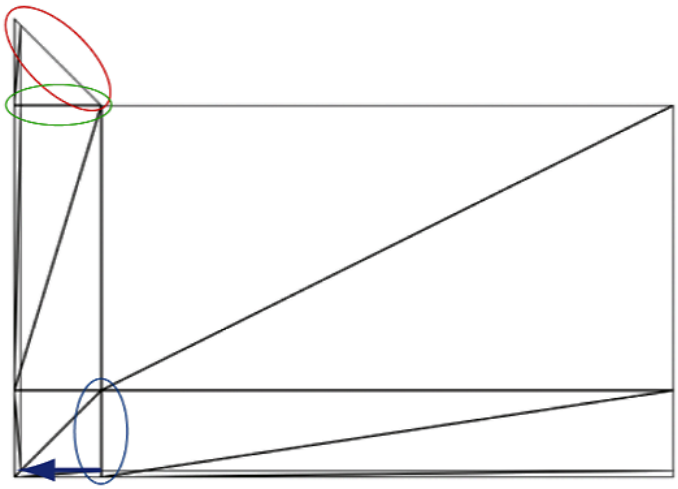

Abbildung 100: Berechnung der Verbindungselemente

Abbildung 100 zeigt den rechteckigen Grundriss eines Hauses, an dessen Wänden Gesimse appliziert wurden. Diese Komponenten wurden für die Erstellung der Abbildung zum Teil entfernt, um die relevanten Strukturen hervorheben zu können. Die gedrehte Winkelhalbierende ist in der Abbildung durch einen roten Kreis markiert. Da sie die Hypotenuse eines rechtwinkligen Dreiecks definiert, ist sie länger als die Profillänge. In Abbildung 97 wird diese Länge bestimmt durch die Länge der Kante, die die Vertices mit den maximalen y-Werten miteinander verbindet. Diese Kante verläuft typischerweise senkrecht zu der Kante, die bei der Positionierung des Gesimses an der Wand an dieser anliegt (in der Abbildung durch einen grünen Kreis markiert). An der Ecke selber muss das Profil um die Eckkante gedreht werden, damit beide anliegenden Gesimse bündig aneinander anschließen. Die Umsetzung dieser Drehung wird durch das oben beschriebene Verfahren realisiert. Dabei wird das Profil skaliert, da innerhalb der Profilkante kein Bruch entstehen darf. Faktisch wird durch die Rotation aus der grün umkreisten Kante die rot umkreiste Kante errechnet, die nach obiger Konstruktion exakt auf der oben erläuterten rotierten Winkelhalbierenden liegt. In Abbildung 100 stellt die rote Kante somit eine

Draufsicht auf die Ebene dar, die durch das beschriebene Einklappen der gemeinsamen Kante von Wand- und Anschlusspolygon entsteht. Die Seite, die für die Eckelementbildung eingeklappt wird, ist in der Draufsicht blau eingekreist. Dabei handelt es sich um eine Seitenfläche, die vollständig in der Wandebene enthalten ist. Die Vertices dieser Fläche, die eingeklappt werden, werden durch die Bestimmung der gemeinsamen Kante der Fläche mit dem Abschlusspolygon des neu gebildeten Eckelements ermittelt. Beim Einklappen selber werden die Vertices der gemeinsamen Kante in Richtung der Wandnormalen verschoben, in der Skizze wird dies durch den blauen Pfeil dargestellt. Dadurch erhält man die im oberen Teil der Abbildung sichtbaren abgeschrägten Elemente, an die nachfolgend ein weiteres Gesims anschließen kann. Abbildung 101 zeigt das Rendering eines Gurtgesimses, das innerhalb des Prototyps für ein Gebäude mit rechteckigem Grundriss durch das erläuterte Verfahren errechnet und anschließend gerendert wurde.

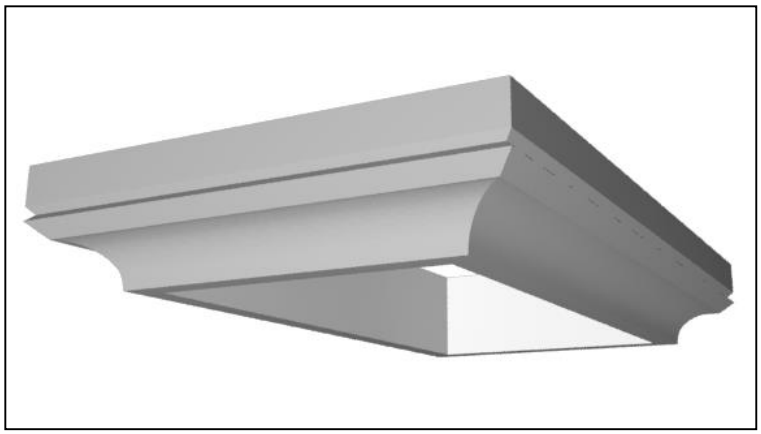

Abbildung 101: Gerendertes Gurtgesims

11 Dacherzeugung – der Weighted-Straight-Skeleton-Algorithmus

Die Berechnung realistischer Dachkonstruktionen ist für ein System zur prozeduralen Gebäudegenerierung ein wichtiger Schritt in der Verarbeitungslogik. Die Bedeutung der Berechnungskomponente für Dächer kann man sich leicht verdeutlichen, indem man sich überlegt, welche Aspekte eines 3D-Modells einen Betrachter davon überzeugen, dass es sich um ein Hausmodell handelt. Betrachtet man den einfachsten Fall eines Quaders beliebiger Ausdehnungen, so wird erst durch das Dach ersichtlich, was dieses Objekt darstellt. Ein Verfahren zur Bestimmung solcher Dachformen muss in der Lage sein, Dächer für unterschiedliche Arten von Grundrissen zu bestimmen. Das bekannteste und am weitesten verbreitete Verfahren ist der Straight-Skeleton-Algorithmus, der in seiner ursprünglichen Variante 1995 von Aichholzer et al. vorgestellt wurde [Ai95].

Das Verfahren ist verwandt mit dem Mittelachsentransformationsalgorithmus, der ebenfalls für Polygone beliebiger Struktur ein topologisches Skelett bestimmt und vorab bereits kurz thematisiert wurde (s. Abschnitt „Erzeugung von Ornamenten“ in ProcMod). Allerdings verwendet dieses Verfahren bei konkaven Polygonen Kurvensegmente zur Beschreibung des topologischen Skeletts, was für die Verwendung zur Dachbestimmung ungeeignet ist. Hier hat der Straight-Skeleton-Algorithmus den Vorteil, dass unabhängig von der Form des Eingabegrundrisses das entstehende Skelett ausschließlich aus geradlinigen Segmenten besteht. Da das so berechnete Skelett Grundlage der abschließend zu berechnenden Dachelemente ist, eignet sich dieses Verfahren sehr gut für die Dachegenerierung.

Anschaulich geht der Algorithmus so vor, dass er das Polygon schrittweise in sich zusammenschrumpfen lässt. Das ursprüngliche Verfahren basiert dabei auf der Verwendung der Winkelhalbierenden an jedem Vertex des Eingabeelements. Der Schrumpfprozess terminiert, sobald entweder der Flächeninhalt oder die Kantenlänge des Polygons einen Wert von Null erreicht. Unter der Voraussetzung konvexer Polygone ergeben der Straight-Skeleton-Algorithmus und das Mittelachsen-Verfahren das gleiche Ergebnis, dies ändert sich bei konkaven Polygonen.

In der Variante von Aichholzer et al. besitzt der Algorithmus die Einschränkung, dass er ausschließlich Dachflächen mit einer Steigung von 45° bestimmen kann. Weiterhin gilt, dass alle Kanten mit der gleichen Geschwindigkeit schrumpfen, wobei sie immer parallel zu ihrer Ausgangskante verlaufen. Die hier entwickelte Variante hebt diese Einschränkung auf, indem es möglich ist, Kanten des Eingabepolygons unterschiedlich zu gewichten und

dadurch den Schrumpfungsprozess einzelner Kanten gezielt zu steuern. Dies ermöglicht die Erzeugung von Dachformen mit Neigungswinkeln, die nicht 45° betragen. Zum grundsätzlichen Verständnis des Algorithmus soll allerdings zunächst der Ansatz von Aichholzer et al. erläutert werden, da die Bezeichnungen für das Verständnis des hier vorgestellten Verfahrens eine wichtige Rolle spielen

11.1 **Ereignisse im Straight-Skeleton-Algorithmus**

Der Ansatz von Aichholzer et al. basiert auf der Berechnung unterschiedlicher Arten von Ereignissen, die während des Schrumpfungsprozess des Polygons auftreten können [Ai95]. Diese Ereignisse werden bezüglich des Zeitpunkts ihres Auftretens sortiert und anschließend chronologisch verarbeitet. Das Eintreten eines Ereignisses führt zur Änderung der Struktur des Polygons, was Anpassungen an den verwendeten Datenstrukturen zur Verwaltung des Prozesses erfordert. In der ursprünglichen Fassung unterscheiden die Autoren nur zwei verschiedene Arten von Ereignissen, die sie als *Edge*- und *Split-Event* bezeichnen.

Ein Edge-Event tritt genau dann auf, wenn eine Kante während des Berechnungsprozesses auf eine Länge von null schrumpft, so dass die ursprünglich zu dieser Kante adjazenten Kanten nach der Berechnung direkte Nachbarn sind [Ai95].

Split-Events erfordern gegenüber Edge-Events einen höheren Aufwand zur Aktualisierung der Nachbarschaftsstrukturen. Ein Ereignis dieser Art tritt auf, wenn ein Vertex während des Schrumpfens eine Kante innerhalb des Polygons zerteilt. Events dieser Art können nur bei Vertices auftreten, die bezüglich ihrer adjazenten Kanten einen überstumpfen Winkel besitzen. Aichholzer et al. bezeichnen solche Vertices als *Reflex-Vertex,* der Name ist eine Ableitung von der englischen Bezeichnung *reflex angle* für einen überstumpfen Winkel. Im Falle eines Split-Events wird das Ausgangspolygon in zwei neue Polygone geteilt, die anschließend rekursiv weiter verarbeitet werden.

Nach Eppstein et al. kann bei speziellen konkaven Polygonen eine weitere Art von Ereignis auftreten [EE98]. Die Autoren nennen dieses Ereignis *Vertex-Event*. Es tritt auf, wenn mehrere Reflex-Vertices im gleichen Punkt zusammenlaufen. Vertex-Events sind die einzigen Ereignisse, bei denen neue Reflex-Vertices entstehen können, bei Split-Events teilt das Reflex-Vertex die getroffene Kante. In den neu entstehenden Polygonen sind die Kind-Vertices eines solchen Reflex-Vertex keine Reflex-Vertices mehr. Bei Vertex-Events ist dies dagegen nicht der Fall. Weiterhin ist die Anzahl der neu entstehenden Polygone nicht wie

bei Split-Events auf zwei pro Event beschränkt, vielmehr entspricht sie der Anzahl simultaner Events ausgelöst von Reflex-Vertices.

Die hier vorliegende Implementation integriert eine weitere Art von Ereignis. Hierbei handelt es sich um das *Change-Slope-Event*. Dieses Event wird verwendet, um während der Berechnungen die Steigung der Kanten zu ändern. Ein solches Event führt im Gegensatz zu den vorherigen Ereignissen nicht zu einer Änderung der Polygonstruktur, sondern ausschließlich zu einer Änderung der Ebenenneigung vor der nächsten Iteration. Dieser Ansatz ist erforderlich, um spezielle Dachformen wie Mansardendächer mittels des Weighted-Straight-Skeleton-Verfahrens erzeugen zu können. Konzeptuell ähnelt dieser Eventtyp dem von Kelly et al. [KW11] eingeführten *Edge Direction Event*, der im Abschnitt „Interactive Architectural Modeling with Procedural Extrusion [KW11]“ diskutiert wurde. Analog zum hier eingeführten Change-Slope-Event führt die Verarbeitung eines Edge Direction Events nicht zu einer Änderung der Polygonstruktur, sondern zu einer Änderung der Ebenensteigung an den jeweiligen Kanten. Im Gegensatz zum Semantic Building Modeler spielt für die Auslösung eines Edge Direction Events nicht die aktuelle Höhe des schrumpfenden Polygons eine Rolle für die Eventauslösung, sondern der Verlauf einer nutzerdefinierten Profilkurve. Diese kann aus einer Reihe verschiedener Liniensegmente bestehen. Ändert sich die Steigung dieser Kurve beim Übergang auf ein anderes Liniensegment, wird ein Edge Direction Event erzeugt und verarbeitet. Da der Semantic Building Modeler den Straight-Skeleton-Algorithmus nur für die Dach- und nicht für die vollständige Gebäudegenerierung einsetzt, wird der Change-Slope-Event als vereinfachte Fassung in den Algorithmus integriert.

11.1.1 **Erkennen von Ereignissen**

Kern des gesamten Verfahrens ist das Erkennen und Behandeln von Ereignissen im Verlauf der Berechnung. Zunächst wird hier das Vorgehen zum Erkennen der Ereignisse nach Aichholzer et al. [Ai95] beschrieben, bevor anschließend auf die Besonderheiten der hier verwendeten Implementation eingegangen wird.

11.1.2 **Definition von Distanz**

Zentral für die Berechnung ist der Zeitpunkt an dem ein Ereignis eintritt. Algorithmisch geht man dabei so vor, dass man in jeder Iteration alle Ereignisse berechnet und anschließend das

Ereignis verarbeitet, das zum frühesten Zeitpunkt eintritt. Gibt es mehrere Ereignisse, die zeitgleich eintreten, so werden alle diese Ereignisse verarbeitet. In der ungewichteten Variante des Verfahrens laufen alle Kanten parallel zu ihren jeweiligen Ausgangskanten mit einer konstanten Geschwindigkeit in das Innere des Polygons. Darum definiert man den Zeitpunkt, an dem ein Ereignis auftritt, als den senkrechten Abstand der auslösenden Kante zu ihrer Ausgangskante [Ai95]. Verwendet man ein gewichtetes Eingabepolygon, so beeinflussen die kantenspezifischen Gewichte die Schrumpfgeschwindigkeit der Kanten, was die Bestimmung unterschiedlicher Dachneigungen zur Folge hat.

In dem hier verwendeten Verfahren wird die Distanz anders definiert. Anstatt die Abstände von der Ausgangskante als Grundlage der Bestimmung des Zeitpunkts zu verwenden, nutzt das Verfahren die Höhe in Bezug auf das planare Eingabepolygon. Je größer die Höhe, desto später der Zeitpunkt, an dem ein Ereignis eintritt. Dieses Vorgehen ist aus dem Grunde reizvoll, als es keine zusätzlichen Distanzberechnungen erfordert, die Distanz geht direkt aus den verwendeten Schnittpunktberechnungen hervor und kann ohne weitere Rechenschritte als Maß für die chronologische Sortierung der eintretenden Ereignisse verwendet werden.

11.1.3 **Erkennen von Edge-Events**

Die Bestimmung von Edge-Events stellt den einfachsten Fall des Algorithmus dar. Ein Edge-Event tritt auf, wenn eine Kante während des Schrumpfungsprozesses auf eine Länge von 0 schrumpft. Im ungewichteten Fall, bei dem alle Kanten des Ausgangspolygons mit einer konstanten Geschwindigkeit auf das Innere des Polygons zulaufen, bewegen sich alle Vertices auf ihren Winkelhalbierenden auf das Innere des Polygons zu. Die Bestimmung eines Edge-Events benötigt demnach nur die Berechnung von Schnittpunkten der Winkelhalbierenden von Vertices, die eine gemeinsame adjazente Kante besitzen. Wird ein Schnittpunkt gefunden, so schrumpft genau diese Kante während der Berechnung in sich zusammen und die Ausgangsvertices werden zu einem neuen Vertex zusammengefasst. Für jeden derart gefundenen Edge-Event berechnet man anschließend seine Distanz und sortiert ihn in die Liste der bereits vorhandenen Events ein [Ai95].

11.1.4 **Erkennen von Split-Events**

Split-Events treten auf, wenn ein Reflex-Vertex während des Schrumpfens in eine andere Kante hineinläuft und diese in zwei Teile spaltet. Die Berechnung von Split-Events ist somit

nur in konkaven Polygonen erforderlich, in konvexen Grundrissen kann dieses Ereignis nicht auftreten. Die Bestimmung von Split-Events ist deutlich aufwendiger als die Berechnung von Edge-Events. Es gibt mehrere unterschiedliche Verfahren zur Bestimmung, die sich unter anderem darin unterscheiden, ob sie die Berechnung im R^2 oder im R^3 vornehmen. Ein Verfahren geht dabei zurück auf die Arbeit von Eppstein et al. [EE98] und soll kurz skizziert werden, da es das hier verwendete Verfahren inspiriert hat. Die Autoren verwenden Ebenen, die im 45° Winkel zur Grundebene stehen und für alle Polygonkanten berechnet werden. Für jede Kante begrenzt man die Ebenenausdehnung durch herausrotierte Winkelhalbierenden am Start- und Endvertex der Kante. Der so eingeschränkte Bereich entspricht nun der Fläche, die die Kante während des Schrumpfungsprozesses einnehmen kann. Dieser Bereich kann zwei unterschiedliche Formen haben. Ist keines der beiden Vertices ein Reflex-Vertex, so schneiden sich die rotierten Winkelhalbierenden und die mögliche Fläche hat die Form eines Dreiecks. Wird die Kante an einer Seite dagegen durch ein Reflex-Vertex begrenzt, so ist die mögliche Fläche unbegrenzt und verläuft ins Unendliche. Für das Verfahren von Eppstein ist diese Unterscheidung irrelevant. Basierend auf den so berechneten Ebenen durchläuft man alle Reflex-Vertices im Polygon und berechnet Schnittpunkte der ebenfalls um 45° herausrotierten Winkelhalbierenden dieser Vertices mit allen Ebenen an allen Kanten des Polygons. Wurde ein Schnittpunkt mit einer Ebene gefunden, testet man, ob sich dieser Schnittpunkt innerhalb der möglichen Fläche der zugehörigen Kante befindet, also während des Schrumpfungsprozesses eingenommen werden kann. Ist dies der Fall, wurde ein Split-Event gefunden. Problematisch an diesem Ansatz ist, dass er nur Dachformen mit einer Dachneigung von 45° erzeugen kann und musste darum für das hier verwendete Verfahren abgewandelt werden.

11.1.5 **Erkennen von Vertex-Events**

Betrachtet man die Beschreibung von Vertex-Events, so drängt sich ein Ansatz zur Bestimmung dieses Ereignistyps auf, der keinerlei geometrische Berechnungen erfordert. Ein Vertex-Event tritt genau dann in einem Punkt auf, wenn mehrere Reflex-Vertices innerhalb dieses Punktes zusammenlaufen. Der einfachste Ansatz basiert darum auf der Analyse sämtlicher während der Berechnung aufgetretener Split-Events. Besitzen mehrere Split-Events denselben Schnittpunkt, so fasst man diese zu einem Vertex-Event zusammen und ersetzt die Quell-Events durch den neu erstellten Vertex-Event.

11.1.6 **Erkennen von Ereignissen in der vorliegenden Implementation**

Die hier Umsetzung des Verfahrens unterscheidet sich in der Berechnung von Ereignissen von den zuvor vorgestellten Ansätzen. Ziel dieses Ansatzes ist die möglichst einfache Integration von Kantengewichten in die Berechnung und die Unabhängigkeit der algorithmischen Logik von der Struktur dieser Gewichte. Die Verwendung von gewichteten Kanten dient der Erzeugung von Dachflächen mit Neigungswinkeln, die von 45° abweichen und erlaubt darüber hinaus unterschiedliche Gewichtungen für die Dachkanten, so dass die Neigungswinkel für verschiedene Dachflächen variiert werden können. Bei Bedarf ist es möglich, unterschiedliche Neigungswinkel für unterschiedliche Iterationen des Verfahrens einzusetzen, so dass im entstehenden Dachmodell die einzelnen Dachkomponenten in unterschiedlichen Winkeln zur Grundfläche angeordnet sind. Für die automatische Generierung von Stadtmodellen ist dies für den Realismus der berechneten Modelle zwingend erforderlich, um beispielsweise zufallsgesteuerte Dachneigungen einsetzen zu können und dadurch die visuelle Vielfalt zu erhöhen.

Die Idee des hier vorgestellten Ansatzes basiert auf dem vorab vorgestellten Ansatz von Eppstein et al. [EE98] zur Bestimmung von Split-Events im R^3. Anstatt einen solchen Ansatz aber nur für die Berechnung eines bestimmten Ereignistyps zu verwenden, wird die vorgestellte Technik für alle Arten eingesetzt. Aus diesem Grund wurde als Ausgangsevent eine Abstraktion von den von Aichholzer et al. [Ai95] und Eppstein et al. [EE98] vorgestellten Ereignissen eingeführt, die als *Intersection-Event* bezeichnet wird. Die Berechnung solcher Intersection-Events basiert auf der Zuweisung von Ebenen mit einer nutzerdefinierten Steigung zu jeder Kante des Eingabepolygons. Diese Ebenen sind zunächst unbegrenzt, ihr Steigungswinkel wird durch den Nutzer im Bogenmaß angegeben. Die Konfiguration erlaubt Steigungen im Bereich von 0.0rad bis 1.571rad, was in etwa einer Steigung zwischen 0° und 90° in Grad entspricht. Konzeptuell kann man den Ablauf der Berechnung auch als Sweep-Plane-Ansatz auffassen, wie ihn Kelley et al. in ihrer Arbeit verwenden, um die Gebäudestruktur anhand unterschiedlicher Profilkurven zu beschreiben [KW11].

Die Ermittlung der Ereignisse verläuft nicht mehr wie bei Aichholzer et al. [Ai95] oder Eppstein et al. [EE98] über die Durchführung geometrischer Berechnungen für die einzelnen Arten von Ereignissen, sondern basiert auf der ereignisunabhängigen Berechnung von Intersection-Events. Hierfür berechnet man Schnittpunkte zwischen je drei Ebenen innerhalb des Polygons. Dabei gilt, dass nicht alle möglichen Kombinationen von Ebenen auf

Schnittpunkte untersucht werden müssen, vielmehr reicht es aus, je zwei adjazente Ebenen mit jeder anderen Ebene auf Schnitte zu untersuchen. Dies reduziert die kombinatorische Komplexität deutlich, trotzdem liegt die Anzahl der notwendigen Berechnungen für n Kanten bei $n(n-2)$ und besitzt somit eine Komplexität von $O(n^2)$. Jeder gefundene Schnittpunkt zwischen drei Ebenen wird innerhalb des Verfahrens zunächst vergleichbar zum Ansatz von Eppstein et al. [EE98] validiert. Hierfür prüft man, ob sich der Schnittpunkt innerhalb der möglichen Fläche der Ebenen befindet und greift dafür auf *Same-Side-Of-Ray*-Tests zurück. Als Referenzstrahlen dienen die Winkelhalbierenden an den Vertices, die allerdings nur im Falle gleicher Kantengewichte echte Winkelhalbierende in dem Sinne sind, dass sie den Winkel am jeweiligen Vertex in der Mitte teilen. Verwendet man unterschiedliche Gewichte, so wird die Bestimmung der Halbierenden schwieriger, da diese bezüglich ihres Winkels die unterschiedlichen Gewichte ihrer adjazenten Kanten widerspiegeln müssen. Die Verwendung von Ebenen an den Ausgangskanten erlaubt die Berechnung dieser wichtigen Strahlen durch die Bestimmung der Schnittgeraden zweier adjazenter Ebenen an jedem Vertex im Eingabepolygon.

Der Reiz dieses Ansatzes liegt darin, dass auf diese Art das Kantengewicht über die Steigung der verwendeten Ebenen direkt in die Halbierende einfließt, so dass keinerlei weitere Berechnungen durchgeführt werden müssen. Die so bestimmten Strahlen fungieren anschließend bei der Validierung der Intersection-Event-Schnittpunkte als Referenzen zur Bestimmung der möglichen Fläche, genau wie dies auch bei Eppstein et al. der Fall ist. Durch diesen Ansatz ist man in der Lage, Schnittpunkte auf generische Weise zu ermitteln und zu validieren. Die Validierung selber basiert auf der Berechnung von Referenzpunkten und der Analyse des Verhältnisses zwischen diesen Referenzpunkten, den errechneten Schnittpunkten und den Vertex-Halbierenden. Ist ein berechneter Schnittpunkt gültig, so wird zunächst ein Intersection-Event für dieses Ereignis erstellt und zum globalen Event-Buffer hinzugefügt. Nachdem sämtliche möglichen Kombinationen der Eingabeebenen auf Schnittpunkte untersucht wurden, beinhaltet der Event-Buffer eine Menge von Intersection-Events, die in den nachfolgenden Berechnungen dahingehend kategorisiert werden müssen, um was für konkrete Eventarten es sich handelt. Dies ist erforderlich, da jede Event-Art, also Edge-, Split- oder Vertex-Events andere Verarbeitungen erfordern. Speziell in Bezug auf die Aktualisierungen der Nachbarn eventauslösender Vertices ist die Eventart entscheidend für die Strukturänderung des Polygons und somit für den weiteren Ablauf des Algorithmus. Der einzige Eventtyp, der nicht auf der Berechnung von Intersection-Events basiert, sind die

Change-Slope-Events. Für diese Eventart ist nur die aktuelle Höhe des schrumpfenden Polygons relevant und nicht die Schnitte der einzelnen Ebenen.

Da die Berechnung der Intersection-Events auf einem generischen Ansatz basiert, ist es erforderlich, Kriterien zu definieren, die verwendet werden können, um Intersection-Events in eine der drei Basis-Event-Typen einzusortieren. Hierfür analysiert man die Events im Event-Buffer in mehreren Schritten und entfernt zunächst ungültige Events. Ein sehr einfaches Kriterium, das bereits eine Reihe von Events aussortiert, betrifft Edge-Events. Solche Events können nur in Schnittpunktkonfigurationen auftreten, in denen alle beteiligten Kanten adjazent sind, sie also einen geschlossenen Linienzug bilden. Die Kante, die auf eine Länge von null zusammenschrumpft, ist dann immer die Mittlere der drei Kanten. Im Umkehrschluss bedeutet dies, dass alle Intersection-Events, bei denen das Event-Vertex kein Reflex-Vertex ist und bei denen die beteiligten Kanten nicht adjazent sind, automatisch ungültig sind und somit aussortiert werden können.

Ein weiteres Kriterium betrifft das Erkennen von Edge-Events. Diese werden durch den Intersection-Event-Ansatz nicht wie bei Aichholzer at al. direkt berechnet, vielmehr funktioniert der hier verwendete Berechnungsansatz derart, dass mehrere Intersection-Events für adjazente Kanten bestimmt werden, die dann zu einem Edge-Event zusammengefasst werden. Bei Aichholzer et al. ermittelt man Schnittpunkte der Winkelhalbierenden und bekommt dadurch automatisch die am Edge-Event beteiligten Vertices geliefert. Im vorliegenden Ansatz wird ein Edge-Event dagegen durch Analyse der Intersection-Events bestimmt. Nach dieser Logik liegt ein Edge-Event vor, wenn zwei Intersection-Events durch adjazente Vertices ausgelöst werden, auf den gleichen Schnittpunkt abbilden und keine Reflex-Vertices sind. Sind alle diese Bedingungen erfüllt, handelt es sich zwingend um einen Edge-Event. Die so bestimmten Kriterien reichen aus, um die unterschiedlichen Eventarten, konkret Split- und Edge-Events zu unterscheiden. Die Bestimmung von Vertex-Events folgt anschließend der Logik, die bereits in einem früheren Abschnitt erläutert wurde. Vertex-Events treten auf, wenn zwei oder mehr Reflex-Vertices im gleichen Punkt zusammenlaufen, somit reduziert sich die Erkennung von Vertex-Events auf die Analyse sämtlicher ermittelter Split-Events und deren Zusammenfassung zu Vertex-Events, sofern sie die Voraussetzungen erfüllen.

Eine besondere Rolle spielen die Change-Slope-Events. Wie vorab erwähnt, ist dies der einzige Eventtyp, der nicht durch Änderungen der Geometrie des schrumpfenden Polygons ausgelöst wird. Somit spielen für diese Ereignisart auch Intersection-Events keine Rolle, da

diese nur an Positionen auftreten, an denen eine Änderung dieser Struktur auftreten könnte. Darum kann man bei diesem Eventtyp auch nicht von einem „Erkennen“ von Events sprechen. In der Implementation wird der Change-Slope-Event dadurch umgesetzt, dass man dem Eventbuffer zu Beginn jeder Iteration ein Event mit der nutzerdefinierten Höhe hinzufügt, an der die Steigungsänderung erfolgen soll. Anschließend berechnet das Verfahren wie vorab geschildert sämtliche auftretenden Intersection-Events und führt die regulären Berechnungsschritte durch. Sofern der hinzugefügte Change-Slope-Event von allen während der Iteration berechneten Events derjenige ist, der zum frühesten Zeitpunkt auftritt, wird er verarbeitet.

11.1.7 **Kernschritte jeder Iteration des Verfahrens**

Die Kernschritte des Verfahrens sind in jeder Iteration identisch. Bevor auf diese detailliert eingegangen wird, werden zunächst die Hauptschritte kurz dargestellt. Dies erleichtert die nachfolgende Erläuterung und ihre Einordnung in den Gesamtablauf. Die genannten Schritte werden bei jeder Iteration des Verfahrens durchgeführt, bis das Abbruchkriterium erfüllt ist und das Verfahren terminiert.

1. Hinzufügen eines Change-Slope-Events zum Eventbuffer (sofern eine Steigungsänderung innerhalb der Berechnung erfolgen soll)
2. Ermittlung sämtlicher Intersection-Events
3. Entfernung ungültiger Events, Kategorisierung aller ermittelten Intersection-Events in Edge-, Split- und Vertex-Events
4. Sortierung aller gültigen Events basierend auf dem Zeitpunkt ihres Auftretens
5. Verarbeitung der Events mit dem frühesten Auftrittszeitpunkt
    a. Erzeugung von Kindelementen für alle Vertices
    b. Aktualisierung der Nachbarschaftsverhältnisse (falls erforderlich)
6. Validierung der Nachbarschaftsstrukturen, bei Bedarf Entfernen von Vertices aus der weiteren Verarbeitung.
7. Falls ein Split- oder Vertex-Event verarbeitet wurde, Aufspaltung des Polygons in zwei oder mehr neue Polygone. Für jedes der so entstandenen Polygone wird eine neue Iteration des Algorithmus berechnet.
8. Prüfung des Abbruchkriteriums

11.2 Eventtypabhängige Aktualisierung der Nachbarschaftsverhältnisse

Die Ermittlung der Ereignisse selber stellt nur einen ersten Schritt im Ablauf des Algorithmus dar. Der nächste zentrale Schritt nach der so erfolgten Eventermittlung ist Schritt 5.b im skizzierten Algorithmus. Um die Bedeutung der Nachbarschaftsverhältnisse für das Verfahren zu erfassen, soll kurz die verwendete Datenstruktur erläutert werden. Diese entspricht der von Felkel et al. [PS98] als *Set of circular Lists of active Vertices* (SLAV) bezeichneten Graphenstruktur. Auf jeder Ebene der Iteration hält man eine Liste von Vertices vor, die Zeiger auf ihre beiden Nachbarn speichern. Diese Vertices werden innerhalb der aktuell berechneten Iteration als *aktive Vertices* bezeichnet. Im konvexen Fall gilt, dass auf jeder Ebene nur eine einzige Liste verwaltet wird, in diesem Fall vereinfacht sich die Datenstruktur zu einer *List of active Vertices* (LAV) Struktur. Treten dagegen bei konkaven Polygonen Split- oder Vertex-Events auf, so werden die Polygone aufgespalten und mehrere neue Listen für die nächste Iteration berechnet. In diesem Fall erhält man eine Menge solcher LAV-Strukturen. Die hier verwendete Datenstruktur speichert zusätzlich für jedes Vertex Zeiger auf dessen Kindknoten. Dadurch ist es möglich, zu jedem Zeitpunkt jeden Knoten innerhalb des gesamten Graphs aufzusuchen, indem man beim Eingabepolygon startet und anschließend den Eltern-Kind- und Nachbarschaftsbeziehungen folgt. Eine solche Struktur ist für die abschließende Berechnung der Ergebniselemente unerlässlich, da diese es erfordert, genau nachvollziehen zu können, welche Elemente die gleichen Elternknoten besitzen, um unter anderem kontinuierliche Texturierungen durchzuführen.

Betrachtet man die erläuterten Datenstrukturen, so wird die Bedeutung sorgfältiger Aktualisierungen der Nachbarschaftsverhältnisse nach jeder Iteration offensichtlich. Nachfolgend wird die Aktualisierung abhängig vom Eventtyp erörtert. Die Reihenfolge der Erläuterung entspricht hierbei auch der Abfolge der Nachbarschaftsaktualisierungen in der konkreten Implementation.

Die einfachsten Updates betreffen Vertices, die selber keinen Event ausgelöst haben. Für diese erzeugt man zunächst Kinder auf der Höhe, die durch den verarbeiteten Event festgelegt wurde und weist anschließend dem erzeugten Kind seine Nachbarknoten zu. Im einfachsten Fall ändern sich die Nachbarschaftsverhältnisse nicht, die Nachbarn des Kindes eines Knotens entsprechen den Kindern der Nachbarn seines Elternknotens. Somit reduziert sich die Aktualisierung der Nachbarschaftsverhältnisse auf die Ermittlung der Kindknoten

der Nachbarn des eigenen Elternknotens. Ein solches einfaches Update führt allerdings nur dann zu korrekten Ergebnissen, wenn die Nachbarn selber keine Events ausgelöst haben. Ist dies nicht der Fall, wird das Update aufwendiger. Die Behandlung der Nachbarschaftsaktualisierungen im Event-Fall wird allerdings erst in einem späteren Schritt durchgeführt. Die Kernidee dabei ist, dass ein Event-Vertex selber am besten entscheiden kann, welche Nachbarn es zugewiesen bekommt, da dies vom Typ des Events abhängt. Aus diesem Grund werden Nachbarschaftsupdates bei Eventvertices immer rekursiv durchgeführt. Wenn also ein Eventvertex ein anderes Vertex als Nachbarn zugewiesen bekommt, so ändert es automatisch auch die Nachbarschaftsstrukturen des neuen Nachbarn. Dadurch soll garantiert werden, dass jede Beziehung zwischen Vertices innerhalb der Graphenstruktur bidirektional ist.

11.2.1 **Nachbarschaftsupdates für Change-Slope-Events**

Da Change-Slope-Events nicht zu einer Änderung der Polygonstruktur führen, erfordern sie auch keine Nachbarschaftsupdates. Bei der Verarbeitung werden für sämtliche Vertices der aktuellen Iteration Kindvertices erzeugt. Da zu jedem Ausgangsvertex genau ein Kindvertex gehört und sich für keines der Kinder die Nachbarschaft ändert, sind keine speziellen Verarbeitungsschritte für die Nachbarschaftsstruktur erforderlich. Change-Slope-Events stellen somit bezüglich des Verarbeitungsaufwands die einfachste Event-Variante dar

11.2.2 **Nachbarschaftsupdates für Edge-Events**

Nachdem für sämtliche Nicht-Event-Vertices Nachbarschaftsaktualisierungen durchgeführt wurden, besteht der nächste Schritt in der Verarbeitung von Edge-Events. Auch bei diesen gestaltet sich das Update vergleichsweise einfach. Zunächst erzeugt man ein neues Vertex als Kind der beiden Vertices, die das Edge-Event ausgelöst haben. Nun setzt man für dieses neue Kindvertex die Nachbarn. Da durch das Edge-Event die Kante zwischen den beiden Elternknoten in der neuen Graphenstruktur verschwunden ist, müssen die Nachbarn aktualisiert werden. Um ein besseres Verständnis der Aktualisierung zu bekommen, sei kurz erläutert, wie die Nachbarn innerhalb des Vertex selber gespeichert werden. Hierbei unterscheidet man die Nachbarn durch Indices. der Nachbarknoten mit Index 0 ist immer der nächste Knoten im Uhrzeigersinn, der Nachbarknoten mit Index 1 dagegen der nächste Knoten entgegen dem Uhrzeigersinn und somit der direkte Vorgänger. Bei Edge-Events ist

einer der Elternknoten Nachbar mit Index 0 des anderen Elternknoten. Das Kindvertex bekommt das Kind von dessen Nachbarn mit Index 0 als Nachbar mit Index 0 und analog das Kind des Nachbarn mit Index 1 als Nachbar mit Index 1. Anschaulich löscht man also die Kante aus dem Graph, so dass die vorab durch die Kante getrennten Kanten im nächsten Iterationsschritt adjazent sind.

11.2.3 **Nachbarschaftsupdates für Split-Events**

Die Aktualisierung von Split-Event-Vertices ist aus dem Grunde deutlich aufwendiger als das Edge-Event-Update, da nicht nur Nachbarschaften von Kindvertices aktualisiert werden müssen, sondern es außerdem erforderlich ist, den Kindknoten selber zu vervielfältigen. Diese Anforderung hängt mit der Zerteilung des Polygons in zwei neue Polygone zusammen, die durch einen Split-Event ausgelöst wird. Da Split-Events Kanten des Eingabepolygons teilen, entstehen durch sie neue Polygonzüge, die unabhängig voneinander weiter verarbeitet werden. Der Kindknoten eines Split-Event-Vertex muss dabei in beiden neuen Polygonen vorkommen. Somit gibt es zwei Ansätze, um dieses Problem zu lösen. Entweder man erhöht den Grad des Knotens innerhalb der Graphenstruktur und erlaubt mehr als zwei Nachbarn oder man vervielfältigt den Kindknoten und fügt in beide entstehenden Polygone einen der erzeugten Kindknoten ein.

Die erste Variante hat den Nachteil, dass die Interpretation der Nachbarschaftsverhältnisse erschwert wird. Steigt der Grad eines Knotens, ist die vorab erläuterte indexbasierte Interpretation nicht mehr ohne zusätzliche Informationen möglich. Man müsste zusätzlich zu dem Index auch noch weitere Informationen speichern, beispielsweise auf welches Polygon sich die jeweilige Indizierung bezieht. Weiterhin könnten die durch das Splitting neu entstehenden Polygone nicht mehr unabhängig voneinander verarbeitet werden, da sie sich mindestens einen Knoten teilen. Dies ist aus Sicht der Parallelisierbarkeit der Verarbeitung nicht wünschenswert, weshalb hier ein Ansatz verwirklicht wird, bei dem der Kindknoten vervielfältigt wird. Die Beziehung zwischen Eltern- und Kindknoten besitzt dabei weiterhin nur den Grad 1, das Elternvertex speichert nur einen Zeiger auf das originale Kind, den Zwilling kennt es nicht. Die Verbindung zwischen den Geschwistern wird auf der Kindebene aufgelöst, indem die Geschwisterknoten eine bidirektionale Beziehung realisieren. Hat man die Kindproblematik derart gelöst, kann man sich mit den Nachbarschaftsupdates befassen. Die Zuweisung der Nachbarn zu Kind und Zwilling erfolgt dabei nach einem festgelegten Schema, so dass nach der Nachbarschaftsberechnung zwei

getrennte Polygonzüge entstehen, die nicht durch Nachbarschaftsbeziehungen miteinander verbunden sind.

11.2.4 **Nachbarschaftsupdates für Vertex-Events**

Vertex-Events sind konzeptuell ähnlich zu Split-Events, da auch sie zu einer Aufspaltung der Polygonstruktur führen. Die Verarbeitung ähnelt darum auch den Arbeitsschritten bei Split-Events, was damit zusammenhängt, dass Split-Events Basis der Bestimmung von Vertex-Events sind. Theoretisch ist die Anzahl von Split-Events, die in einem Vertex-Event zusammengefasst werden, unbegrenzt. In der Praxis ist dies für die Verarbeitung ungünstig, da wiederum das bereits beschriebene Problem von Knoten mit einem Grad größer als zwei entstehen würde. Aus diesem Grund werden Vertex-Events derart eingeschränkt, dass nur zwei Split-Events jeweils zu einem Vertex-Event zusammengefasst werden können. Treten mehr als zwei Split-Events im gleichen Punkt auf, so werden mehrere Vertex-Events erzeugt, um diese Split-Events vollständig aufnehmen zu können.

Wie bei Split-Events werden auch bei Vertex-Events die Kindknoten der auslösenden Vertices vervielfältigt und anschließend spezifische Nachbarschaftsupdates ausgeführt. Diese Aktualisierungen sind einer der Hauptgründe für die Einschränkung der Vertex-Events auf maximal zwei Quell-Split-Events. Begrenzt man diese Zahl nicht, so bekommt man massive Probleme bei der Zuweisung der Nachbarschaften, da man alle Quellvertices berücksichtigen und ihre Nachbarn mit in die Berechnung einfließen lassen muss. Verwendet man die beschriebene Einschränkung, so kann deterministisch festgelegt werden, welche Nachbarn für das originale Kindvertex und welche für das Zwillingsvertex verwendet werden. Steigt die Anzahl der beteiligten Split-Events, so muss zwingend auch die Anzahl der Kinder steigen, man kommt mit einem einzelnen Zwilling nicht mehr aus, die Verwaltung der entstehenden Strukturen wird deutlich erschwert. Die Einschränkung vereinfacht diese Situation und verschiebt die Problematik auf die Ebene der Eventverarbeitung. Jedes entstehende Vertex-Event wird unabhängig von den anderen Events verarbeitet, egal, ob es sich um das gleiche Kindvertex handelt, oder nicht. Hierdurch kann es sehr wohl dazu kommen, dass mehrere originale und Zwillings-Vertices im gleichen Punkt erzeugt werden, allerdings weiß jedes Event, welches Vertex zu ihm gehört und auch die Eltern-Kind-Beziehung bleibt eindeutig auflösbar.

11.3 **Problemematische Nachbarschaftsverhältnisse – der Virtual Edge-Ansatz**

Je nach Struktur der Eingabepolygone kann es bei der Aktualisierung der Nachbarschaftsverhältnisse zu Problemen kommen, die erkannt und behandelt werden müssen, um am Ende jeder Iteration eine valide Struktur garantieren zu können. Diese Problematik sei am Beispiel mehrerer Split-Events auf der gleichen Kante, aber nicht im gleichen Punkt, erläutert. Wird eine Polygonkante durch mehrere Reflex-Vertices im gleichen Iterationsschritt zerteilt, wird das Nachbarschaftsupdate deutlich komplizierter. Dies liegt darin begründet, dass die Ausgangsbeziehungen nicht mehr ausreichen, um die neuen Nachbarn der Kinder korrekt zu berechnen. Bei Split-Events basiert dieses Update darauf, einerseits die Kinder der Vertices als Nachbarn zu verwenden, die auf Elternebene die Kante begrenzen. Wird aber nun eine Kante mehrfach durch Split-Events zerteilt, so ist der direkte Nachbar eines Kindes auf dieser Kante nicht mehr zwingend das Kind des Start- oder Endvertex auf der Kante, sondern kann auch durchaus das Kind eines anderen Split-Events sein. Da jeder Split-Event unabhängig von den anderen Events verarbeitet wird, besitzt der einzelne Event keinerlei Informationen über andere Events und kann somit auch den Problemfall nicht behandeln. Ein einzelner Event dieser Art kennt schließlich nur die zerteilte Kante sowie die direkten Nachbarn seines Elternknotens. Im Fall einzelner Splits auf der gleichen Kante reichen diese Informationen aus, um eine valide Struktur zu garantieren, bei mehreren Events auf der gleichen Kante ist dies dagegen nicht mehr der Fall.

Um dieses Problem zu lösen, implementiert der Semantic Building Modeler ein Verfahren, das als *Virtual-Edge-Ansatz* bezeichnet wird. Eine solche *virtuelle Kante* ist definiert als ein Strahl mit einem Startpunkt und einer Richtung. Auf diesem Strahl kann sich eine beliebige Anzahl von Vertices befinden, die basierend auf ihrer Distanz zum Startknoten sortiert werden. Für jeden Split- und Vertex-Event wird vor Aktualisierung der Nachbarschaftsverhältnisse eine solche Kante erzeugt. Anschaulich handelt es sich dabei um eine Zusammenfassung mehrerer adjazenter Kanten mit exakt gleicher Ausrichtung zu einer virtuellen Kante. Für einen Split-Event erzeugt man beispielsweise eine Kante, die der zerteilten Kante entspricht und fügt das Kind zu der virtuellen Kante hinzu. Wird nun ein weiterer Split-Event verarbeitet, verwendet man den *Virtual-Edge-Manager* der aktuellen Iterationsebene und versucht zunächst, eine Kante zu finden, die sich mit der geteilten Kante überlappt. Wird eine solche Kante gefunden, fügt man dieser die Vertices des Split-Events hinzu, sonst wird eine neue Kante angelegt. Kernidee der virtuellen Kanten ist es nun, bei

Nachbarschaftsupdates über den Virtual-Edge-Manager nachzuschlagen, ob es zwischen dem zu aktualisierenden Vertex und seinem ursprünglichen Nachbarn noch weitere Vertices auf dieser Kante gibt. Wird ein solches Vertex gefunden, wird über dieses die Nachbarschaft aktualisiert.

In einer frühen Phase der Implementation wurden sowohl originale Kinder als auch ihre duplizierten Zwillinge zur virtuellen Kante hinzugefügt. Die Distanz des Zwillings wurde dabei immer um einen sehr kleinen Betrag gegenüber der Distanz des originalen Vertex erhöht. Dadurch wurde garantiert, dass die Anordnung der Vertices auf der virtuellen Kante immer eine exakte Wahl des jeweils nächsten Vertex zuließ. Das Problem, das sich aus diesem Ansatz ergab, besteht in der Richtungsabhängigkeit der Anordnung. Je nachdem, in welche Richtung diese verläuft, ist der Nachbar eines Vertex entweder der Zwilling oder das originale Vertex. Die Auflösung solcher Ambiguitäten erforderte die Definition zusätzlicher Regeln, beispielsweise „Zwillingsvertices bekommen immer Zwillinge“. Allerdings führte auch der Einsatz einer solchen regelbasierten Nachbarauswahl nicht immer zu korrekten Ergebnissen, was zu einer geringen Robustheit der Berechnung führte. Es war nicht möglich, die korrekte Terminierung des Verfahrens bei unterschiedlichen Grundrissen zu garantieren, da die Struktur der Nachbarschaftsbestimmung speziell bei komplexen Grundrissen zu labil war. Aus diesem Grund wurde der Ansatz verallgemeinert.

Anstatt auch die Zwillinge in die Berechnungen einzubeziehen, werden nun alle Nachbarschaften zunächst auf der Ebene der originalen Vertices berechnet. Aus diesem Grund werden auch den virtuellen Kanten ausschließlich originale Vertices hinzugefügt, wodurch die Richtungsabhängigkeit entfällt. Vertices, die über Zwillinge verfügen, bekommen im ersten Schritt vier Nachbarn. Nachdem alle Nachbarn berechnet wurden, werden den Zwillingen solcher Vertices jeweils der dritte und vierte Nachbar zugewiesen. Dabei werden alle Updates der Nachbarschaften rekursiv durchgeführt, wenn also ein Vertex einen Nachbarn auf Nachbarslot `0` zugewiesen bekommt, so bekommt dieser Nachbar umgekehrt eine Referenz auf Nachbarslot `1`. Dieses Verfahren benötigt keinerlei Sonderregeln und ist deutlich robuster. Es führt bei allen Konfigurationen zu korrekten Lösungen. Ausnahme ist eine Konfiguration, die als *Quadrat-Problem* bezeichnet und im Abschnitt „Das „Quadrat-Problem““ erläutert wird. Allerdings ist bei diesem Problem die auslösende Konfiguration bekannt und kann somit im Hauptverfahren vermieden werden.

11.4 Validierung der Polygonstruktur – Lösch-Events

Nachdem sämtliche Kindvertices berechnet und die Nachbarschaften bestimmt wurden, erfolgt eine Validierung der Polygonstruktur. Für konvexe Polygone ist dieser Schritt nicht erforderlich, da hier ausschließlich Edge-Events auftreten, die unkompliziert verarbeitet werden können. Die Nachbarschaftsbestimmung ist aufgrund der Tatsache, dass weder Split- noch Vertex-Events möglich sind, unkritisch, so dass nach jeder Iteration garantiert ist, dass man ein gültiges Polygon erhält. Bei konkaven Polygonen kann es allerdings dazu kommen, dass Vertices aus der weiteren Berechnung entfernt werden müssen. Dies hängt von der Nachbarstruktur der Vertices ab. Ein Vertex wird genau dann als ungültig betrachtet, wenn es entweder zweimal den gleichen Nachbarn zugewiesen bekommt oder wenn das Vertex und seine beiden Nachbarn auf einer Kante liegen. In beiden Fällen kann man das Vertex aus der Verarbeitung entfernen, da es keinerlei weiteren Beitrag zur Struktur des topologischen Skeletts leisten wird.

Im Falle gleicher Nachbarn ist das Vertex nur über eine Kante mit seinen Nachbarn verbunden. Dies macht die Berechnung einer Schnittgerade am Vertex selber unmöglich, diese ist aber auch für den weiteren Berechnungsablauf nicht erforderlich. Hängt ein solches Vertex nur noch an einer Kante, so kann es auch nicht weiter schrumpfen und keine neuen Kinder mehr zugewiesen bekommen. Somit wird das Vertex aus dem Vertexbuffer für die nächste Verarbeitungsstufe entfernt und bei der nächsten Iteration nicht mehr berücksichtigt. Konzeptuell kann man das Ganze mit dem globalen Abbruchkriterium des Verfahrens vergleichen, bei dem die Berechnung terminiert, sobald das entstehende Polygon einen Flächeninhalt von null besitzt. Betrachtet man das ungültige Vertex mit seinem Nachbar als eigenständiges Subpolygon, so beträgt dessen Flächeninhalt ebenfalls null und eine weitere Berechnung ist nicht erforderlich. Somit ändert auch das Löschen dieses Vertex aus dem Kindbuffer nichts an der Struktur des finalen Linienzuges.

Dies gilt auch für den zweiten Fall, in dem Vertices aus der weiteren Verarbeitung entfernt werden. Liegen das Vertex und seine beiden Nachbarn auf einer Kante, so kann man das Vertex entfernen, ohne dass dies Auswirkungen auf die weitere Berechnung hat. Vielmehr würde man auch in diesem Fall Probleme bei der wichtigen Schnittgeradenbestimmung bekommen, da beide adjazenten Ebenen identisch sind und somit eine Schnittgerade mathematisch nicht definiert ist. Entfernt man das Vertex, ändert dies dagegen nichts an der Kante selber und sämtliche weiteren Berechnungsschritte können unverändert durchgeführt werden. Während man im Falle gleicher Nachbarn keinerlei Updates der Nachbarn

durchführt, werden im Fall linearer Nachbarn deren Nachbarschaftsverhältnisse aktualisiert. Ein solches Update ist trivial, der Nachbar des gelöschten Vertex mit Index `0` bekommt den Nachbarn des gelöschten Vertex mit Index `1` als neuen Nachbarn auf Nachbarposition `1` und umgekehrt. Dies erzeugt eine valide Polygonstruktur, die in den weiteren Iterationen weiter verarbeitet werden kann.

11.5 **Berechnung der Ergebnisstruktur**

Das Modul zur Berechnung der Dachstruktur ist nur eine Komponente des Gesamtsystems, die möglichst lose an dieses gekoppelt werden soll. Aufgrund der Komplexität des Verfahrens sind auch die Datenstrukturen und Klassen, die zur Repräsentation der einzelnen Bestandteile verwendet werden, komplex und algorithmenspezifisch. Die Verwendung solcher Strukturen innerhalb des Hauptsystems ist darum unerwünscht. Um die Wiederverwendbarkeit des Moduls zu maximieren, ist es erforderlich, Ein- und Ausgabestrukturen so einfach wie möglich zu gestalten. Die Eingabe in das Modul besteht aus einer Menge von Kantengewichten und Vertices, die ausschließlich durch ihre Position im dreidimensionalen Raum definiert sind und im Uhrzeigersinn angegeben werden müssen. Die Angabe der Vertices in einer festen Reihenfolge ist aus dem Grund zentral, da diese für die Bestimmung der Nachbarn innerhalb der ersten Iteration erforderlich ist. Die nachfolgenden Iterationen greifen dann nur noch auf die vorab berechneten Nachbarschaftsbeziehungen zurück und nutzen diese für die weiteren Berechnungsschritte. Auch die Ergebniselemente sollen leicht verständlich und möglichst problemlos in andere Komponenten integrierbar sein. Deshalb wird nach der Terminierung des Verfahrens als Abschluss eine Ergebnisstruktur erstellt, die sämtliche Dachflächen enthält. Die Berechnung dieser Ergebnisflächen startet bei den ursprünglichen Eingabekanten. Wie bereits bei der Erläuterung der verwendeten Datenstrukturen beschrieben, speichert jedes Vertex Zeiger auf seine beiden Nachbarn sowie auf sein Kind, sofern ein solches vorhanden ist. Das grundsätzliche Vorgehen besteht darum darin, für alle Kanten des Eingabepolygons durch die gesamte Datenstruktur zu iterieren und Ergebnis-Elemente zu erstellen. Diese Ergebnis-Elemente sind entweder Vier- oder Dreiecke. Dreiecke kommen dabei nur als letztes Element vor, da es anschließend keine weitere Kante gibt, von der ausgehend weitere Elemente berechnet werden können.

Das Ergebnis dieser Berechnungen für eine Kante ist eine Menge adjazenter Elemente, die alle einheitlich texturiert werden. Für die Bestimmung der Elemente selber ist deren Struktur

irrelevant, für die Berechnung der Texturkoordinaten für jedes Vertex muss allerdings zwischen verschiedenen Arten von Vierecken unterschieden werden, da dies direkten Einfluss auf die Bestimmung der Koordinaten im Texturraum hat. Mögliche auftretende Formen sind dabei Trapeze sowie Parallelogramme als deren Sonderfall. Weiterhin muss unterschieden werden, ob bei Trapezen die untere oder obere Kante die Längere ist und in welche Richtung die Parallelogramme geneigt sind. Basierend auf diesen Berechnungen ist es dann möglich, *Offsets* im Texturraum zu berechnen, die es erlauben, Texturen derart zu applizieren, dass zwischen zwei Elementen keine Brüche entstehen und in der u-Dimension kein Versatz feststellbar ist. Die Berechnung und Verwendung solcher Offsets ist für das visuelle Ergebnis von zentraler Bedeutung.

Neben Problemen durch den Versatz einzelner Texturen ist die Skalierung der Textur ein weiterer Faktor, der berücksichtigt werden muss. Das Ziel ist es, das gesamte Dach mit der gleichen Textur zu belegen und diese immer im gleichen Größenverhältnis aufzutragen. Vernachlässigt man diesen Aspekt, so werden die Dachziegel auf benachbarten Dachflächen unterschiedlich groß sein. Aus diesem Grund aktualisiert das System permanent einen *Scaling-Faktor*, der aus dem Verhältnis von Element- zu Texturhöhe, beziehungsweise Element- zu Texturbreite errechnet und aktualisiert wird. Für jedes Element bestimmt man diese Proportionen, verwendet immer den kleineren Wert und aktualisiert den globalen Scaling-Faktor nur dann, wenn dieser größer als der aktuell berechnete Wert ist. Nachdem alle Ergebnis-Elemente verarbeitet wurden, besitzt man einen Faktor, mit dem man sämtliche berechneten Texturkoordinaten uniform skalieren kann, so dass die Ausdehnungen der Texturen auf den Elementflächen immer gleich sind. Anschaulich bedeutet dies, dass man eine konstante Größe der Dachziegel erhält, die aber wiederum abhängig von der konkret berechneten Dachform ist. Somit gilt, dass zwei unterschiedliche Ergebnisstrukturen aus unterschiedlichen Eingaben in den Algorithmus trotz der Verwendung der exakt gleichen Textur unterschiedliche Dachziegelgrößen generieren werden, da diese eine Funktion der Dachstruktur sind.

11.5.1 **Probleme und Schwierigkeiten bei der Berechnung der Ergebniselemente**

Wiederum funktioniert das Verfahren zur Bestimmung der Ergebniselemente bei konvexen Polygonen einwandfrei. Der Grund liegt darin, dass garantiert ist, dass jedes Vertex zwingend ein Kind besitzt. Findet man zwei Vertices ohne Kinder, so terminiert die Berechnung und man hat eine Seite vollständig verarbeitet. Im degenerierten Fall bei

konkaven Polygonen können allerdings Lösch-Events auftreten, es werden also Vertices aus dem Vertex-Buffer entfernt und dadurch aus der weiteren Verarbeitung ausgeschlossen. Die Probleme, die dies für die Ergebnis-Berechnung nach sich zieht, sind offensichtlich. Besitzt ein Vertex kein Kind, so muss dies nicht zwingend zu einer Terminierung führen. Wiederum sind Kanten, die durch mehrere Split-Events zerteilt werden, ein anschauliches Beispiel für die Problematik. Man startet beispielsweise bei der Kante, die in einem späteren Iterationsschritt von mehreren Reflex-Vertices gleichzeitig, aber nicht im gleichen Punkt getroffen wird. In der ersten Phase der Berechnung ergibt sich ein Trapez als Ergebnis-Element. Nun stellt man aber fest, dass eines der beiden Vertices auf der nächsten Ebene kein Kind mehr besitzt, weil es durch einen Lösch-Event aus der weiteren Verarbeitung entfernt wurde. Dies bedeutet allerdings nicht zwingend, dass auch die Verarbeitung der Kante terminiert, weil es möglich ist, dass eines der Vertices auf der unterteilten Kante sehr wohl über ein Kind verfügt. Würde man an diesem Punkt die Berechnung terminieren lassen, würde das Ergebnis-Element auf dieser Kante nicht erkannt und das Ergebnis würde ein Loch enthalten, für das keinerlei Texturkoordinaten berechnet werden. Wiederum muss also für diesen Fall ein Ansatz gefunden werden, der diese Problematik abfängt und auch bei solchen Konfigurationen die Ergebnis-Element-Berechnung zu einem gültigen Ende führt. Wie auch bei Nachbarschaftsupdates auf mehrfach gesplitteten Kanten bietet sich der Virtual-Edge-Ansatz für die Auflösung dieser Ambiguitäten an. Da die virtuellen Kanten auf jeder Iterations- und Polygonebene durch einen spezifischen Manager verwaltet werden, der ausschließlich Kanten dieser Iteration speichert und berücksichtigt, ist es zunächst erforderlich, alle virtuellen Kanten im Controller des Algorithmus zu aggregieren und diese bei der Ergebnis-Element-Berechnung auszuwerten. Eine solche Aggregation ist unproblematisch, da durch das Schrumpfen des Polygons, also durch die Berechnung der Kinder auf jeder Ebene, die Vertex-Mengen disjunkt sind. Somit müssen auch die Mengen der virtuellen Kanten disjunkt sein. Hat man die Kanten derart zusammengefasst, lässt sich auch das vorab erläuterte Problem bei der Ergebnis-Berechnung elegant lösen. Findet man ein Vertex ohne Kind, so fragt man beim globalen Virtual-Edge-Manager an, ob es eine Kante zwischen dem kinderlosen Vertex und seinem aktuellen Nachbarn gibt. Wenn dies der Fall ist, lässt man sich den ersten Knoten zwischen den beiden Vertices geben, der über Kinder verfügt. Sollte auch auf diese Art kein gültiger Knoten gefunden werden, wurde diese Dachseite vollständig verarbeitet und das Verfahren startet erneut mit der nächsten Eingabekante.

Die Ergebnis-Strukturen selber bestehen der Einfachheit halber nur aus Arrays von drei oder vier Vertices. Jedes dieser Vertices speichert zunächst nur seine Position. Die Texturkoordinaten werden in einer Hashmap vorgehalten. Der Grund für die vom Vertex losgelöste Speicherung der so berechneten Werte liegt in der Wiederverwendbarkeit der Vertices. Während die Position eines Vertex zwischen benachbarten Flächen absolut ist, sich demnach verschiedene Flächen das gleiche Vertex teilen können (ein Vertex kann in maximal vier Flächen gleichzeitig vorkommen), sind die Texturkoordinaten relativ zur jeweiligen Dachfläche. Ein Vertex, das in mehreren Elementen vorkommt, besitzt in jedem solchen Element vollständig andere Texturkoordinaten, die somit auch auf Ebene der Fläche und nicht auf der Ebene des Vertex gespeichert werden sollten. Der Zugriff auf die Texturkoordinaten innerhalb der Hashmap erfolgt über den Index des Vertex im Ergebnis-Element. Auch bei diesen ist die Abfolge der Vertices festgelegt, außerdem müssen sie wie auch die Eingabevertices im Uhrzeigersinn angegeben werden.

11.6 **Bekannte Probleme der Implementation**

11.6.1 **Das „Quadrat-Problem“**

Das Verfahren erreicht bei einer Vielzahl untersuchter Konfigurationen eine hohe Robustheit und produziert zuverlässig korrekte Dachformen mit zugehörigen Texturkoordinaten. Während der Arbeit mit diesem Modul zeigte sich allerdings eine spezielle Konfiguration, bei der das Verfahren in seiner Berechnung scheitert. Bei der Berechnung der Dachformen für eine solche Konfiguration tritt ein Fall auf, bei dem ein Reflex-Vertex zwei Split-Events im selben Schnittpunkt auslöst. Dieser Fall kann nur dann auftreten, wenn durch das Schrumpfen genau eine Ecke getroffen wird und das Verfahren davon ausgeht, dass beide adjazenten Kanten dieser Ecke aufgespalten werden. Die Problematik besteht darin, dass ein einzelnes Eventvertex mehr als ein Kind bekommen müsste, auch die Nachbarschaftsupdates scheitern in diesem Fall. Abbildung 102 zeigt eine schematische Darstellung einer Konfiguration, die zum genannten Fehler führt. Die Skizze zeigt eine Eingabekonfiguration, wie sie durch das Hauptprogramm in das Dach-Modul eingeben wird. Hierbei ist festzuhalten, dass die tatsächliche Eingabe nur in den Außenkanten des Grundrisses besteht. Die Kanten des grau hinterlegten Quadrats treten in der Berechnung der Dachstrukturen nicht auf. Während des Schrumpfungsprozesses schrumpft nun das graue Quadrat in sich zusammen, das blau und das rot umkreiste Vertex bewegen sich aufeinander zu und treffen sich genau in der Mitte des Quadrates. Anschaulich

spaltet ein Vertex die adjazenten Kanten des anderen Vertex genau am zugehörigen Eckpunkt, wodurch eine eindeutige Auflösung der ausgelösten Split-Events nicht mehr möglich ist.

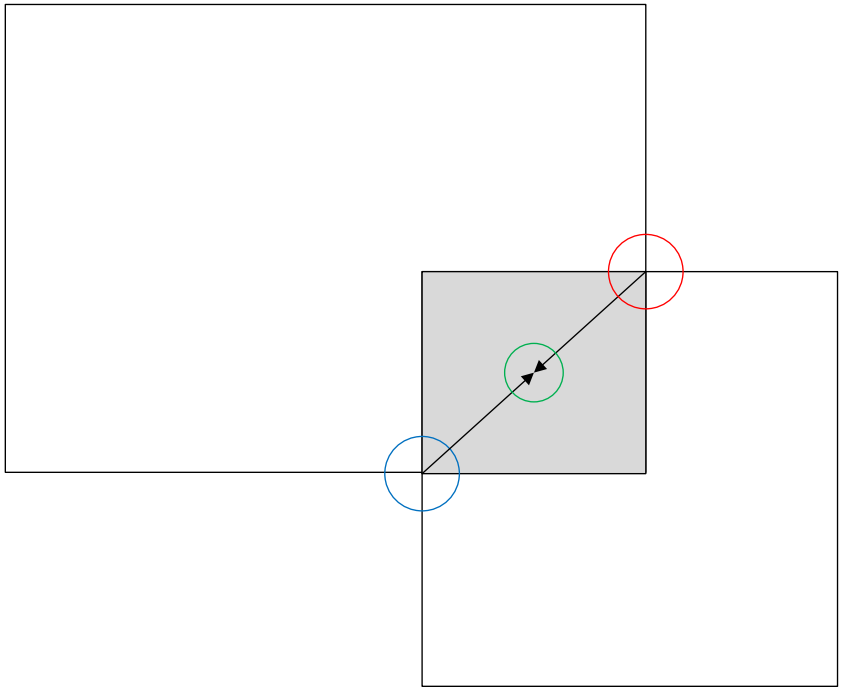

Abbildung 102: Quadratproblem-Konfiguration

Die Lösung dieses Problemfalls im Verfahren selber ist sehr aufwendig und erfordert die Definition einer Vielzahl von Ausnahmeregeln, um genau diese Problematik zu beheben. Deutlich einfacher ist es dagegen, die Eingabe zu modifizieren und dafür zu sorgen, dass das Straight-Skeleton-Verfahren nur solche Strukturen bekommt, die es auch korrekt verarbeiten kann. Im Beispiel reicht es bereits aus, eines der beiden großen Rechtecke um eine Einheit zu verschieben oder in den Ausdehnungen zu modifizieren. Dadurch wird der Problemfall vermieden und das Verfahren errechnet wiederum korrekte Dachformen. Eine andere Möglichkeit besteht in der Verwendung leicht unterschiedlicher Kantengewichte für die beteiligten Kanten. Auch diese Modifikation der Eingabe führt zu einer korrekten Berechnung.

Zusammenfassend seien hier noch einmal kurz die Vorbedingungen für das Auftreten des Problemszenarios aufgelistet:

1. Alle Kanten besitzen das gleiche Gewicht, so dass alle Ebenen die gleiche Steigung besitzen

2. Die Eingabe bildet eine Form, bei der mindestens ein Reflex-Vertex auftritt und eine Ebenenschnittgerade die Ebenenschnittgerade des gegenüberliegenden Vertex schneidet
3. Der Split-Event muss verarbeitet werden, die Distanz muss also die geringste Distanz aller in der aktuellen Iteration auftretenden Events besitzen

Speziell Punkt 2.) ist dabei selten, da das Problem nicht generell für quadratische Grundrisse gilt, sondern nur für solche, bei denen ein Eckpunkt des Quadrats ein Reflex-Vertex ist, da dies Voraussetzung für das Auftreten von Split-Events ist. Ein einfacher Ansatz zur Vermeidung des Problems innerhalb des Algorithmus besteht darin, die Gewichtungen der Eingabekanten alle innerhalb eines kleinen Bereichs zu variieren. Dies würde Voraussetzung 1.) aushebeln. Alternativ könnte man die Grundrisseingaben in das Verfahren analysieren und bei der Feststellung einer problematischen Konfiguration die Eingabebausteine entweder in ihrer Größe oder ihrer Position minimal ändern. Auch dies wäre für die Problemvermeidung ausreichend.

Um im Hauptprogramm auf das Auftreten solcher Fehler reagieren zu können, werden bei jeder Iteration alle auftretenden Split-Events untersucht. Sollte eine irreguläre Konfiguration entdeckt werden, so wird eine `Exception` ausgelöst, die im Hauptprogramm ausgewertet werden kann und weitere Verarbeitungsschritte ermöglicht.

11.6.2 **Floating-Point- und Rundungsprobleme**

Ein weiteres bekanntes Problem des Verfahrens besteht in der Anfälligkeit für Rundungsfehler ausgelöst durch die Verwendung von Gleitpunktzahlen (*floating point numbers*). Ursächlich hierfür ist die Problematik, dass es nicht möglich ist, irrationale Zahlen als Teilbereich der reellen Zahlen innerhalb eines Computers exakt abzubilden. Aus diesem Grund verwendet man typischerweise einen Ansatz, bei dem die Genauigkeit der Zahlendarstellung variiert wird, indem die vorhandenen Bits flexibel auf den ganzzahligen und den gebrochenen Anteil verteilt werden. Diese Darstellungsform erlaubt den Einsatz solcher Zahlen in vollkommen unterschiedlichen Problem-Domänen. Während man beispielsweise in der Astronomie mit sehr großen ganzzahligen Komponenten arbeitet, bei denen der gebrochene Anteil typischerweise eine geringe Bedeutung besitzt, spielen in der Pharmazie sehr exakte Nachkommastellen eine wichtige Rolle. Der Einsatz von Gleitpunktzahlen ermöglicht eine ideale Ausnutzung der vorhandenen Bits, so dass diese der

jeweiligen Problemstellung entgegenkommen [GS02]. Eine exakte Darstellung ist möglich, falls es sich bei den verwendeten Zahlen um rationale oder ganze Zahlen handelt. Irrationale Zahlen werden dagegen nur näherungsweise abgebildet.

Diese Problematik führt bei den mathematischen Berechnungen innerhalb des Systems zu nicht vorhersehbaren Abweichungen. Solche Abweichungen können an verschiedenen Stellen zu fehlerhaften Berechnungen führen, kritisch sind in diesem Zusammenhang beispielsweise Berechnungen, bei denen zwei Vektoren auf Gleichheit geprüft werden müssen. Solche Positionsvektoren werden durch Schnittpunktberechnungen unter Verwendung trigonometrischer Operationen bestimmt und können dadurch leicht variieren. Das System verwendet aus diesem Grund gerundete Werte. Solche Rundungen führen dazu, dass Abweichungen im tausendstel Bereich nicht ins Gewicht fallen und Schnittpunkte als gleich angesehen werden. Allerdings geht mit jeder Rundung ein Rundungsfehler einher, der sich über eine Kette von Operationen akkumuliert, so dass bei späteren Rechnungen mit *falschen* Werten operiert wird. Prinzipiell ist das System tolerant, Abweichungen werden innerhalb eines Toleranzbereiches interpretiert, der durch den Nutzer angepasst werden kann. In den meisten Fällen führt dies zum richtigen Ergebnis, bei bestimmten Konfigurationen ist der aufsummierte Rundungsfehler in Kombination mit Floating-Point-Ungenauigkeiten aber unter Umständen so groß, dass es zu falschen Ergebnissen kommt und die Berechnung scheitert.

Als weitere Ursache für Ungenauigkeiten sind *Casts* zwischen den verschiedenen Datentypen, die Java zur Darstellung von Gleitpunktzahlen zur Verfügung stellt. Hierbei ist `Double` der genauere Datentyp, da er 64 Bit zur Kodierung von Gleitpunktzahlen verwendet, während `Float` nur 32 Bit nutzt. Casts, also Typkonvertierungen zwischen diesen verschiedenen Datentypen, sind erforderlich, da verschiedene eingesetzte Bibliotheken leider unterschiedliche Datentypen verwenden. So arbeitet die Java-Mathe-Bibliothek mit einer `Double`-Präzision, gleiches gilt für den verwendeten Matrix-Solver *Colt*[15], der verwendet wird, um Gleichungssysteme innerhalb des Mathemoduls zu lösen. Hierbei handelt es sich um eine kritische Komponente, da sie unter anderem zur Berechnung der Intersection-Events zum Einsatz kommt und dadurch in jeder Iteration eine zentrale Rolle spielt. Die verwendete Vektorbibliothek greift dagegen auf `Float` als Datentyp zurück, so dass an verschiedenen Stellen Konvertierungen erforderlich sind. Überraschenderweise wirken sich diese Konvertierungen nur sehr marginal auf die Genauigkeit aus. Vergleiche

[15] Colt: http://acs.lbl.gov/software/colt/

der Abweichungen bei über 20 verschiedenen Konfigurationen ergaben, dass die Unterschiede, sofern sie überhaupt vorlagen, erst im 10000stel Bereich auftraten. Außerdem ließ sich nicht belegen, dass ohne Casts die Abweichungen durchweg geringer wären, teilweise zeigten sich kleinere Abweichungen bei der Verwendung von `Float`. Da die Verwendung des `Double`-Datentyps keine Vorteile brachte, wurde `Float` als ressourcenschonenderer Datentyp beibehalten.

Dies hat zudem den Vorteil, dass für das Rendering keine Casts von `Double` nach `Float` durchgeführt werden müssen, da Processing ausschließlich `Float` für Vertexpositionsangaben akzeptiert. Dadurch spart man die vergleichsweise teuren Cast-Operationen [Si04] in jedem neuen Frame für jede einzelne Vektorkomponente. Gesetzt den Fall, dass für einen Grundriss aus 20 Vertices je nach Struktur durchschnittlich 60 Vertices errechnet werden, spart man somit für das Zeichnen des Ergebnisses 180 Cast-Operationen pro Frame, was durchaus einen Performancegewinn darstellt. Bei steigender Komplexität der Grundrisse und speziell bei mehr Berechnungsiterationen wächst auch die Anzahl der zu zeichnenden Vertices, so dass die Anzahl der Cast-Operationen ebenfalls zunimmt.

Die Vermeidung fehlerhafter Berechnungen durch akkumulierte Rundungsfehler ist schwierig bis unmöglich. Eine potentielle Quelle solcher Ungenauigkeiten sind Kindberechnungen bei Vertices, die keine Events ausgelöst haben. Vertices dieser Art erhalten ihre Kinder anhand von Berechnungen, die das Vertex in Richtung seiner Ebenenschnittgerade verschieben, bis es die Höhe erreicht, die durch die umgesetzten Events vorgegeben wird (man verwendet immer den oder die Events mit der geringsten Distanz und schrumpft das Polygon, bis alle Kindknoten diese Distanz erreichen). Leider führt der einfache Weg, also die tatsächliche Verschiebung der Vertices entlang ihrer Ebenenschnittgerade, nicht zu korrekten Ergebnissen. Tatsächlich ist festzustellen, dass die derart berechneten Kinder nicht exakt in einer Ebene liegen, die Abweichungen sind zwar relativ klein, aber dennoch vorhanden. Aus diesem Grund wird ein aufwendigerer Ansatz verwendet, bei dem man die beiden adjazenten Kanten eines Vertex in Richtung ihrer Slopeplane verschiebt, bis sie die gewünschte Höhe erreichen. Anschließend berechnet man auf dieser Höhe Schnittpunkte zwischen jeweils adjazenten Kanten und erhält dadurch die Position des neuen Kindknotens. Der Vorteil dieses Ansatzes besteht darin, dass Vertices, die direkte Nachbarn sind, immer auf der exakt gleichen Höhe liegen, da die sie verbindende Kante immer mit der gleichen Steigung verschoben wird.

Tatsächlich ergaben Tests mit den unterschiedlichen Verfahren, dass die entstehende Abweichung bei Einsatz des aufwendigen Verfahrens durchweg geringer ist, als bei einer einfachen Verschiebung der Vertices entlang ihrer Ebenenschnittgerade bis zum Erreichen der Zielhöhe. Die dadurch erreichte höhere Robustheit wird in diesem Fall als wichtiger erachtet als die Laufzeitvorteile des einfachen Ansatzes.

Da Berechnungsfehler aufgrund der vorab dargestellten Schwierigkeiten nicht vollständig vermieden werden können, wird zumindest versucht, sie zu erkennen. Tritt während der Verarbeitung ein beliebiger Fehler auf, prüft das System, ob er auf Abweichungsprobleme zurückzuführen sein kann. Hierfür wird ein einfacher Ansatz verwendet, bei dem man einen Strahl zwischen allen Eingabevertices und ihren ersten Kindern berechnet. Da sich Fehler mit steigender Iterationszahl weiter akkumulieren, sollte die erste Kindebene diejenige sein, die am ehesten einem idealen Verlauf entspricht. Nun durchläuft man die Hierarchie bis zu dem Punkt, an dem ein Vertex selber einen Event ausgelöst hat und berechnet den Abstand dieses Vertex zu der fast idealen Gerade. Dieser Abstand erlaubt eine ungefähre Abschätzung der akkumulierten Abweichungen und stellt somit ein Indiz für eine potentielle Ursache eines Berechnungsfehlers dar. Der Grund, weshalb man die Hierarchie nicht bis zum Blatt-Kind durchläuft, ist dabei die Tatsache, dass das Auftreten eines Events in einem Knoten zu einer Richtungsänderung der Geraden zwischen Kind und Elternknoten führt. Somit besitzt die Abweichung des Kindes eines Event-Vertex von der idealisierten Gerade keinerlei Bedeutung und bietet somit auch keinen Aufschluss über potentielle Fehlerursachen. Dieses Vorgehen ist für eine grobe Schätzung entstehender Abweichungen ausreichend.

11.7 **Dachbeispiele**

Nachdem im vorliegenden Abschnitt das verwendete Verfahren inklusive der vorgenommenen Anpassungen vorgestellt wurde, sollen in diesem Abschnitt abschließend einige Dachbeispiele gezeigt werden. Sämtliche nachfolgend präsentierten Dächer wurden mittels des Weighted-Straight-Skeleton-Verfahrens erzeugt und als 3D-Objekt aus dem Semantic Building Modeler exportiert. Danach wurden sie in 3ds Max geladen und gerendert. Dabei wurde explizit auf die Verwendung von Texturen verzichtet, um die generierten Dachformen besser erkennen zu können.

Abbildung 103 zeigt Renderings eines komplexen Dachgrundrisses mit zwei rechteckigen Erkern und zwei Zylinderkomponenten. Dieser Grundriss wurde durch den Objectplacement-Algorithmus prozedural erzeugt und anschließend durch den Semantic Building Modeler weiterverarbeitet. Die Eingabe in den Weighted-Straight-Skeleton-Algorithmus enthält unterschiedliche Kantengewichte, wie an den Dachneigungen gut zu erkennen ist. Die Konfiguration verwendet einen Neigungswinkel von ca. 90° für die Seiten des Gebäudes und einen Winkel von ca. 45° für die Hauptflächen. Dies ist speziell in der Draufsicht auf das Dach gut zu erkennen.

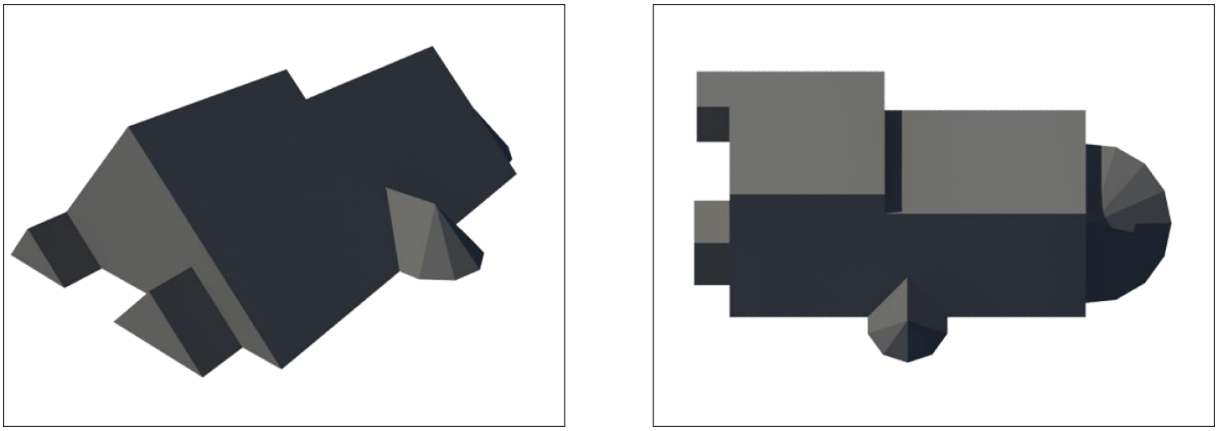

Abbildung 103: Komplexer Dachgrundriss mit unterschiedlichen Dachneigungswinkeln

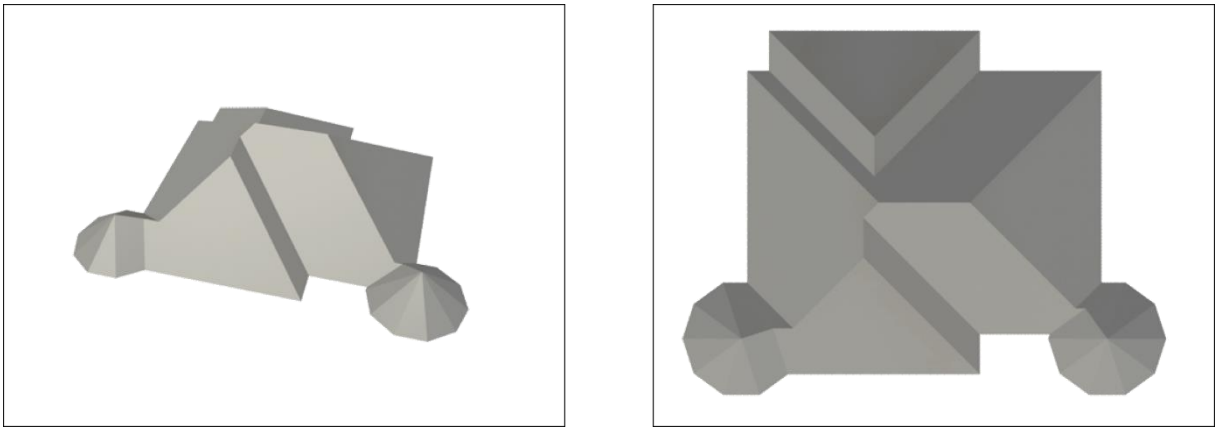

Abbildung 104: Komplexer Dachgrundriss mit identischen Kantengewichten

Abbildung 104 zeigt ein Dach, dessen Grundriss ebenfalls durch das Objectplacement-Verfahren erzeugt wurde. Im Gegensatz zur vorherigen Abbildung sind in diesem Fall alle Kanten gleich gewichtet.

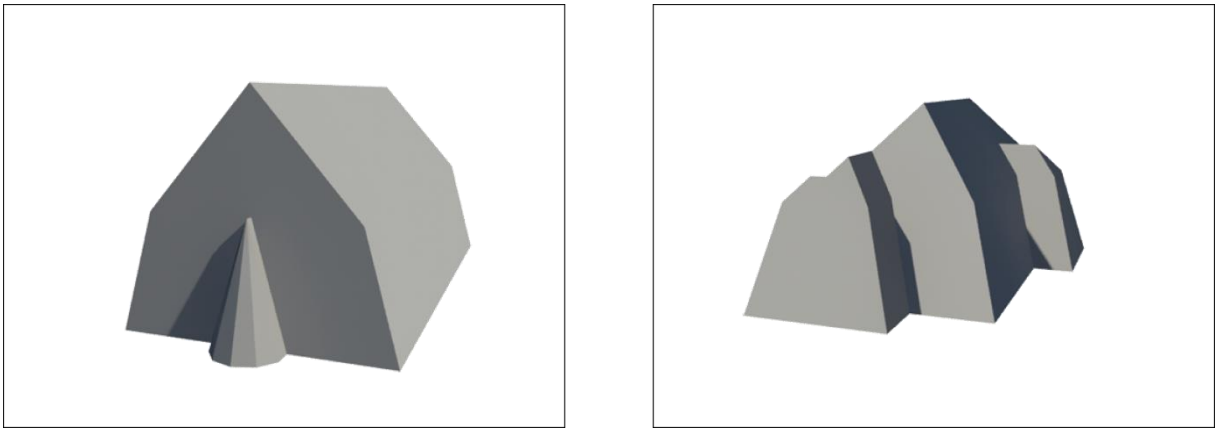

Abbildung 105: Mansardendach mit komplexem Grundriss

Abbildung 105 enthält Dachbeispiele, bei dem die vorgestellte Change-Slope-Event-Technik eingesetzt wurde, um die Dachneigung bei Erreichen einer nutzerdefinierten Höhe zu modifizieren. Offensichtlich ist es durch diese Technik möglich, auch Mansardendächer mittels Weighted-Straight-Skeleton-Algorithmus zu erzeugen. Dabei hat die Implementation den großen Vorteil, dass der Change-Slope-Event nahtlos in die reguläre Verarbeitung integriert werden kann. Dadurch bleibt die generische Natur des Verfahrens und somit dessen Einsetzbarkeit für beliebig strukturierte Grundrisse erhalten.

12 Semantische Gebäudekonstruktion

Eine Kernkompetenz des Semantic Building Modelers, die ihn von anderen Systemen zur prozeduralen Gebäudekonstruktion abgrenzt, ist die Fähigkeit der Anwendung, semantische Gebäudebeschreibungen zu verstehen und aus diesen 3D-Gebäudemodelle zu erzeugen. Dies unterscheidet die vorliegende Software von anderen Vertretern wie der CityEngine, da diese auf der Verwendung von Shape-Grammatiken basiert. Wie vorab erläutert, handelt es sich bei diesen Grammatiken um eine Form der Ersetzungssysteme, bei denen aber anstelle von Zeichenketten Formen ersetzt werden. Durch die wiederholte Anwendung eines nutzerdefinierten Regelsystems entsteht aus einem meist einfachen Massekörper ein komplexes Gebäude. Der Unterschied solcher Ansätze lässt sich am Vergleich des Semantic Building Modelers mit der CityEngine besonders gut herausarbeiten. Dies ist bereits bei vergleichsweise einfachen Basisstrukturen von Gebäuden gut erkennbar. Während der Nutzer im Semantic Building Modeler über die Angabe eines Wertes für `height` direkt eine Gebäudehöhe festlegt, tut er dies in der CityEngine zunächst über die Angabe der Höhe des Massekörpers. Intern arbeitet auch der Semantic Building Modeler mit einem solchen Körper, dies wird aber vor dem Nutzer versteckt. Ein weiteres Beispiel ist die Erzeugung von Stockwerken in den jeweiligen Systemen. Im Semantic Building Modeler definiert der Nutzer über eine Reihe von Parametern, welchen Grundriss und welche Höhe ein Stockwerk haben soll und an welcher Position im Gebäude es sich befindet. Hierfür stehen Parameter zur Verfügung, die bei der Gebäudebeschreibung durch den Nutzer festgelegt werden. Wiederum bildet die Konfiguration direkt die Semantik der Gebäudekonstruktion ab und ist somit deutlich näher an der Vorstellung, die der Anwender sich von den Beschreibungsparametern macht, die für die Stockwerkskonstruktion erforderlich sind. In der CityEngine erzeugt man Stockwerke dagegen unter Verwendung einer horizontalen Unterteilung des Massekörpers. Aus technischer Sicht ist dieser Ansatz gleichwertig, allerdings stellt er höhere Anforderungen an einen Anwender. Dieser muss von der Abstraktion eines Gebäudestockwerks umdenken auf dessen konkrete Konstruktion durch geometrische Operationen. Dies ist ein generelles Problem, unter dem grammatikbasierte, prozedurale Ansätze leiden. Sie setzen beim Nutzer die Fähigkeit voraus, sich die Konstruktion eines Gebäudes oder allgemein eines beliebigen 3D-Objekts als Abfolge von Ersetzungsoperationen vorzustellen. Der Semantic Building Modeler führt hier eine semantische Zwischenebene ein, die genau diese Voraussetzungen aufhebt. Dadurch gestattet das System dem Nutzer, sich die Gebäudekonstruktion auf eine Art und Weise

vorzustellen, die deutlich näher an seiner Erfahrungswelt liegt. Dies hat den Vorteil, dass es unerfahrenen Nutzern deutlich leichter fallen wird, sich mit dem Semantic Building Modeler auseinanderzusetzen und schnell Ergebnisse zu erzielen. Die Kosten dieser Einfachheit und deutlich besseren Verständlichkeit gehen aber zu Lasten der Flexibilität. Um neue Funktionen und Strukturen in das System zu integrieren, muss die Zwischenschicht um eben genau diese Funktionalität erweitert werden. Dies ist bei Systemen wie der CityEngine nicht erforderlich. Durch die Verwendung der Shape-Grammatiken ist es erfahrenen Nutzern möglich, beliebige Formen durch das System zu erzeugen.

Offensichtlich spielt bei einem System, das die Beschreibung eines Gebäudes durch eine Menge semantischer Parameter anstrebt, eine Komponente zur Verarbeitung und Validierung der nutzergenerierten Konfiguration eine zentrale Rolle. Da der Semantic Building Modeler kein grammatikbasiertes System darstellt, ist es für den Nutzer nicht möglich, neue Parameter zu definieren. Dies wäre zwar technisch durchaus umsetzbar, würde aber der Zielsetzung des Systems zuwider laufen. Nur durch die Kopplung vorgegebener Parameter an deren Verwendung im Programmcode lassen sich diese mit Semantik anreichern. Existiert eine solche Kopplung nicht, besitzen die Parameter auch keine Bedeutung. Sie können zwar wie in Shape-Grammatiken verwendet werden, um Werte zu speichern und kommen dadurch dem Variablenkonzept von Programmiersprachen nahe, allerdings ist der Nutzer selber dafür verantwortlich, eine Semantik zu definieren. Diese resultiert aus der Art, wie die Variable verwendet wird. Genau an dieser Stelle setzt der Semantic Building Modeler an. Anstatt vom Nutzer zu verlangen, eine solche Semantik durch die Festlegung einer Folge geometrischer Operationen implizit festzulegen, gibt er sie explizit vor. Die Bedeutung eines Parameters ist dadurch fest und durch den Nutzer nicht beeinflussbar. Ein Parameter `roofSlopeMain` wird nur verwendet, um die Steigung eines Daches an der Hauptseite des Gebäudes festzulegen. Die im System umgesetzten Algorithmen greifen auf diesen Parameter zu und verwenden ihn ausschließlich für die Dachkonstruktion. Es ist nicht möglich, ihn für irgendeine andere Funktionalität einzusetzen oder auszuwerten. Durch diese Festlegung und Abbildung im Programmcode entsteht die eindeutige Semantik, durch die sich der Semantic Building Modeler auszeichnet. Ein Nutzer hat eine direkte Vorstellung davon, welche Auswirkungen die Änderung des zugewiesenen Wertes haben wird, sie führt zu einem flacheren oder steileren Dach. Eine solche Vorstellung existiert in grammatikbasierten System nicht. Hier ist es zwar möglich, einen Variable zu deklarieren, die `roofSlopeMain` heißt und einen Wert aufnimmt, ob diese aber

für die Konstruktion des Daches oder die Wandstärke verwendet wird, hängt vollständig davon ab, wie der Nutzer sie einsetzt. Wiederum wird offensichtlich, dass die Fixierung der Semantik automatisch zu einer Reduktion der Freiheit des Nutzers führt. Allerdings erleichtert sie auch sein Verständnis und erlaubt es, sich auf die Struktur eines Gebäudes zu konzentrieren, anstatt sich über die erforderlichen, geometrischen Konstruktionsschritte Gedanken machen zu müssen.

12.1 **Ermittlung der Gebäudeparameter**

Dieser große Vorteil des Semantic Building Modeler gegenüber anderen System erfordert somit in einem ersten Schritt die Überlegung, welche Parameter man überhaupt benötigt, um ein Gebäude zu konstruieren. Dazu gehören neben offensichtlichen Parametern wie den Gebäudedimensionen auch eine Vielzahl weiterer, teils gebäudetypspezifischer Parameter. Dabei müssen nicht nur die Parameter selber ermittelt und in ihrer Struktur beschrieben werden, sondern unter Umständen auch Beziehungen zwischen diesen. Die Aufgabe, eine Menge von Parametern für die möglichst vollständige Beschreibung unterschiedlichster Gebäudetypen zu ermitteln, ist nicht trivial. Prinzipiell kann man hier zwei verschiedene Ansätze verfolgen.

Ein *Top-Down-Vorgehen* würde versuchen, eine Menge von Parametern zu bestimmen, die unabhängig vom beschriebenen Gebäudetyp sind und für die Beschreibung einer großen Anzahl unterschiedlicher Gebäude eingesetzt werden können. Offensichtliche Parameter, die über einen solchen Ansatz gefunden werden können, beschreiben beispielsweise die Ausdehnungen des Gebäudes. Allerdings stößt man mit einem solchen Ansatz schnell an Grenzen. Dies wird deutlich, wenn man sich überlegt, dass es sich nicht nur um Parameter handeln muss, die verschiedenen Gebäudetypen einer Zeitepoche gemein sein müssen, sondern möglichst auch über beliebige Zeitepochen relevant sind. Welche gemeinsamen Beschreibungskomponenten haben beispielsweise ein römisches Atriumhaus, ein Wohnhaus im Jugendstil und ein modernes Hochhaus? Erschwerend kommt hinzu, dass für die computerbasierte Konstruktion eines Gebäudes möglicherweise weitere Parameter relevant sein können, die bei der Betrachtung der Gebäude zunächst nicht offensichtlich sind.

Aus diesem Grund wurden die meisten zur Verfügung stehenden Parameter über einen *Bottom-Up-Ansatz* entwickelt. Bei diesem Vorgehen geht man von einem konkreten Gebäudetyp aus und ermittelt sämtliche Parameter, die für die Konstruktion von Instanzen

dieses Typs erforderlich sind. Durch die Implementation spezifischer Gebäudetypen lassen sich so einerseits Parameter aufdecken, die für die Berechnung des konkreten Typs notwendig sind, andererseits findet man auch weitere Parameter, die zumindest einer Gruppe von Gebäuden gemeinsam sind, wenn nicht sogar allgemein gültig für alle Gebäude. Aus diesem Grund wurden zunächst verschiedene Gebäudetypen im System umgesetzt, anhand derer es anschließend möglich war, eine Menge von Parametern zu extrahieren, die Teil des Konfigurationssystems wurden.

Die Konfigurationskomponente selber ist innerhalb des Systems als eigenständiges und völlig unabhängiges Model implementiert. Eine solche Modularisierung ist aus verschiedenen Gründen wünschenswert und hilfreich. Hier ist zunächst eine deutlich höhere Wiederverwendbarkeit der Komponente zu nennen. Da der Semantic Building Modeler selber ebenfalls aus einer Reihe unterschiedlicher Submodule besteht, lag der Wunsch nahe, diese möglichst unabhängig voneinander entwickeln und testen zu können. Dabei sollte es aber möglich sein, sämtliche Module über ein einheitliches Konfigurationssystem zu steuern, gleichgültig ob dies nun das Modul für die Dachberechnung oder das Modul für die ähnlichkeitsbasierte Grundrisserzeugung war. Die Konfigurationen sämtlicher Module wurden darum in einem zentralen Konfigurationsmodul zusammengeführt. Dieses kann an die einzelnen Module gekoppelt werden, die sich die für ihre Arbeit erforderlichen Konfigurationsparameter aus dem Konfigurationsmodul laden. Eine solche lose Kopplung beschleunigt zum einen die Entwicklung der Submodule und erleichtert zum andern die spätere Zusammenführung der Module innerhalb eines einzigen Systems, dem Semantic Building Modeler. Aus dieser Überlegung heraus wurde darum ein Konfigurationssystem entwickelt, das XML-basierte Konfigurationen lesen und verarbeiten kann. Anschließend stellt es die Konfigurationen über standardisierte Schnittstellen als Service für gekoppelte Module bereit. Nachfolgend soll auf die Struktur des Konfigurationssystems und eine Reihe zentraler Bestandteile der Konfigurationen selber eingegangen werden.

12.2 **Das Konfigurationsmodul**

Das Konfigurationsmodul erfüllt im Kontext des Semantic Building Modeler drei zentrale Aufgaben, auf die im folgenden Abschnitt detailliert eingegangen wird. Die erste Aufgabe besteht im einfachen Laden eines XML-Dokuments, das eine Menge von Konfigurationsparametern enthält. Im zweiten Schritt wird dieses Dokument mittels eines XML Schema-Dokuments validiert. Sofern die Gültigkeitsprüfung erfolgreich war, liest das

Modul sämtliche Parameter aus dem Dokument und stellt diese anschließend anderen Modulen zur Verfügung.

12.2.1 **Schritt 1: Laden des XML-Dokuments**

Der erste Schritt in der Verarbeitungsabfolge des Konfigurationsmoduls besteht im Laden eines vorgegebenen XML-Dokuments. XML wurde aus verschiedenen Gründen als Konfigurationsformat gewählt. Ein wichtiger Aspekt ist die Tatsache, dass XML nicht nur für Maschinen, sondern auch für Menschen lesbar und verständlich ist. Die hierarchische Baumstruktur eines XML-Dokuments ermöglicht die Zusammenfassung und Gruppierung logisch zusammenhängender Komponenten und ist darum gut geeignet, hierarchische Konfigurationsstrukturen auszudrücken. Darüber hinaus kann XML als Textformat bereits mit rudimentären Textverarbeitungssystemen editiert werden. Allerdings existieren auch sehr fortschrittliche XML-Systeme, die den Nutzer bei der Editierung von XML-Dokumenten unterstützen. Beispielsweise zu nennen sind hier der *<oXygen/> XML Editor*[16] der Firma *Synchro Soft* oder *XML Spy*[17] der Firma *Altova*. Diese Werkezuge sind unter anderem in der Lage, Dokumente über vorhandene XML Schemas zu validieren, was im Kontext des Semantic Building Modelers eine wichtige Rolle spielt. Speziell für das Verfassen komplexer XML-Dokumente, die über umfangreiche Vokabulare verfügen, sind solche Softwarewerkzeuge unabdingbar, da sie durch Funktionalitäten wie *Code Completion* und die direkte Auswertung von angeschlossenen XML Schemas dem Nutzer einerseits Hilfestellungen bei der Auswahl erforderlicher Elemente geben und ihn andererseits darauf hinweisen können, sobald das XML-Dokument einen Fehler enthält, der zu einem Scheitern der Validierung führen würde. Prinzipiell ist es zwar möglich, Konfigurationen für den Semantic Building Modeler mit einem einfachen Texteditor zu erstellen, allerdings benötigt man ohne die unterstützenden Funktionen von XML-Editoren deutlich mehr Zeit. Neben der Menschenlesbarkeit hat der XML-Standard eine Reihe weiterer Vorteile, die bereits zu Beginn dieser Arbeit intensiv erörtert wurden. Aus diesem Grund soll an dieser Stelle auf eine erneute Diskussion des Standards verzichtet werden. Das Hauptargument für die Verwendung von XML als Konfigurationsformat ist allerdings der XML Schema-Standard, mittels dessen XML-Dokumente validiert werden können. Auf die Mächtigkeit dieser Technik und die Vorteile, die sie bietet, wird in Schritt 2 weiter eingegangen.

[16] <oXygen/>: http://www.oxygenxml.com/
[17] Altova XMLSpy: http://www.altova.com/de/xmlspy.html

Eine sehr wichtige Eigenschaft des Konfigurationsmoduls ist es, dass dieses die Herkunft der Daten, die es anderen Modulen zur Verfügung stellt, vor diesen versteckt. Für ein angeschlossenes Modul ist es nicht möglich zu erkennen, ob die Parameter, die der Konfigurationsservice bereit hält, von der lokalen Festplatte des Computers stammen oder von einer entfernten Ressource geladen wurden. Der Zugriff auf die vorhandenen Parameter erfolgt immer über die gleichen Methoden. Das Modul implementiert beide Ansätze, es kann XML-Dokumente von der lokalen Festplatte lesen oder diese über das Internet von anderen Computern zunächst herunterladen und anschließend verarbeiten. Im zweiten Fall gibt der Nutzer die URL an, an welcher sich das zu ladende XML-Dokument befindet. Diese Technik ist eine wichtige Säule, über die das System den Aufbau einer *verteilten Bibliothek* von Gebäudekonfigurationen ermöglicht. In einem ersten Schritt gestattet sie es nämlich, eine Startkonfiguration von einem potentiell entfernten Rechner zu laden. Innerhalb einer solchen Startkonfiguration ist es dann möglich, weitere externe Ressourcen zu referenzieren und diese miteinander zu kombinieren. Darauf wird an späterer Stelle im Kontext der Verarbeitung eingegangen. In diesem Zusammenhang werden auch die Sprachmittel vorgestellt, die der Semantic Building Modeler für diesen Zweck zur Verfügung stellt.

12.2.2 **Schritt 2: Validieren des XML-Dokuments**

Wie vorab erwähnt, ist die Möglichkeit, XML-Dokumente mittels XML Schema zu validieren, ein Hauptgrund für die Wahl von XML als Konfigurationsformat. Die Entwicklung von XML Schema als Nachfolger für den DTD-Standard wurde bereits ausführlich geschildert. Dabei wurde auch auf die Vorteile des neueren Standards eingegangen, speziell das ausführliche Datentypkonzept in Kombination mit unterschiedlichen Vererbungsmechanismen macht XML Schema zum Mittel der Wahl. Dies gilt umso mehr, je komplexer die XML-Dokumente sind, die validiert werden sollen.

Prinzipiell existieren für Anwendungen, die auf komplexe, nutzergenerierte Konfigurationen angewiesen sind, zwei unterschiedliche Möglichkeiten der Gültigkeitsprüfung. Die erste Variante ist die Überprüfung der Eingabeparameter im Programmcode. Bis zur Entwicklung von XML Schema war dies für XML-basierte, komplexe Konfigurationen das einzig wirklich einsetzbare Mittel, um zu garantieren, dass die angegeben Werte gültig waren. Dies hängt damit zusammen, dass DTDs über kein differenziertes Typkonzept verfügen, so dass es nicht möglich ist zu überprüfen, ob ein Wert den Anforderungen entspricht. DTDs können ausschließlich testen, ob ein Wert vorhanden ist, weitere Aussagen können nicht

validiert werden. Die Validierung im Programmcode hat eine Reihe gravierender Nachteile. Zunächst macht sie diesen komplexer und umfangreicher, was automatisch auch die Fehleranfälligkeit des Systems erhöht. Dies hängt damit zusammen, dass die Überprüfung der Gültigkeit eines Wertes voraussetzt, dass dem System bekannt ist, welche Werte gültig sind und welche nicht. Ein naives Vorgehen würde diese Festlegung direkt innerhalb des Programmcodes vornehmen. Dort würde der Wert eines bestimmten Parameters ausgelesen, anschließend würde man ihn in einen bestimmten Datentyp umwandeln, um dann zu testen, ob er innerhalb eines gültigen Wertebereichs liegt. Ändert sich für einen Parameter nun im Laufe der Entwicklung der Datentyp oder stellt man fest, dass eine Änderung des Wertebereichs erforderlich ist, so muss man diese Änderungen direkt im Programmcode vornehmen. Dies wiederum ist für Nutzer ohne Programmiererfahrung unmöglich, kann aber auch für erfahrene Entwickler sehr aufwendig werden. Elaboriertere Ansätze können dieses Problem lösen, indem man den Datentyp und Wertebereich eines Parameters durch eine eigene Beschreibungssprache ausdrückt, die dann innerhalb der Software automatisiert verarbeitet werden kann. Dadurch wäre es möglich, Anpassungen an Parameterspezifikationen ohne Eingriff in den Programmcode vorzunehmen. Allerdings sind die Entwicklung einer Beschreibungssprache und auch die Implementation der Verarbeitungslogik innerhalb der Software nicht trivial und zeitaufwendig. Neben dem Implementationsaufwand hat die Integration der Parameterspezifikation in die Software einen weiteren gravierenden Nachteil. Dieser besteht darin, dass die Spezifikation unabhängig von konkreten Nutzerkonfigurationen ist und somit für sämtliche Anwender Gültigkeit besitzt. Der vorab genannte Parameter `roofSlopeMain` ist immer eine Gleitpunktzahl, deren Wertebereiche alle positiven, rationalen Zahlen umfasst. Eine nachträgliche Änderung ist somit für alle Nutzer relevant und nicht nur für eine lokale Instanz der Software. Daraus erwächst automatisch der Wunsch, die Parameterspezifikationen an einem zentralen Ort vorzuhalten und nur dort zu modifizieren. Tut man dies nicht, beispielsweise weil die Spezifikationen im Programmcode implementiert ist oder in Form einer lokalen Konfigurationsdatei vorliegt, erfordert eine nachträgliche Anpassung ein Update sämtlicher bereits ausgelieferter Softwareinstanzen. Je mehr Nutzer bereits mit dem vorliegenden System arbeiten, desto aufwendiger wird ein solcher Aktualisierungsprozess.

Die Verwendung von XML Schema löst eine ganze Reihe der vorab genannten Probleme. Durch das ausgefeilte Datentypkonzept ist es möglich, nicht nur den Typ eines Parameters

festzulegen, sondern auch differenzierte Aussagen über dessen Wertebereich zu treffen. Auf die zur Verfügung stehenden Sprachmittel in XML Schema wurde dabei zu Beginn dieser Arbeit detailliert eingegangen, darum wird an dieser Stelle auf eine erneute Diskussion der Ansätze zur Ableitung durch Einschränkung oder Erweiterung verzichtet. Es sei aber festgehalten, dass eine Festlegung, wie diese am Beispiel des `roofSlopeMain`-Parameters vorab genannt wurde, in XML Schema problemlos umsetzbar ist. Daneben stellt XML Schema weitere Technologien bereit, die in der Lage sind, nutzergenerierte Werte mit Hilfe regulärer Ausdrücke oder Wertelisten zu validieren. Die Implementation solcher Ansätze innerhalb des Systems selber ist aufwendig und dadurch fehleranfällig. Der Rückgriff auf eine bewährte Technologie wie XML Schema und die weit verbreiteten und robusten Java-Bibliotheken zur Verarbeitung von XML und XML Schema stellen hier einen deutlich besseren und effektiveren Weg dar.

Neben dem Vorhandensein eines solchen umfangreichen Datentypkonzepts eignen sich XML Schema–Dokumente auch gut für eine zentralisierte Wartung. Die verwendete XML-Bibliothek *JDOM* wird seit dem Jahr 2000 [jd] entwickelt und ist ein sehr robustes Werkzeug. Sie bietet eine umfangreiche Funktionalität unter anderem für das Laden entfernter XML Schema-Dokumente und kann diese direkt verwenden, um beliebige XML-Dokumente zu validieren. Hierbei handelt es sich um eine der Kernfunktionen des Konfigurationsmoduls des Semantic Building Modelers.

12.2.3 **Schritt 3: Verarbeiten des geladenen XML-Dokuments**

Nachdem das XML-Dokument geladen und validiert wurde, kann es durch die Software weiterverarbeitet werden. Hierfür greift man auf die Baumrepräsentation zurück, die durch die JDOM-Bibliothek beim Einlesen des Dokuments erzeugt wurde. Dieser Baum kann mit Hilfe der bereitgestellten Methoden durchlaufen und ausgewertet werden. Da das Modul nicht ausschließlich für die Verarbeitung von Stadtbeschreibungen dient, sondern auch für die Konfiguration der unterschiedlichen Submodule des Semantic Building Modelers eingesetzt wird, stellt es unterschiedliche Verarbeitungsmethoden bereit, die sich darin unterscheiden, welche Art von Konfiguration sie verarbeiten. Jede dieser Methoden validiert das ihr übergebene XML-Dokument mit dem passenden Schema und erzeugt für dieses eine Baumrepräsentation. Anschließend wird eine Instanz einer Konfigurationsklasse erzeugt, der die Wurzel des XML-Baumes übergeben wird. Jede Konfigurationsklasse weiß dabei, wie sie die ihr übergebenen Strukturen verarbeiten muss und ist dadurch in der Lage, das

Übergabedokument auszuwerten und die Parameter in Form von Variablen der Konfigurationsklasse zu speichern. Hierbei ist festzuhalten, dass auch die interne Repräsentation der Konfiguration deren hierarchische Struktur widerspiegelt. Dies sei am Beispiel einer Städtekonfiguration illustriert.

Abbildung 106 zeigt eine schematische Darstellung des `sbm_ci:city`-Elements. Dieses ist das Wurzelelement jeder Stadt-Konfiguration. Es muss mindestens eine Instanz des `sbm_ci:buildingDescriptor`-Elements enthalten.

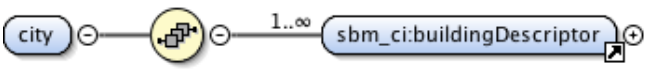

Abbildung 106: Schematische Darstellung des City-Schemas

Jedes `sbm_ci:buildingDescriptor`-Element beschreibt die Struktur für genau einen Gebäudetyp. Zur Auswahl stehen in der vorliegenden Version ein vollkommen frei definierbares Gebäude, ein Gebäude im Jugendstil und eine Gebäudeklasse, die Tempel dorischer Ordnung mit einem Doppelantentempel-Grundriss konstruiert. Bei den frei definierbaren Gebäuden existieren keinerlei Vorgaben innerhalb der Software. Es ist dem Anwender vollkommen freigestellt, welche Technologien er verwendet, um das jeweilige Gebäude zu erzeugen. Dies gilt auch für die eingesetzten Gebäudekomponenten. Ein solches freies Gebäude wird innerhalb eines `sbm_bu:building`-Elements konfiguriert und bietet dem Anwender im Vergleich zu den typspezifischen Gebäudeklassen die größte Flexibilität. Typischerweise nimmt mit der größeren Freiheit in Bezug auf die Gebäudespezifikation auch die Anzahl der Parameter zu, die durch den Nutzer festgelegt werden müssen.

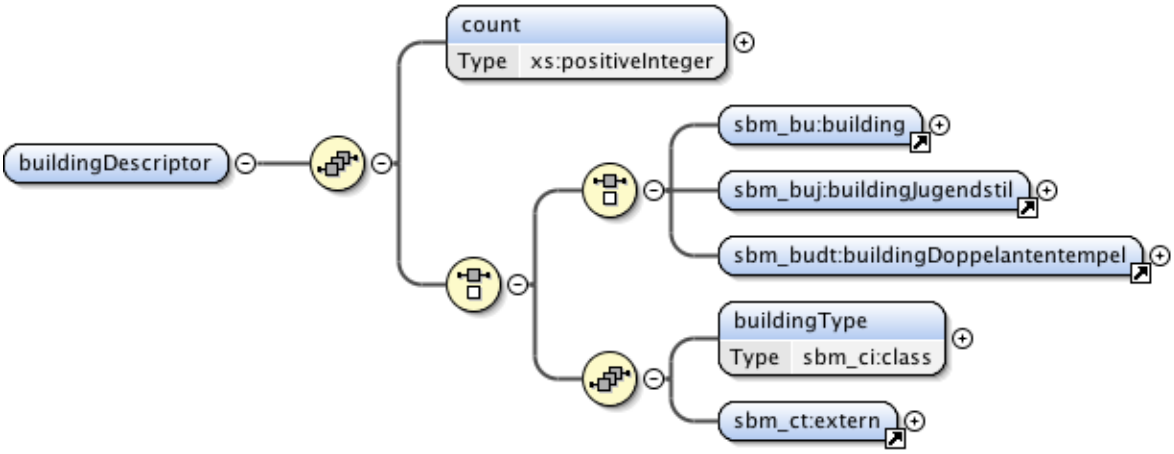

Abbildung 107: Schematische Darstellung der `sbm_ci:buildingDescriptor`-Komponente des City-Schemas

Spezifischere Gebäudetypen, wie sie durch `sbm_buj:buildingJugendstil`- oder `smcb:budt:buildingDoppelantentempel`-Elemente deklariert werden, enthalten durch den speziellen Typ bereits eine Reihe von Festlegungen. Diese können durch den Nutzer nicht geändert werden. Dadurch reduziert sich die Anzahl der Parameter, die der Nutzer festlegen muss und somit die Komplexität der Konfiguration.

Abbildung 107 zeigt eine schematische Darstellung des `sbm_ci:buildingDescriptor`-Elements. Neben dem Gebäudetyp enthält dieses eine Angabe der Anzahl an Gebäuden, die durch das System für den jeweiligen Typ errichtet werden sollen. Auf die einzelnen Elemente wird an späterer Stelle noch differenzierter eingegangen. Wichtig für die Verarbeitungsabfolge ist, dass innerhalb des Konfigurationsmoduls jede der in den vorherigen Abbildungen gezeigten Hierarchiestufen innerhalb der internen Repräsentation des Moduls durch eine Menge von Objekten abgebildet wird. Auf der obersten Ebene befindet sich eine Instanz der `CityConfiguration`-Klasse, die mindestens eine Instanz der `BuildingDescriptor`-Klasse enthält. Jede Instanz der `BuildingDescriptor`-Klasse beinhaltet für den jeweiligen Gebäudetyp, den sie konstruieren soll, eine typspezifische Klasseninstanz. Dieses Schema geht hinunter bis auf die Ebene der tatsächlichen Parameter.

Die Abbildung der XML-Struktur in Form einer Java-Klassenhierarchie hat für die Verarbeitung innerhalb der Module, die mit dem Konfigurationsmodul kommunizieren, eine Reihe wichtiger Vorteile. Als Alternative könnte man auch direkt auf dem von JDOM erzeugten Baum arbeiten, dies ist allerdings aus verschiedenen Gründen ungünstig. Zunächst ist hier aus softwaretheoretischer Sicht eine Kapselung der verwendeten Backendstrukturen als großer Vorteil zu sehen. Die Module, die auf den Konfigurationsservice zugreifen, besitzen keinerlei Wissen über die dahinterliegenden Strukturen. Weder kennen sie die verwendete XML-Bibliothek noch ist ihnen bekannt, dass es sich um eine XML-basierte Konfiguration handelt. Dadurch ist es möglich, die vollständige Konfigurationsstruktur umzustellen, beispielsweise auf Serialisierungen der Java-Objekte oder alternative Konfigurationsformate wie beispielsweise die *JavaScript Object Notation*[18] (JSON) oder *YAML Ain't Markup Language*[19] (YAML). Eine Anpassung der Module, die den Service verwenden, ist nicht erforderlich. Dies ist der erste wichtige Grund, weshalb der Konfigurationsservice mehr leistet, als ausschließlich den verarbeiteten XML-Baum an die anderen Module weiterzureichen. Der zweite Grund besteht in den zusätzlichen

[18] JSON: http://www.json.org/
[19] YAML: http://www.yaml.org/

Verarbeitungsschritten, die innerhalb der unterschiedlichen Konfigurationsklassen durchgeführt werden. Die Bandbreite dieser Verarbeitung reicht von einfachen Parameterberechnungen bis hin zum vollständigen Nachladen entfernter XML-Dokumente, die zusätzliche Konfigurationen enthalten. Durch die Integration dieser zusätzlichen Aufgaben in das Konfigurationsmodul werden diese vollständig vor den anderen Modulen versteckt, was zu einer sauberen Modularisierung führt und dadurch sowohl die Wartbar- als auch die Wiederverwendbarkeit erhöht.

12.2.3.1 **Reflection im Konfigurationsmodul**

Eine wichtige Zielsetzung des gesamten Semantic Building Modelers und des Konfigurationsmoduls im Speziellen ist eine möglichst einfache Erweiterbarkeit des Systems. Dies gilt sowohl für die Integration neuer Gebäudetypen im Hauptmodul als auch für die entsprechenden Konfigurationsklassen im Konfigurationsservice. Die Basistechnik, die hierfür eingesetzt wird, ist die Reflection-Fähigkeit von Java, auf die zu Beginn dieser Arbeit eingegangen wurde. Kern dieser Technologie ist die Metaklasse `Class`, deren Instanzen eine Vielzahl an Informationen über die Struktur von Klassen enthalten. Dazu gehören beispielsweise die vorhandenen Felder und Methoden, aber auch Konstruktoren oder Superklassen. Darüber hinaus stellt `Class` mit der `newInstance`-Methode eine Möglichkeit bereit, ohne genau Kenntnis der konkreten Klassenstruktur Instanzen einer beliebigen Klasse zu erzeugen. Diese Technik ist Kern vieler konfigurierbarer Plug-In-Architekturen und ebenso zentraler Bestandteil des Semantic Building Modelers.

Betrachtet man Abbildung 107 genauer, so fällt auf, dass neben den konkreten Gebäudetypen eine weitere Alternative zur Verfügung steht, die aus einem Element `sbm_ci:buildingType` und einem Element `sbm_ct:extern` besteht. Auf das zweite Element wird später eingegangen, vereinfacht gesagt erlaubt es die Angabe einer beliebigen Ressource, die vom System geladen wird. Dazu gehören an dieser Stelle ausschließlich Gebäudetypen. Damit das System weiß, welche Konfigurationsklasse es für einen solchen Gebäudetyp verwenden muss, ist es erforderlich, diesen explizit anzugeben. Hierfür sind die Werte im Element `sbm_ci:buildingType` erforderlich. Dieses enthält einen Eintrag aus einer Aufzählungsliste über den es möglich ist, mittels Reflection Instanzen der jeweils angeforderten Gebäudekonfigurationsklassen zu erzeugen. Soll später eine zusätzliche Konfigurationsklasse für einen neuen Gebäudetyp hinzugefügt werden, so muss diese zunächst implementiert werden. Anschließend reicht es aus, einen Bezeichner für die neue

Klasse in eine Mapstruktur einzutragen, die die Beziehung zwischen Bezeichner und `Class`-Instanzen der Konfigurationsklassen herstellt. Weitere Anpassungen am Programmcode sind nicht erforderlich.

Ein weiterer Einsatzort für Reflection ist die Verarbeitung der XML-Konfigurationen. Vorab wurde die Fähigkeit von XML hierarchische Strukturen abzubilden, als großer Vorteil bezeichnet. Für die Übersetzung dieser XML-Baumhierarchie in eine Hierarchie von Klasseninstanzen, die unterschiedliche Teile der Konfiguration repräsentieren, eignet sich der Reflection-Ansatz ebenfalls sehr gut. Das Konfigurationsmodul ist dabei so strukturiert, dass für einen Großteil der Konfigurationsbestandteile, die selber keine Blattknoten sind, eigene Konfigurationsklassen implementiert wurden. Diese sind von einer gemeinsamen abstrakten Basisklasse `AbstractConfigurationObject` abgeleitet, was eine einheitliche Verarbeitung und Speicherung ermöglicht. Beim Durchlaufen der Kindelemente eines beliebigen Knotens innerhalb der JDOM-Baumrepräsentation prüft das System, ob es sich beim vorliegenden Knoten um ein Blatt oder einen inneren Knoten handelt. Sofern ein innerer Knoten gefunden wurde, wird getestet, ob es eine Konfigurationsklasse gibt, die für die Verarbeitung dieses Knotens zuständig ist. Wenn dies der Fall ist, wird mittels Reflection eine Instanz der gefundenen Klasse erzeugt, dieser wird der innere Knoten als Wurzelknoten übergeben und die Verarbeitung kann fortfahren. Dadurch lässt sich die Größe der Codebasis deutlich reduzieren, was wiederum zu einer verringerten Fehleranfälligkeit und dadurch zu einer erhöhten Robustheit führt.

12.3 **Ausgewählte Elemente des XML Schemas**

Nachdem im vorherigen Abschnitt auf die Schritte eingegangen wurde, die das Konfigurationsmodul bei der Verarbeitung eines XML-Dokuments durchführt, sollen nachfolgend exemplarisch zentrale Komponenten des XML Schemas für die Stadtkonstruktion vorgestellt und ihre Funktion erläutert werden.

Die oberste Ebene der Konfigurationshierarchie wurde bereits im vorherigen Abschnitt thematisiert und durch Abbildung 106 visualisiert. Die Wurzel einer solchen Konfiguration ist immer das `sbm_ci:city`-Element, welches eine beliebige Anzahl von Gebäudedeskriptoren enthält. Jeder Gebäudedeskriptor definiert einen Gebäudetyp und die Anzahl an Instanzen dieses Gebäudetyps, die durch den Semantic Building Modeler erzeugt werden sollen. Diese Struktur ist Gegenstand von Abbildung 107. Für die Vorstellung der

weiteren vorhandenen Strukturen wird nun exemplarisch die Konfiguration eines freien Gebäudes vorgestellt. Auf die Besonderheiten der anderen Gebäudetypen wird anschließend eingegangen.

12.3.1 **Konfiguration des allgemeinen Gebäudetyps `sbm_bu:building`**

Der allgemeine Gebäudetyp ist derjenige, der dem Nutzer die größten Freiheiten bezüglich der Gebäudekonstruktion überlässt. Er trifft keinerlei Vorannahmen über typspezifische Gebäudekomponenten und –strukturen. Darum können für diesen Typ sämtliche vorhandenen Verfahren und Algorithmen eingesetzt werden, die innerhalb des Systems implementiert sind. Dies gilt sowohl für die unterschiedlichen Technologien zur Grundrisserzeugung und –modifikation als auch für Verfahren zur Erzeugung von Komponenten wie Gesimse oder Dächern. Aufgrund dieses großen Funktionsumfangs ist dieser Gebäudetyp am besten geeignet, um einen Überblick über die vorhandenen Konfigurationsstrukturen zu bieten.

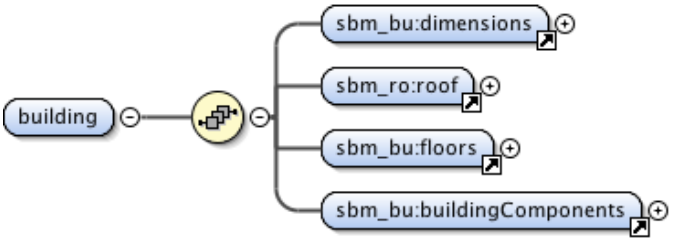

Abbildung 108: Schematische Darstellung der `sbm_bu:building`-Komponente

Abbildung 108 zeigt eine schematische Darstellung der `sbm_bu:building`-Komponente. An dieser erkennt man sehr gut die grundsätzliche Struktur, in der Gebäude innerhalb des Semantic Building Modelers definiert werden. Jedes Gebäude dieses allgemeinen Typs wird beschrieben durch seine Dimensionen (`sbm_bu:dimensions`), eine Konfiguration für die Berechnung des Dachs (`sbm_ro:roof`), mindestens eine Stockwerkskonfiguration innerhalb des Stockwerkcontainers (`sbm_bu:floors`) sowie eine Menge von Gebäudekomponenten, wie Türen, Fenster oder Gesimse (`sbm_bu:buildingComponents`).

12.3.1.1 **Gebäudeausdehnungen – `sbm_bu:dimensions`**

Die `sbm_bu:dimensions`-Komponente ist das einzige Element innerhalb des Systems, deren Parameter vollständig durch einen Top-Down-Ansatz ermittelt wurden. Hierbei handelt es

sich um Parameter, die sämtlichen bisher implementierten Gebäudetypen gemein sind und mit hoher Wahrscheinlichkeit auch für eine Vielzahl noch offener Gebäudetypen Gültigkeit besitzen. Die Werte für Länge (`height`) und Breite (`width`) des Gebäudes werden für die Skalierung und Berechnung der Grundrisse eingesetzt, die Höhe (`height`) für die Bestimmung der Stockwerkshöhen. Ebenfalls allen aktuell vorhandenen Gebäudetypen gemein ist ein Parameter (`wallThickness`), der für die abschließende Erzeugung von Wänden mit einer bestimmten Breite eingesetzt wird.

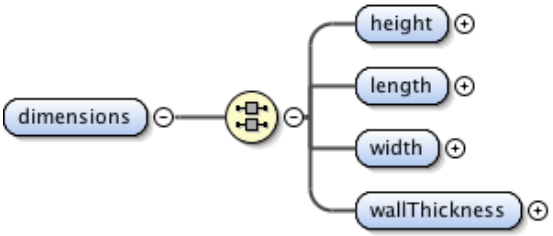

Abbildung 109: Schematische Darstellung der `sbm_bu:dimensions`-Komponente

12.3.1.2 Dachkonfiguration – sbm_ro:roof

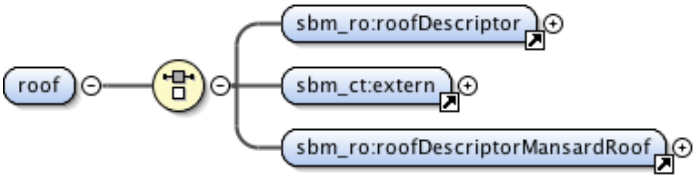

Abbildung 110: Schematische Darstellung der `sbm_ro:roof`-Komponente

Abbildung 110 zeigt die Struktur des `sbm_ro:roof`-Elements. Dieses wird für die Konfiguration von Dächern eingesetzt, die mittels des vorab beschriebenen Weighted-Straight-Skeleton-Verfahrens für beliebige Grundrisse errechnet werden. Bei einer solchen Konfiguration hat der Anwender drei unterschiedliche Elemente zur Verfügung, aus denen er ein Einzelnes auswählen muss. Die allgemeine Form der Dachberechnung basiert auf den Kernparametern, die für das Weighted-Straight-Skeleton-Verfahren erforderlich sind. Eine solche Dachform wird innerhalb des `sbm_ro:roofDescriptor`-Elements konfiguriert. Das Element enthält unter anderem Parameter für die Steigung an der Dachhaupt- und Dachnebenseite. Diese werden innerhalb des Verfahrens verwendet, um die Steigung der Ebenen an den Dachkantenkanten festzulegen. Über diese Parameter ist es möglich, eine Vielzahl unterschiedlicher Dachformen zu erzeugen. Darüber hinaus ist es aufgrund der

algorithmischen Berechnung nicht erforderlich, auf Besonderheiten der jeweiligen Grundrisse zu achten. Das Verfahren ist unabhängig von der Struktur des Dachgeschossgrundrisses und kann für beliebige Grundrissstrukturen eingesetzt werden. Dies ist von zentraler Bedeutung, da durch die unterschiedlichen Verfahren für Grundrisskonstruktion und –modifikation und der teils randomisierten Berechnungen a priori nicht vorhersehbar ist, wie der fertige Grundriss des Dachgeschosses aussehen wird.

Eine spezielle Dachform, die durch die Standardparameter nicht realisiert werden kann, ist das Mansardendach. Bei diesem handelt es sich um ein „geknicktes Dach zur besseren Ausnutzung des Dachraumes“ [Me08]. Die Besonderheiten bezüglich der Konstruktion eines solchen Dachs wurden im Kontext des Weighted-Straight-Skeleton-Verfahrens vorgestellt. Zentraler Unterschied ist die Tatsache, dass bei einem Mansardendach für jede Seite zwei Steigungen angegeben werden müssen. Zusätzlich muss der Nutzer die Höhe festlegen, an der der Steigungswechsel stattfindet. Basierend auf diesen Parametern kann das System anschließend Mansardendächer für beliebige Eingabegrundrisse berechnen. Die letzte zur Auswahl stehende Kindkomponente des `sbm_ro:roof`-Elements ist das `sbm_ct:extern`-Element. Dieses Element ist ein Element, das nicht nur innerhalb des `sbm_ro:roof`-Elements vorkommen darf, sondern auch an verschiedenen anderen Stellen innerhalb der Schemadokumente. Außerdem spielt es eine zentrale Rolle für das Konzept einer verteilten Konfigurationsbibliothek und wird darum im nachfolgenden Abschnitt detailliert beschrieben.

12.3.1.3 **Einbinden externer Konfigurationskomponenten – `sbm_ct:extern`**

Vorab wurde bereits an verschiedenen Stellen erwähnt, dass das Konfigurationsmodul in der Lage ist, entfernte Komponenten zu laden. Ein Ansatz wurde im Kontext der Verarbeitungsschritte des Konfigurationsmoduls bereits erläutert. Er besteht im vollständigen Laden einer Stadtkonfiguration von einem entfernten Server. Hierbei gibt der Nutzer über eine URL die Position der Zielkonfiguration an, die anschließend durch den Service heruntergeladen und temporär gespeichert wird. Nachfolgend wird sie dann über die Mechanismen für lokal vorhandene Konfigurationen verarbeitet und zur Stadtkonstruktion genutzt. Diese Technik alleine ist für eine verteilte Konfiguration nicht ausreichend, da sie nur das Laden ganzer Konfigurationen erlaubt. Dabei ist der Einsatz nicht nur auf Stadtkonfigurationen beschränkt, vielmehr können auch Konfigurationen für die unterschiedlichen Submodule geladen und eingesetzt werden. Dies ist zwar ein erster

wichtiger Schritt, allerdings wäre es wünschenswert, nicht nur vollständige Konfigurationen laden und einsetzen zu können, sondern auch beliebige Konfigurationsteile miteinander zu kombinieren und dadurch eine eigene Stadtkonfiguration basierend auf einer beliebigen Anzahl verschiedener Gebäudekonfigurationen zusammenzusetzen. Hat ein Nutzer eine Konfiguration für einen bestimmten Gebäudetyp erstellt, so kann er sie online verfügbar machen und es anderen Anwendern erlauben, diese Konfiguration für ihre eigenen Städte zu verwenden. Dadurch werden Konfigurationen wiederverwendbar und können problemlos zwischen Nutzern ausgetauscht werden. Je mehr Nutzer einen solchen Mechanismus verwenden, desto einfacher wird es für den einzelnen Anwender, eine Stadtkonfiguration zu erstellen. Sehr sinnvoll wäre in diesem Zusammenhang der Aufbau eines Webservice, der als zentrales Repository fungiert, in dem Anwender ihre Konfigurationen ablegen und beispielsweise mit einer Reihe von Schlagwörtern beschreiben können. In Kombination mit einer Suchfunktion wäre es dann möglich, sehr schnell eine potentiell große Anzahl von Konfigurationen zu durchsuchen. Als Ergebnis könnte das System dann einfach eine Reihe von URLs zu passenden Dokumenten liefern, die der Nutzer direkt in seine eigene Stadt einbauen könnte.

Kern dieser Technologie ist das `sbm_ct:extern`-Element, welches in Abbildung 111 exemplarisch dargestellt wird. Dieses Element hat einen Datentyp, der über Ableitung durch Einschränkung aufgrund des Basisdatentyps `xs:anyURI` erzeugt wurde und ausschließlich Verweise in das lokale Dateisystem (beginnend mit `file://`) oder auf Webressourcen (beginnend mit `http://`) akzeptiert. Sobald innerhalb einer Konfiguration ein solches Element gefunden wird, verwendet das Servicemodul die Funktionen der JDOM-Bibliothek und lädt die entfernte Ressource. Das geladene Konfigurationsfragment wird automatisch validiert, so dass auch in diesem Fall die Gültigkeit der Konfiguration garantiert werden kann.

Abbildung 111: Schematische Darstellung des `sbm_ct:extern`-Elements

Aktuell ist das `sbm_ct:extern`-Element an verschiedenen Stellen innerhalb des Moduls zulässig. Die nachfolgende Tabelle gibt einen Überblick über die validen Elternelemente und erläutert den jeweiligen Einsatzzweck. An den Einträgen in Tabelle 11 erkennt man, dass der Einsatz des `sbm_ct:extern`-Elements größtenteils für das Nachladen potentiell

umfangreicher Fragmente eingesetzt wird. Dazu gehören neben vollständigen Gebäudekonfigurationen auch Parameterlisten, die für die verschiedenen Verfahren zur Grundrissberechnung verwendet werden.

Elternelement	Funktion
`sbm_ci:buildingDescriptor`	Gebäudebeschreibungen
`sbm_ro:roof`	Dachkonfiguration
`sbm_fl:floor`	Stockwerkskonfiguration
`sbm_fp:footprint`	Grundrisskonfiguration
`sbm_ct:polygon`	Polygonbeschreibung bestehend aus einer Liste von mindestens drei Vertices
`sbm_ct:modelFile`	Verweis auf den Speicherort einer 3D-Modell-Datei
`sbm_co:component`	Konfiguration einer Gebäudekomponente beliebigen Typs
`sbm_buj:objectplacementConfiguration`	Konfiguration für den Objectplacement-Algorithmus für die Grundrisserzeugung / -modifikation

Tabelle 11: Einsatz des `sbm_ct:extern`-Elements zum Nachladen externer Konfigurationsfragmente

Speziell im Zusammenhang mit diesen Algorithmen ist die Wiederverwendung von Konfigurationen sinnvoll, da die Verfahren stark parametrisiert sind. Sofern eine Konfiguration entwickelt wurde, die Grundrisse eines bestimmten Typs generiert, ist es sinnvoll, diese vorzuhalten und wiederzuverwenden.

Neben der Möglichkeit zum Aufbau einer verteilten Bibliothek von Konfigurationsfragmenten erlaubt das `sbm_ct:extern`-Element die Aufteilung und Modularisierung eigener Konfigurationen in Form verschiedener Dateien. Speziell bei umfangreichen Konfigurationen mit einer Vielzahl unterschiedlicher Gebäudetypen ist es sehr hilfreich, einzelne Fragmente in separate Dateien auszulagern. Dadurch können auch innerhalb eines lokalen Städteprojekts Komponenten wiederverwendet werden. Dies

wiederum erhöht die Wartbarkeit des eigenen Projekts, da die Größe der Konfiguration abnimmt, Änderungen nur noch an einer zentralen Stelle vorgenommen werden müssen und auf bereits getestete Konfigurationen zurückgegriffen werden kann.

12.3.1.4 **Stockwerksdefinition – sbm_bu:floors**

Die nächste wichtige Komponente für eine Gebäudekonfiguration ist die Konfiguration der Stockwerke. Hierfür steht als Container-Element das `sbm_bu:floors`-Element zur Verfügung. Dieses muss mindestens die Konfiguration für ein einzelnes Stockwerk enthalten, maximal sind drei Stockwerksbeschreibungen möglich. Abbildung 112 zeigt die schematische Darstellung dieses Elements.

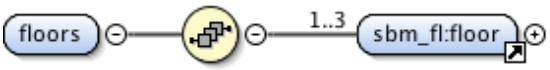

Abbildung 112: Schematische Darstellung des `sbm_bu:floors`-Element

Für die Festlegung der Stockwerksstrukturen existieren zwei verschiedene Ansätze. Entweder man definiert ein Stockwerk und verwendet diese Definition für sämtliche innerhalb des Gebäudes zu erzeugenden Stockwerke oder man erstellt unterschiedliche Stockwerkskonfigurationen für die verschiedenen Stockwerkstypen. Welche Variante verwendet wird, teilt man dem System durch den Wert des `sbm_fl:floorposition`-Elements mit. Weist man diesem den Wert `ALL` zu, wird die angegebene Konfiguration für alle Stockwerke verwendet. Sonst kann man durch die Werte `GROUND`, `INTERMEDIATE` und `TOP` anzeigen, für welche Stockwerke innerhalb des Gebäudes die jeweilige Konfiguration als Konstruktionsplan eingesetzt werden soll.

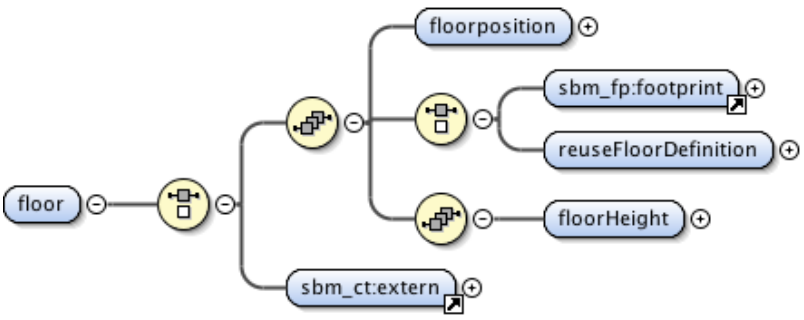

Abbildung 113: Schematische Darstellung des `sbm_fl:floor`-Elements

Abbildung 113 zeigt die Struktur des `sbm_fl:floor`-Elements. Wie bereits vorab erwähnt, können über das `sbm_ct:extern`-Element vollständige Stockwerksdefinitionen von externen Quellen geladen werden. Daneben stehen verschiedene Möglichkeiten zur Festlegung der Stockwerksstruktur zur Verfügung. Definiert man ein Stockwerk von Grund auf, so ist neben der Angabe der Position des Stockwerks innerhalb des Gebäudes und der Höhe die Auswahl eines Verfahrens zur Grundrisserzeugung zentral. Hier stehen verschiedene Verfahren zur Verfügung, aus denen gewählt werden kann. Auf diese wird im nachfolgenden Abschnitt detaillierter eingegangen. Die letzte Möglichkeit für die Festlegung eines Stockwerks ist die Wiederverwendung eines vorab berechneten Grundrisses. Hierfür greift man auf das `sbm_fl:reuseFloorDefinition`-Element zurück. Diesem weist man einen Wert aus einer vordefinierten Aufzählung zu. Anstatt einen neuen Grundriss zu berechnen, greift das System während der Stockwerkserzeugung auf eine Datenstruktur zurück, in der die vorab verwendeten Grundrisse gespeichert sind und indiziert diese mit Hilfe des angegeben Aufzählungselements.

12.3.1.5 **Grundrisserzeugung – sbm_fp:footprint**

Im vorherigen Abschnitt wurde das `sbm_fp:footprint`-Element bereits kurz angesprochen. Es dient der Festlegung von Grundrissstrukturen beziehungsweise der Auswahl und Konfiguration von Verfahren, die für die Generierung von Grundrissen zur Verfügung stehen. Die Fähigkeit, unterschiedliche Algorithmen zur Erzeugung beliebiger Grundrisse einsetzen zu können, ist eine der Stärken des Semantic Building Modelers und eine zentrale Komponente zur Erzeugung visueller und geometrischer Vielfalt in den berechneten Gebäudemodellen. Die verschiedenen Möglichkeiten, aus denen der Nutzer auswählen kann, sind in Abbildung 114 anhand der möglichen Kindelemente erkennbar. Zunächst besteht auch für die Grundrissbeschreibung die Möglichkeit, vollständige Grundrisskonfigurationen mittels des `sbm_ct:extern`-Elements aus anderen Quellen zu laden und innerhalb der Verarbeitung auszuwählen.

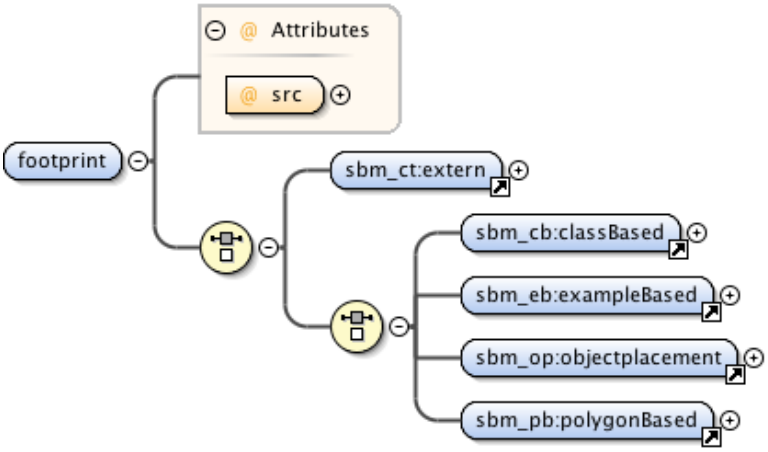

Abbildung 114: Schematische Darstellung des `sbm_fp:footprint`-Elements

Auch bei der Verwendung einer externen Grundrissbeschreibung hat man die Wahl zwischen den vier Varianten, die innerhalb der Abbildung zu erkennen sind. Das `sbm_cb:classBased`-Element dient zur Konfiguration von Grundrissen, die innerhalb des Systems als Programmcode vorliegen. Dies ist beispielsweise bei der Grundrissdefinition für Doppelantentempel-Typ der Fall, da dieser Gebäudetyp zusätzlich zur rechteckigen Grundform über eine Innenraumstruktur verfügt. Diese Innenraumstruktur wird innerhalb des Codes realisiert, indem Raum- und Portalstrukturen festgelegt werden. Die Verwendung eines solchen vordefinierten Grundrisses erfolgt über die Angabe des Klassennamens der Klasse, die den Grundriss implementiert. Die zweite nicht-algorithmische Variante für die Grundrisserzeugung ist das direkte Laden von Grundrissen aus 3D-Modell-Dateien. In diesen müssen die Grundrisse als geschlossener Polygonzug vorliegen. Das System lädt die Dateien, extrahiert die geometrischen Strukturen und überführt diese in die intern verwendete Repräsentationsform. Da innerhalb des `sbm_pb:polygonBased`-Elements ein `sbm_ct:extern`-Element für die Angabe des Ortes verwendet wird, an dem die Modelldatei liegt, kann auch diese von beliebigen Orten geladen werden und muss nicht lokal auf der Festplatte des Nutzers liegen. Auch vorgefertigte Grundrisse, die als serialisierte Polygone vorhanden sind, können auf diese Art und Weise in einen Webservice integriert und anderen Nutzern zur Verfügung gestellt werden. Neben diesen beiden statischen Varianten, bei denen der Grundriss in seiner Struktur bereits existiert und nur durch das System skaliert werden muss, verfügt der Semantic Building Modeler über zwei weitere prozedurale Verfahren, mittels derer Grundrisse neu erzeugt oder im Falle des Objectplacement-Verfahrens auch modifiziert werden können. Die Verfahren wurden vorab bereits intensiv erläutert, für eine detaillierte Darstellung sei darum auf die betreffenden Abschnitte verwiesen. Prinzipiell handelt es sich bei beiden Technologien um parametrisierte

Verfahren, die über eine Vielzahl unterschiedlicher Parameter gesteuert werden. Das Objectplacement-Verfahren kann dabei sowohl eingesetzt werden, um neue Grundrisse zu erzeugen, aber auch um vorherige Grundrisse zu modifizieren. Da durch die Positionierung zusätzlicher Subkomponenten im Normalfall die ursprüngliche Struktur erhalten bleibt, bietet sich dieses Verfahren auch für die oberen Stockwerke eines Gebäudes an. Diese Annahme ist allerdings nur unter der Voraussetzung gültig, dass die finale Bestimmung des Stockwerkgrundrisses nicht auf einem Verfahren zur Berechnung konvexer Hüllen basiert. In diesem Fall kann eine Kompatibilität der Grundrisse nicht garantiert werden. Dies ist bei der ähnlichkeitsbasierten Grundrisserzeugung meist auch der Fall. Dieses Verfahren verwendet den Eingabegrundriss ausschließlich zur Ermittlung einer initialen Regelmenge. Anschließend erfolgt die Regelauswahl entweder sequentiell oder heuristisch, je nachdem, welche Synthesevariante der Nutzer ausgewählt hat. In den meisten Fällen wird das Ergebnis der Synthese inkompatibel mit dem Eingabegrundriss sein, da der Algorithmus nur Grundrisse erzeugt, die der Eingabe ähneln. Allerdings können diese in Bezug auf Größe und Form stark vom Basispolygon abweichen und sind somit nur bedingt geeignet für die Verwendung als Grundrisse für Zwischen- und Obergeschosse.

12.3.1.6 **Gebäudekomponenten – sbm_bu:buildingComponents**

Der letzte Teil der Gebäudekonfiguration sind die Gebäudekomponenten, die der Nutzer festlegen muss. Gebäudekomponente ist hierbei ein Sammelbegriff für sämtliche Bestandteile, die nicht durch die Grundrissextraktion und Stockwerksgenerierung erzeugt werden. Dazu gehören sowohl Komponenten wie Fenster oder Türen, die direkt aus 3D-Modell-Dateien geladen und anschließend appliziert werden, als auch prozedural berechnete Elemente wie Gesimse oder Fensterbänke, die durch die Extrusion eines Querschnitts generiert werden. Bei der Festlegung der Gebäudeelemente kann der Nutzer eine beliebige Anzahl solcher Komponenten spezifizieren.

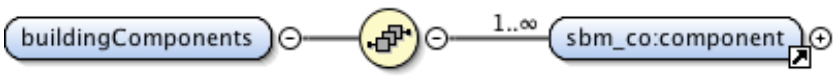

Abbildung 115: Schematische Darstellung des `sbm_bu:buildingComponents`-Elements

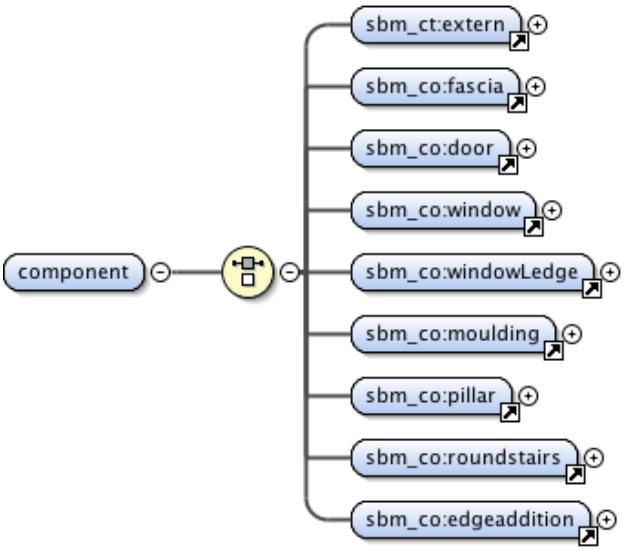

Abbildung 116: Schematische Darstellung des `sbm_co:component`-Elements

Abbildung 115 zeigt die schematische Darstellung des `sbm_bu:buildingComponents`-Elements. Hierbei handelt es sich um ein Container-Objekt, das eine beliebige Anzahl einzelner Komponentenbeschreibungen aufnehmen kann. Die aktuell vorhandenen Komponenten und die für diese verwendeten Elemente sind in Abbildung 116 dargestellt.

Tabelle 12 listet alle acht vorhanden Komponentenelemente auf und enthält für jeden Typ einen kurzen Verweis auf das verwendete Erzeugungsverfahren.

Jede dieser Komponenten enthält als Kindelemente eine Reihe unterschiedlicher Parameter, die für die Erzeugung benötigt werden. Während sich die Parameter bei den prozedural erzeugten Elementen aufgrund unterschiedlicher Erzeugungsalgorithmen voneinander unterscheiden, ist sämtlichen Objekten, die aus 3D-Modellen geladen werden, das `sbm_co:componentModelSource`-Element als Kind gemeinsam.

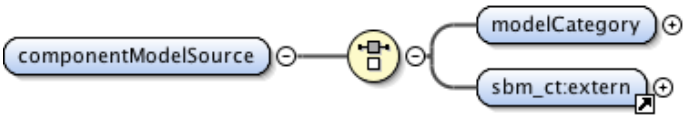

Abbildung 117: Schematische Darstellung des `sbm_co:componentModelSource`-Elements

Dieses wird in Abbildung 117 gezeigt. Die Quelle eines 3D-Modells kann demnach auf zwei unterschiedliche Arten angegeben werden. Zum einen ist es möglich, den Lademechanismus zu verwenden, dessen Ziel über eine URI-Angabe innerhalb des `sbm_ct:extern`-Elements angegeben wird. Auch für Komponenten-3D-Modelle ist es demnach möglich, diese sowohl von der lokalen Festplatte als auch von entfernten Servern zu laden und in die eigene Gebäudekonstruktion zu integrieren. Im Zusammenhang mit Komponentenmodellen steht

darüber hinaus noch eine weitere Variante zur Verfügung, die zwar konzeptuell einfach, aber speziell für die Erzeugung variabler Gebäude sehr effizient ist. Die Kernidee hinter diesem Ansatz ist es, Modelle und Texturen in Kategorien zusammenzufassen. Jede Kategorie kann eine beliebige Anzahl an Elementen eines bestimmten Typs enthalten.

Komponente	Schema-Element	Herkunft
Fascia	`sbm_co:fascia`	Prozedurale Berechnung basierend auf einem Fascia-Querschnitt
Tür	`sbm_co:door`	3D-Modell
Fenster	`sbm_co:window`	3D-Modell
Fensterbank	`sbm_co:windowledge`	Prozedurale Berechnung basierend auf einem Fensterbank-Querschnitt
Gesimse	`sbm_co:moulding`	Prozedurale Berechnung basierend auf einem Gesimse-Querschnitt
Säule	`sbm_co:pillar`	3D-Modell
Stufenfundament	`sbm_co:roundstairs`	Prozedurale Berechnung basierend auf einem initialen Grundriss, aus dem durch wiederholte Skalierung und Extrusion Treppenstufen erzeugt werden.
Kantenverzierung / -hervorherbung	`sbm_co:edgeadditions`	Prozedurale Berechnung basierend auf einem Querschnitt, der in Richtung einer Menge von Zielkanten extrudiert wird

Tabelle 12: Unterstützte Komponenten

Anstatt nun ein konkretes Modell über dessen Dateinamen anzugeben, ermöglicht der Semantic Building Modeler die Festlegung einer Kategorie, aus der das System ein Modell

auswählen soll. Die Auswahl erfolgt zufallsbasiert. Legt ein Nutzer eine Gebäudekonfiguration über einen Gebäudedeskriptor fest, bietet ihm der kategorienbasierte Ansatz eine Möglichkeit, um über die Komponentenfestlegung automatisch Variationen in den Instanzen des beschriebenen Gebäudetyps zu erzeugen. Je größer dabei die Kategorien sind, je mehr unterschiedliche Modelle sie enthalten, desto größer wird auch die Variation, die das System erzeugt. Auch in diesem Zusammenhang bietet sich die Implementation eines webbasierten Services an, innerhalb dessen Kategorien inklusive der zugehörigen Modelle und Texturen zusammengefasst und geteilt werden können. Dabei wäre es nicht zwingend erforderlich, die eigentlichen Elemente, also Texturen oder 3D-Modelle auf den Webservern selber vorzuhalten. Diese könnten auch als reine URI-Sammlungen fungieren, aus denen sich der Semantic Building Modeler für eine angefragte Kategorie eine Menge von URIs auswählt und erst in einem nächsten Schritt über die implementierten Mechanismen die tatsächlichen Modelle nachlädt. In der aktuellen Version müssen allerdings noch sämtliche Elemente einer Modellkategorie lokal auf der Festplatte des Anwenders vorhanden sein, damit die Auswahl funktioniert. Aus technischer Sicht stellt aber eine Umstellung auf eine webbasierte Komponentenauswahl keine Hürde dar.

12.3.2 **Konfiguration spezieller Gebäudetypen**

Nachdem im vorherigen Abschnitt auf den allgemeinen Gebäudetyp und die wichtigsten vorhandenen Komponenten für dessen Konfiguration eingegangen wurde, widmet sich dieser Abschnitt den speziellen Gebäudetypen und den Unterschieden in der Konfiguration. Anhand des implementierten Tempels sollen die grundsätzlichen Unterschiede in der Gebäudespezifikation herausgearbeitet werden, die sich in der Art und Menge der festzulegenden Parameter niederschlagen. Allgemein kann man vorab bereits festhalten, dass mit steigender Spezifität eines Gebäudes die Freiheitsgrade für dessen Konfiguration abnehmen. Betrachtet man vergleichend einen Doppelantentempel mit einem anderen spezifischen Gebäudetyp innerhalb des Semantic Building Modelers, dem Jugendstilgebäude, so zeigt sich diese Tendenz sehr deutlich. Während für den allgemeinen Gebäudetyp des vorherigen Abschnitts noch sämtliche implementierten Verfahren für die Grundrisserzeugung zur Verfügung standen, kann für Jugendstilgebäude nur noch das Objectplacementverfahren eingesetzt werden. Darüber hinaus bietet der Jugendstilgebäudetyp eine Reihe typspezifischer Parameter, die beispielsweise für die Steuerung der Grundrissmodifikation eingesetzt werden. Hier existieren verschiedene

Parameter, über die der Nutzer angeben kann, mit welcher Wahrscheinlichkeit der rechteckige Basisgrundriss modifiziert wird und ob eine Wiederverwendung der Basisgrundrisse auf den höheren Stockwerken erfolgen soll. Diese Parameter werden als Kindelemente des `sbm_buj:buildingJugendstil`-Elements angegeben, welches in Abbildung 118 zu sehen ist.

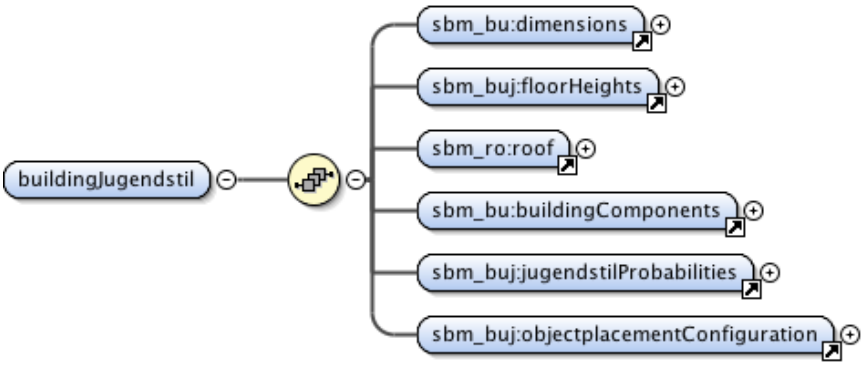

Abbildung 118: Schematische Darstellung des `sbm_buj:buildingJugendstil`-Elements

Daneben erkennt man in der Abbildung, dass sich das Jugendstil- und das allgemeine Gebäude nur das `sbm_ro:roof`-Element, die Gebäudedimensionen im `sbm_bu:dimensions`-Element und die Gebäudekomponenten im `sbm_bu:buildingComponents`-Element teilen. Die restlichen Komponenten sind gebäudetypspezifisch und auf die Besonderheiten der Jugendstil-Gebäudestruktur zugeschnitten. Dies macht es für einen Anwender leichter, eine Konfiguration für ein solches Gebäude festzulegen, da die Anzahl der erforderlichen Parameter geringer ist. Außerdem ist die Wahrscheinlichkeit inkompatibler Stockwerksgrundrisse durch die Einschränkung auf das Objectplacement-Verfahren bei der Stockwerksmodifikation deutlich geringer. Somit ist davon auszugehen, dass es für einen unerfahrenen Nutzer einfacher sein wird, ein Jugendstilgebäude errechnen zu lassen, als ein allgemeines Gebäude vollständig zu definieren. Festzuhalten ist allerdings, dass der allgemeine Gebäudetyp durchaus verwendet werden kann, um Jugendstilgebäude zu errichten, da er auf sämtliche Berechnungsverfahren zurückgreifen kann. Umgekehrt ist dies aufgrund des eingeschränkten Parametersatzes des Jugendstilgebäudes nicht der Fall. Beispielsweise wird durch die Festlegung auf das Objectplacementverfahren zur Grundrissmodifikation bereits eine ganze Reihe potentieller Gebäudeformen ausgeschlossen.

Noch kleiner werden die Konfigurationsmöglichkeiten des Gebäudes, sobald man die Gebäudeklasse des Doppelantentempels betrachtet. Dieser griechische Tempel hat einen

festgelegten Grundriss, durch den er sich von anderen griechischen Tempeltypen wie dem Prostylos oder dem Tholos abhebt [SP04]. Da die Form des Grundrisses eines der definierenden Merkmale für diesen Tempeltyp ist, macht der Einsatz grundrissmodifizierender Algorithmen keinen Sinn.

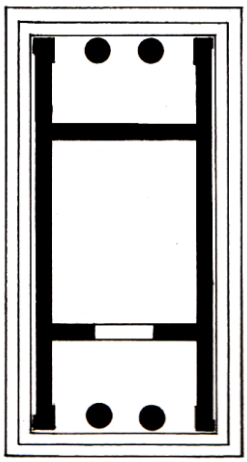

Abbildung 119: Grundriss eines Doppelantentempels [SP04]

Abbildung 119 zeigt den Grundriss eines Doppelantentempels. In der aktuellen Implementation wird eine Sonderform dieses Tempeltyps umgesetzt. Diese verwendet den gezeigten Grundriss, ergänzt ihn aber um eine umlaufende Säulenhalle. Dadurch ist es möglich, auch andere Grundrisstypen in ihren Strukturen zu implementieren und diese als Basis für die Konstruktion weiterer Tempeltypen zu verwenden.

Nicht nur in Bezug auf die Grundrissfestlegung ist die Konfigurierbarkeit des Doppelantentempels im Vergleich zu den anderen Gebäudetypen kleiner.

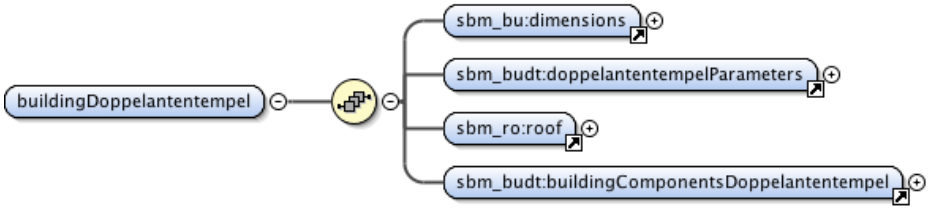

Abbildung 120: Schematische Darstellung des `sbm_budt:buildingDoppelantentempel`-Elements

Betrachtet man die schematische Darstellung des `sbm_budt:buildingDoppelantentempel`-Elements in Abbildung 120, so teilt sich dieses mit dem Wurzelelement des allgemeinen Gebäudetyps nur noch das `sbm_bu:dimensions`-

und das `sbm_ro:roof`-Element. Die Spezifikation von Stockwerksstrukturen, beispielsweise durch die Festlegung von Parametern für die Grundrissalgorithmen oder die Angabe von Stockwerkspositionen, macht im Kontext eines Tempelbaus wenig Sinn. Aus diesem Grund existiert auch für diesen Typ eine Reihe spezifischer Parameter, die direkt aus der architektonischen Beschreibung hervorgehen. Der Nutzer gibt bei der Konfiguration demnach nicht die Höhe eines Stockwerks an, sondern spezifiziert die Höhe der Metope oder des Architravs.

Wiederum ist dies ein gutes Beispiel für die Verwendung semantisch bedeutsamer Parameter, mit denen ein mit der Struktur griechischer Tempel kundiger Nutzer problemlos umgehen kann. Intern handelt es sich um eine weitere Abstraktion der geometrischen Operationen, die der Semantic Building Modeler bei der Gebäudekonstruktion einsetzt. Das System verwendet intern die gleichen Strukturen, die es auch für die Erzeugung allgemeiner Gebäude nutzt. So ist ein Architrav für den Semantic Building Modeler nichts anderes als ein Stockwerk mit rechteckigem Grundriss und nutzerdefinierter Höhe. Durch die Verwendung semantischer Parameter ist dies allerdings für den Anwender vollständig transparent. Konzeptuell handelt es sich hierbei um eine weitere Abstraktionsschicht, die für die Konstruktion des Tempels eingesetzt wird.

Auch in Bezug auf die Gebäudekomponenten, die für einen Tempel verwendet werden können, macht das System eine Reihe von Einschränkungen. Bei der Betrachtung von Abbildung 121 zeigt sich, dass für einen Tempel dieses Typs nur vier der eigentlich acht implementierten Komponententypen zur Verfügung stehen. Dabei ist es im Gegensatz zum freien Gebäude nicht möglich, beliebig aus den vorhandenen Komponenten zu wählen. Vielmehr ist es für die Erzeugung einer validen Konfiguration für Doppelantentempel zwingend erforderlich, sämtliche in der Abbildung erkennbaren Gebäudeelemente vollständig festzulegen.

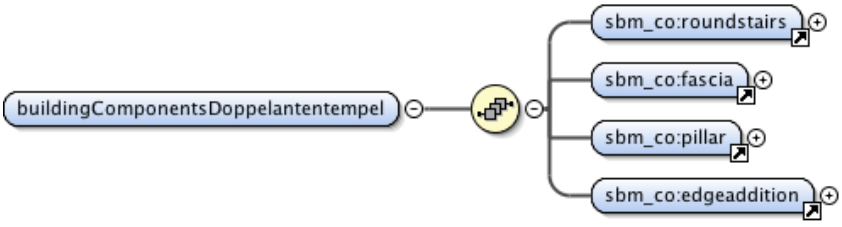

Abbildung 121: Schematische Darstellung des `scmb_budt:buildingComponentsDoppelantentempel`-Elements

An diesem Beispiel zeigt sich, wie spezielle Gebäudetypen verwendet werden können, um den Nutzer bei der Erstellung von Konfigurationen zu unterstützen. Einerseits ermöglichen die strukturellen Besonderheiten der jeweiligen Gebäude typischerweise eine Reduzierung der Parametermenge, die durch den Nutzer spezifiziert werden muss. Andererseits ist es möglich, bestimmte Festlegungen zu erzwingen, die charakteristisch für die jeweiligen Strukturen sind.

12.4 Fazit zu Konfigurationsmodul und XML Schema

Der vorherige Abschnitt befasste sich mit der technischen Implementation und Funktionalität des Konfigurationsmoduls des Semantic Building Modelers, ging in diesem Zusammenhang zusätzlich auch auf die wichtigsten Elemente ein, die bei der XML-basierten Konfiguration von Stadtmodellen eingesetzt werden können. Besondere Bedeutung hat die Verwendung von XML Schema zur Beschreibung der Konfigurationsstruktur des Systems. XML Schema eignet sich aufgrund des umfangreichen Datentypkonzepts hervorragend für die Validierung nutzergenerierter Strukturen und macht dadurch eine Überprüfung der Konfigurationsparameter innerhalb des Systems weitgehend unnötig, was die Codekomplexität reduziert und dadurch zu einer höheren Robustheit des Systems führt. Um einen besseren Eindruck der Konfigurationsstruktur zu erhalten, wurden die zur Verfügung stehenden Hauptkomponenten für den allgemeinen Gebäudetyp vorgestellt und bei zentralen Komponenten auf deren Funktion eingegangen. Dabei verblieb die Diskussion meist auf der Ebene komplexer Elemente und ging nicht herab bis zu den tatsächlichen Parameterspezifikationen. Eine vollständige Dokumentation sämtlicher vorhandener Schema-Elemente ist online verfügbar unter: Für eine vollständige Dokumentation aller innerhalb des Systems vorhandener Parameter ist online verfügbar unter http://pgunia.github.io/SemanticBuildingModeler_Documentation/.

13 Gebäudebeispiele

Nachdem in den vorherigen Kapiteln die Systemstruktur, die entwickelten Algorithmen und die XML-basierte Konfiguration des Semantic Building Modelers vorgestellt wurden, werden in diesem Abschnitt Beispiele für die durch das System erzeugten Modelle präsentiert. Die hierfür verwendeten, vollständigen Konfigurationsdateien sind online verfügbar unter https://github.com/pgunia/SemanticBuildingModeler_Ressources/tree/public/.

Die Konfigurationen dienten der Erstellung von 3D-Gebäudemodellen durch den Semantic Building Modeler. Diese wurden anschließend aus der Software in das OBJ-Format exportiert und in das 3D-Modellierungsprogramm Autodesk 3ds Max geladen. Innerhalb dieses Programms wurden die einzelnen Gebäude manuell angeordnet und zu einer einfachen Stadtszene zusammengesetzt. Sämtliche innerhalb der Szenen sichtbaren Komponenten, bei denen es sich nicht um Gebäude handelt (beispielsweise Vegetation oder die verwendete Grundebene), wurden ebenfalls manuell in 3ds Max erzeugt. Komponenten dieser Art werden nicht durch den Semantic Building Modeler generiert, dieser ist nur für die Produktion der Gebäudegeometrie zuständig.

Das Rendering selber erfolgt unter Verwendung der *NVIDIA Mental Ray*-Engine. Diese Render-Engine wird von *Mental Images* entwickelt, einer Unterabteilung von NVIDIA. Mental Ray ist bereits seit vielen Jahren fester Bestandteil von 3ds Max und wird kontinuierlich weiterentwickelt [Bo13]. Die Renderengine implementiert verschiedene Algorithmen für die physikalisch korrekte Berechnung globaler Beleuchtung.

Vereinfacht gesagt unterscheiden sich globale von lokalen Beleuchtungsmodellen darin, dass bei lokalen Modellen nur die direkte Interaktion eines Objektes mit den Lichtquellen einer Szene berücksichtigt wird, um die Beleuchtung an einem beliebigen Oberflächenpunkt des Objekts zu bestimmen. Indirekte Beleuchtungseffekte, wie sie beispielsweise durch die Reflektion von Licht entstehen, können durch solche lokalen Beleuchtungsmodelle nicht simuliert werden. Globale Beleuchtungsmodelle sind hierzu in der Lage. Innerhalb dieses Bereichs existieren zwei große Verfahrensgruppen, das *Ray Tracing* und das *Radiosity-Verfahren*, auf die hier nicht weiter eingegangen werden soll. Die mit globalen Beleuchtungsmodellen erzeugten Bilder wirken aufgrund der physikalisch basierten Beleuchtungsberechnungen deutlich realistischer als die Ergebnisse lokaler Beleuchtungsmodelle, allerdings sind die Berechnungen auch um ein Vielfaches

aufwendiger. Darum setzen auch heute noch viele 3D-Renderengines speziell in der Computerspielindustrie auf fortschrittlichere Varianten lokaler Modelle, da für diese die Echtzeitfähigkeit der Beleuchtungsberechnung die Hauptanforderung an die Engine darstellt [Wa02]. Die Mental Ray-Engine implementiert eine Reihe von Verfahren für die korrekte Berechnung globaler Beleuchtung und erzeugt dadurch sehr realistische Bilder, ist allerdings nicht echtzeitfähig. Für die Erzeugung von Beispielbildern ist dies aber auch nicht erforderlich, weshalb auf diese Engine für das Rendering der Stadtszenen zurückgegriffen wurde.

13.1 **Beispiele für Gebäude des freien Gebäudetyps**

Der freie Gebäudetyp ist die allgemeinste Gebäudevariante, die innerhalb des Prototyps des Semantic Building Modelers implementiert ist. Bei der Erstellung von Konfigurationen für diesen Gebäudetyp kann der Nutzer auf sämtliche Funktionen der Software zurückgreifen, dies gilt sowohl für die Grundriss- und Dachgenerierung als auch für die Applikation beliebiger Komponenten. Da der Objectplacement-Algorithmus Grundlage der Grundrissmodifikation des Jugendstil-Gebäudetyps ist, für den im nachfolgenden Abschnitt Beispiele vorgestellt werden, konzentrieren sich die hier gezeigten Renderings auf die ähnlichkeitsbasierte Grundrisserzeugung. Dieses Verfahren eignet sich im Gegensatz zum Objectplacement-Algorithmus ausschließlich für die Generierung neuer Grundrisse, bereits vorhandene können dagegen nicht über das Verfahren modifiziert werden. Aus diesem Grund wird für Gebäude, deren Grundriss über die ähnlichkeitsbasierte Grundrisserzeugung generiert wird, nur für das Erdgeschoss ein Grundrisspolygon berechnet, das dann auf den nachfolgenden Stockwerken wiederverwendet wird. In diesem Zusammenhang ist anzumerken, dass auch eine Kombination der beiden Verfahren, also der Objectplacement- und der ähnlichkeitsbasierten Berechnung möglich ist. So könnte der Erdgeschossgrundriss ähnlichkeitsbasiert anhand eines beliebigen Eingabegrundrisses erzeugt werden. Auf den nachfolgenden Stockwerken wäre dann eine Modifikation dieses initialen Grundrisses mittels Objectplacement-Verfahren möglich. Dadurch lässt sich eine Vielzahl unterschiedlicher und variantenreicher Grundrisse erstellen. Um die Verwendbarkeit der ähnlichkeitsbasierten Grundrissgenerierung für die Gebäudekonstruktion zu verdeutlichen, wird darauf aber in den nachfolgenden Beispielen explizit verzichtet. Die gezeigten Gebäude wurden ausschließlich über den ähnlichkeitsbasierten Algorithmus errechnet. Der hierfür verwendete Eingabegrundriss ist in Abbildung 122 zu sehen. Es handelt sich um einen

einfachen rechteckigen Grundriss mit einem Erker an der Frontseite. Dieses Grundrisspolygon wurde bereits im Abschnitt „Ähnlichkeitsbasierte Grundrisserzeugung“ für die Erzeugung der Beispielgrundrisse eingesetzt.

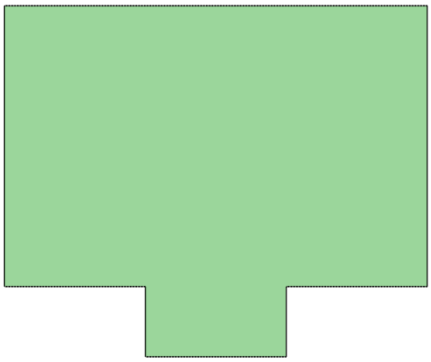

Abbildung 122: Eingabegrundriss für ähnlichkeitsbasierte Grundrisserzeugung

Die Gebäude selber sind von ihrem Erscheinungsbild durch die deutsche Nachkriegsarchitektur inspiriert, wie sie beispielsweise im Rheinland weit verbreitet ist. Diese verwendet typischerweise recht einfache Grundrisse. Die Fassaden sind meist schmucklos und verputzt, auch die Fenster sind von ihrer Struktur her weniger aufwendig als beispielsweise diejenigen in den nachfolgend vorgestellten Jugendstilgebäuden. Die häufigste Variante dieses Gebäudes hat einen einfachen rechteckigen Grundriss, der über die Stockwerke hinweg nicht variiert wird. Die Verwendung eines solchen Grundrisses als Beispiel für die Eingabe in die ähnlichkeitsbasierte Grundrisserstellung würde in den meisten Fällen der Synthese ebenfalls zu rechteckigen Grundrissen unterschiedlicher Größe führen. Um das Potential des Verfahrens besser zu verdeutlichen, wurde darum der in Abbildung 122 gezeigte, rechteckige Grundriss mit Erker verwendet. Als Komponentenmodelle greift der Semantic Building Modeler auf einfache Fenster- und Türmodelle zurück. Außerdem wurde in den meisten Fällen die Fassade einfarbig gehalten, anstatt auf Ziegeltexturen zurückzugreifen, da Gebäude dieser Art häufig über eine verputzte, einfarbige Fassade verfügen.


Abbildung 123: Beispiele für Gebäude des freien Gebäudetyps


Abbildung 124: Beispiele für Gebäude des freien Gebäudetyps


Abbildung 125: Beispiele für Gebäude des freien Gebäudetyps

Die vorherigen Abbildungen zeigen eine Reihe beispielhafter Renderings einer Stadtszene, deren Gebäude vollständig über den freien Gebäudetyp erzeugt wurden. Die dargestellten Grundrisse wurden über das ähnlichkeitsbasierte Verfahren erstellt. Bezüglich der

Gebäudekomponenten konnte das System aus zwei verschiedenen Fenstermodellen und einem Türmodell wählen. Manche der gezeigten Gebäude wurden mit Gesimsen versehen, wobei für diese keine Verbindungselemente an den Gebäudeecken berechnet wurden. Teilweise beschränkt sich die Gesimsapplikation auf bestimmte Stockwerke (bsw. Dachgeschoss) und bestimmte Wandausrichtungen (bsw. Gebäudefront). Außerdem wurden nur an einem Teil der berechneten Gebäude Fensterbänke erzeugt. Für die Erstellung der Fensterbänke und der Gesimse wurde auf das gleiche Profilpolygon zurückgegriffen, das aus einem einfachen Rechteck besteht. Für die Dacherzeugung wurden zwei unterschiedliche Konfigurationen verwendet. Die erste Konfiguration erzeugt Dächer, bei denen sämtliche Dachflächen eine Neigung von etwa 57° haben. Bei der zweiten Konfiguration sind dagegen unterschiedliche Neigungen möglich. An der Gebäudefront und –rückseite variiert die Steigung zwischen 45° und 69°, an den verbleibenden Gebäudeseiten sind Neigungswinkel im Bereich von 63° bis 90° möglich.

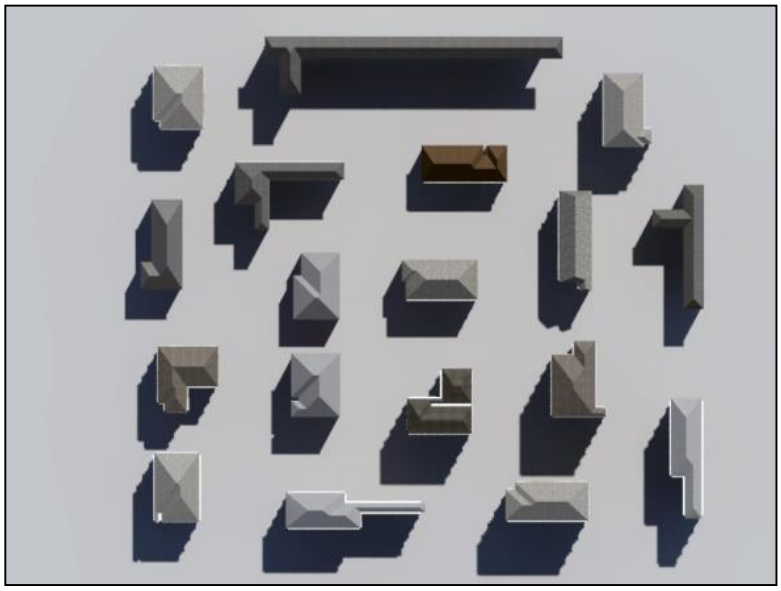

Abbildung 126: Draufsicht auf Gebäude des freien Gebäudetyps (ähnlichkeitsbasierte Grundrisserzeugung)

Abbildung 126 zeigt eine Draufsicht auf eine Auswahl der Gebäude, mittels derer die vorherigen Beispielrenderings erzeugt wurden. Dadurch soll ein besserer Eindruck der Grundrisse entstehen, die durch das ähnlichkeitsbasierte Verfahren anhand des vorab gezeigten Eingabepolygons errechnet wurden. Gut zu erkennen ist an dieser Ansicht, wie speziell die ermittelten Regeln an den Reflex-Vertices in der Synthesephase dazu führen, dass der Algorithmus erkerähnliche Strukturen generiert. Diese entsprechen allerdings nicht exakt dem Eingabegrundriss, sondern sind diesem nur in Bezug auf bestimmte

Charakteristika ähnlich. Dies gilt speziell für die Vertexregeln, die die spezifischen Eigenschaften der Reflex-Vertices abbilden. Die Anwendung dieser Regeln während der Synthese generiert neue Reflex-Vertices, über die Länge der zu diesen Vertices adjazenten Kanten treffen die Regeln dagegen keinerlei Aussage. Diese hängt allein von der nachfolgenden Auswahl weiterer Regeln ab. Dadurch können Strukturen entstehen, wie sie in Abbildung 126 gut zu erkennen sind, beispielsweise Grundrisse mit Erkern aber auch Polygone, die sich durch langgezogene Gebäudeteile auszeichnen. Für beide Grundrissarten sind in der Abbildung Beispiele zu sehen.

Anhand der vorab gezeigten Beispiele erkennt man, dass die ähnlichkeitsbasierte Grundrisserzeugung ein fruchtbares Verfahren ist, das durchaus realistische Grundrisse generieren kann. Allerdings sind die Ergebnisse im Vergleich zum Objectplacement-Verfahren willkürlicher, da die Kontrolle des Algorithmus schwieriger ist. Dadurch entstehen mehr unrealistische, teils stark verschachtelte Grundrisse als dies beim Objectplacement-Verfahren der Fall ist. Für eine detailliertere Diskussion des Verfahrens sei allerdings auf die vorherigen Abschnitte verwiesen.

Der nachfolgende Abschnitt befasst sich mit Beispielen für Gebäude des Jugendstil-Gebäudetyps. Im Gegensatz zum freien Gebäudetyp kann dieser ausschließlich auf den Objectplacement-Algorithmus zurückgreifen.

13.2 **Beispiele für Gebäude des Typs Jugendstil**

Der Jugendstil-Gebäudetyp ist ein Beispiel für einen speziellen Typ, bei dem eine Reihe sonst frei wählbarer Gestaltungsmerkmale durch das System festgelegt ist. Der Konstruktionsplan dieses speziellen Gebäudes liegt in Form einer Java-Klasse vor, Hauptunterschiede im Vergleich zum freien Gebäudetyp liegen unter anderem in den zur Verfügung stehenden Algorithmen für die Grundrisskonstruktion und -modifikation. Außerdem werden für diesen Typ nur spezielle 3D-Modelle für Fenster, Türen und Gesimse verwendet, die aus der Kategorie `Jugendstil` stammen. Diese Festlegungen erleichtern die Erstellung einer Konfiguration für Gebäude dieser Klasse, da weniger Parameter spezifiziert werden müssen.

Der Grundriss der Jugendstil-Gebäudeinstanzen wird durch den Objectplacement-Algorithmus errechnet. Ausgangspunkt ist ein rechteckiger Basisgrundriss nutzerdefinierten Ausmaßes. Für jeden Stockwerkstyp (`GROUND`, `INTERMEDIATE`, `TOP`) gibt der Nutzer

innerhalb der Konfigurationsdatei eine Wahrscheinlichkeit an, anhand derer entschieden wird, ob eine Modifikation des vorhergehenden Grundrisses erfolgen soll. Zusätzlich dazu können verschiedene Höhenbereiche für die Stockwerkstypen angegeben werden.

Wie auch bei anderen Gebäudetypen, ist es für Jugendstil-Konfigurationen erforderlich, eine Dachkonfiguration zu spezifizieren und dem System zur Verfügung zu stellen. Zur Auswahl stehen das Standard- und das Mansardendach, für die jeweils nach vorab erläutertem Schema Parameter wie die Neigungswinkel definiert werden.

Die nachfolgenden Abbildungen zeigen Renderings von Gebäuden, die anhand zweier unterschiedlicher Jugendstilkonfigurationen erzeugt und anschließend in 3ds Max importiert wurden. Der einzige Unterschied zwischen diesen Konfigurationen besteht in der verwendeten Dachkonfigurationen. Die erste Gebäudevariante verwendet Standarddächer mit Steigungen, die innerhalb eines vorgegebenen Bereiches durch das System variiert werden, die zweite Variante nutzt eine Konfiguration zur Berechnung von Mansardendächern.

Für die Erzeugung der Gebäude wurden zwei verschiedene Fenster- und ein einziges Türmodell verwendet. Weiterhin wurden die Fascia-Komponenten durch ein einzelnes Profil generiert. Trotz dieser sehr kleinen Asset-Bibliothek wirken die berechneten Modelle abwechslungsreich. Eine umfangreichere Bibliothek an Gebäudekomponenten kann in diesem Zusammenhang die Vielfalt der Ergebnisgebäude sehr leicht steigern, ohne dass Eingriffe in den Programmcode oder Anpassungen an der Konfiguration erforderlich sind. Das Konzept der kategorienbasierten Asset-Auswahl erweist sich in diesem Zusammenhang als einfache und effiziente Möglichkeit, die visuelle Vielfalt der Gebäude zu steigern.

Neben den verwendeten Komponenten spielt auch der eingesetzte Algorithmus für die Grundrissgenerierung eine wichtige Rolle. Wie vorab erwähnt, kommt für Jugendstil-Gebäude nur das Objectplacement-Verfahren zum Einsatz. Abbildung 130 zeigt eine Sicht auf eine vereinfachte Szenerie, die nur Gebäude enthält, die für die Erstellung der Abbildungen verwendet wurden. Alle in 3ds Max hinzugefügten Komponenten wie Vegetation, Bodenfläche oder Straßen wurden der Übersichtlichkeit wegen aus der Szene entfernt. Die verwendete Perspektive ist eine Ansicht von oben auf die Szene, anhand derer sich ein guter Überblick über die generierten Grundrissstrukturen gewinnen lässt. Daran lassen sich zwei Eigenschaften des Systems erkennen. Zum einen zeigt diese Perspektive die Vielfalt der Grundrisse, die sich durch das Objectplacement-Verfahren realisieren lässt.


Abbildung 127: Beispiele für Gebäude des Typs "Jugendstil"


Abbildung 128: Beispiele für Gebäude des Typs "Jugendstil"


Abbildung 129: Beispiele für Gebäude des Typs "Jugendstil"

Die verwendete Konfiguration generiert realistische Grundrisse, die für den Jugendstil-Gebäudetyp sehr gut geeignet sind. Neben der Eignung des Objectplacement-Verfahrens für diesen speziellen Typ zeigt sich auch die Mächtigkeit des adaptierten Weighted-Straight-

Skeleton-Verfahrens. Dieses ist in der Lage, auch für komplexe Grundrisse realistische Dachstrukturen automatisch zu generieren. Zu erkennen sind in diesem Zusammenhang sowohl die unterschiedlichen Dachtypen, also Mansarden- und Standarddach als auch die Auswirkungen der zufallsbasierten Neigungswinkelschwankungen, die eine hohe Variabilität in der Dacherzeugung erreichen.

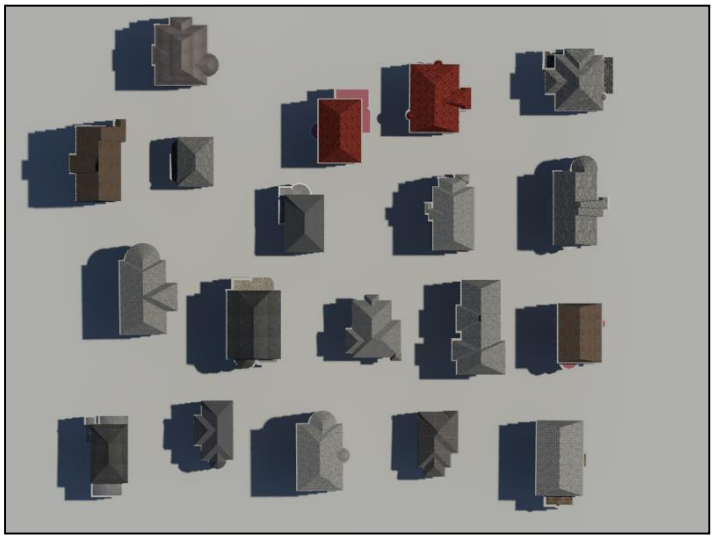

Abbildung 130: Draufsicht auf Jugendstil-Gebäude

Nachdem in den beiden vorherigen Abschnitten Beispiele für Wohngebäude unterschiedlicher Art vorgestellt wurden, bei denen der Fokus auf einer möglichst hohen Variabilität bei gleichzeitig großem Realismus liegt, befasst sich der nun folgende, letzte Beispielabschnitt mit einem völlig anderen Gebäudetyp, nämlich dem griechischen Tempel dorischer Ordnung, wie er beispielsweise von Schmidt-Colinet et al. beschrieben wird [SP04].

13.3 **Beispiele für den Gebäudetyp Tempel**

Neben dem freien und dem Jugendstil-Gebäudetyp implementiert der Semantic Building Modeler noch einen dritten Typ, der sich in verschiedenen Aspekten grundlegend von den vorab genannten Typen unterscheidet. Zunächst handelt es sich um den am stärksten spezialisierten Typ, der dem Nutzer die geringsten Freiheitsgrade bezüglich der Konfigurationserstellung bietet. Dies hängt damit zusammen, dass der implementierte Tempel in Bezug auf eine ganze Reihe von Parametern bereits vorab festgelegt ist.

Zunächst ist hier der Grundriss zu nennen. Der vorliegende Tempeltyp ist das bisher einzige Gebäude, für das eine Innenraumstruktur verwendet wird. Die hierfür umgesetzten Verfahren und Algorithmen wurden bereits im Kapitel „Innenraumkonstruktion und Wanderzeugung“ vorgestellt. Als Vorbild für die Konstruktion der Innenraumstrukturen wurde der Grundriss des griechischen Doppelantentempels verwendet, der in Abbildung 131 zu sehen ist.

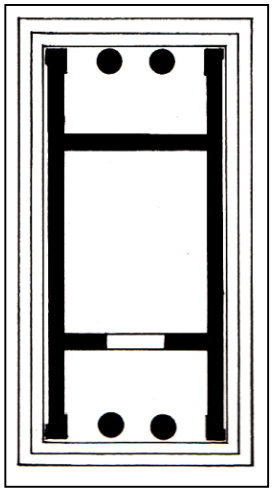

Abbildung 131: Grundriss des griechischen Doppelantentempels [SP04]

Die Doppelantentempel-Klasse verwendet aber im Gegensatz zum dargestellten Grundriss nur die eingezeichneten Wandstrukturen, positioniert dagegen die Säulen in Form einer vollständig umlaufenden Säulenreihe. Dabei sind die vorhandenen Methoden zur Tempelkonstruktion sehr allgemein formuliert und gestatten die Integration anderer Innenraumstrukturen ohne großen Aufwand. Exemplarisch wurde im vorliegenden Prototyp allerdings nur der Doppelantentempelgrundriss realisiert.

Der Konstruktionsalgorithmus erzeugt zunächst einen rechteckigen Grundriss nutzerdefinierbarer Größe, der als Basis für die Erzeugung der Treppe genutzt wird, die zum Tempel hinauf führt. Dabei ist der rechteckige Basisgrundriss für den Nutzer nicht änderbar. Er kann im Gegensatz zu den vorherigen Wohngebäuden nicht auf die implementierten Verfahren für die Grundrissmodifikation zurückgreifen, einzig die Ausdehnungen des Rechtecks können beliebig variiert werden, die Grundform ist dagegen unveränderbar. Auch die weiteren Konstruktionsschritte sind im Vergleich zu den Wohngebäuden sehr stark festgelegt, einzig die Höhe der einzelnen Tempelbestandteile bei Tempeln dorischer Ordnung [SP04] können angegeben werden. Dazu gehören beispielsweise die Höhe des

Architravs, der Metope oder des Geisons. Die Säulen werden basierend auf dem vorab erläuterten Kategorienverfahren durch das System geladen und anschließend anhand der Konfigurationsparameter positioniert. Für die nachfolgend gezeigten Beispielrenderings wurde durch den Semantic Building Modeler ein Tempel generiert, bei dem aus zwei verschiedenen dorischen Säulentypen gewählt werden konnte. Analog erfolgte auch die Applikation der Triglyphen, auch diese wurden als 3D-Modelle geladen und anschließend basierend auf den Nutzerangaben positioniert. Als letzten Schritt bei der Erstellung einer validen Konfiguration für die Konstruktion eines solchen Tempels müssen Konfigurationswerte für die Dachkonstruktion angegeben werden. Dies erfolgt analog zu den vorab genannten Gebäudetypen. Für den dargestellten Tempel wurde eine Steigung von ca. 90° an der Vorder- und Rückseite des Gebäudes verwendet und eine Neigung von ca. 15° an den Tempelseiten.

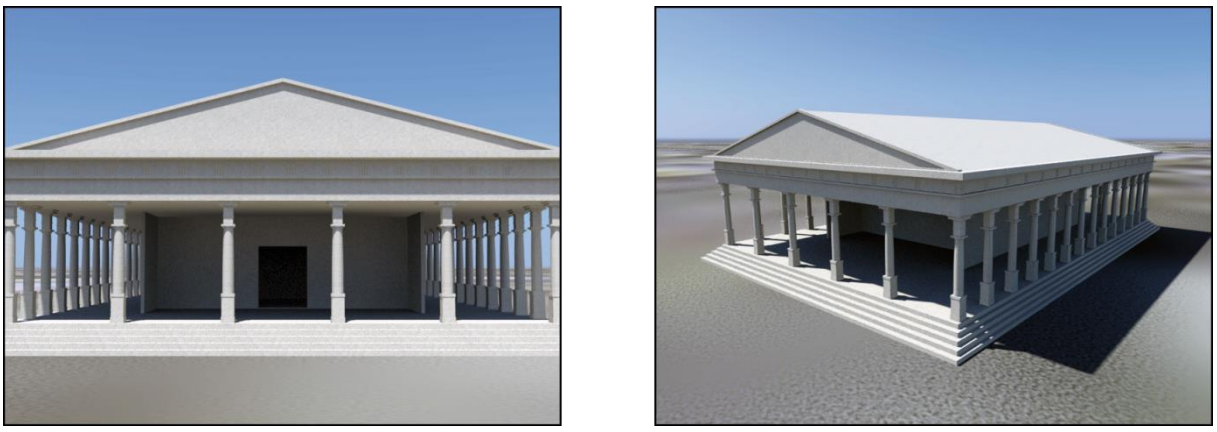

Abbildung 132: Beispiele für einen Tempel dorischer Ordnung mit Doppelantentempel-Grundriss

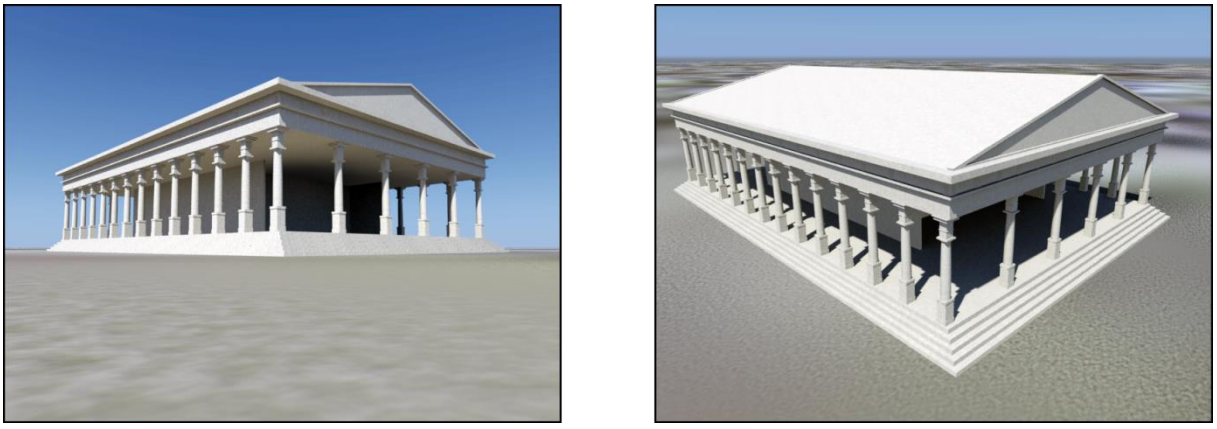

Abbildung 133: Beispiele für einen Tempel dorischer Ordnung mit Doppelantentempel-Grundriss

Die vorab gezeigten Renderings wurden analog zu den in den vorherigen Abschnitten vorgestellten Bildern der Wohngebäudetypen vollständig in 3ds Max generiert, nachdem die durch den Semantic Building Modeler berechneten Modelle importiert und zu einer Szene kombiniert wurden.

Die ursprüngliche Motivation für die Implementation einer Klasse zur Erzeugung von Tempelstrukturen bestand darin, einen Gebäudetyp zu entwickeln, der sich in seiner Struktur stark von den Standardgebäuden unterscheidet. Ein Hauptziel war dabei die Identifikation von Gemeinsamkeiten zwischen den unterschiedlichen Gebäudeklassen. Dies bezieht sich sowohl auf gemeinsame Konfigurationsparameter als auch auf die intern verwendeten Algorithmen und Strukturen, die für die Berechnung der Gebäude verwendet werden. Dabei zeigte sich, dass die Umsetzung einer Tempelklasse viele der bereits vorhandenen Methoden wiederverwenden konnte. So werden die einzelnen Komponenten des Tempels, beispielsweise die Metope oder der Architrav, intern als Stockwerke umgesetzt. Auch für die Erzeugung des Daches mussten keinerlei Adaptionen am verwendeten Algorithmus vorgenommen werden, vielmehr stellt der rechteckige Grundriss eine sehr einfache Eingabe für das Weighted-Straight-Skeleton-Verfahren dar. Die Positionierung und Ausrichtung der Säulen und Triglyphen konnte in großen Teilen ebenfalls auf Methoden und Prozeduren zurückgreifen, die ursprünglich für die Positionierung von Fenstern in Wohngebäuden implementiert wurden.

Zusammenfassend zeigte die Umsetzung des Tempelgebäudes, dass viele der vorhandenen Algorithmen und Strukturen auch für Gebäudetypen wiederverwendet werden konnten, die auf den ersten Blick wenige Ähnlichkeiten mit den zunächst umgesetzten Wohngebäuden aufweisen. Dies kann als Indiz gewertet werden, dass der Semantic Building Modeler mit seinen Strukturen und Methoden auch für die Implementation weiterer Gebäudeklassen gute Voraussetzungen bietet. Durch die Wiederverwendung vorhandener Ressourcen kann der Implementationsaufwand mit hoher Wahrscheinlichkeit signifikant reduziert werden.

14 Zusammenfassung

14.1 Gebäudekonstruktion und -vielfalt

Die bereits in der Einleitung dieser Arbeit diskutierte, stetig wachsende Bedeutung computergenerierter Gebäude- und Stadtmodelle ist eine wichtige Triebfeder für die Entwicklung prozeduraler Technologien zur automatischen Generierung solcher Modelle. Hierbei lassen sich die verschiedenen Ansätze in unterschiedlicher Art und Weise differenzieren. Eine vorab bereits zitierte Unterscheidung stammt von Oliver Deussen, der prozedurale und regelbasierte Systeme voneinander unterscheidet [De03]. Der Semantic Building Modeler gehört zur Gruppe der prozeduralen Systeme, bei denen der Konstruktionsprozess in Form von Algorithmen innerhalb der Software implementiert ist.

Dies ist Voraussetzung für die Verwendung semantisch basierter Parameter, für die der Nutzer bei der Erstellung von Konfigurationen Werte festlegen kann. Die Möglichkeit, automatisierte Gebäudekonstruktionsprozesse durch die Verwendung semantisch bedeutsamer Parameter zu steuern, unterscheidet den Semantic Building Modeler von regelbasierten Systemen und ist zentraler Faktor für die Namenswahl des Systems. Auf die Vorteile, die die Nutzung eindeutig festgelegter Parameter mit ausdrucksstarken Bezeichnern aus Sicht eines Nutzers hat, wurde bereits an früherer Stelle ausführlich eingegangen. Auch die potentiellen Nachteile in Bezug auf die geringere Modellierungsmächtigkeit solcher Verfahren gegenüber regelbasierten Systemen wurden diskutiert. Als Fazit sollen trotzdem noch einmal die wichtigsten Aspekte kurz thematisiert werden.

14.1.1 Semantische Konfigurationsparameter vs. Modellierungsmächtigkeit

Prozedurale Systeme wie der Semantic Building Modeler erlauben es dem Nutzer nicht ohne Weiteres, eigene Konstruktionsprozesse zu formulieren, sondern geben die verwendeten Algorithmen und Konstruktionsschritte in Form von Programmcode vor. Der Nutzer ist in der Lage, diesen Prozess durch die Verwendung von Parametern zu steuern. Die Tatsache, dass die Parameter durch das System vorgegeben sind, hat zwei Konsequenzen. Zum einen besitzen die Parameter eine feste Bedeutung, da die ihnen zugewiesenen Werte innerhalb der Algorithmen ausgewertet und für die Berechnungen eingesetzt werden. Beispielsweise wird ein Parameter `height` immer für die Festlegung der Höhe des zu berechnenden Gebäudes eingesetzt. Eine alternative Verwendung kann durch den Nutzer ohne Eingriff in den

Programmcode nicht festgelegt werden. Dadurch erhält der Parameter eine Semantik, aus der seine Bedeutung und Rolle innerhalb der Konstruktion für den Nutzer direkt ersichtlich ist. Regelbasierte Systeme gründen dagegen auf der Theorie der Textersetzungssysteme und somit auf einem Formalismus, innerhalb dessen der Nutzer eine Regelmenge spezifiziert, deren iterative oder parallele Anwendung einen Start- in einen Zielzustand überführt. Typischerweise bestehen solche Regelsysteme aus geometrischen Operationen wie Unterteilungen und Formersetzungen. Die Aufgabe des Nutzers ist die sinnvolle Aneinanderreihung solcher Regeln, so dass am Ende eine Form entsteht, die seinen Vorstellungen entspricht. Die Möglichkeit, direkt mit solchen geometrischen Operationen zu arbeiten, erhöht die Modellierungsmächtigkeit, allerdings auch die Komplexität.

Innerhalb des Semantic Building Modeler konnte gezeigt werden, dass für den speziellen Bereich der Gebäudekonstruktion die Modellierungsmächtigkeit ausreicht, um Gebäude hoher Variabilität und ansprechendem visuellen Eindruck zu generieren. Die unterschiedlichen Möglichkeiten zur Erzeugung und Modifikation von Grundrissen erlauben die Berechnung vielfältiger Gebäudetypen, ohne dass Eingriffe in den Programmcode erforderlich sind, die Wahl geeigneter Parameterwerte ist ausreichend. Um diese Zielsetzung zu erreichen, implementiert der Semantic Building Modeler einen allgemeinen Gebäudetyp, der als `ArbitraryBuilding` bezeichnet wird. Dieser Gebäudetyp bietet dem Nutzer die größte Freiheit bei der Spezifikation der Gebäudestrukturen. Dies gilt sowohl für die Konstruktion der Gebäude- und Stockwerksgrundrisse, als auch für die Erzeugung von Dächern unterschiedlichen Typs. Gebäudekomponenten können über den kategorienbasierten Ansatz zufallsbasiert durch das System abhängig vom Komponententyp entweder prozedural erzeugt oder direkt als 3D-Modell geladen und positioniert werden. Dadurch lassen sich Variationen auch für Nutzer ohne Programmierkenntnisse leicht realisieren.

Für spezielle Gebäudetypen, deren Strukturen nicht durch die bereitgestellten Technologien realisierbar sind, sind dagegen Eingriffe in den Programmcode notwendig. Ein gutes Beispiel für einen solchen Gebäudetyp ist der implementierte Tempel. Dieses Gebäudes unterscheidet sich in verschiedenen Aspekten von den beiden anderen Gebäudetypen, dem freien und dem Jugendstil-Gebäudetyp. Aufgrund der Struktur des Tempels ist es erforderlich, mehrere Grundrisse anzugeben, konkret ist die zusätzliche Festlegung von Innenraumstrukturen notwendig. Diese unterscheiden sich in einer Reihe von Punkten von den verwendeten Standardgrundrissen. Der wichtigste Unterschied ist die Tatsache, dass sie

nicht mehr durch geschlossene Polygonzüge dargestellt werden können und darüber hinaus die Definition von Verbindungen zwischen den einzelnen Räumen vorgenommen werden muss. Eine einfache Extrusion des Basisgrundrisses für die Erzeugung der Stockwerke ist für solche Strukturen nicht möglich. Dadurch wird der Berechnungsprozess aufwendiger. Dies erforderte die Implementation eines Generierungsverfahrens für Doppelantentempelstrukturen, auf das vorab bereits detailliert eingegangen wurde. Allerdings konnte für den neuen Gebäudetyp auf eine Vielzahl bereits vorhandener Funktionen und Algorithmen zurückgegriffen werden, unter anderem für die Dachgenerierung, die Berechnung der einzelnen Tempelbestandteile oder die Triglyphen- und Säulen-Positionierung.

Zusammenfassend können bereits mit den vorhandenen Gebäudetypen vielfältige Gebäude konstruiert werden. Dies ist in den meisten Fällen ohne Eingriffe in den Programmcode des Semantic Building Modelers möglich. Besitzen die Gebäude eine spezielle Form, wie dies beim implementierten Tempel der Fall ist, so ist zwar mit hoher Wahrscheinlichkeit die Umsetzung eines neuen Gebäudetyps in Form einer Java-Klasse erforderlich, allerdings kann dabei auf umfangreiche Algorithmen- und Verfahrensbibliotheken zurückgegriffen werden, so dass für einen erfahrenen Programmierer der Aufwand überschaubar sein sollte. Dadurch kann der Semantic Building Modeler auch als Framework zur Gebäudegenerierung betrachtet werden, das verwendet werden kann, um mit Hilfe der vorhandenen Funktionen die Konstruktion weiterer Gebäude zu beschreiben und diese anschließend prozedural erzeugen zu lassen.

Typischerweise erfordern spezielle Gebäudetypen auch die Einführung neuer Konfigurationsparameter. Damit eine der großen Stärken des Systems, nämlich die Verwendung ausdrucksstarker, semantischer Parameter erhalten bleibt, sollten diese sorgfältig gewählt und bezeichnet werden, damit die Bedeutung für den Nutzer nachvollziehbar und verständlich ist.

14.2 **Grundrisskonstruktion für Gebäude und Stockwerke**

Eine Grundlage der vorab dargestellten, einfachen Erzeugung und Spezifikation von Gebäuden sind neben der Verwendung semantischer Parameter auch die Möglichkeiten für die automatisierte Erstellung von Grundrissen. Insgesamt bietet das System vier verschiedene Technologien zur Grundrisserzeugung, von denen ein Verfahren auch für die

Modifikation bereits vorhandener Grundrisse einsetzbar ist. Die ersten beiden Verfahren stellen im eigentlichen Sinne keine prozeduralen Ansätze dar. Die erste Variante lädt einen Grundriss aus einer Modelldatei und verwendet diesen dann für die Stockwerkskonstruktion. Bei der zweiten Variante liegt der Grundriss in Form einer Java-Klasse vor. Dieser Grundrisstyp ist für spezielle Gebäudetypen hilfreich und wird unter anderem für die Konstruktion des Tempels eingesetzt.

Neu für die Grundrisskonstruktion sind einerseits das Objectplacement-Verfahren und andererseits das Verfahren für die ähnlichkeitsbasierte Grundrisserzeugung. Das Objectplacement-Verfahren ist ein stark parametrisierter Algorithmus, bei dem einfache geometrische Grundkörper miteinander kombiniert werden, um dadurch komplexe Formen zur erzeugen. Für die Zusammenfassung der positionierten Komponenten kann entweder auf den Footprint-Merger-Algorithmus zurückgegriffen werden, bei dem es sich um ein Verfahren handelt, das die Kanten der einzelnen Komponenten durchläuft und das als Ergebnis ein geschlossenes, konvexes oder konkaves Polygon liefert. Alternativ steht auch der Graham-Scan-Algorithmus zur Verfügung, mittels dessen die konvexe Hülle um die Eckpunkte aller positionierten Komponenten berechnet wird. Die Verwendung dieses Verfahrens garantiert konvexe Polygone als Ergebnis der Placement-Berechnungen. Betrachtet man die im vorherigen Abschnitt gezeigten Beispiele gerenderter Gebäude und ihrer Grundrisse, speziell diejenigen des Jugendstil-Gebäudetyps, so erkennt man, dass dieses Verfahren sehr gut geeignet ist, um realistische, variationsreiche Grundrisse zu generieren. Dies sind zwei Kernforderungen, die das Verfahren innerhalb des Semantic Building Modelers erfüllen muss. Darüber hinaus ist der Algorithmus sehr gut über Steuerparameter kontrollierbar und somit geeignet, auch andere Gebäudetypen und ihre Grundrisse zu berechnen. Durch die Verwendung des Configuration-Services ist es möglich, einmal erstellte Konfigurationen für das Placement-Verfahren auszulagern und für andere Gebäude wiederzuverwenden oder diese mit anderen Nutzern zu teilen.

Dies gilt auch für den zweiten prozeduralen Algorithmus, der für die Grundrisserzeugung eingesetzt werden kann. Bei diesem handelt es sich um eine Erweiterung des von Merrell et al. [MM08] vorgestellten Continuous Model Synthesis-Verfahrens. Die grundsätzliche Idee besteht in der Extraktion von Regeln, anhand derer die charakteristischen Eigenschaften von 3D-Modellen beschrieben werden. Während der Synthesephase des Algorithmus wird durch die wiederholte Anwendung der ermittelten Regeln ein Objekt generiert, das dem Eingabeobjekt ähnlich ist. Merrell et al. wenden ihr Verfahren auf beliebige 3D-Objekte an

und sind dadurch in der Lage, interessante Varianten des Eingabeobjekts zu generieren. Für die Grundrisskonstruktion ist der Algorithmus in seiner ursprünglichen Form allerdings zu willkürlich. Aus diesem Grund wurde er innerhalb des Semantic Building Modelers um verschiedene, neue Komponenten erweitert, durch die eine größere Kontrolle über das Syntheseergebnis ermöglicht wird. Dazu gehört unter anderem ein Verfahren zur kontrollierten Komponentenauswahl, das zu deutlich gleichmäßigeren und weniger stark verschachtelten Strukturen führt. Darüber hinaus implementiert die hier vorgestellte ähnlichkeitsbasierte Grundrissberechnung einen Ansatz zur Regelklassifikation, der zu einer größeren Steuerbarkeit der Synthese führt. Für eine detaillierte Darstellung dieser Komponenten und eine ausführliche Diskussion der Eignung des Verfahrens für unterschiedliche Grundrisstypen sei auf den Abschnitt „Continuous Model Synthesis [MM08]“ verwiesen. Zusammenfassend stellt das Verfahren zur ähnlichkeitsbasierten Grundrisserzeugung einen fruchtbaren Ansatz dar, der gut geeignet ist, um variantenreiche Grundrisse zu generieren.

Im Vergleich zum Objectplacement-Verfahren besitzt er allerdings eine Reihe von Schwachpunkten, die Gegenstand zukünftiger Arbeiten sein können. Der erste Schwachpunkt besteht darin, dass das Verfahren nur für die Synthese neuer Grundrisse und nicht für die Modifikation bereits bestehender einsetzbar ist. Dies resultiert aus der Natur des Verfahrens und lässt sich nur schwer vermeiden. Die Trennung der Regelextraktions- von der Synthesephase und die Struktur der Regelanwendung im Zuge der Grundrissgenerierung macht eine Anpassung des Verfahrens für die Grundrissmodifikation schwierig. Betrachtet man darüber hinaus die Ergebnisse des Syntheseverfahrens für unterschiedliche Eingabegrundrisse und Konfigurationen der Parameter, so wird deutlich, dass das Verfahren auch unter Verwendung der zusätzlichen Komponenten teilweise unrealistische Grundrisse erzeugt. Im Vergleich zur Ausgangsimplementation, einer zweidimensionalen Umsetzung des Continuous Model Synthesis-Verfahrens von Merrell et al. [MM08], stellt die ähnlichkeitsbasierte Grundrisssynthese allerdings einen großen Schritt in die richtige Richtung dar. Zukünftige Arbeiten könnten hier versuchen, den Syntheseprozess noch stärker kontrollierbar zu machen. Hierbei handelt es sich allerdings um eine Gratwanderung, da zu große Kontrolle leicht zu einer geringeren Variation der erzeugten Grundrisse führen kann. Dies zeigte sich auch in den Versuchen mit den unterschiedlichen Kontrollmechanismen, die innerhalb dieser Arbeit in den Ausgangsalgorithmus integriert wurden. Betrachtet man allerdings die Arbeit von Merrell et al. und die hier vorgenommene

Anpassung des Verfahrens für die Grundrissgenerierung, so lässt sich das große Potential erkennen, das in dem Verfahren steckt, dessen Reiz in der vergleichsweise einfachen Struktur und den daraus erzeugten, komplexen Ergebnissen steckt. Zielführend könnten in diesem Zusammenhang Versuche sein, den Samplingraum der Synthesephase stärker zu kontrollieren, beispielsweise durch die Einführung einer Mindestgröße der während der Faceextraktion ermittelten Polygone. Dies könnte helfen, ein Hauptproblem der erzeugten Grundrisse abzuschwächen oder sogar ganz zu vermeiden, nämlich die Entstehung zu stark verschachtelter Subkomponenten.

14.3 **Dachgenerierung**

Für die Konstruktion realistischer Gebäude ist die Berechnung realistischer Dächer von großer Bedeutung. Die zentralen Anforderungen an den verwendeten Algorithmus sind einerseits, dass er möglichst generisch sein soll, also für beliebig komplexe Grundrisse eingesetzt werden kann, andererseits soll er unterschiedliche Dachformen erzeugen können, beispielsweise Dächer mit unterschiedlich geneigten Dachflächen oder Dachsonderformen wie das Mansardendach. Ein weit verbreitetes Verfahren für die Dacherzeugung ist der Straight-Skeleton-Algorithmus, der auf die Arbeit von Aichholzer et al. zurückgeht [Ai95]. In der ursprünglichen Variante konnte das Verfahren nur Walmdächer mit einer Steigung von 45° erzeugen. Die hier verwendete Implementation basiert auf einem anderen Ansatz, bei dem die Schrumpfungsberechnung nicht über die Bestimmung von Winkelhalbierenden realisiert wird, sondern durch Ebenen, die die einzelnen Kanten des Dachgrundrisses enthalten. Dabei ist es möglich, für jede dieser Kanten unterschiedliche Steigungen festzulegen, wodurch sich Dachformen mit Neigungswinkeln realisieren lassen, die von 45° abweichen.

Kern des Straight-Skeleton-Verfahrens ist das Erkennen von Ereignissen. Jedes Ereignis führt zu einer Änderung der topographischen Struktur des in sich zusammenschrumpfenden Polygons. Aichholzer et al. unterschieden zwei verschiedene Eventtypen, nämlich Edge- und Split-Events [Ai95], Eppstein et al. [EE98] erkannten, dass noch eine weitere Eventvariante auftreten kann, die sie als Vertex-Events bezeichneten. Die vorliegende Implementation führt einen weiteren Eventtyp ein, der Change-Slope-Event genannt wird und der im Gegensatz zu den anderen Events nicht die topographische Struktur des Polygons ändert, sondern zu einer Zuweisung neuer Kantengewichte führt. Erst durch die Einführung dieses Eventtyps ist die Erzeugung von Mansardendächern mit Hilfe des Weighted-Straight-

Skeleton-Algorithmus möglich, ohne die entstehenden Strukturen nachträglich zu modifizieren.

Betrachtet man die Gebäudebeispiele aus dem vorherigen Abschnitt, so ist die breite Einsetzbarkeit des Verfahrens gut zu erkennen. Der Algorithmus ist in der hier vorgestellten, ebenenbasierten Variante robust und berechnet realistische Dachkonstruktionen für beliebige Grundrisse. Für eine detaillierte Diskussion des Verfahrens, potentieller Schwachpunkte und der hier vorgenommen Implementation sei auf den Abschnitt „*Dacherzeugung* – der Weighted-Straight-Skeleton-Algorithmus“ verwiesen.

14.4 **Verteilte Konfigurationsstrukturen**

In den vorherigen Abschnitten wurde wiederholt auf die Bedeutung semantischer Konfigurationsparamater, die für deren Umsetzung erforderlichen Strukturen und die Vorteile für den Nutzer eingegangen. Die Einführung solcher Parameter ist eine der zentralen Leistungen des Semantic Building Modelers. Hierzu zählen nicht nur die reine Verwendung benannter Konfigurationsparameter, sondern in einer Vorstufe auch deren Identifikation und Umsetzung, entweder basierend auf einem Bottom-Up- oder einem Top-Down-Vorgehen. Dadurch war es möglich, eine Reihe allgemeingültiger Parameter zu identifizieren, die unabhängig von einem konkreten Gebäudetyp sind und zur Beschreibung einer Vielzahl unterschiedlicher Gebäudeinstanzen eingesetzt werden können. Zusätzlich zeigte sich im Verlauf dieser Arbeit, dass auch die Einführung gebäudetypspezifischer Parameter sinnvoll ist, da solche spezifischen Parameter sehr häufig eng mit der Semantik der Gebäudekonstruktion verbunden sind. Als Beispiel sei auf den Tempel-Gebäudetyp verwiesen. Dessen Beschreibung erfordert die Angabe einer Reihe von Höhenangaben, konkret die Höhe des Architravs (`architraveHeight`), der Metope (`metopeHeight`) und des Geisons (`geisonHeight`). Offensichtlich ergeben diese Parameter nur im Zusammenhang mit der Konstruktion eines Tempels dieser Bauart Sinn. Für einen mit der Architektur griechischer Tempel vertrauten Nutzer ist aufgrund der Semantik und klaren Benennung der Gebäudeparameter deren Bedeutung direkt verständlich. Somit sollte es für einen solchen Anwender keine Probleme bereiten, eine korrekte Konfiguration zu erstellen, die durch das System ausgewertet und in ein Tempelmodell umgesetzt werden kann.

Die Kombination allgemeiner mit gebäudetypspezifischen Parametern erweist sich als zielführend, um mit einer möglichst geringen Anzahl verfügbarer Parameter eine möglichst

große Anzahl unterschiedlicher Gebäudemodelle parametrisieren zu können. Grundsätzlich ist es aus Gründen der Übersichtlichkeit wünschenswert, die Gesamtzahl der festzulegenden Parameter auf ein Minimum zu reduzieren. Diese Zielsetzung ist auch einer der Hauptgründe für die Einführung spezifischer Gebäudetypen wie des Jugendstil-Gebäudes. Prinzipiell können Instanzen dieses Gebäudetyps auch durch die Verwendung des freien Gebäudetyps spezifiziert werden, allerdings sind die hierfür festzulegenden Parameter bezüglich ihrer Anzahl deutlich umfangreicher. Da sich der Semantic Building Modeler stärker an eine in Bezug auf die Erstellung von 3D-Gebäudemodellen unerfahrene Nutzergruppe richtet, verfolgt das System das Ziel, die Erstellung der Basiskonfigurationen so einfach wie möglich zu halten. Je mehr Informationen über einen Gebäudetyp implizit innerhalb der Erzeugungsalgorithmen kodiert sind, desto weniger Informationen müssen durch den Nutzer angegeben werden und desto einfacher wird ihm auch die Erstellung der Konfigurationen fallen.

Die Reduktion der anzugebenden Parameter wird innerhalb des Semantic Building Modelers auch noch durch einen weiteren Mechanismus ermöglicht, der das vorgestellte System von allen bisherigen Systemen dieser Art unterscheidet. Konkret ist dies die Möglichkeit, die XML-basierte Konfiguration zu modularisieren. Die hierfür umgesetzten Technologien erlauben die Kombination einzelner Konfigurationskomponenten, die von völlig unterschiedlichen Orten geladen werden können. Dadurch entsteht eine Wiederverwendbarkeit, die den Aufbau von nutzergenerierten Konfigurationsbibliotheken ermöglicht. Hat ein Anwender beispielsweise eine Konfiguration für das Objectplacement-Verfahren entwickelt, die besonders gut geeignet ist, um eine bestimmte Art von Grundrissen zu generieren, so kann er diese anderen Nutzern durch die Angabe einer URI zur Verfügung stellen. Die anderen Nutzer können das Konfigurationsfragment anschließend in ihre eigenen Konfigurationen einbinden und es dadurch wiederverwenden. Im Idealfall könnte sich der Nutzeraufwand darauf reduzieren lassen, dass unerfahrene Nutzer Konfigurationen aus verschiedenen Fragmenten zusammenbauen und dadurch komplexe Konfigurationen erstellen können, ohne sich mit den einzelnen Parametern auseinandersetzen zu müssen.

Die vorab genannten Mechanismen erweisen sich als sinnvoll und zielführend in Bezug auf die Zielsetzung, die Konfigurationserstellung für den Nutzer so einfach wie möglich zu gestalten. Dies gilt sowohl für die sorgfältige Auswahl und Spezifikation von Parametern, als auch für die Implementation spezifischer Gebäudetypen direkt innerhalb des Systems.

Auch die umgesetzten Mechanismen zur Modularisierung und Wiederverwendung von Konfigurationsfragmenten stellen aus Sicht des Nutzers wünschenswerte Features dar, die eine erfolgreiche Auseinandersetzung mit dem Semantic Building Modeler deutlich erleichtern. Auf potentielle Erweiterungsmöglichkeiten des Systems, durch die der Nutzer zusätzlich bei der Arbeit unterstützt werden kann, wird im späteren Ausblick eingegangen.

15 Ausblick

Nachdem im vorherigen Abschnitt zusammenfassend auf die zentralen Komponenten des Semantic Building Modelers eingegangen wurde, widmet sich dieser Abschnitt abschließend möglichen Erweiterungen des Systems und der darin verwendeten Verfahren.

15.1 **Integriertes Softwarewerkzeug**

In der prototypischen Implementation, die als *Proof-of-Concept* der entwickelten Algorithmen und Technologien die Basis dieser Arbeit darstellt, benötigt der Nutzer verschiedene Werkzeuge, um als Ergebnis seiner Bemühungen eine fertige Stadt zu erhalten. Aktuell werden drei voneinander getrennte Komponenten benötigt. Dies ist zunächst der Semantic Building Modeler selber, dann eine beliebige Software mittels derer die XML-Konfigurationen erstellt und bearbeitet werden können und zuletzt ein 3D-Modellierungswerkzeug, das in der Lage ist, OBJ-Modelldateien zu laden und zu verarbeiten. Für sämtliche innerhalb der Arbeit gezeigten Renderings wurde der <oXygen/> XML Editor 14.2 für die Editierung und Validierung der XML-Konfigurationen verwendet. Das Compositing der durch die Software erzeugten Modelle zu einer Stadtszene erfolgte mittels Autodesk 3ds Max 2013. Zukünftige Versionen des Semantic Building Modelers könnten eine integrierte Entwicklungsumgebung bereitstellen, die die Funktionalitäten der einzelnen Softwarekomponenten miteinander vereinigt. Dies würde eine für den Anwender wünschenswerte Erweiterung darstellen, da er weniger einzelne, teure Softwarepakete verwenden muss, um eine realistische Stadtszene zu erstellen. Allerdings ist in diesem Fall der Aufwand der Implementation nicht zu unterschätzen. Betrachtet man den enormen Funktionsumfang einer 3D-Modellierungsumgebung wie Autodesk 3ds Max, so wird schnell klar, dass die vollständige Integration eines solchen Systems nicht umsetzbar ist. Allerdings ist sie auch nicht erforderlich. Der Semantic Building Modeler unterscheidet sich von regelbasierten Systemen unter anderem dadurch, dass sein Anwendungsgebiet sehr klar umrissen ist. Vergleicht man seine Modellierungsmächtigkeit beispielsweise mit der von Sven Havemann vorgestellten GML [Ha05], so wird klar, dass diese Sprache bezüglich ihrer Modellierungsmöglichkeiten deutlich umfangreicher ist. Allerdings steigt mit der größeren Ausdrucksstärke typischerweise auch die Komplexität. Dies ist einer der Gründe, warum der Semantic Building Modeler explizit auf die Erzeugung von Gebäuden beschränkt ist. Andere 3D-Modelle können und sollen nicht durch ihn erstellt werden. Dies erleichtert speziell für ungeübte Nutzer die Bedienung und somit auch den Einstieg in die Gebäudemodellierung.

Da der Semantic Building Modeler als prozedurales System Gebäude algorithmisch erstellt, benötigt er auch nur einen Bruchteil der Fähigkeiten, die Modellierungsumgebungen wie 3ds Max besitzen. Als Einstieg wäre es beispielsweise wünschenswert, wenn der Nutzer die errechneten Gebäude direkt innerhalb des Semantic Building Modelers verschieben, skalieren oder rotieren könnte. Im Prototyp kann er nur die Kamera um die berechneten Gebäude drehen, um sich diese aus unterschiedlichen Perspektiven anzuschauen. Die Integration rudimentärer Editierungsmöglichkeiten stellt eine umsetzbare Erweiterung für zukünftige Implementationen dar, die den Einsatz elaborierter Softwarewerkzeuge wie 3ds Max unnötig machen könnte. Weiterhin wünschenswert und unter Verwendung der integrierten Import-Funktionalität bereits theoretisch möglich wäre das Laden und Verarbeiten von zusätzlichen Modellen wie beispielsweise Bäumen, Parkbänken oder ähnlichen, typischerweise in einer Stadt vorhandenen Gegenständen. Wie man bereits an den Renderings innerhalb des Beispielabschnitts erkennt, spielen solche Komponenten eine zentrale Rolle für den Realismus einer gerenderten Stadtszene.

Aus Sicht der Konfigurationserstellung wird ebenfalls nur ein kleiner Teil der Fähigkeiten benötigt, den moderne XML-Editoren wie <oXygen/> zur Verfügung stellen. Denkbar wären vergleichsweise einfache aber hilfreiche Features wie eine auf dem Schema basierende Code-Vervollständigung und die Möglichkeit, die erzeugten Konfigurationen direkt unter Verwendung des Basisschemas zu validieren. Prinzipiell kann für die Erstellung der Konfigurationen auf jeden beliebigen Volltexteditor zurückgegriffen werden. Dies kann bei umfangreichen Konfigurationen aber schnell mühsam und arbeitsintensiv werden. Die Implementation der vorab genannten Basisfeatures würde hier bereits eine deutliche Erleichterung darstellen.

15.2 **Aufbau von Bibliotheken**

Der Aufbau von webbasierten Bibliotheksservices stellt für viele Konzepte des Semantic Building Modelers eine sinnvolle Erweiterung dar. Solche Webservices sind in verschiedenen Varianten denkbar. Zunächst könnten sie eingesetzt werden, um Konfigurationsfragmente zu verwalten, damit diese zwischen Anwendern des Systems ausgetauscht werden können. Dadurch könnte mit technisch einfachen Mitteln eine umfangreiche Bibliothek aufgebaut werden, aus der sich Nutzer bedienen können. Denkbar ist sowohl eine dezentrale Struktur, bei der der Webservice als reiner *URI-Aggregator* fungiert und die eigentlichen Fragmente auf beliebigen Webservern auf der ganzen Welt

liegen, als auch eine zentralisierte Struktur. Bei der zentralisierten Architektur würden sämtliche Konfigurationsteile auf den vom Webservice verwalteten Webrepositorys vorgehalten, der Service würde dann die URIs bereitstellen, mittels derer sich die lokalen Semantic Building Modeler-Instanzen die Fragmente zunächst herunterladen und anschließend in die eigenen Konfigurationen integrieren können. Eine solche zentralisierte Form müsste in ihrer einfachsten Form User-Interfaces bereitstellen, über die die Nutzer Fragmente hochladen und bereits vorhandene Fragmente editieren könnten. Beide Varianten haben architekturspezifische Vor- und Nachteile.

Die dezentrale Form ist einfacher zu implementieren. Konzeptuell würde sie aus einer Datenbank mit sehr einfacher Struktur bestehen, die neben den Fragment-URIs auch Volltextbeschreibungen der Fragment-Funktion enthält. Durch die Anbindung einer *Meta-Search-Engine* wie *Apache Solr*[20] oder *elasticsearch*[21] an die Datenbank könnten die Nutzer die vorhandenen Fragmente durchsuchen und eine Menge für sie geeigneter Konfigurationsfragmente zurückgeliefert bekommen. Der einfachen Implementation und Wartung gegenüber steht der Nachteil, dass diese Architektur voraussetzt, dass sämtliche Nutzer des Systems, die bereit sind, ihre Fragmente mit anderen Anwendern zu teilen, auch über die Möglichkeit verfügen, diese online bereitzustellen. Hierfür erforderlich ist der Zugriff auf einen Webspace. Im Idealfall sollte darüber hinaus garantiert werden, dass einerseits die Fragment-URIs andererseits auch der -inhalt stabil bleiben, sich diese also nach der Veröffentlichung nicht mehr ändern. Hier bietet die zentralisierte Variante große Vorteile, da sie sämtliche Fragmente verwaltet und die für den Zugriff erforderlichen URIs bereitstellt. Auch für das Problem der *Inhaltsstabilität* der Fragmente sind diverse Ansätze denkbar. So ist eine zentralisierte Architektur deutlich besser in der Lage, eine Versionsverwaltung der Konfigurationen vorzunehmen, beispielsweise indem das System sämtliche Versionen eines Fragments vorhält und für jede neue auch eine neue URI bereitstellt. Allerdings ist für ein solches System der Implementations- und Wartungsaufwand deutlich höher und somit naturgemäß auch die Kosten für Betrieb und Weiterentwicklung.

Der Aufbau solcher Bibliotheken ist nicht nur für die Verwaltung und den Austausch von Konfigurationsteilen sinnvoll. Denkbar ist der Einsatz solcher Webservices auch für die Gebäudekomponenten, die in Form von 3D-Modellen vorliegen. Auch die applizierbaren

[20] Apache Solr: http://lucene.apache.org/solr/
[21] elasticsearch: http://www.elasticsearch.org/

Texturen könnten durch einen solchen Service verwaltet und bereitgestellt werden. Wie in den vorherigen Abschnitten wiederholt betont wurde, stellt das kategorienbasierte Laden von 3D-Modell-Komponenten eine einfache, aber sehr wirkungsvolle Möglichkeit dar, ohne großen Aufwand Variationen innerhalb der erzeugten Geometrie zu realisieren. Betrachtet man beispielsweise die verwendeten 3D-Modelle für Fenster oder Türen, so stellen diese einen sehr einfachen Ansatz dar, um eine große visuelle Vielfalt in den berechneten Gebäuden umzusetzen. Das Kernkonzept besteht in der Organisation vergleichbarer Komponenten in Kategorien, aus denen der Semantic Building Modeler während der Gebäudeerzeugung zufallsbasiert auswählen kann. Die Kategorien beschränken sich nicht auf rein funktionale Unterscheidungen wie *Tür* oder *Fenster*, sondern können beliebig fein untergliedert werden. Denkbar wären Kategorien wie *Vierelementige Jugendstilfenster* oder ähnliches. Je mehr Modelle für die jeweiligen Kategorien zur Verfügung stehen, desto größer wird auch der Variantenreichtum sein, der basierend auf diesen erzeugt werden kann. In Kombination mit den verschiedenen vom Semantic Building Modeler bereitgestellten Algorithmen, beispielsweise den unterschiedlichen Verfahren zur Grundrisserzeugung und –modifikation, lassen sich über einen solchen Ansatz abwechslungsreiche Stadtszenerien erzeugen.

Auch für die Verwaltung von 3D-Komponentenmodellen und Texturen sind wiederum die vorab erwähnten Architekturvarianten denkbar. Allerdings verschärft sich bei wachsendem Umfang der Bibliotheken die Problematik des erforderlichen Speicherplatzes in der zentralisierten Variante. Da sowohl 3D-Modelle als auch Texturen deutlich mehr Speicherplatz benötigen als textbasierte Konfigurationsfragmente, sollte die Wahl für eine der beiden Architekturvarianten den notwendigen Speicherplatz und die daraus erwachsenden Kosten berücksichtigen. Grundsätzlich entspricht die Konzeption des Services allerdings der skizzierten Funktionalität der Konfigurationsfragmente. Die Objekte werden entweder auf einem verteilten oder zentralen Repository abgelegt und die URIs für die lokalen Semantic Building Modeler-Instanzen bereitgestellt. Zu jedem gespeicherten Objekt wird eine Beschreibung auf dem Server abgelegt, die den Nutzer bei der Suche nach geeigneten Kategorien unterstützt. Auch der Aufbau einer solchen Asset-Bibliothek ist aus technischer Sicht vergleichsweise einfach umzusetzen. Allerdings bestehen natürlich auch hier deutliche Unterschiede bezüglich Implementations- und Wartungsaufwand zwischen der zentralen und der dezentralen Servicearchitektur.

Unabhängig von der Fragestellung, ob die Webservices für die Verwaltung von Konfigurationsfragmenten, 3D-Modellen oder Texturen eingesetzt werden und ob sie zentral oder dezentral organisiert sind, wäre auch für diese Funktionalität eine Integration in einer Softwarekomponente wünschenswert. Somit ergibt sich eine weitere mögliche Forderung an eine integrierte Entwicklungsumgebung, die die aktuell noch getrennten Softwaresysteme miteinander kombiniert. Ein solches System sollte neben den Basisfunktionen eines 3D-Modellierungswerkzeuges und einer XML-Entwicklungsumgebung darüber hinaus auch noch die Möglichkeit bieten, online verfügbare Bibliotheken mit Konfigurationsfragmenten, 3D-Modellen und Texturen zu durchsuchen und die gefundenen Objekte direkt in die zu erstellenden Gebäude- und Stadtkonfigurationen zu integrieren.

15.3 **Erweiterung des Semantic Building Modelers**

Neben den vorab genannten potentiellen Erweiterungen, die größtenteils auf die Implementation zusätzlicher Services und die Integration aktuell externer Funktionen in das System abzielen, gibt es einige Aspekte, um die der Semantic Building Modeler selber erweitert werden könnte.

Zunächst ist die Implementation weiterer Funktionen für die Integration zusätzlicher Komponententypen wünschenswert. Denkbar wären beispielsweise Balkone, Regenrinnen oder Schornsteine. Bei jedem Komponententyp sollte sorgfältig überlegt werden, wie er verarbeitet werden kann. Bereits für die im Prototyp vorhandenen Komponenten ist eine Unterscheidung in solche Objekte erforderlich, die prozedural erzeugt werden und in solche, bei denen das Objekt vollständig geladen und anschließend bei Bedarf skaliert und ausgerichtet wird, bevor es am Gebäudemodell appliziert wird. Balkone sind ein gutes Beispiel für solch einen Komponententyp, der vorgefertigt geladen wird. Eine prozedurale Erzeugung macht für solche Objekte wenig Sinn, da die Beschreibung des Objektes und die Implementation der für die Berechnung erforderlichen Algorithmen deutlich aufwendiger sein wird, als die Modellierung innerhalb eines Softwaresystems wie 3ds Max. Der Semantic Building Modeler stellt für deren Verarbeitung bereits eine Reihe von Algorithmen und Methoden zur Verfügung. Diese Funktionsbibliotheken umfassen Prozeduren für das Laden, Skalieren, Ausrichten und Positionieren der Komponenten am Gebäude selber. Je nach Komponententyp sind hier eventuell Modifikationen und Erweiterungen erforderlich. Allerdings sollten diese in den meisten Fällen bezüglich ihres Umfangs überschaubar sein, da die Kernfunktionen bereits vorhanden sind.

Dies ist auch bei vielen anderen Komponenten der Fall, bei denen die Konstruktion zumindest in Teilen prozedural erfolgt. Die vorab genannten Regenrinnen sind ein gutes Beispiel für Objekte dieser Art. Sie ähneln in ihrer Konstruktion den bereits innerhalb des Systems vorhandenen Gesimsen, bei denen nur das Profil als Polygon aus einer Modelldatei geladen wird. Bei solchen Objekten reicht der einfache Skalierungsansatz nicht aus, da das System uniform skaliert. Vergrößert oder verkleinert man nun aber ein Regenrinnen- oder Gesimssegment gleichmäßig in alle Richtungen, so ändern sich mit der Länge des Objekts automatisch auch Höhe und Breite. Eine solche Berechnung ist offensichtlich für Komponenten dieser Art nicht wünschenswert. Vielmehr sollte nur die Länge frei wählbar sein, damit eine Regenrinne an einer Wand platziert werden kann und dabei Höhe und Breite konstant bleiben. Hier eignet sich das für Gesimse implementierte Extrusionsverfahren sehr gut, da es in der Lage ist, Regenrinnen derart zu erstellen, dass sie exakt der Geometrie der Zielfassade entsprechen. Offensichtlich kann also auch in diesem Fall auf die vorhandenen Funktionen des Systems zurückgegriffen werden, so dass die Implementation des neuen Komponententyps vergleichsweise einfach realisiert werden kann. Auch für die notwendigen Verbindungsstücke, die bei Regenrinnen erforderlich sind, wenn diese von einer horizontalen in eine vertikale Ausrichtung übergehen, existieren bereits Methoden innerhalb des Semantic Building Modelers. Als Beispiel kann hier die Berechnung von Verbindungsstücken zwischen Gesimsen an Hauswänden mit unterschiedlicher Ausrichtung herangezogen werden. Für die Erzeugung solcher um das gesamte Gebäude laufende Fascias ist es erforderlich, Verbindungselemente an den Gebäudeecken zur berechnen. Diese Problemstellung ähnelt in gewisser Weise dem Verbindungsproblem bei Regenrinnen und könnte unter Rückgriff auf den implementierten Algorithmus in ähnlicher Art und Weise umgesetzt werden.

An den beiden vorab genannten Beispielen wird deutlich, dass die Erweiterung des Systems um neue Komponenten wünschenswert ist und dabei auf vorhandene Funktionen und Algorithmen zurückgegriffen werden kann. Nichtsdestotrotz sind hierfür Programmierkenntnisse und ein Basiswissen in analytischer Geometrie erforderlich. Sobald die erforderlichen Routinen durch einen Entwickler implementiert sind, stehen sie für den Anwender zur Verfügung, der diese dann über einfache Parameter steuern und verwenden kann. Dadurch wird der Punkt erreicht, an dem der normale Nutzer nicht mehr länger über mathematische Grundlagen und eine gute räumliche Vorstellung verfügen muss, um Gebäudekonstruktionen vorzunehmen, sondern das System über semantisch basierte

Parameter steuert. Jede Erweiterung dieser Art erhöht somit die Modellierungsmächtigkeit für den Nutzer, ohne dass sich dieser in die teils komplizierte Algorithmik einarbeiten muss. Aus Sicht der Implementation des Semantic Building Modelers ist dieser explizit mit der Zielsetzung einer möglichst hohen Erweiterbarkeit konzipiert, sowohl in Bezug auf die vorhandene Klassenstruktur als auch in Bezug auf den Zugriff auf bereits vorhandene Algorithmen und Methoden.

16 Literaturverzeichnis

[Ab01] Abdelguerfi, M.: 3D synthetic environment reconstruction. Kluwer Academic Publishers, Boston, 2001.

[AD86] Abelson, H.; DiSessa, A. A.: Turtle geometry. The computer as a medium for exploring mathematics. MIT Press, Cambridge, Mass, 1986, c1980.

[AC03] ACM: Proceedings of the nineteenth annual symposium on Computational geometry, 2003.

[Ad94] Adobe: PostScript language. Tutorial and cookbook. Addison-Wesley, Reading, Mass. [u.a.], 1994.

[Ai95] Aichholzer, O. et al.: A Novel Type of Skeleton for Polygons. In J. UCS, 1995, 1; S. 752–761.

[AAP02] Aichholzer, O.; Aurenhammer, F.; Palop, B.: Quickest paths, straight skeletons, and the city Voronoi diagram: Proceedings of the eighteenth annual symposium on Computational geometry. ACM, New York, NY, USA, 2002; S. 151–159.

[ARB07] Aliaga, D. G.; Rosen, P. A.; Bekins, D. R.: Style Grammars for Interactive Visualization of Architecture. In IEEE Transactions on Visualization and Computer Graphics, 2007, 13; S. 786–797.

[AVB08] Aliaga, D. G.; Vanegas, C. A.; Beneš, B.: Interactive example-based urban layout synthesis: ACM SIGGRAPH Asia 2008 papers. ACM, New York, NY, USA, 2008; S. 160:1-160:10.

[Ap08] Apetri, M.: 3D-Grafik-Programmierung. [alle mathematischen Grundlagen ; von einfachen Rasteralgorithmen bis hin zu Landscape-Generation ; 3D-Grafik in C++, optimaler Einstieg in OpenGL und Direct3D ; inklusive CD]. Mitp, Heidelberg, 2008.

[Ar91] Arvo, J. Hrsg.: Graphics gems II. Program of computer graphics. Academic Pr., Boston [u.a.], 1991.

[As09] Aschwanden, G. et al.: Evaluation of 3D city models using automatic placed urban agents: CONVR Conference Proceedings, University Of Sydney, 2009.

[BR99] Baeza-Yates, R.; Ribeiro-Neto, B.: Modern information retrieval. ACM Press; Addison-Wesley, 1999, New York, Harlow, England.

[Ba05] Balzert, H.: Lehrbuch der Objektmodellierung. Analyse und Entwurf mit der UML 2 ; mit CD-ROM und e-learning-Online-Kurs. Elsevier, Spektrum, Akad. Verl, Heidelberg, München, 2005.

[BY03] Barequet, G.; Yakersberg, E.: Morphing between shapes by using their straight skeletons: Proceedings of the nineteenth annual symposium on Computational geometry. ACM, New York, NY, USA, 2003; S. 378–379.

[Ba07] Batty, M.: Cities and Complexity: Understanding Cities with Cellular Automata, Agent-Based Models, and Fractals. The MIT Press, 2007.

[Ba72] Baumgart, B. G.: Winged edge polyhedron representation. Stanford University, Stanford, CA, USA, 1972.

[BA05] Bekins, D.; Aliaga, D. G.: Build-by-Number: Rearranging the Real World to Visualize Novel Architectural Spaces. In Visualization Conference, IEEE, 2005, 0; S. 19.

[BB06] Bender, M.; Brill, M.: Computergrafik. Ein anwendungsorientiertes Lehrbuch. Hanser, München [u.a.], 2006.

[BAH09] Benstead, L.; Astle, D.; Hawkins, K.: Beginning OpenGL game programming. Course Technology, Boston, MA, 2009.

[Be08] Berg, M. d.: Computational geometry. Algorithms and applications. Springer, Berlin, 2008.

[Be08] Berg, M. d. et al.: Computational Geometry: Algorithms and Applications. Springer-Verlag TELOS, Santa Clara, CA, USA, 2008.

[Bi01] Birch, P. J. et al.: Rapid procedural-modelling of architectural structures: Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage. ACM, New York, NY, USA, 2001; S. 187–196.

[Bl08] Bloch, J.: Effective Java. Programming language guide. Addison-Wesley, Upper Saddle River, NJ, 2008.

[Bo13] Boardman, T.: Getting started in 3D with 3ds Max. Model, texture, rig, animate, and render in 3ds Max. Focal Press, New York, 2013.

[BWS10] Bokeloh, M.; Wand, M.; Seidel, H.-P.: A connection between partial symmetry and inverse procedural modeling. In ACM Trans. Graph., 2010, 29; S. 104:1-104:10.

[BS02] Bolz, J.; Schröder, P.: Rapid evaluation of Catmull-Clark subdivision surfaces: Proceedings of the seventh international conference on 3D Web technology. ACM, New York, NY, USA, 2002; S. 11-17.

[Bo89] Borgida, A. et al.: CLASSIC: a structural data model for objects. In SIGMOD Rec, 1989, 18; S. 58–67.

[BD74] Braun, H.; Drixelius, W.: Formen der Kunst. Eine Einführung in die Kunstgeschichte. Martin Lurz, München, 1974.

[Br64] Braunfels, W.: Knaurs Weltkunstgeschichte: Frühchristliche Kunst, Byzanz, Mittelalter, Renaissance, Barock, Moderne Kunst. Droemer, 1964.

[Br00] Brenner, C.: Towards fully automatic generation of city models. In International Archives of Photogrammetry and Remote Sensing, 2000, 33; S. 84–92.

[BM05] Bruns, K.; Meyer-Wegener, K.: Taschenbuch der Medieninformatik. Carl Hauser Verlag GmbH & CO. KG; Fachbuchverl. Leipzig im Carl-Hanser-Verl., München, Wien, 2005.

[Bü10] Bütow, S.: Konzeption und Implementierung webbasierter Tools zur Unterstützung der WLAN-Projektierung in „indoor"-Szenarien. Diplomarbeit, Dresden, 2010.

[Ca04] Cacciola, F.: A CGAL implementation of the Straight Skeleton of a Simple 2D Polygon with Holes: 2nd CGAL User Workshop, Polytechnic Univ., Brooklyn, New York, USA, 2004.

[CC98] Catmull, E.; Clark, J.: Recursively generated B-spline surfaces on arbitrary topological meshes. Seminal graphics. ACM, New York, NY, USA, 1998; S. 183-188.

[CV07] Cheng, S.-W.; Vigneron, A.: Motorcycle Graphs and Straight Skeletons. In Algorithmica, 2007, 47; S. 159–182.

[CSW95] Chin, F.; Snoeyink, J.; Wang, C. A.: Finding the medial axis of a simple polygon in linear time: Algorithms and Computations. Springer, 1995; S. 382–391.

[Ch56] Chomsky, N.: Three models for the description of language. In IRE Transactions on Information Theory, 1956, 2; S. 113–124.

[Ch02] Chomsky, N.: Syntactic structures. Mouton de Gruyter, Berlin, New York, 2002.

[Ch12] Christian Klaß: Prozedurale Schöpfung. Künstliche Welten auf Knopfdruck. http://www.golem.de/news/prozedurale-schoepfung-kuenstliche-welten-auf-knopfdruck-1212-96323.html, 25.04.2013.

[CKM04] Clingman, D.; Kendall, S.; Mesdaghi, S.: Practical Java game programming. Charles River Media, Hingham, Mass, 2004.

[Co10] Cormen, T. H.: Algorithmen - eine Einführung. Oldenbourg, München [u.a.], 2010.

[DYB98] Debevec, P.; Yu, Y.; Boshokov, G.: Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping. University of California at Berkeley, Berkeley, CA, USA, 1998.

[DTM96] Debevec, P. E.; Taylor, C. J.; Malik, J.: Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1996; S. 11-20.

[DKT98] DeRose, T.; Kass, M.; Truong, T.: Subdivision surfaces in character animation: Proceedings of the 25th annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1998; S. 85-94.

[De03] Deussen, O.: Computergenerierte Pflanzen. Technik und Design digitaler Pflanzenwelten. Springer, Berlin, 2003.

[De98] Deussen, O. et al.: Realistic modeling and rendering of plant ecosystems: Proceedings of the 25th annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1998; S. 275–286.

[DG02] Devillers, O.; Guigue, P. Devillers, O.; Guigue, P.: Faster Triangle-Triangle Intersection Tests, 2002.

[Di59] Dijkstra, E. W.: A Note on Two Problems in Connexion with Graphs. In NUMERISCHE MATHEMATIK, 1959, 1; S. 269-271.

[Du07] Dummer, S.: Modellierung gotischer Pfeiler mittels modifzierter L-Systeme. Diplomarbeit, Frankfurt am Main, 2007.

[Dy08] Dylla, K. et al.: Rome reborn 2.0: A case study of virtual city reconstruction using procedural modeling techniques. In Computer Graphics World, 2008, 16; S. 25.

[Eb02] Ebert, D. S. et al.: Texturing and Modeling. A Procedural Approach. Morgan Kaufmann [Imprint]; Elsevier Science & Technology Books, San Diego, 2002.

[En10] Enterprise Lab Wiki: Welcome to the User-Manual for ColladaLoader. https://wiki.enterpriselab.ch/el/projects:colladaloader, 28.03.2013.

[EE98] Eppstein, D.; Erickson, J.: Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions: Proceedings of the fourteenth annual symposium on Computational geometry. ACM, New York, NY, USA, 1998; S. 58-67.

[Er05] Ericson, C.: Real-time collision detection. Elsevier, Amsterdam, Boston, 2005.

[EEP02] Erk, K.; Erk-Priese; Priese, L.: Theoretische Informatik. Eine umfassende Einführung. Springer, Berlin [u.a.], 2002.

[Es] Esri: CityEngine Features. http://www.esri.com/software/cityengine/features.html, 22.05.2012.

[Es11] Esri: Esri Acquires 3D Software Company Procedural. http://www.esri.com/news/releases/11-3qtr/esri-acquires-3D-software-company-procedural.html, 22.05.2012.

[Fa10] Faculty of mechanical engineering: Cyberwalk. Ungehindertes Gehen in virtuellen Welten. http://www.amm.mw.tum.de/forschung/abgeschlossene-projekte/cyberwalk/.

[Fi08] Finkenzeller, D.: Modellierung komplexer Gebäudefassaden in der Computergrafik. Universitätsverlag Karlsruhe, Karlsruhe, 2008.

[Fo00] Fowler, M.: Refactoring. Wie Sie das Design vorhandener Software verbessern. Addison-Wesley, München [u.a.], 2000.

[Fr08] Fry, B.: Visualizing data. O'Reilly Media, Inc., Beijing, Cambridge, 2008.

[Fu04] Funkhouser, T. et al.: Modeling by example. In ACM Trans. Graph., 2004, 23; S. 652–663.

[Ga12] Gamma, E.: Design patterns. Elements of reusable object-oriented software. Addison-Wesley, Boston, 2012.

[GJ71] George Stiny; James Gips: Shape Grammars and the Generative Specification of Painting and Sculpture: Segmentation of Buildings for 3DGeneralisation. In: Proceedings of the Workshop on generalisation and multiple representation Leicester, 1971.

[Gl98] Glassner, A. S.: Graphics gems. Academic Press, San Diego, CA, 1998.

[GW87] Gonzales, R. C.; Wintz, P.: Digital Image Processing. Addison-Wesley Publishing Company, Inc., 1987.

[Go] Gottschalk, S. A.: Collision queries using oriented bounding boxes.

[GXK13] Greenberg, I.; Xu, D.; Kumar, D.: Processing. Creative coding and generative art in processing 2. Friends of Ed; Springer [distributor], Berkeley, Calif, London, 2013.

[Gr09] Gregory, J.: Game engine architecture. A K Peters, Wellesley, Mass, 2009.

[Gr03] Greuter, S. et al.: Real-time procedural generation of ‘pseudo infinite’ cities: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia. ACM, New York, NY, USA, 2003; S. 87-ff.

[GPZ04] Gross, M.; Pauly, M.; Zwicker, M.: Point-based computer graphics siggraph 2004 course notes, 2004.

[GHH01] Gruben, G.; Hirmer, M.; Hirmer, A.: Griechische Tempel und Heiligtümer. Hirmer, München, 2001.

[GS02] Gumm, H.-P.; Sommer, M.: Einführung in die Informatik. Oldenbourg, München, Wien, 2002.

[HMv09] Haegler, S.; Müller, P.; van Gool, L.: Procedural modeling for digital cultural heritage. In Journal on Image and Video Processing, 2009, 2009; S. 7.

[HBW06] Hahn, E.; Bose, P.; Whitehead, A.: Persistent realtime building interior generation: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames. ACM, New York, NY, USA, 2006; S. 179-186.

[Ha99] Harold, E. R.: XML bible. IDG Books Worldwide, Foster City, Calif, 1999.

[Ha12] Hartmann, P.: Mathematik für Informatiker. Ein praxisbezogenes Lehrbuch. Vieweg & Teubner, Wiesbaden, 2012.

[Ha05] Havemann, S.: Generative Mesh Modeling. Dissertation, Braunschweig, 2005.

[HF01] Havemann, S.; Fellner, D.: A versatile 3D model representation for cultural reconstruction: Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage. ACM, New York, NY, USA, 2001; S. 205–212.

[HF04] Havemann, S.; Fellner, D. W.: Generative parametric design of Gothic window tracery: Proceedings of the 5th International conference on Virtual Reality, Archaeology and Intelligent Cultural Heritage. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2004; S. 193–201.

[He94] Heckbert, P. S.: Graphics gems IV. Academic Press Professional, Boston [etc.], 1994.

[Ho96] Hoppe, H.: Progressive meshes: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1996; S. 99-108.

[Ho97] Hoppe, H.: View-dependent refinement of progressive meshes: Proceedings of the 24th annual conference on Computer graphics and interactive techniques. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, 1997; S. 189–198.

[Ho98] Hoppe, H.: Efficient implementation of progressive meshes. In Computers & Graphics, 1998, 22; S. 27–36.

[HYN03] Hu, J.; You, S.; Neumann, U.: Approaches to Large-Scale Urban Modeling. In IEEE Comput. Graph. Appl., 2003, 23; S. 62–69.

[Ja07] Janusch, S.: Konzeption & Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden. Diplomarbeit, Darmstadt, 2007.

[jd] jdom: Who We Are, 22.03.2013.

[Jo99] Josuttis, N. M.: The C++ standard library. A tutorial and reference. Addison-Wesley, Harlow, 1999.

[KW11] Kelly, T.; Wonka, P.: Interactive architectural modeling with procedural extrusions. In ACM Trans. Graph, 2011, 30; S. 14:1-14:15.

[Ki92] Kirk, D. Hrsg.: Graphics gems III. [IBM version]. AP Professional, San Diego, Calif. [u.a.], 1992.

[Kl05] Klawonn, F.: Grundkurs Computergrafik mit Java. Die Grundlagen verstehen und einfach umsetzen mit Java 3D. Vieweg, Wiesbaden, 2005.

[Ko06] Koch, W.: Baustilkunde. Wissen-Media-Verl., Gütersloh, München, 2006.

[Ko06] Korpela, J. K.: Unicode explained. O'Reilly, Sebastopol, CA, 2006.

[KS09] Krüger, G.; Stark, T.: Handbuch der Java-Programmierung. Addison-Wesley, München, Boston, Mass. [u.a.], 2009.

[KN09] Krumke, S. O.; Noltemeier, H.: Graphentheoretische Konzepte und Algorithmen. Vieweg+Teubner Verlag / GWV Fachverlage GmbH, Wiesbaden, Wiesbaden, 2009.

[Ku10] Kunz, A.: Web-3D-Welten systematisch erzeugen. Diplomica-Verl, Hamburg, 2010.

[L.00] L. Denise Pinnel et al.: Design of Visualizations for Urban Modeling: IN PROCEEDINGS OF VISSYM '00 — JOINT EUROGRAPHICS - IEEE TCVG SYMPOSIUM ON VISUALIZATION, 2000.

[La12] Lang, H.-W.: Algorithmen in Java. Sortieren, Textsuche, Codierung, Kryptografie. Oldenbourg, München, 2012.

[LD03] Laycock, R. G.; Day, A. M.: Automatically generating large urban environments based on the footprint data of buildings: Proceedings of the eighth ACM symposium on Solid modeling and applications. ACM, New York, NY, USA, 2003; S. 346–351.

[LD03] Laycock, R. G.; Day, A. M.: Automatically Generating Roof Models from Building Footprints: WSCG, 2003.

[LDG01] Legakis, J.; Dorsey, J.; Gortler, S.: Feature-based cellular texturing for architectural models. In Proceedings of the 28th conference on Computer graphics and interactive techniques, 2001; S. 309–316.

[Le11] Lengyel, E.: Game engine gems 2. CRC Press, Natick, Mass, 2011.

[Li1968b] Lindenmayer, A.: Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. In Journal of Theoretical Biology Journal of Theoretical Biology, 1968b, 18; S. 300–315.

[Li68] Lindenmayer, A.: Mathematical models for cellular interaction in development: Parts I. In Journal of Theoretical Biology, 1968, 18.

[Li74] Lindenmayer, A.: Adding Continuous Components to L-Systems: L Systems, Most of the papers were presented at a conference in Aarhus, Denmark. Springer-Verlag, London, UK, UK, 1974; S. 53-68.

[Li11] Lipp, M. et al.: Interactive Modeling of City Layouts using Layers of Procedural Content. In Comput. Graph. Forum, 2011, 30; S. 345–354.

[LWW08] Lipp, M.; Wonka, P.; Wimmer, M.: Interactive visual editing of grammars for procedural architecture: ACM SIGGRAPH 2008 papers. ACM, New York, NY, USA, 2008; S. 102:1-102:10.

[Ma09] Martin, R. C.: Clean-Code. Refactoring, Patterns, Testen und Techniken für sauberen Code ; [Kommentare, Formatierung, Strukturierung, Fehler-Handling und Unit-Tests, zahlreiche Fallstudien, best practices, Heuristiken und Code-smells]. Mitp, Heidelberg, München, Landsberg, Frechen, Hamburg, 2009.

[MP96] Měch, R.; Prusinkiewicz, P.: Visual models of plants interacting with their environment: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1996; S. 397–410.

[MB11] Meier, S.; Borkowski, A.: Geometrie stochastischer Signale. De Gruyter, Berlin, New York, 2011.

[Me07] Merrell, P.: Example-based model synthesis: Proceedings of the 2007 symposium on Interactive 3D graphics and games. ACM, New York, NY, USA, 2007; S. 105-112.

[MM08] Merrell, P.; Manocha, D.: Continuous model synthesis: ACM SIGGRAPH Asia 2008 papers. ACM, New York, NY, USA, 2008; S. 158:1-158:7.

[MM09] Merrell, P.; Manocha, D.: Constraint-based model synthesis: 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling. ACM, New York, NY, USA, 2009; S. 101-111.

[MM11] Merrell, P.; Manocha, D.: Model Synthesis: A General Procedural Modeling Algorithm. In IEEE Transactions on Visualization and Computer Graphics, 2011, 17; S. 715–728.

[Me08] Meyer-Bohe, W.: Baukonstruktion. Ein Kompendium. Kohlhammer, Stuttgart, 2008.

[MS02] Müller, M.; Schwarzer, S.: Eine Einführung in C++. http://www.icp.uni-stuttgart.de/Courses_and_Lectures/C++/script/node3.html, 18.05.2012.

[Mü99] Müller, P.: Prozedurales Modellieren einer Stadt. Semesterarbeit, Zürich, 1999.

[Mü01] Müller, P.: Design und Implementation einer Preprocessing Pipeline zur Visualisierung prozedural erzeugter Stadtmodelle. Diplomarbeit, Zürich, 2001.

[Mü06] Müller, P. et al.: Procedural modeling of buildings. In ACM Trans. Graph, 2006, 25; S. 614-623.

[Mü07] Müller, P. et al.: Image-based procedural modeling of facades: ACM SIGGRAPH 2007 papers. ACM, New York, NY, USA, 2007.

[Mu11] Murdock, K.: 3ds Max 2012 bible. Wiley, Indianapolis, Ind, 2011.

[Na99] Nagl, M.: Softwaretechnik mit Ada 95. Entwicklung grosser Systeme. Vieweg, Braunschweig, 1999.

[NH99] North, S.; Hermans, P.: XML in 21 Tagen. Schritt für Schritt Einstieg in Fähigkeiten und Konzepte ; praktischer Einsatz anhand beispielhafter XML-Applikationen. Markt-und-Technik-Verlag, München, 1999.

[O'85] O'Rourke, J.: Finding minimal enclosing boxes. In International journal of computer & information sciences, 1985, 14; S. 183–199.

[OM04] Orlamünder, D.; Mascolus, W.: Computergrafik und OpenGL. Eine systematische Einführung ; mit 26 Übungen. Fachbuchverl. Leipzig im Carl-Hanser-Verl., München, Wien, 2004.

[OHM00] Orwant, J.; Hietaniemi, J.; Macdonald, J.: Algorithmen mit Perl. O'Reilly, Beijing, Cambridge, Farnham, Köln, Paris, Sebastopol, Taipei, Toyko, 2000.

[PS98] Petr Felkel; Stepan Obdrzalek: Straight Skeleton Implementation: Proceedings of Spring Conference on Computer Graphics, 1998; S. 210–218.

[Pi83] Pischel, G.: Grosse Kunstgeschichte der Welt. Malerei ; Plastik ; Architektur ; Kunsthandwerk. Südwest-Verlag, München, 1983.

[Pi97] Pixar: Geri´s Game. http://www.pixar.com/shorts/gg/index.html, 23.05.2012.

[PK02] Prinz, P.; Kirch-Prinz, U.: C++ lernen und professionell anwenden. [gezielter Lernerfolg durch überschaubare Kapiteleinheiten ; vollständige Darstellung Schritt für Schritt ; konsequent objektorientiert programmieren ; Microsoft-Visual-C++-book-Edition, Programmierbeispiele und Musterlösungen]. Mitp, Bonn, 2002.

[PJM94] Prusinkiewicz, P.; James, M.; Měch, R.: Synthetic topiary: Proceedings of the 21st annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1994; S. 351-358.

[PL96] Prusinkiewicz, P.; Lindenmayer, A.: The algorithmic beauty of plants. Springer-Verlag New York, Inc, New York, NY, USA, 1996.

[Pu07] Puntigam, F.: Skriptum zu Objektorientierte Programmierung. http://www.complang.tuwien.ac.at/franz/objektorientiert/skript07-1seitig.pdf, 18.05.2012.

[RF07] Reas, C.; Fry, B.: Processing. A programming handbook for visual designers and artists. MIT Press, Cambridge, Mass, 2007.

[Re06] Rechenberg, P.: Informatik-Handbuch. Hanser, München [u.a.], 2006.

[Re83] Reeves, W. T.: Particle systems—a technique for modeling a class of fuzzy objects: Proceedings of the 10th annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1983; S. 359-375.

[RB85] Reeves, W. T.; Blau, R.: Approximate and probabilistic algorithms for shading and rendering structured particle systems: Proceedings of the 12th annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1985; S. 313-322.

[Ri09] Richter, M.: Ein Simulator zur zufallsgesteuerten Generierung von archäologisch / historischen 3D Gebäudeensembles aus generischen Gebäudemodellen. Diplomarbeit, Köln, 2009.

[RSJ96] Rob Ingram; Steve Benford; John Bowers: Building Virtual Cities: Applying Urban Planning Principles to the : PROCEEDINGS OF THE ACM SYMPOSIUM ON VIRTUAL REALITY SOFTWARE AND TECHNOLOGY (VRST96. ACM Press, 1996; S. 83–91.

[Ro96] Roberto Ierusalimschy; Luiz Henrique de Figueiredo; Luiz Henrique; Figueiredo Waldemar; Waldemar Celes Filho Roberto Ierusalimschy et al.: Lua - an Extensible Extension Language, 1996.

[Ro10] Rome Reborn Project: About. http://www.romereborn.virginia.edu/about.php, 25.04.2013.

[Sa07] Sagheb, K.: Generierung gotischer Maßwerkfenster durch fraktale Geometrien. Diplomarbeit, Frankfurt am Main, 2007.

[SP04] Schmidt-Colinet, A.; Plattner, G. A.: Antike Architektur und Bauornamentik. Grundformen und Grundbegriffe. WUV, Wien, 2004.

[SE02] Schneider, P. J.; Eberly, D.: Geometric Tools for Computer Graphics. Elsevier Science Inc, New York, NY, USA, 2002.

[SE02] Schneider, P. J.; Eberly, D.: Geometric Tools for Computer Graphics. Elsevier Science Inc, New York, NY, USA, 2002.

[Sc07] Schrader, R.: Prozedural unterstützte Generierung von Gebäudemodellen für interaktive Anwendungen. Diplomarbeit, Koblenz, 2007.

[Sh00] Sharp, B.: Subdivision Surface Theory. http://www.gamasutra.com/view/feature/3177/subdivision_surface_theory.php, 22.05.2012.

[Sh02] Shepherd, D.: XML in 21 Tagen. Daten plattformübergreifend austauschen und modellieren ; praktische XML-Entwicklung und Implementierung ; DTD, XPath, Xlink, SAX, DOM, XSLT u.v.m. Markt-und-Technik-Verlag; SAMS, München/Germany, [München], 2002.

[Sh08] Shreiner, D.: OpenGL programming guide. The official guide to learning OpenGL, version 2.1. Addison-Wesley, Upper Saddle River, NJ, 2008.

[Si04] Simmons, R.: Hardcore Java. O'Reilly, Sebastopol, CA, 2004.

[Si03] Sippach, F.: Geometrisch-physikalische Geometrisch-physikalische Gebäudebeschreibung im Internet mit XML. Studienarbeit, Weimar, 2003.

[SWW11] Skulschus, M.; Wiederstein, M.; Winterstone, S.: XML-Schema. Comelio-Medien, Berlin, 2011.

[Sm84] Smith, A. R.: Plants, fractals, and formal languages: Proceedings of the 11th annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1984; S. 1–10.

[SB87] Snyder, J. M.; Barr, A. H.: Ray tracing complex models containing surface tessellations: Proceedings of the 14th annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1987; S. 119–128.

[So11] Socher, R.: Mathematik für Informatiker. Mit Anwendungen in der Computergrafik und Codierungstheorie ; mit 6 Tabellen, 76 Beispielen und 294 Aufgaben. Fachbuchverl. Leipzig, München, 2011.

[St98] Stam, J.: Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values: Proceedings of the 25th annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, 1998; S. 395–404.

[SJ03] Stefan Greuter; Jeremy Parker: Undiscovered Worlds – Towards a Framework for Real-Time: In Proc. of the Fifth Intern. Digital Arts and Culture Conference. Press, 2003.

[St75] Stiny, G.: Pictorial and formal aspects of shape and shape grammars. Birkhaeuser, Basel, Stuttgart, 1975.

[SD05] Sven Havemann; Dieter W. FellnerSven Havemann; Dieter W. Fellner: Progressive Combined BReps, 2005.

[TV03] Tanase, M.; Veltkamp, R. C.: Polygon decomposition based on the straight line skeleton: Proceedings of the nineteenth annual symposium on Computational geometry, 2003; S. 58–67.

[Ta09] Tanenbaum, A. S.: Moderne Betriebssysteme. Pearson Studium, München, Boston [u.a.], 2009.

[Te07] Text Encoding Initiative: Introducing the Guidelines. http://www.tei-c.org/Support/Learn/intro.xml, 01.03.2013.

[Te07] Text Encoding Initiative: TEI:History. http://www.tei-c.org/About/history.xml, 28.02.2012.

[va02] van der Vlist, E.: XML Schema. O'Reilly, Sebastopol, CA, 2002.

[Va10] Vanegas, C. A. et al.: Modelling the appearance and behaviour of urban spaces: Computer Graphics Forum, 2010; S. 25–42.

[Va12] Vanegas, C. A. et al.: Inverse design of urban procedural models. In ACM Trans. Graph., 2012, 31; S. 168:1-168:11.

[Va12] Vanegas, C. A. et al.: Procedural Generation of Parcels in Urban Modeling. In Comp. Graph. Forum, 2012, 31; S. 681–690.

[Vo07] Vonhoegen, H.: Einstieg in XML. [Grundlagen, Praxis, Referenzen ; für Entwickler und XML-Einsteiger ; Formatierung, Transformation, Schnittstellen ; XML-Schema, DTD, XSLT 1.0/2.0, XPath 1.0/2.0, DOM, SAX, SOAP, Open XML]. Galileo Press, Bonn, 2007.

[W304] W3C: XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/xmlschema-0/, 12.03.2013.

[W304] W3C: XML Schema Part 1: Structures Second Edition. http://www.w3.org/TR/xmlschema-1/, 12.03.2013.

[W304] W3C: XML Schema Part 2: Datatypes Second Edition. http://www.w3.org/TR/xmlschema-2/, 07.03.2013.

[Wa08] Watson, B. et al.: Procedural urban modeling in practice. In Computer Graphics and Applications, IEEE, 2008, 28; S. 18–26.

[Wa02] Watt, A.: 3D-Computergrafik. Pearson Studium, München, 2002.

[We09] Weber, B. et al.: Interactive Geometric Simulation of 4D Cities. In Comput. Graph. Forum, 2009, 28; S. 481–492.

[Wi10] Wiley Online Library: Computer Graphics Forum, 2010.

[Wo03] Wonka, P. et al.: Instant architecture. In ACM Trans. Graph, 2003, 22; S. 669-677.

[Ya02] Yap, C. K. et al.: Different Manhattan project: automatic statistical model generation, 2002; S. 259–268.

[Yo04] Yong, L. et al.: Semantic modeling project: building vernacular house of southeast China: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry. ACM, New York, NY, USA, 2004; S. 412–418.

[ZDA04] Zerbst, S.; Düvel, O.; Anderson, E.: 3D-Spiele-Programmierung. Professionelle Entwicklung von 3D-Engines und -Spielen. Markt und Technik, München/Germany, 2004.

17 Abbildungsverzeichnis

18 Tabellenverzeichnis

19 Abkürzungsverzeichnis

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BNF	Backus-Naur-Form
B-Rep	Boundary Representation
B-Rep-Meshes	Boundary-Representation Meshes
BSP-Baum	Binary Space Partition-Baum
CAAD	Computer-Aided Architectural Design
CAD	Computer-Aided Design
CERN	Conseil Européen pour la Recherche Nucléaire
CNC	Computerized Numerical Control
CPU	Central Processing Unit
CSG	Constructive Solid Geometry
CSV	Comma-Separated Value
DOM	Document Object Model
DTD	Dokumenttypdefinition
EBNF	Erweiterte Backus-Naur-Form
GIS	Geographic Information Systems
GLU	OpenGL Utility Library
GML	Generative Modeling Language
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IBR	Image Based Modeling
IP	Internet Protocol
JDK	Java Development Kit
JSON	JavaScript Object Notation
LAV	List of active Vertices
LoD	Level of Detail
L-System	Lindenmayer-System
MAT	Mittelachsentransformation
OBB	Objektausgerichtete Bounding Box
OOP	Objektorientierte Programmierung
Pixel	Picture Element
RTF	Rich Text Format
SGML	Standard Generalized Markup Language
SLAV	Set of circular Lists of active Vertices
SMesh	Subdivision Mesh
SQL	Standard Query Language
STL	Standard Template Library
TCP	Transmission Control Protocol

TEI	Text Encoding Initiative
Tgraph	Typisierter Graph
URI	Uniform Ressource Identifier
URL	Uniform Ressource Locator
UTF	Unicode Transformation Format
Voxel	Volume Element
VRML	Virtual Reality Modeling Language
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	eXtensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
YAML	YAML Ain´t Markup Language

20 Verfügbare Onlineressourcen

Sämtliche zur Verfügung gestellten Ressourcen werden unter den Lizenzbedingungen der GNU General Public License (GPL), Version 2.0 veröffentlicht und zur Verfügung gestellt. Alle geltenden Bedingungen für die Vervielfältigung, den Betrieb und die Modifizierung der Inhalte sind einzusehen unter: GNU General Public License (GPL), Version 2.0

20.1 XML Schema-Dokumentation

Die vollständige Dokumentation sämtlicher vorhandenen Schema-Komponenten ist verfügbar unter:

City-Schema des Semantic Building Modeler

20.2 Verwendete XML- / XML Schema-Dokumente

Sämtliche für die Erzeugung der Beispiele verwendeten XML-Konfigurationen sind verfügbar unter:

XML-Konfigurationen

Weiterhin kann auf alle XML Schema-Dokumente zugegriffen werden über:

XML Schema-Dokumente

20.3 Quellcode aller Module des Semantic Building Modelers

Der Java-Quellcode aller implementierten Module des Semantic Building Modeler ist verfügbar unter:

Repository für alle vorhandenen Module des Semantic Building Modeler

20.4 Beispielrenderings entwickelter Stadt- und Gebäudeszenen

Die nachfolgenden Videos zeigen einfache Stadtszenen, die aus Gebäuden bestehen, die durch den Semantic Bilding Modeler erzeugt wurden.

Video 1: Semantic Building Modeler - Street View Rendering

Video 2: Semantic Building Modeler - City Rendering

Video 3: Semantic Building Modeler - Temple Rendering