Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Denisz Mihajlov

Ein Prozedurales Regelsystem zur Generierung und Positionierung von Räumen und Möbeln in Gebäudehüllen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Denisz Mihajlov

Ein Prozedurales Regelsystem zur Generierung und Positionierung von Räumen und Möbeln in Gebäudehüllen

Masterarbeit eingereicht im Rahmen der Masterprüfung

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

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Peer Stelldinger

Eingereicht am: 23. Juni 2023

Denisz Mihajlov

Thema der Arbeit
Ein Prozedurales Regelsystem zur Generierung und Positionierung von Räumen und Möbeln in Gebäudehüllen

Stichworte
Prozedurale Generierung, Räumeerstellung, Figur-Grammatik, CGA.

Kurzzusammenfassung
In der Computergrafik werden bei der Erstellung von Gebäuden oft nur die Fassaden berücksichtigt und generiert. Doch wie können auch Innenräume in diese Gebäuden integriert werden? Können diese Innenräume bestimmte Typen bekommen und können sie mit Möbeln ausgestattet werden? Diese Fragestellungen werden oft einzeln betreachtet. Diese Masterarbeit konzentriert sich auf die Entwicklung von Konzepten zur Generierung und 3D-Modellierung von Innenräumen in Gebäudehüllen, zur Typisierung dieser Räume und zur Platzierung von Möbeln darin. Dabei werden all diese Probleme als ein einziges Problem in dieser Arbeit untersucht und mithilfe von prozeduralen Generierungsmethoden gelöst. Insgesamt werden in der Arbeit drei Konzepte vorgestellt, die die Raumpartitionierung, Raumtypisierung und Möbelplatzierung ermöglichen. Jedes Konzept verwendet unterschiedliche Methoden zur Lösung der entsprechenden Probleme. Diese Konzepte bauen aufeinander auf, um alle drei Probleme gemeinsam zu lösen. Die entwickelten Konzepte wurden erfolgreich durch Prototypen bestätigt. Diese Prototypen wurden in einer Figur-Grammatik in Form der Operationen hinzugefügt, um aus einer Gebäudehülle ein Gebäude mit Etagen und Räumen zu erzeugen.
Denisz Mihajlov

Title of the paper
A procedural rule system for generating and positioning rooms and furniture within building envelopes.

Keywords
Procedural generation, room creation, shape grammar, CGA.

Abstract
In computer graphics, when creating buildings, often only the facades are considered and generated. But how can interior spaces be integrated into these buildings? Can these interior spaces have specific types and be furnished with furniture? These questions are often examined

separately. This master's thesis focuses on the development of concepts for generating and 3D modeling interior spaces within building envelopes, categorizing these rooms, and placing furniture in them. All these problems are investigated as a single problem in this thesis and solved using procedural generation methods. In total, three concepts are presented in this thesis, which enable room partitioning, room categorization, and furniture placement. Each concept utilizes different methods to solve the respective problems. These concepts build upon each other to collectively solve all three problems. The developed concepts have been successfully validated through prototypes. These prototypes were added in the form of operations to a shape grammar to generate a building with floors and rooms from a building envelope.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Die prozedurale Generierung ist ein Prozess, mit dem mit kleinem Aufwand und wenigen Eingangsinformationen unterschiedliche Programminhalte, wie z.B. Texturen, 3D-Objekte oder Töne generiert werden können. Damit können auch die ganzen Welten oder Gebäude modelliert werden. Die prozedurale Generierung kann in unterschiedlichen Branchen benutzt werden, wie zum Beispiel in Spielen, 3D-Modellierungssoftware oder Musik. Die Inhalte werden nicht als statische Objekte manuell erstellt, sondern ausgehend von definierten Vorgaben und Regeln vom Computer erzeugt. Verglichen mit der eigentlichen Programmierung besteht der größte Aufwand bei der Entwicklung eines Spiels in der Erstellung von Grafiken, Karten, Charakteren und anderen Modellen. Bei großen Spielproduktionen und der 3D-Modellierungssoftware können die Entwickler durch die Verwendung prozeduraler Verfahren zur Automatisierung diesen Prozessen unterstützt werden.

Ein weiteres Anwendungsfeld der prozeduralen Generierung ist die Erstellung von Gebäudeinnenräumen. Ein Innenraum ist ein Raum, der vor Witterungseinflüssen geschützt ist und größtenteils von Wänden und Dachflächen umgeben ist. Der Raum kann prozedural nicht nur erstellt, sondern auch typisiert werden, zusätzlich können anhand der Raumtypen auch Möbel in der Räume platziert werden.

Das Ziel dieser Masterarbeit ist die Entwicklung von Konzepten, die die prozedurale Generierung von Räumen, die Typisierung der Räume und die Platzierung von Möbeln innerhalb der Räume realisieren. Die Algorithmen, die diesen Konzepten zugrunde liegen, werden zu Operationen in einer Figur-Grammatik zusammengefasst, damit sie auf die Gebäudehüllen angewendet werden können. Im Rahmen dieser Arbeit sollen alle drei entwickelten Konzepte in einer Software benutzt und prototypisch umgesetzt werden, um die Probleme von der Raumpartitionierung, Raumtypisierung und Möbelplatzierung gemeinsam zu lösen. Die Konzepte werden aufeinander aufgebaut, um ihre Kompatibilität innerhalb einer Figur-Grammatik zu gewährleisten.

Die Fragestellung, die in dieser Arbeit untersucht wird, lautet: Welche Konzepte können entwickelt werden und welche Methoden werden entwickelt, um die Prozesse der Raumerstellung, Typisierung und Möbelplatzierung in einem Gebäude durchführen zu können? Können

diese Konzepte prototypisch umgesetzt werden und in einer Software zusammenarbeiten, um ein 3D-Modell des Innenraums mit einem Typ und den Möbeln zu generieren und darzustellen? Wie können diese Prozesse auch von Benutzern gesteuert werden und inwieweit ist das möglich?

Der erste Teil bezieht sich auf die Partitionierung eines Gebäudes in Etagen und Räume. Dafür wird das Gebäude erstmal in Etagen unterteilt, danach werden mithilfe des Partitionierungsalgorithmus die Etagen partitioniert und die Räume erstellt. Die Erstellung passiert in zwei Phasen, die erste Phase umfasst die Erstellung von Räumen basierend auf einer Datenstruktur für planare Graphen mit dem Namen "Halbkanten-Datenstruktur". Der Algorithmus, der für die Partitionierung benutzt wird, ist ein BSP-Algorithmus. Die zweite Phase ist die Umwandlung von erstellten Räumen in das 3D-Mesh.

Im zweiten Teil der Arbeit werden bereits erstellte Räume im Gebäude weiterentwickelt. Wenn diese Räume erstellt sind, haben sie am Anfang keine Bedeutung für das generierte Modell. Sie können unterschiedliche Funktionalitäten und Typen haben. Wenn den Räumen unterschiedliche Typen zugewiesen werden, kann der Nutzer der Software besser erkennen, wofür jeder Raum gedacht ist und eine weitere Entwicklung dieser Räume ist einfacher. Eine solche Typisierung kann auch prozedural erreicht und vom Nutzer gesteuert werden. Die Typisierung kann nicht nur bei der allgemeinen Wahrnehmung der erstellten Welt (des Gebäudes) helfen, sondern auch beim Verbindungsaufbau zwischen den Räumen.

Dieses Konzept bezieht sich auf die Algorithmen, die die prozedurale Typisierung der Räume ermöglichen und anhand dieser Typisierung die Verbindungen zwischen den Räumen darstellen können. Es wird gezeigt, wie die Typisierung von bereits existierenden Räumen durchgeführt werden kann. Der Prozess der Raumerstellung und die Typisierung der Räume sind komplett voneinander getrennt. Die Idee besteht darin, die Typisierung als ein regelbasiertes, iteratives Verfahren darzustellen, das mit den Regeln für verschiedene Raumtypen aus einer Konfigurationsdatei arbeitet und dann versucht, alle Regeln nacheinander anzuwenden. Die Regeln zeigen, welche Kriterien ein Raum erfüllen muss, um als bestimmter Typ klassifiziert zu werden. Sobald die Typisierung abgeschlossen ist, werden die erstellten Regeln für den Verbindungsaufbau zwischen den Räumen benötigt. Die typisierten und verbundenen Räume werden anschließend in 3D modelliert, damit die Software das erstellte 3D-Modell anzeigen kann.

Das Konzept für das letzte Teil der Arbeit umfasst die Platzierung von den Möbeln im Raum. Das ist ebenfalls ein regelbasierter iterativer Prozess, der eine Konfigurationsdatei mit den Regeln erfordert. Im Gegensatz zu den Regeln aus dem zweiten Teil, beschreiben diese Regeln nicht den Raumtyp, sondern die Position eines bestimmten Möbelstücks im Raum.

Diese Regeln erhalten Informationen über den Typ des Raums, in dem das Möbelstück platziert werden kann, sowie die Positionseigenschaften des Möbelstücks im Raum und die möglichen Beziehungen zwischen unterschiedlichen Möbelstücken. Damit die Möbelstücke nach der Platzierung benutzt werden können und genügend Abstand zu anderen Möbelstücken haben, werden auch die Informationen zu der Abstandsflächen für jedes Möbelstück in den Regeln hinzugefügt.

Am Anfang sollen die benötigten Möbelstücke bereits zufällige Positionen im Raum erhalten. Diese Positionen werden anschließend mithilfe des Separation Steering Algorithmus optimiert. Erst danach werden alle Regeln angewendet und die Positionen der Möbelstücke entsprechend der Regeln verändert.

Die Arbeit besteht aus sieben Kapiteln. Das nächste Kapitel zeigt verwendete Arbeiten, die dabei geholfen haben, die Konzepte und die Implementierung für alle Teile der Arbeit zu erstellen. Zusätzlich werden hier die Grundlagen zum Basisprojekt sowie die verwendeten Algorithmen wie der BSP-Algorithmus, der Separation Steering Algorithmus und die Halbkanten-Datenstruktur dargestellt. Kapitel 3 beschreibt die Konzepte und Anforderungen aller drei Teile der Masterarbeit. Kapitel 4 beschreibt die verwendete Software und die Bibliotheken, die benutzt wurden, um die benötigten Algorithmen zu erstellen. Zusätzlich wird hier auch die Software beschrieben, die für die Erstellung von 3D-Modellen von Möbelstücken verwendet wurde. Kapitel 5 stellt die Implementierungen und Algorithmusbeschreibungen aller Prozesse dar, die innerhalb der Masterarbeit behandelt wurden. Kapitel 6 zeigt die durchgeführten Tests und die Ergebnisse der Implementierung. Hier wird auch beschrieben, ob alle im Kapitel 3 dargestellten Anforderungen erfüllt wurden. Das letzte Kapitel ist die Zusammenfassung, in der auf die Ergebnisse und die entwickelte Software eingegangen wird. Es wird auch beschrieben, ob die Ziele, die in der Arbeit definiert wurden, erreicht wurden.

2 Stand der Technik

In diesem Kapitel werden die Artikel und Veröffentlichungen, die bei der Entwicklung von Konzepten und der Implementierung der Software geholfen haben, vorgestellt. Das Kapitel ist in drei Teile gegliedert: Generierung von Räumen, Typisierung von Räumen und Möbelplatzierung. In jedem Teil werden Veröffentlichungen zu dem entsprechenden Thema aufgeführt und kurz erläutert.

2.0.1 Generierung der Räume

Es gibt viele unterschiedliche Methoden, um Räume innerhalb eines Gebäudes zu generieren. Im Folgenden werden einige dieser Methoden vorgestellt.

Logan M. Bond hat für die Generierung von Räumen den Separation Steering Algorithmus von Craig Reynolds genutzt [1]. Dabei wurden Vierecke als Objekte verwendet, um die Räume zu repräsentieren. Zunächst wurden alle Räume an derselben Position erstellt. Mithilfe der Trennungsphase des Algorithmus wurden die Räume so lange auseinandergezogen, bis sie sich nicht mehr überschneiden und ihre Kanten sich nicht mehr berührten. Die Räume wurden danach nicht mehr weiterbewegt, weshalb die Phasen Ausrichtung und Zusammenhalt nicht mehr benötigt wurden. Dieser Ansatz ermöglichte die Erstellung von Räumen mit beliebiger Fläche und Ausrichtung. Dieser Ansatz wurde in den Konzepten nicht benutzt, da er vordefinierte viereckige Räume benötigt. In dem entwickelten Konzept von der Raumpartitionierung sind vordefinierte Räume nicht vorausgesetzt. Bei dem Konzept von der Möbelplatzierung wurde aber der Separation Steering Algorithmus verwendet, um die Position von Möbelstücken im Raum zu optimieren.

Random Room Placement ist eine weitere Methode, die für die Raumgenerierung verwendet werden kann [2]. Dabei handelt es sich um einen Brute-Force-Algorithmus zur Erzeugung von Räumen. Zunächst wird eine zufällige Breite und Höhe für den Raum ausgewählt und dieser wird erzeugt. Anschließend wird ein zufälliger Punkt auf der entsprechenden Ebene ausgewählt, der die obere linke Ecke des Raums darstellt. Bevor der Raum an dieser Stelle platziert wird, wird geprüft, ob er mit anderen Räumen kollidiert. Dieser Algorithmus kann auch für die Erstellung von Räumen und Dungeons verwendet werden. Ein Nachteil davon

ist der ungenutzte Platz zwischen den Räumen, der durch die zufällige Platzierung entstehen kann. Eine Optimierungsmöglichkeit besteht darin, dass nachdem alle Räume platziert sind, diese gleichmäßig vergrößert werden, bis kein freier Platz mehr auf der Fläche vorhanden ist. Im Artikel [2] werden auch weitere Methoden vorgestellt, die für die Erstellung von Dungeons verwendet werden können und teilweise auch für die Erstellung von Räumen geeignet sind. Um diese Methode zu nutzen, sollen auch vordefinierte Räume erstellt werden und im Gebäude platziert werden, das entwickelte Konzept versucht aber die Räume aus einer gegebenen Gebäudefläche zu produzieren.

Der Growth-Algorithmus ist ein weiterer Algorithmus, der für die Erstellung von Räumen oder Dungeons verwendet werden kann. Eine Variante dieses Algorithmus kann für die Raumerstellung mit gegebener Fläche genutzt werden. Dieser Ansatz erfolgt in drei Schritten: Zerlegung der Fläche in kleine Zellen, Platzierung der anfänglichen Raumkerne, Vergrößerung der Räume. Der Raum kann in gleich große Kästchen zerlegt werden. Die Zerlegung mit diesem Ansatz kann einfach durch die Division der Höhe und Breite des gesamten Raums durch die Höhe und Breite des Kästchens erfolgen. Die Platzierung der Räume ist ein sehr wichtiger Teil dieses Growth-Algorithmus. Die Räume können in Abhängigkeit von anderen Räumen und dem Abstand zu Außenwänden platziert werden. Die Räume werden auf die Kästchen platziert, von denen sie am besten wachsen können. Am Ende können die Räume iterativ vergrößert werden. Die Räume können zuerst als Vierecke vergrößert werden, bis die gewünschte Größe erreicht ist. Danach können die Räume so vergrößert werden, dass sie anschließend L-förmig sind. Dieser Ansatz wurde von Lopes in seinem Artikel [3] vorgestellt. Lopes ermöglicht die Vergrößerung von nur einem Raum bei jeder Iteration. Die Auswahl der Räume hängt von der gewünschten Größe ab. Die Räume, die am weitesten von der gewünschten Größe entfernt sind, werden zuerst vergrößert.

Es gibt auch Methoden, mit denen Räume durch Subdivision erstellt werden können. Hahn et al. haben ein Verfahren zur Generierung von Bürogebäuden vorgestellt, bei dem der Raum im Gebäude iterativ auf einheitliche Weise halbiert wird, bis die Höhe jeder Scheibe innerhalb einer gegebenen Schwelle liegt [4]. Der gleiche Artikel verwendet auch eine Unterteilung zur Erzeugung der Grundrisse auf jeder einzelnen Etage des Bürogebäudes. Hierbei werden auf jeder Etage gerade oder geschwungene Flure platziert und der Raum zwischen den Fluren und den Wänden in kleinere, auf Achsen ausgerichtete Kästen unterteilt, die Bürocluster darstellen. Obwohl beide Methoden auf Unterteilung basieren, unterscheiden sie sich sowohl in der Implementierung als auch im Ergebnis, da die eine Büroetagen und die andere den Innengrundriss für eine einzelne Bürogebäudeebene generiert. Diese Methode mit dem Halbieren von Gebäudeflächen ist ähnlich zu dem Konzept der Raumpartitionierung, das in dieser

Masterarbeit erstellt wurde, in dem Konzept wird aber kein Flur verwendet, um die Räume zu partitionieren.

Merrell et al. [5] stellt eine Methode vor, die mithilfe von maschinellem Lernen und stochastischer Optimierung realistische Wohnungsgrundrisse generiert [6]. Die Autoren schlagen vor, ein Bayes'sches Netzwerk [7] zu trainieren, indem reale Architekturdaten verwendet werden, um relevante Aspekte des architektonischen Entwurfs zu lernen. Mit einem Satz flexibler Benutzeranforderungen wäre das Bayes'sche Netzwerk dann in der Lage, ein Architekturprogramm zu erzeugen. Unter Verwendung dieses Architekturprogramms kann ein Grundriss durch die Anwendung von stochastischer Optimierung erzeugt werden. Das ist eine interessante Lösung zur Raumpartitionierung mithilfe des maschinellen Lernens. Das in dieser Arbeit entwickelte Konzept nutzt kein maschinelles Lernen zur Partitionierung, daher sind keine komplizierten Konstrukte wie Bayes'sches Netzwerk erforderlich, um das Ziel der Arbeit zu erreichen.

Koenig und Knecht [8] stellen eine Methode zur Generierung von Grundrissen mithilfe von dichtem Packen vor. Die Idee hinter ihrer Lösung besteht darin, einen Satz von Rechtecken zu haben, die Räume repräsentieren, die innerhalb einer vorgegebenen Fassade platziert werden sollen. Um die Ergebnisse ihrer Lösung zu verbessern, weisen Koenig und Knecht den Rechtecken virtuelle Federn zu. Diese Federn funktionieren ähnlich wie die Massen-Feder-Dämpfer-Systeme, die von Arvin und House [9] vorgestellt wurden. In dem Artikel von Koenig und Knecht werden die Federn als Darstellung des Grades verwendet, bis zu dem ein Raum verformbar ist und sie dienen auch dazu, einen Druckausgleich zwischen mehreren Räumen zu schaffen, so dass die Gesamtverformung gleichmäßig auf alle Räume verteilt wird.

2.0.2 Raumtypisierung

In diesem Abschnitt werden einige Veröffentlichungen erwähnt, die sich mit der Typisierung von Räumen befassen. In diesen Arbeiten werden verschiedene Methoden verwendet, um eine erfolgreiche Typisierung innerhalb eines Gebäudes zu ermöglichen. Hierfür können beispielsweise Graphen, Konfigurationsdateien oder Blueprints genutzt werden.

Die Arbeit [10] beschäftigt sich mit der Generierung von Städten, Gebäuden und Räumen. Dabei wird jedem Raum im generierten Gebäude ein Typ zugewiesen. Hierfür werden ein Blueprint und Raum-Metadaten verwendet. Der Blueprint enthält die Angaben zur Raumgröße und die Raum-Metadaten enthalten eine Eigenschaft, die angibt, ob ein Raum durchgängig ist oder nicht. Wenn ein Raum existiert, der zur Größe des Blueprints passt, wird ihm ein Typ aus dem Blueprint zugewiesen. In dem Konzept aus dieser Masterarbeit wird eine ähnliche Idee, wie bei den Blueprints verfolgt, die aber mehr Konfigurationsparametern hat und nicht nur die Raumgröße.

In der Veröffentlichung [11] werden die Räume anhand ihrer Größe und den drei Kategorien (Dienstraum, privates Zimmer, öffentliches Zimmer) typisiert. Die Flächen werden mit diesen drei Kategorien markiert und die Zimmer werden hierarchisch verteilt (in der Veröffentlichung: erst das Wohnzimmer und dann alle anderen Räume). Die Typen der Räume werden nach einer vordefinierten Priorität ermittelt. Manchmal ist die Typisierung auch von der Funktionalität der anderen Räume abhängig, zum Beispiel soll das Badezimmer mit den öffentlichen Räumen verbunden werden, und ein extra Badezimmer soll mit einem Schlafzimmer verbunden werden. Für die Typisierung wird ein regelbasierter Algorithmus verwendet, der die Räume in einem hierarchischen Graph hinzufügt und dadurch die Typen setzt. Zusätzlich arbeitet der Algorithmus mit einer vordefinierten Konfiguration, die angibt, wie viele Räume von welchem Typ im Gebäude erstellt werden sollen. Die Anzahl der Räume eines bestimmten Typs hängt dabei von der Gesamtanzahl der Räume ab.

In der Arbeit [12] wird die grundlegende Struktur eines Hauses durch einen Graphen repräsentiert, bei dem jeder Knoten einem Raum entspricht und jede Kante einer Verbindung zwischen den Räumen. Der Graph selbst wird generiert, indem die Haustür als Wurzelknoten verwendet wird, gefolgt von der Erstellung aller öffentlichen Räume und schließlich der Erstellung aller privaten Räume. Die spezifischen Raumtypen werden zunächst nicht zugewiesen, sondern unmittelbar nach der Verteilung der öffentlichen Räume entsprechend einer Reihe vom Benutzer bereitgestellter Attributen definiert. Die Verwendung des Graphen geht jedoch über die Raumtypisierung hinaus. Der Graph wird verwendet, um Räume richtig zu platzieren und zu modellieren, nachdem die Typisierung abgeschlossen ist. In dem in dieser Arbeit entwickelten Konzept wird der Graph auch benutzt, allerdings hat der Graph im Konzept nur eine zusätzliche Funktion und wird nicht für die Typisierung verwendet.

Die Veröffentlichung [13] beschäftigt sich mit einem Ansatz zur Generierung von Hausgrundrissen mit semantischen Informationen. Der Grundgedanke dieses Modells basiert auf dem Squarified-Treemaps-Algorithmus [14], welcher ursprünglich zur Erstellung grafischer Darstellungen auf Basis hierarchischer Informationen wie Verzeichnisstrukturen oder Organisationsstrukturen genutzt wurde. Dieser Ansatz ermöglicht die Erstellung interner Hausstrukturen unter Berücksichtigung von Informationen über deren Eigenschaften und Funktionalitäten. Zur Umsetzung des Ansatzes wird das zu generierende Haus in Bereiche unterteilt, welche den Kategorien Sozial, Service und Private zugeordnet werden. Dabei werden die Bereiche mithilfe des Squarified-Treemaps-Algorithmus aufgeteilt. Im Anschluss wird der Algorithmus auf jeden Bereich erneut angewendet, um die Räume zu generieren. Während der Generierung der Räume werden diese bereits typisiert. Die Typisierung kann dabei vom Benutzer gesteuert werden und hängt vom Bereich ab, in welchem die Räume erstellt werden. Im Sozialbereich können

beispielsweise Wohnzimmer, Esszimmer und Toiletten platziert werden. Im Servicebereich hingegen können die Küche, die Speisekammer und die Waschküche untergebracht werden. Der private Bereich kann das Schlafzimmer, das Hauptschlafzimmer, ein Badezimmer sowie einen optionalen Nebenraum umfassen, der auf unterschiedliche Weise genutzt werden kann, beispielsweise als Bibliothek. Diese Liste ist nicht festgelegt und kann vom Benutzer angepasst werden. Da die Raumtypisierung hier während der Raumerstellung geschieht, ist es nicht für dieses Projekt geeignet, da diese zwei Prozesse getrennt werden sollen.

2.0.3 Möbelplatzierung

In der bisherigen Forschung wurden verschiedene Methoden vorgestellt, um virtuelle Szenen in Echtzeit mit Möbeln zu füllen. Es gibt zum Beispiel Artikel, die optimierungsbasierte Methoden und datengesteuerte Ansätze verwenden, um Möbel in Räumen zu platzieren. Eine Methode zur automatischen Anordnung von Möbeln in großflächigen virtuellen Umgebungen wurde von Germer und Schwarz [15] vorgeschlagen. Ihre Methode verwendet eine agentenbasierte Lösung, um Objektlayouts automatisch zu generieren.

Eine weitere Methode zur Generierung von Möbellayouts wurde von Kari Anne Høier Kjølaas präsentiert [16]. Hier wird ein gegebener Raum als eine verschachtelte Hierarchie rechteckiger Vorlagen dargestellt, die durch acht vorgegebene Mutationsfunktionen ausgetauscht werden können. Leere Kästen werden vor Türen und Fenstern platziert, um den freien Raum darzustellen. Diese Methode ist jedoch auf rechteckige Räume beschränkt und jeder Vorlagentyp sowie der Satz entsprechender Parameter müssen sorgfältig entworfen werden.

Die Entwicklung eines interaktiven Möbellayout-Systems wird in [17] beschrieben, das den Benutzern dabei hilft, neue Möbelanordnungen zu entwerfen. Ausgehend vom ursprünglichen Layout und Benutzerbeschränkungen generiert das System neue Layout-Vorschläge mithilfe des Monte-Carlo-Samplers der Markov-Kette. Innenarchitekturregeln werden verwendet, um eine Kostenfunktion zu bilden, die vom Sampler untersucht wird. Diese Methode erfordert die Unterstützung des Benutzers, um das endgültige Innendesign zu generieren.

Es wurden verschiedene optimierungsbasierte Verfahren vorgeschlagen, um realistische Möbelanordnungen zu erzeugen. Diese Verfahren optimieren das Möbellayout in Bezug auf eine Kostenfunktion, die ergonomische, ästhetische und funktionale Aspekte berücksichtigt. In der Vergangenheit wurden mehrere Methoden vorgeschlagen, um diese Kostenfunktion mithilfe evolutionärer Berechnungsverfahren zu optimieren.

Beispielbasierte Methoden nutzen Informationen aus vorhandenen Raumlayouts, um neue plausible Möbelanordnungen zu generieren. Dazu benötigen sie eine Reihe von vom Benutzer erstellten Layoutbeispielen [18; 19]. Einige prozedurale Methoden basieren auf Constraints wie

Platzierungs-, Nähe-, Unterstützungs- und Kontaktbeschränkungen und erfordern aufgrund der Komplexität dieser Beschränkungen eine Offline-Berechnung [20; 21]. Eine andere Lösung für die automatische Anordnung von Möbeln besteht darin, sie aus der Perspektive von Empfehlungssystemen anzugehen. Diese Lösung generiert zunächst prozedural eine Sammlung von Layouts und bewertet sie dann mit einer Scoring-Funktion, um den rentabelsten Kandidaten auszuwählen [22].

Es gibt noch einen weiteren Artikel [23], der sich mit Möbelanordnungen in Gebäuden beschäftigt. Dieser Artikel schlägt eine automatische Methode zur Möbelanordnung vor, die die gierige Kostenminimierung nutzt. Die Methode optimiert die Möbelauswahl und -anordnung für ästhetische und funktionale Regeln und integriert auch prozedurale Methoden für die lokale Anordnung kleiner Objekte. Sie erreicht vergleichbare Ergebnisse wie eine kürzlich vorgestellte automatische Innenraumdesign-Methode in Bezug auf Benutzerpräferenzen, ist jedoch um eine Größenordnung schneller. Die vorgeschlagene Methode eignet sich zur Generierung großer Innenraum-Virtual-Environments während der Laufzeit.

Es gibt einen weiteren Artikel, der die Möbelanordnung mittels Augmented-Reality-Darstellung präsentiert [24]. In diesem Ansatz werden prozedurale Regeln verwendet, um das Design eines Innenraums automatisch zu generieren. Die Regeln sind in einem hierarchischen Baum organisiert, wobei die oberste Ebene die globalen Designentscheidungen enthält, wie zum Beispiel die Anordnung der Möbel im Raum. Die unteren Ebenen enthalten detailliertere Regeln, wie zum Beispiel die Wahl von Farben und Texturen für jeden Möbelgegenstand. Das System nutzt Augmented Reality, um dem Benutzer eine immersive Erfahrung zu bieten. Im Konzept, das in dieser Masterarbeit erstellt wurde, werden ebenfalls Regeln verwendet, um die Möbel im Raum zu positionieren. Allerdings beschreiben diese Regeln keine Designentscheidungen wie Farben und Texturen, sondern nur die Positionen der Möbel im Raum. Die Regeln in diesem Konzept werden einfach iterativ in der Reihenfolge bearbeitet, in der sie eingetroffen sind. Es wird hier kein hierarchischer Baum erzeugt, um die Regeln zu bearbeiten.

2.1 Grundlagen

In diesem Teil werden die Grundlagen der verwendeten Methoden präsentiert. Zusätzlich wird am Ende des Abschnitts das Basisprojekt vorgestellt, das im Rahmen dieser Masterarbeit erweitert wurde.

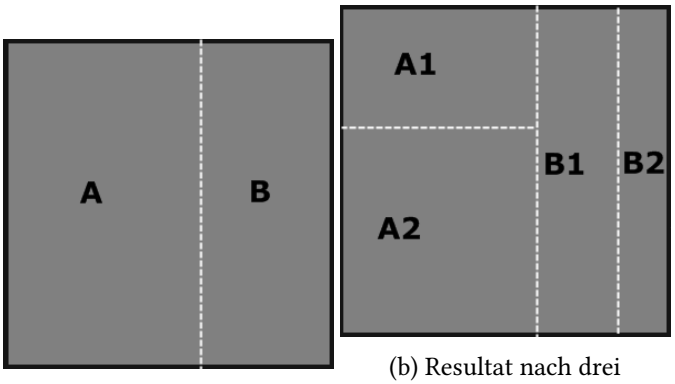
2.1.1 Figur-Grammatiken

Figur-Grammatiken (im Englischen Shape grammars) sind eine Art von Produktionssystem in der Informatik zur Generierung von geometrischen Figuren. Üblicherweise werden Figur-Grammatiken zur Erzeugung von zwei- bzw. dreidimensionalen Figuren eingesetzt, heutzutage vor allem in den Bereichen der Architektur und Computergrafik. Die Grundlage für die Figur-Grammatiken wurde in einem Seminar-Artikel von George Stiny und James Gips im Jahr 1971 geschaffen [25].

Eine Figur-Grammatik besteht aus Regeln sowie einer Generierungsengine, die die Regeln auswählt, verarbeitet. Eine Regel definiert, wie eine existierende Figur im geometrischen Raum transformiert werden kann. Eine Figur-Grammatik generiert eine Figur durch rekursives Anwenden der Regeln, beginnend mit der Startfigur. Das Ergebnis der angewendeten Regel auf eine vorhandene Figur ist immer eine neue Figur. Eine Figur-Grammatik besteht aus mindestens drei Produktionsregeln: einer Startregel, mindestens einer Transformationsregel und einer Terminierungsregel. Die Startregel ist erforderlich, um den Generierungsprozess zu starten, während die Terminierungsregel notwendig ist, um die Generierung abzuschließen.

Ursprünglich wurden Figur-Grammatiken für Gemälde und Skulpturen angewendet. Sie eignen sich besonders für kleine, klar definierte Zielsetzungen, wie zum Beispiel das Struktur- und Layout-Design von Innenräumen oder Fassaden von Gebäuden. Figur-Grammatiken bestehen oft aus einer großen Anzahl von Regeln. Beispielsweise besteht die Figur-Grammatik von William Mitchell zur Generierung einer Villa im Stil des italienischen Architekten Andrea Palladio aus 69 Regeln, die in acht Durchführungsschritten angewendet werden [26]. Ähnlich zur Anwendung in der Architektur haben Figur-Grammatiken auch in der Computergrafik in den letzten Jahrzehnten an Bedeutung gewonnen. Sie werden vor allem bei der prozeduralen Modellierung von Gebäuden [27] oder Städten [28; 29] eingesetzt. Figur-Grammatiken bilden die Grundlage für zahlreiche entwickelte Systeme, die mithilfe von Produktionsregeln eine Variation von unterschiedlichen 3D-Modellen generieren können. Sowohl realistisch aussehende Straßenpläne als auch Fassaden oder Innenräume von Gebäuden können mithilfe von Figur-Grammatiken prozedural erstellt werden.

2.1.2 Binary Space Partition



(a) Resultat nach einer Teilung

(b) Resultat nach drei Teilungen

Abbildung 2.1: Partitionierung nach der Teilung der Fläche

Der BSP-Baum (Binary Space Partition) ist eine Datenstruktur, die erstmals von Henry Fuchs in den frühen 1980er Jahren entwickelt wurde [30]. BSP ist ein rekursiver Prozess, bei dem eine Struktur in zwei Teile aufgeteilt wird und die Unterabschnitte in einer Binärbaumstruktur gespeichert werden. BSP wurde bei einer Vielzahl von Problemen im Zusammenhang mit Computergrafik eingesetzt. Die BSP-Bäume können für die Erstellung von Dungeons oder Räumen verwendet werden. Damit können auch 3D Objekte und Figuren erzeugt oder verbessert werden. Auf Basis dieses Verfahrens wurde auch ein neuronales Netz namens BSP-Net entwickelt, das 3D-Objekte durch konvexe Zerlegung repräsentieren kann. Das Netz wird darauf trainiert, eine Form unter Verwendung eines Satzes von konvexen Formen zu rekonstruieren, die von einem BSP-Baum erhalten werden, der auf einem Satz von Ebenen aufgebaut ist. Die erzeugten 3D-Objekte sind kompakt und eignen sich gut zur Darstellung von scharfen Geometrien [31].

Die Methode und der Fortgang der BSP-Teilung werden benötigt, um bei jeder Raumgenerierung unterschiedliche Partitionierungen der Ebene zu erhalten. Um dies zu erreichen, müssen ausgewählte Aspekte der BSP-Teilung randomisiert werden. Zunächst muss bei der Aufteilung ausgewählt werden, ob es sich um eine horizontale oder eine vertikale Aufteilung handelt. Die Position der Teilung kann auch randomisiert werden und einen beliebigen Abstand von links oder von oben haben, abhängig von der Orientierung der Teilung.

Mit der BSP-Technik ist es jedoch nicht möglich, eine gewünschte Anzahl von zu generierenden Räumen festzulegen, da nur die Anzahl der auftretenden BSP-Teilungen kontrolliert werden kann. Dies kann auch ein Nachteil sein, da der Benutzer nicht die Kontrolle über die Anzahl der Räume hat.

Diese Methode wird für das erste entwickelte Konzept in der Masterarbeit eingesetzt, um die Etagen in Räume zu zerlegen. Um die erstellten Räume erfolgreich in 3D darstellen zu können, wird eine Datenstruktur benötigt, die die Informationen zu jeder Teilungslinie (Wand) erfolgreich, effizient und bequem speichern kann. Diese Datenstruktur wird im weiteren Verlauf der Arbeit dargestellt.

2.1.3 Halbkanten-Datenstruktur

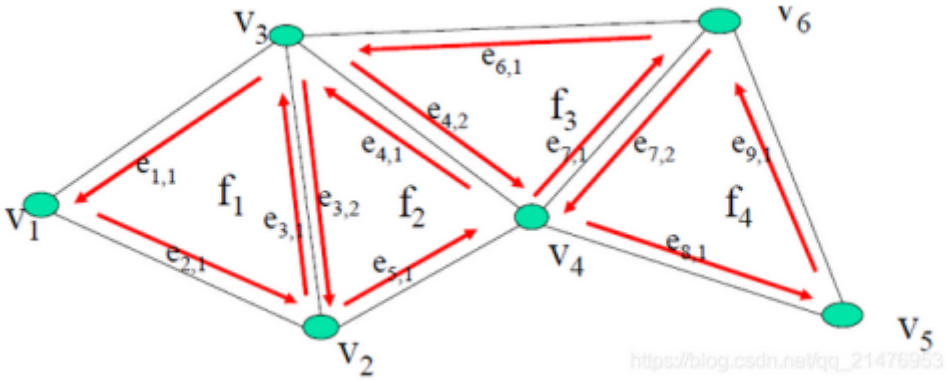


Abbildung 2.2: Halbkanten-Datenstruktur

Eine Half-Edge-Datenstruktur ist eine Datenstruktur für planare Graphen. Sie besteht aus Knoten, Halbkanten und Flächen, die jeweils in einer Liste abgelegt sind [32]. Mit dieser Struktur ist es leicht durch das Polygonnetz zu navigieren und Nachbarschaftsbeziehungen zu suchen, da die Vorgängerkante, die Nachfolgerkante und die angrenzenden Flächen eindeutig bestimmt sind. Sie eignet sich daher besonders für unstrukturierte, räumliche Datensätze. Diese Datenstruktur kommt oft zum Einsatz in der Computergrafik und in den Polygonnetzen im Allgemeinen.

In der Abbildung 2.2 ist eine solche Datenstruktur graphisch dargestellt. Die Variablen v stehen für Punkte, die mit Kanten e verbunden sind. Es ist zu sehen, dass es zwischen zwei Punkten immer zwei Kanten gibt, die in entgegengesetzte Richtungen orientiert sind: Diese Kanten sind die Halbkanten. Es gibt auch Facetten, die zu diesen Kanten zugewiesen sein können. In der Abbildung sind das die Variablen f. Für diese Arbeit sind die Facetten jedoch nicht relevant und werden daher nicht benutzt oder implementiert. Im Rahmen dieser Arbeit hilft diese Datenstruktur, die korrekte Raumverteilung zu modellieren und anschließend in 3D zu präsentieren.

Die Punkte in der Datenstruktur sind die einfachen Punkte im 3D-Koordinatensystem mit X-, Y- und Z-Werten. Diese Punkte haben auch einen Verweis auf die Halbkante, die sie als Startpunkt haben. Die Halbkante besteht aus zwei Punkten: einem Startpunkt und einem Endpunkt. Diese Punkte zeigen auch die Richtung an, in welche die Kante ausgerichtet ist. Zusätzlich hat jede Halbkante einen Verweis auf den Nachfolger und gegebenenfalls eine gegenübergestellte Halbkante. Falls eine Kante keine gegenübergestellte hat, so ist diese Kante eine Grenze des Objekts.

2.1.4 Separation Steering Algorithmus

Der Separation Steering Algorithmus kann eine Technik für die Generierung von Räumen sein. Das einfachste Beispiel für diesen Algorithmus sind Boids. Boids sind die Bezeichnung für bestimmte interagierende Objekte in einer Computersimulation. Der Name „Boid“ entspricht einer verkürzten Version von „bird-oid object“, was sich auf ein vogelähnliches Objekt bezieht [33]. Die Bezeichnung stammt von einem bahnbrechenden Künstliches-Leben-Programm, das 1986 von Craig Reynolds entwickelt wurde, um das Schwarmverhalten von Vögeln zu simulieren [34]. Der Algorithmus von Reynolds hat die Bewegungen von Vögeln erfolgreich modelliert, ohne die Bewegung für jeden Vogel separat zu programmieren. Der Separation Steering Algorithmus besteht aus 3 Teilen: Trennung, Ausrichtung und Zusammenhalt.

Um zu bestimmen, wie die Objekte in der Simulation bewegt werden, wird der Nachbarkreis von jedem Objekt untersucht. Jedes Objekt, das sich in diesem Radius befindet, wird zur Liste der Nachbarn für das entsprechende Objekt hinzugefügt. Die Trennungsphase wird auf diese Liste angewendet. Das Ziel der Trennung ist es, die Objekte auf minimale Distanz voneinander zu bringen, unter der Voraussetzung, dass sich die Objekte nicht überlappen dürfen. In dieser Phase wird die Entfernung von jedem Nachbarn zum betrachteten Objekt berücksichtigt. Der Richtungsvektor jedes Nachbarn wird dann normalisiert und durch die Entfernung zum betrachteten Objekt geteilt.

Die Ausrichtungsphase versucht, nach der Trennung die Richtung jedes Boids in Übereinstimmung mit seinen Nachbarn beizubehalten. Es wird erneut durch die Liste der Nachbarn für jedes Objekt iteriert und der Mittelwert von allen Richtungsvektoren wird berechnet. Dieser Mittelwert wird vom Richtungsvektor des Objekts subtrahiert und erneut dem Objekt zugewiesen.

Die letzte Phase ist der Zusammenhalt. In dieser Phase werden die Objekte zu ihren Nachbarn bewegt. Es wird wieder durch die Liste der Nachbarn für jedes Objekt iteriert, aber jetzt wird der durchschnittliche Positionsvektor für jeden Nachbarn berechnet. Danach werden die

Positionsvektoren der entsprechenden Objekte aktualisiert. In der Abbildung 2.3 werden alle Phasen des dargestellten Algorithmus visualisiert.

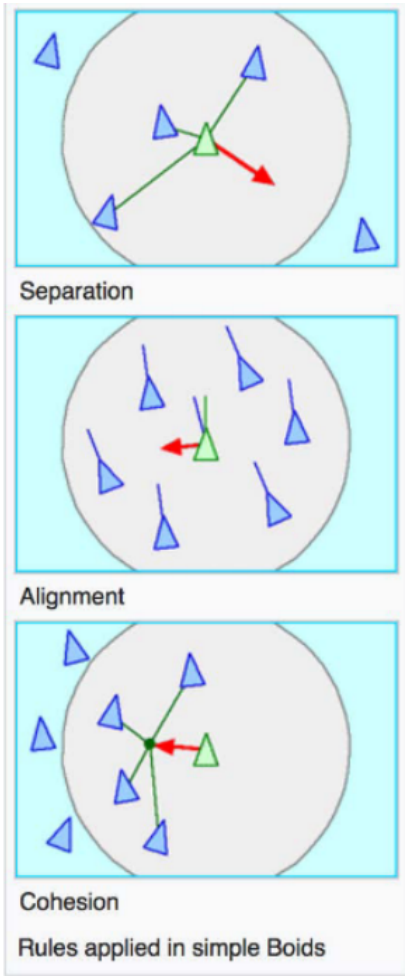


Abbildung 2.3: Phasen vom Separation Steering Algorithmus

2.1.5 Basis PCG Projekt

In diesem Abschnitt wird ein kurzer Überblick über das verwendete Basisprojekt gegeben. Als Basisprojekt für die Implementierung wurde ein Procedural Content Generation Projekt verwendet, das von Studenten der HAW Hamburg entwickelt wurde [35]. Das Projekt beschäftigt sich mit CGA unter Verwendung einer Shape-Grammatik. CGA steht für **Computer Generated Architecture** und bezeichnet eine prozedurale Modellierung von Architekturen. Dieses Projekt ermöglicht die prozedurale Generierung von Gebäuden. Im Rahmen dieser

Arbeit wird gezeigt, wie das Projekt erweitert werden kann, um Räume innerhalb der erstellten Gebäude zu produzieren, zu typisieren und Möbel innerhalb der erstellten Räume zu platzieren. Das Basisprojekt basiert auf einer Shape-Grammatik, mit der Regeln definiert werden, die bei der Gebäudeerstellung angewendet werden.

Das Projekt beinhaltet drei Hauptkomponenten für den Grammatikansatz:

1. Parser
2. Evaluator
3. Mesh Generator

Ein Parser wird verwendet, um eine gegebene Grammatik zu lesen und zu überprüfen, ob sie korrekt geschrieben ist. Die Grammatik besteht aus Variablen, Werten und Regeln mit Operationen. Bei der Anwendung jeder Regel, die definiert ist, wird das erstellte Modell entsprechend der Operation verändert. Insgesamt wurden in dem Basisprojekt 11 unterschiedliche Operationen implementiert, mit denen ein Gebäude erstellt werden kann. Es gibt Operationen, um die Formen der verwendeten Objekte zu verändern, Objekte zu transformieren oder neue Objekte wie Dächer oder Fenster zu erstellen.

Der Prozess der Regelanwendung findet im Evaluator statt. Hier werden alle Variablen und Regeln ausgewertet und auf das Basisobjekt angewendet. Das Basisobjekt ist ein Viereck, das mithilfe entsprechender Regeln modifiziert wird. Die Ausgabe des Evaluators ist ein Shape-Baum. Dieser Baum enthält alle Komponenten des Gebäudes. Diese Komponenten werden Shapes genannt und sollen nach weiterer Verarbeitung in 3D-Objekte, sogenannte TriangleMeshes, umgewandelt werden.

Die Umwandlung der Shapes in 3D-Objekte, sogenannte TriangleMeshes, findet im MeshGenerator statt. Ein TriangleMesh ist eine Datenstruktur, die aus Vertices, Dreiecken und Normalen besteht. Die Vertices sind einfache Punkte mit drei Koordinaten (X, Y und Z). Die Dreiecke sind Objekte, die aus drei Vertices und einer Normalen bestehen. Diese Normale zeigt die Richtung an, in die ein Dreieck orientiert ist. Diese Richtung ist wichtig, damit das Dreieck im 3D-Raum korrekt angezeigt wird und das Gebäude richtig modelliert werden kann.

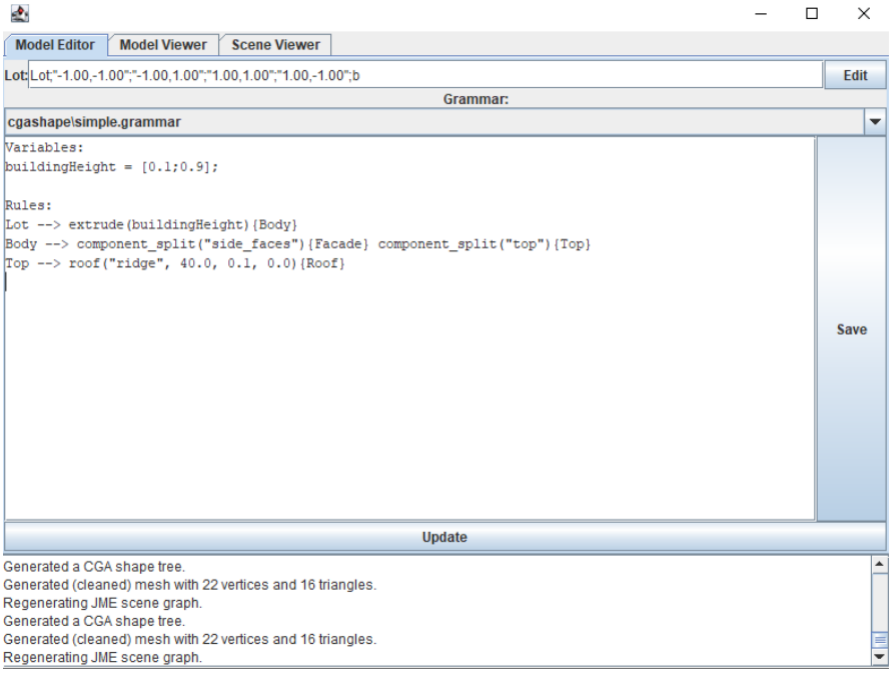


Abbildung 2.4: Grammatik von einem einfachen Gebäude

In der Abbildung 2.4 ist ein Beispiel von einer Shape-Grammatik zu sehen. Sie besteht aus einer Variable *buildingHeight* und drei Regeln. Die Regeln haben immer die gleiche Schreibweise: links vom Pfeil wird das Objekt geschrieben, auf welches die Regel angewendet wird, zum Beispiel *Lot*. Rechts vom Pfeil steht die Operation mit allen Parametern, die benutzt wird (*extrude(buildingHeight)*). Danach in geschweiften Klammern stehen die Objekte, die nach der Ausführung der Operation erstellt werden. Wenn auf das gleiche Objekt mehrere Regeln angewendet werden, sollen diese Regeln auch eine Wahrscheinlichkeit haben, ansonsten ist die Ausführung nicht möglich. Es kann auch die Bedingung hinzugefügt werden, und nur wenn sie erfüllt ist, wird die Regel ausgeführt. Das resultierende 3D-Objekt ist in der Abbildung 2.5 zu sehen.

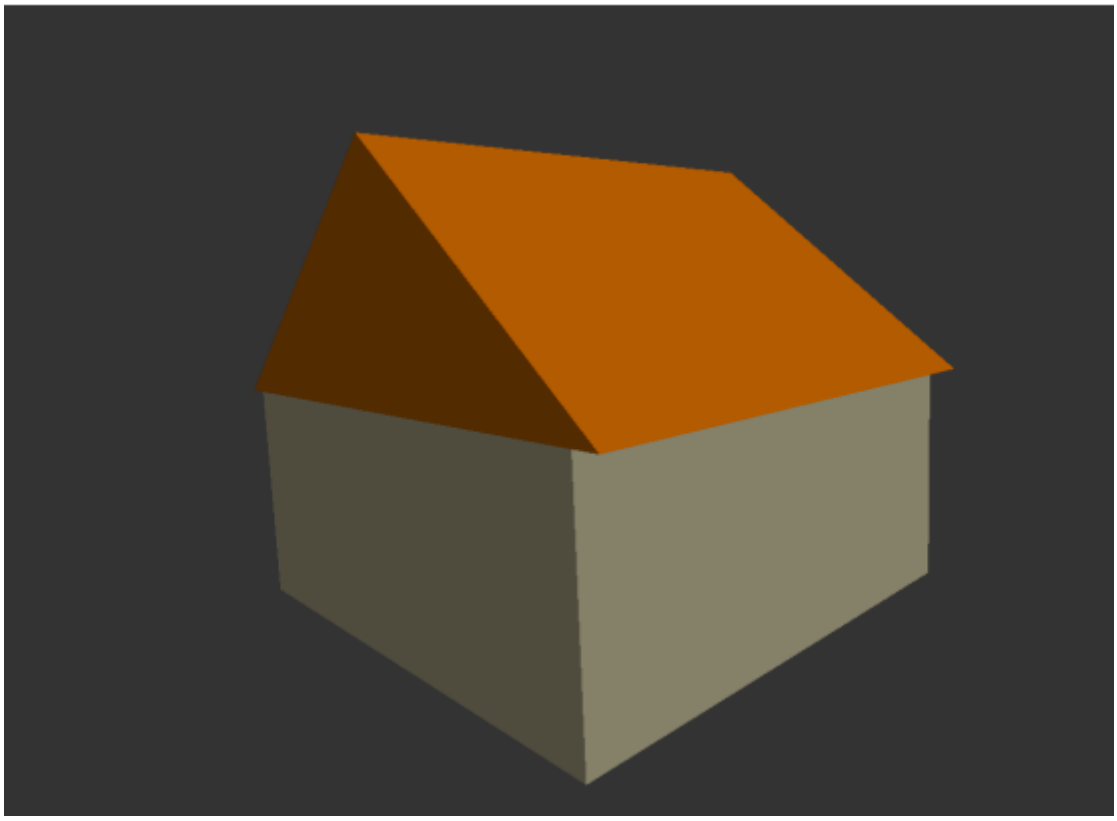


Abbildung 2.5: Das Gebäude aus der Grammatik in der Abbildung 1

Zu den bereits bestehenden Operationen werden dann drei neue Operationen hinzugefügt, die Raumpartitionierung, Raumtypisierung und Möbelplatzierung im Raum ermöglichen. Diese Operationen werden auf Basis von Konzepten entwickelt, die im späteren Kapitel präsentiert werden. Es gibt auch andere Projekte, die CGA-Grammatiken nutzen, um Gebäude in 3D mithilfe von Regeln zu definieren [27]. Mit CGA-Grammatiken können nicht nur einfache Gebäude, sondern auch große Städte generiert werden [36]. In diesem Kontext zeigten Parish und Müller, wie man große städtische Umgebungen generiert, in denen jedes Gebäude aus einfachen Massenmodellen und Shadern für Fassadendetails besteht. Ein weiterer Artikel demonstriert, wie man geometrische Details an Fassaden einzelner Gebäude mit CGA-Grammatiken generiert [37].

3 Anforderungen und Konzepte

3.1 Anforderungen

In diesem Kapitel werden die Konzepte der Methoden vorgestellt, die für die Entwicklung im Rahmen der Masterarbeit benötigt werden. Diese Konzepte beziehen sich sehr stark auf definierte Anforderungen und versuchen diese Anforderungen zu entsprechen. Bei der prototypischen Umsetzung von diesen Konzepten sollen diese Anforderungen erfüllt werden, damit die Ziele der Arbeit erreicht werden können.

1. Anforderungen für die Raumpartitionierung
   - 1.1 Es soll möglich sein, das Gebäude in Etagen aufzuteilen.
   - 1.2 Der BSP-Algorithmus soll implementiert werden.
   - 1.3 Der BSP-Algorithmus soll ein BSP-Baum mit Blättern erstellen.
   - 1.4 Für jede Teilung sollen zwei neue Blätter zum Baum hinzugefügt werden.
   - 1.5 Jedes Blatt ohne Unterblätter soll einen Raum repräsentieren.
   - 1.6 Die Etage soll in Räume zerlegt werden.
   - 1.7 Die Räume sollen vier Wände haben.
   - 1.8 Die Wände sollen aus Halbkanten bestehen.
   - 1.9 Die Wände sollen die Wandbreite haben.
   - 1.10 Die Wände sollen in 3D-Meshes umgewandelt sein.
   - 1.11 Die 3D-Meshes von den Wänden sollen miteinander verknüpft werden.
   - 1.12 Es soll eine neue Operation entwickelt werden, die die Raumverteilung ermöglicht.
   - 1.13 Die Operation soll auf Etagen angewendet werden.
   - 1.14 Es soll zwischen inneren und äußeren Wänden unterschieden werden.
   - 1.15 Es soll eine Methode entwickelt werden, die Halbkanten separieren kann.
   - 1.16 Der BSP-Algorithmus soll eine benutzerdefinierte Anzahl von Teilungen machen

2. Anforderungen für die Raumtypisierung
   - 2.1 Die Konfigdatei für die Raumtypisierung soll erstellt werden.
   - 2.2 Das Programm soll die Konfigdatei richtig ablesen und in Regeln unterteilen.
   - 2.3 Jede Regel soll korrekt interpretiert werden.
   - 2.4 Es soll möglich sein, obligatorische und optionale Regeln zu definieren.
   - 2.5 Jede Regel soll erfüllt werden.
   - 2.6 Der Typisierungsprozess ist nur dann abgeschlossen, wenn alle Regeln verarbeitet und erfüllt sind.
   - 2.7 Falls nicht alle Regeln erfüllt werden können, soll der Korrekturprozess gestartet werden.
   - 2.8 Der Korrekturprozess soll die getroffenen Entscheidungen analysieren und möglicherweise korrigieren.
   - 2.9 Die Verbindung zwischen den Räumen soll in Form eines Graphs dargestellt werden.
   - 2.10 Die Wände sollen die Metadaten haben.
   - 2.11 Die Metadaten sollen sagen, ob eine Tür in der Wand benötigt wird.
   - 2.12 Alle Außenwände sollen die Löcher für die Fenster haben.
   - 2.13 Nur eine Außenwand kann die Eingangstür haben.
   - 2.14 Die Größe von Fenstern und Türen soll konfigurierbar sein.
   - 2.15 Die Anzahl von Fenstern pro Außenwand soll konfigurierbar sein.
3. Anforderungen für die Möbelplatzierung
   - 3.1 Die Konfigdatei mit den Regeln soll erstellt werden.
   - 3.2 Die Konfigdatei mit den Regeln für die Möbel soll geparsed werden.
   - 3.3 Die Regeln sollen die Information über die Möbelposition und Abstandsfläche erhalten.
   - 3.4 Die Regeln sollen zeigen, ob das Möbelstück ein Paar mit einem anderen Möbelstück bildet.
   - 3.5 Die Regeln sollen beschreiben, wie die Paare zueinander positioniert sind.
   - 3.6 Die erstellte Regeln sollen separat in einem iterativen Prozess erfüllt werden.

3.7 Nach der Initialplatzierung sollen die Möbel nur in X- oder Z-Richtung bewegt werden.

3.8 Falls eine Kollision mit anderem Möbelstück stattfindet, soll die entsprechende Meldung auf der Konsole erscheinen.

3.9 Es soll versucht werden, die Kollision durch Bewegung der Möbel zu beheben.

3.10 Sobald die Kollision nicht mehr stattfindet, soll die Abstandsfläche von dem Möbelstück geprüft werden.

3.11 Die Information zur Abstandsfläche für einzelne Möbel soll in den Regeln definiert werden.

3.12 Wenn die Abstandsfläche nicht gewährleistet ist, soll das Möbelstück so lange verschoben werden, bis die Abstandsfläche vorhanden ist und keine Kollisionen verursacht sind.

3.13 Sobald alle Regeln bearbeitet sind, soll der iterative Prozess abgeschlossen werden.

3.14 Es müssen nicht alle Regeln erfüllt werden, aber bei nicht erfüllten Regeln soll eine Ausgabe mit der Erklärung erscheinen.

3.2 Innenraumpartitionierung mit der Binary Space Partition

In diesem Abschnitt wird das Konzept vorgestellt, das für die Raumpartitionierung innerhalb von Gebäuden entwickelt wurde. Das Ziel dieses Teils ist, die Etagenflächen der Gebäude in Räume aufzuteilen. Die Teilung erfolgt mithilfe des beschriebenen BSP-Algorithmus. Die dadurch erstellten Räume werden als separate Entitäten betrachtet und ihre Sammlung bildet die Etage. Die Anzahl der benötigten Teilungen wird vom Benutzer angegeben. Beim Ausführen des BSP-Algorithmus wird ein BSP-Baum erstellt, der alle Räume enthält, die während der Ausführung des Algorithmus erstellt wurden. Zu Beginn besteht der Baum nur aus einem Blatt, der als Wurzel des Baums dient und eine Etage ohne Partitionierung repräsentiert. Nachdem die Partitionierung durchgeführt wurde, werden nur die Räume, die keine Kinderblätter im Baum haben, als separate innere Räume des Gebäudes betrachtet. Der BSP-Algorithmus ist rekursiv, und die Unterteilung erfolgt so lange, bis die gewünschte Anzahl von Teilungen erreicht ist. Die Teilungslinien können als Wände für die Innenräume verwendet werden, um die Ebene in Räume mit unterschiedlichen Größen zu zerlegen. Die Teilung erfolgt abwechselnd in Richtung der X- und Z-Achse, damit die Räume optimaler zerteilt werden können. Die Wände sollen als eigene Entität dargestellt werden. Eine Sammlung von genau vier Wänden definiert einen Raum. Die Wände werden in zwei Kategorien unterteilt: innere und äußere

Wände. Diese Unterteilung ist für die Modellierung wichtig, damit Fenster und Türen an den richtigen Wänden platziert werden können. Die oben beschriebene Halbkanten Datenstruktur wird verwendet, um Wandinformationen (Start- und Endpunkt) sowie die gesamte Struktur der Etage darzustellen. Also jede erstellte Wand kann auch als eine separate Halbkante gesehen werden und enthält alle Informationen, die eine Halbkante liefern kann. Bei jeder Teilung soll verifiziert werden, ob ein Raum aus genau vier Wänden entsteht, die auch gemeinsam hintereinanderstehen und laut der Halbkanten Datenstruktur aufeinander verweisen. Nur in diesem Fall ist der Raum erfolgreich erstellt und kann als Blatt in dem BSP-Baum hinzugefügt werden. Die Halbkanten-Datenstruktur ist auch ein wichtiger Bestandteil der 3D-Modellierung und der partitionierten Räume. Genau diese Datenstruktur wird benötigt, um Wände des Gebäudes in 3D modellieren zu können. Um eine Etage in 3D darzustellen, werden die erstellten Wände und ihre Koordinaten benötigt. Diese Koordinaten beschreiben den Start und Endpunkt der Wand. Eine Wand kann also als eine Halbkante mit Start und Endpunkt und ggf. gegenüberstehender Halbkante dargestellt werden. Im Laufe der Modellierung der Räume werden alle Wände separat betrachtet und mithilfe der Koordinaten aus der Halbkanten-Datenstruktur modelliert. Es wird auch möglich sein, die Breite einer Wand während des Modellierungsprozesses festzulegen, um die modellierte Etage optisch ansprechender zu gestalten. Um den Prozess der Raumpartitionierung, BSP-Baumerstellung und Modellierung erfolgreich und in einer Sequenz innerhalb des Basisprojekts auszuführen, wurde eine neue Operation für die generative Figuren-Grammatik geschrieben, die alle diese Prozesse nacheinander startet und am Ende die gewünschten Gebäude mit Innenräumen produziert und modelliert.

3.3 Typisierung der Räume

Für die Typisierung wird ein regelbasiertes System verwendet. Dazu werden Regeln definiert, die unterschiedliche Eigenschaften des Raumes beinhalten. Zu den Eigenschaften gehören die Größe des Raumes, mögliche Nachbarräume, die Wahrscheinlichkeit, die Verbindung zur Ausgangstür und die Anzahl der Räume dieses Typs im Haus. Zusätzlich wird definiert, ob eine Regel obligatorisch oder optional ist. Optionale Regeln werden nur dann berücksichtigt, wenn ein Haus mehr Räume hat, als es definierte Regeln gibt, die obligatorisch sind. Diese Information über die Regeln wird als Konfigurationsdatei im System abgelegt. Diese Datei wird gelesen und die Regeln werden erstellt. Die Datei kann in Form einer CSV-Datei erstellt werden. Die Informationen in dieser Datei müssen evaluiert und geparsed werden, damit jede Spalte in der Datei als eine vollständige Regel definiert werden kann.

Die erstellten Regeln werden einem iterativen Algorithmus zur Verfügung gestellt. Nur die Räume, die mindestens so groß sind, wie der in der Regel definierte Wert, und die Nachbarschaftsanforderungen der Regel erfüllen, können den Typ aus der Regel erhalten. Um sicherzustellen, dass die Nachbarschaft immer geprüft wird, muss mindestens ein Raum im Gebäude bereits typisiert sein, damit der iterative Algorithmus abgeschlossen werden kann. Um den ersten Raum auszuwählen, wird der Wert des Attributs „Wahrscheinlichkeit“ benötigt. Dieser Wert zeigt die Wahrscheinlichkeit von einem Typ, als ersten vergeben zu werden. Es gibt auch einen Raumtyp, für den diese Prüfung nicht notwendig ist. Dieser Typ wird verwendet, wenn alle obligatorischen und optionalen Regeln erfüllt sind, es aber noch Räume ohne Typen im Gebäude gibt. Diese Räume erhalten dann den sogenannten „Default“-Typ, damit der Algorithmus abgeschlossen werden kann. Die Regeln werden nacheinander bearbeitet und es wird zwischen obligatorischen und optionalen Regeln unterschieden. Zunächst müssen alle obligatorischen Regeln erfüllt werden und danach, wenn es noch Räume ohne Typen im Gebäude gibt, werden die optionalen Regeln bearbeitet und erfüllt, bis alle Räume typisiert sind. Sobald dies der Fall ist, kann der Prozess beendet werden.

Es kann jedoch ein Problem auftreten, wenn nicht alle Regeln im iterativen Prozess erfüllt werden können. In diesem Fall muss ein Mechanismus entwickelt werden, der die vorherigen Entscheidungen untersucht und möglicherweise verbessert, damit alle Regeln erfüllt werden können. Ein solches Problem kann beispielsweise auftreten, wenn die Küche laut Konfiguration nur neben dem Wohnzimmer platziert werden kann, aber alle Räume, die mit dem Wohnzimmer benachbart sind, bereits mit Typen belegt sind. Der Korrekturprozess untersucht dann alle bereits erfüllten Regeln und versucht, die Räume anders zu typisieren, bis die Küche korrekt neben dem Wohnzimmer platziert werden kann. Wenn der Prozess gestartet wird, wird der Typ aus der Regel, die untersucht wird, aus dem entsprechenden Raum entfernt und einem anderen Raum zugewiesen. Zuerst werden die Räume untersucht, denen noch keine Typen zugewiesen wurden. Falls es keine passenden Räume für den Typ gibt, werden die Räume mit bereits gesetzten Typen untersucht. Immer wenn ein Typ erneut vergeben wird, wird geprüft, ob eine Regel, die den Korrekturprozess verursacht hat, erfüllt werden kann. Wenn im Laufe des Prozesses der Typ eines Raums geändert wird, wird die Typisierung für den alten Typ des Raums wiederholt, damit alle Regeln, die bereits erfüllt waren, auch erfüllt bleiben. Nur wenn alle bereits erfüllten Regeln und die Regel, die den Korrekturprozess verursacht hat, erfüllt sind, darf der Prozess beendet werden. Wenn es jedoch nicht möglich ist, alle Regeln durch den Korrekturprozess zu erfüllen, wird der gesamte Typisierungsprozess abgebrochen und eine Meldung wird auf der Konsole ausgegeben, die mögliche Probleme (nicht passende Raumgröße

oder Nachbarschaft) darstellt. Da für die Typisierung nur alle obligatorischen Regeln erfüllt werden müssen, wird der Korrekturprozess nur auf diese Regeln angewendet.

Erst, wenn die Räume typisiert sind, werden sie miteinander verbunden. Dafür werden die Räume zunächst zu einem Graphen hinzugefügt. Jeder Knoten in diesem Graphen repräsentiert einen Raum, und jede Kante stellt eine Verbindung zwischen zwei Räumen dar. Das unterscheidet sich von den Methoden, die im Kapitel 2 beschrieben wurden, da dort der Graph ein Teil der Typisierung ist. In dieser Methode wird der Graph jedoch erst nach der Typisierung erstellt. Er ist von der Typisierung abhängig und wird nur für den Verbindungsaufbau verwendet. Damit der Graph erfolgreich erstellt werden kann, soll durch jeden Raum durchgegangen werden und geprüft werden, ob es in der Nachbarschaft dieses Raums einen Raum gibt, dessen Typ in der Nachbarschaftsliste aus der Regel für den betrachteten Raum existiert. Wenn dies der Fall ist, wird eine Kante zwischen diesen Knoten erstellt. Eine graphische Darstellung des Raumgraphen ist in der Abbildung 3.1 zu sehen.

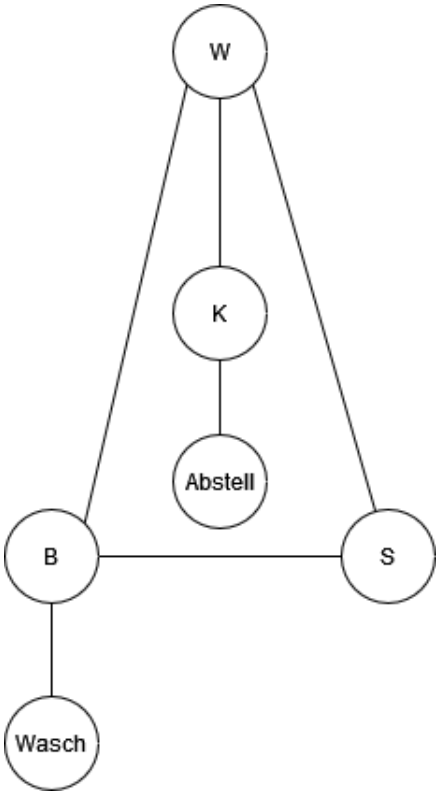


Abbildung 3.1: Raumgraph

In der Abbildung 3.1 ist zu sehen, dass der Knoten W (für Wohnzimmer) mit mehreren Knoten verbunden ist. Die Küche (Knoten K) ist mit dem Wohnzimmer und der Abstellkammer verbunden, während das Badezimmer mit dem Schlafzimmer (Knoten S), dem Waschraum und dem Wohnzimmer verbunden ist. Somit besteht die Verbindung zwischen den Räumen in beide Richtungen.

Wenn alle Raumtypen festgelegt und der Graph für die Verbindungen erstellt wurde, müssen diese Verbindungen im 3D-Mesh modelliert werden. In diesem Konzept sind keine Korridore in den Gebäuden vorgesehen, daher werden einfache Türen in den Wänden zwischen benachbarten Räumen als Verbindungen verwendet. Diese Türen werden als Löcher in den Wänden im 3D-Modell dargestellt. Um die Wände mit Türen und Fenstern darzustellen, soll ein spezielles Polygon benutzt werden, das die Löcher darstellen und ins Dreiecksmesh integrieren kann. Die Information, ob ein Loch in der Wand benötigt wird oder nicht, wird durch Wand-Metadaten dargestellt. Dies kann beispielsweise eine einfache Variable sein, die angibt, ob eine Tür benötigt wird oder nicht. Wände, die als Außenwände bezeichnet werden, können außer der Wand mit der Eingangstür keine Tür erhalten. Diese Wände werden immer mit Löchern modelliert, die Fenster darstellen. Die Größe der Fenster und Türen sowie die Anzahl der Fenster pro Außenwand sind konfigurierbar und können vom Benutzer definiert werden. Die genaue Sequenz des Typisierungsprozesses kann als folgendes Flowdiagramm graphisch dargestellt werden.

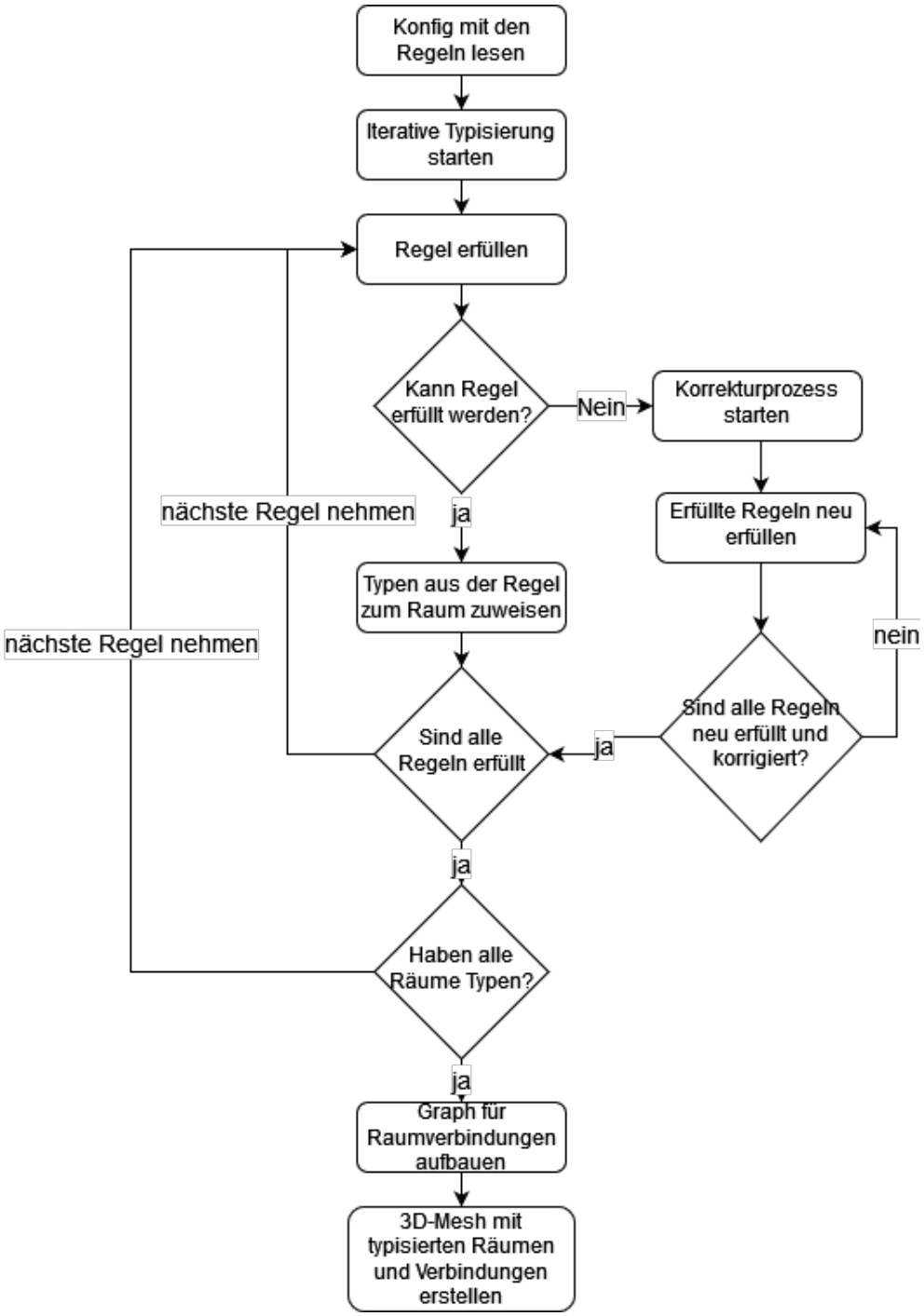


Abbildung 3.2: Flowdiagramm für Raumtypisierungsprozess

Um sicherzustellen, dass die Raumtypisierung und der Verbindungsaufbau nacheinander ausgeführt werden und in die bestehende Grammatik integriert werden können, wird eine Grammatikoperation erstellt. Diese Operation soll die Prozesse von Typisierung und Graphaufbau starten. Zusätzlich soll am Anfang bei dieser Operation die erstellte Konfigdatei gelesen und die Regeln geparsed werden. Diese Operation kann aber nur dann ausgeführt werden, wenn die Etage bereits in Räume partitioniert ist.

3.4 Methode der Möbelausstattung der Räume

Das Ziel dieses Abschnitts ist es, ein Konzept für den Prozess zu vermitteln, der eine erfolgreiche und kollisionsfreie Platzierung der Möbel im generierten Raum ermöglicht. Dieser Prozess kann auch als regelbasierter Prozess dargestellt werden. Die Platzierung hängt von den definierten Regeln und der Abstandsfläche ab. Die Abstandsfläche beschreibt sinnvolle Abstände zwischen den Möbeln, um sicherzustellen, dass alle Möbelstücke im Raum problemlos genutzt werden können. Die Abstandsfläche wird auch in unterschiedliche Richtungen des Möbelstücks geprüft. Die Regeln beschreiben die Eigenschaften der Möbelposition im Raum, zum Beispiel, dass ein Möbelstück neben der Wand oder in der Ecke stehen soll. Zusätzlich werden auch die Paareigenschaften der Möbel über die Regeln beschrieben. Die Position eines Möbelstücks, das als Paar mit einem anderen definiert ist, hängt von seinem Paarmöbelstück ab. Auch die Distanz zwischen den Paaren wird in den Regeln festgelegt. Die Regel wird eine Information beinhalten, wie die Paare zueinander stehen, zum Beispiel: soll ein Möbelstück sich gegenüber von einem anderen befinden, oder sollen sich die Möbelstücke um ein anderes Möbelstück herum oder neben dem Paarmöbelstück befinden. Der Abstand zwischen den Paarmöbelstücken wird ebenfalls in den Regeln definiert. Es ist auch möglich, über die Regeln zu definieren, ob ein Möbelstück Kollisionen verursachen kann. Dies ist zum Beispiel bei einem Teppich im Raum der Fall, der zwar nicht wichtig für die anderen Möbel ist, aber dennoch berücksichtigt werden muss, damit andere Möbelstücke auf ihm platziert werden können, ohne mit ihm zu kollidieren. Die Möbelstücke können auch mehrmals definiert werden, damit mehrere gleiche Möbelstücke im Zimmer platziert werden können, zum Beispiel mehrere Stühle. Türen und Fenster im Gebäude sind eine weitere Art von Objekten, die separat betrachtet werden müssen, um sie korrekt im Raum zu platzieren. Für die Möbelstücke, die Fenster überdecken können, wird eine spezielle Eigenschaft eingefügt, die signalisiert, ob diese Möbelstücke nicht vor dem Fenster platziert werden sollen. Diese Regeln können auch als Konfigurationsdatei definiert werden, die gelesen und geparsed werden soll, damit die Regeln erfolgreich definiert und verwendet werden können.

Die Möbel, die in den Räumen positioniert werden, werden als Dateien vom Typ *.obj* zur Verfügung gestellt. Nachdem die Regeln aus der Konfigurationsdatei gelesen und erstellt wurden, werden die in den Regeln erwähnten Möbelstücke aus diesen Dateien ausgelesen und als Dreiecksmeshes dargestellt. Genau diese Dreiecksmeshes werden am Ende des Prozesses zur Etage hinzugefügt und somit im 3D-Modell dargestellt. Die Möbel können aus dem Internet heruntergeladen und in der Blender-Software bearbeitet werden. Die Voraussetzungen für die Möbel sind eine optimale Größe, die in den Raum passt, sowie eine Ausrichtung entlang der negativen z-Achse. Wenn es eine Regel gibt, für die kein Möbelmodell vorhanden ist, kann der Prozess nicht gestartet werden.

Die Platzierung der Möbel erfolgt in einem iterativen Prozess, der für jede Regel durchgeführt wird. Der Prozess wird so lange durchgeführt, bis alle Regeln bearbeitet sind. Dabei wird jedes Möbelstück separat bewegt, bis alle Eigenschaften der Regel erfüllt sind. Wenn es nicht möglich ist, die Regeln zu erfüllen, wird eine Meldung auf der Konsole ausgegeben, die den Fehler beschreibt und den Benutzer darauf hinweist, was geändert werden kann, damit die Regel erfüllt wird.

Um eine bessere Verteilung der Möbel im Raum zu erreichen, erhalten die Möbel zuerst zufällige Plätze. Dann werden die Möbel mithilfe des Separation Steering Algorithmus gleichmäßig im Raum verteilt. Hierfür werden jedoch nur zwei Phasen des Algorithmus benötigt: Zusammenhalt und Trennung. Diese Optimierung soll für eine begrenzte Zeit durchgeführt werden und kann zum Beispiel nur für eine bestimmte benutzerdefinierte Anzahl von Iterationen erfolgen. Anschließend werden die Positionen der Möbel im Raum anhand der definierten Regeln angepasst.

Nach der initialen Platzierung wird das Möbelstück abhängig von den in den Regeln beschriebenen Eigenschaften der Position so lange bewegt, bis keine Kollisionen zwischen ihm und anderen Möbeln stattfinden. Danach wird die Abstandsfläche geprüft. Falls alles stimmt, wird die Regel als erfüllt markiert und es wird dann zum nächsten Möbelstück übergegangen. Falls ein Möbelstück nicht richtig platziert werden kann, obwohl es noch Platz im Zimmer gibt, wird geprüft, welche Kriterien nicht erfüllt sind. Wenn das Problem beispielsweise mit der Abstandsfläche zusammenhängt, wird versucht, das andere Möbelstück so weit wie nötig von diesem Möbelstück zu verschieben, bis die Abstandsfläche gewährleistet ist. Wenn das Problem darin besteht, Kollisionen mit anderen Möbeln zu vermeiden, wird versucht, die Position des betrachteten Möbelstücks zu ändern. Wenn es jedoch nicht möglich ist, eine Position ohne Kollision und mit entsprechendem Abstand zu finden, wird das Möbelstück nicht hinzugefügt. In diesem Fall erscheint eine entsprechende Meldung für den Benutzer, die erklärt, warum das Möbelstück nicht hinzugefügt werden kann.

Es gibt noch zwei Sonderfälle, bei denen die Positionierung der Möbel optimiert werden muss. Der erste Sonderfall tritt auf, wenn ein Möbelstück neben einer Tür platziert werden soll. Die Tür wird eine feste Abstandsfläche haben, die vom Benutzer definiert wird. Diese Abstandsfläche gibt an, dass das Möbelstück nicht näher an der Tür positioniert werden kann als der definierte Wert. Gleiches gilt auch für den Abstand zwischen den Fenstern und den Möbelstücken, die diese Fenster überdecken können. Bei der Prüfung und ggf. Verschiebung der Möbelstücke, die diese Abstände nicht einhalten, müssen auch die definierten Regeln für diese Möbelstücke berücksichtigt werden.

Bei Möbelstücken, die als Paar definiert sind, wird geprüft, ob ein Möbelstück ein Eltern- oder Kindermöbelstück ist. Elternmöbelstücke sollen nur entlang der Wand oder in der Ecke platziert werden. Gleiches gilt auch für die Möbelstücke, die keine Paare haben. In der folgenden Abbildung ist ein möglicher Prozess der Platzierung der Elternmöbelstücke als Flussdiagramm dargestellt.

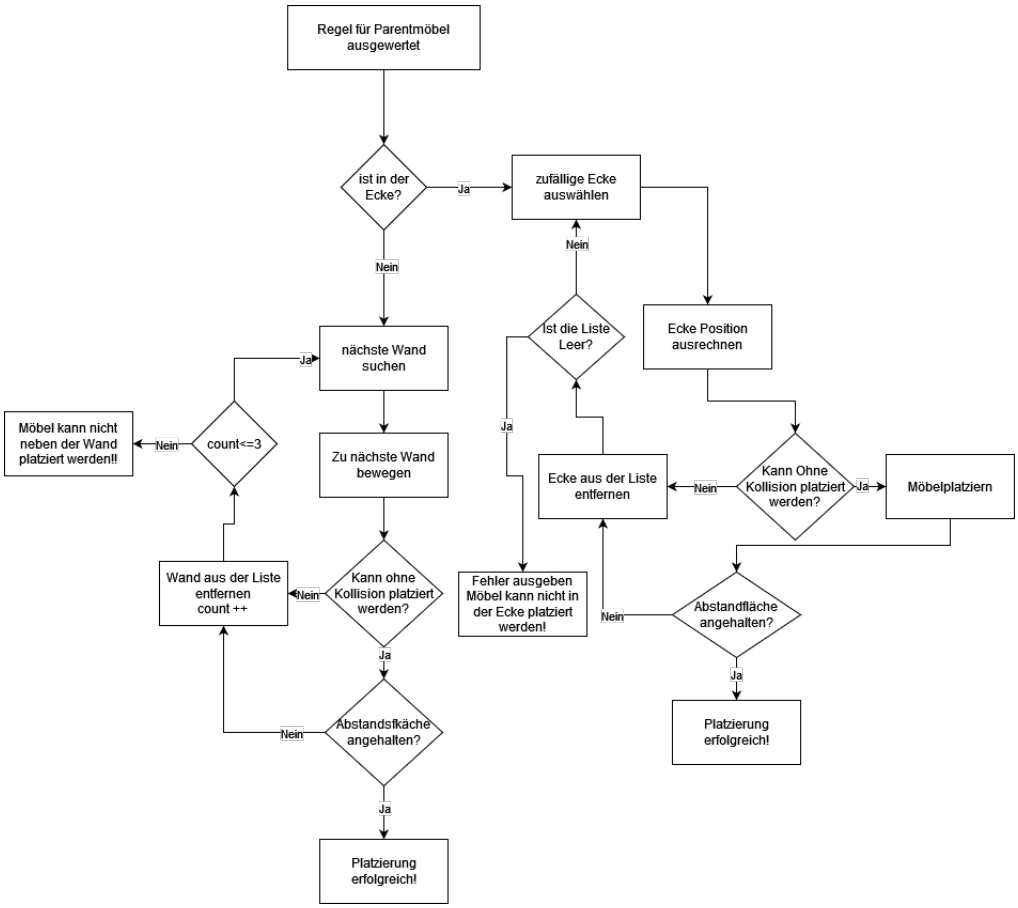


Abbildung 3.3: Flussdiagramm zur Elternmöbelstückplatzierung

Bei den Kinder-Möbelstücken ist der Vorgang komplizierter. Bei diesen Möbelstücken soll noch geprüft werden, wie sie zu Elternmöbelstücken stehen sollen. Wenn ein Kindermöbelstück um ein Elternmöbelstück herum stehen soll, sollen die Seiten des Elternmöbelstücks untersucht werden, um herauszufinden, ob dort Platz ist, um das Kindermöbelstück zu platzieren. Wenn der Platz gefunden ist, soll das Kindermöbelstück dort platziert werden und es soll auf Kollision geprüft werden. Wenn es keine Kollisionen gibt, wird das Kindermöbelstück erfolgreich platziert. Wenn das Kindermöbelstück neben dem Elternmöbelstück platziert werden soll, soll sichergestellt werden, dass nach der Platzierung das Kindermöbelstück keine Kollisionen mit dem Elternmöbelstück oder anderen Möbelstücken hat. Es soll auch geprüft werden, dass das Kindermöbelstück genau neben dem Elternmöbelstück steht. Wenn das Kindermöbelstück einfach nur vor dem Elternmöbelstück platziert werden soll, wird auch die Möglichkeit in Betracht gezogen, es neben der Wand zu platzieren. Diese Kriterien sollen nur für dieses

Kindermöbelstück geprüft werden. Zusätzlich sind Abstände zu Elternmöbelstücken relevant. Wenn eines der oben beschriebenen Kriterien nicht erfüllt werden kann, weil eine Kollision auftritt oder eine Abstandsfläche nicht eingehalten werden kann, sollen die entsprechenden Meldungen in der Konsole erscheinen, um dem Benutzer zu signalisieren, wo das Problem aufgetreten ist. Für dieses Konzept der Kindermöbelstückplatzierung kann auch ein möglicher Prozess als folgendes Flussdiagramm definiert werden.

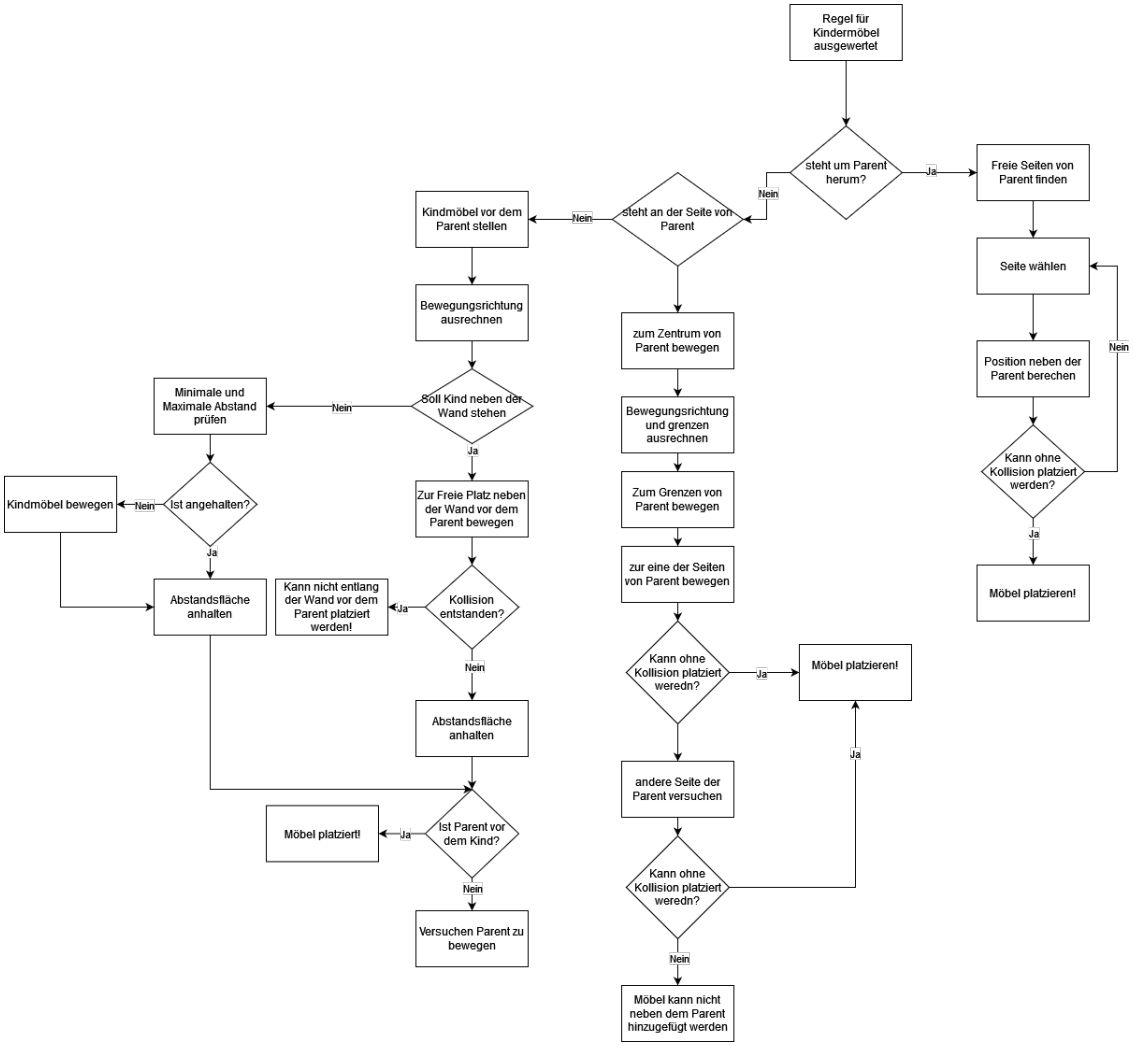


Abbildung 3.4: Flussdiagramm zur Kindermöbelstückplatzierung

Es könnte sein, dass nach der Platzierung der Kindermöbelstücke diese nicht direkt gegenüber dem Elternmöbelstück stehen. Aus diesem Grund wird versucht, die Position des Elternmöbelstücks anzupassen, damit es dem Kindermöbelstück gegenübersteht. Wenn das

Elternmöbelstück ohne Kollisionen und Abstandsmangel verschoben werden kann und gegenüber dem Kindermöbelstück platziert werden kann, wird es auch verschoben. Wenn es jedoch aufgrund von Kollisionen oder Platzmangel nicht möglich ist, das Elternmöbelstück zu platzieren, wird keine Verschiebung durchgeführt. Dieser Prozess soll jedoch keinen Fehler verursachen, sondern nur als Optimierung angesehen werden. Der Prozess wird nur dann gestartet, wenn ein Kindermöbelstück keine explizite Ausrichtung gegenüber dem Elternmöbelstück hat.

Genauso wie in den beiden oberen Abschnitten soll hier eine Operation für die Figur-Grammatik entwickelt werden, mit der der Prozess der Möbelplatzierung gestartet werden kann. Dieser Prozess kann wiederum nur dann ausgeführt werden, wenn die Etage bereits in Räume partitioniert ist, und die Räume auch typisiert sind. Diese Operation soll alle benötigten Informationen erhalten, um den Prozess erfolgreich zu starten. Diese Informationen können beispielsweise den Pfad zur Konfigurationsdatei und die Anzahl der Iterationen bei der Optimierung enthalten.

4 Verwendete Software

In diesem Kapitel werden die Frameworks, die in der Arbeit verwendet wurden, präsentiert. Zusätzlich wird hier kurz die Hardware dargestellt, die für die Tests benutzt wurde.

Die Programmiersprache, in der die Software geschrieben wird, ist Java. Die benutzten Frameworks sind mit dieser Programmiersprache kompatibel. Die Entwicklungsumgebung, die für die Implementierung benutzt wurde, ist IntelliJ.Für die Implementierung wurde die **JMonkeyEngine** verwendet[1]. JMonkeyEngine (kurz jME) ist eine Szenengraph-basierte und komplett in Java geschriebene Spiel-Engine. Viele der in jME umgesetzten Konzepten stammen aus dem Buch „3D Game Engine Design“ von David Eberly [38]. jME wurde entwickelt, um Java-Entwicklern eine voll funktionsfähige Grafikengine zur Verfügung zu stellen. jMonkeyEngine ist nur eine Abstraktionsebene von OpenGL entfernt. Die Engine stellt moderne OpenGL-Fähigkeiten bereit und profitiert aufgrund ihres geringen Abstraktionsniveaus von hoher Leistung. Die jMonkeyEngine ist ein communityzentriertes Open-Source-Projekt, das unter einer BSD-Lizenz steht.

Nicht alle Funktionalitäten aus der oben dargestellten Spiel-Engine werden verwendet. Grundsätzlich sind die Prozesse von Vektor- und Matrizenrechnung aus diesem Framework für die Entwicklung in Rahmen dieser Arbeit relevant. Im Basisprojekt wurden jedoch einige Prozesse implementiert, bei denen mehr Funktionalität aus der jMonkeyEngine genutzt wurde. Zum Beispiel wird sie bei der Erstellung von 3D-Ansicht der Gebäude benutzt.

Damit der Möbelplatzierungsprozess erfolgreich durchgeführt werden kann, sollen die 3D-Modelle als *.obj* Dateien für die Software zur Verfügung gestellt werden. Diese Dateien werde aus Open-Source Quellen im Internet heruntergeladen. Diese Quellen sind Free3D[2] und CGTrader[3], in beiden Webseiten können unterschiedliche 3D-Modelle für alle Zwecke gefunden werden. Es gibt hier sowohl kostenlose als auch bezahlte Modelle und Sammlungen von 3D-Objekten. Die Objekte können in unterschiedlichen Formen heruntergeladen werden.

Bevor die 3D-Objekte von Möbeln verwendet werden können, müssen sie zuerst bearbeitet und angepasst werden. Die Bearbeitung dieser Objekte erfolgt mithilfe der Blender-Software.

[1] JMonkeyEngine (https://hub.jmonkeyengine.org/)
[2] https://free3d.com/
[3] https://www.cgtrader.com/

Blender ist eine kostenfreie 3D-Grafiksuite, die unter der GPL lizenziert ist und mit der Körper modelliert, texturiert und animiert werden können[4]. Diese können in Grafiken, Animationen und Software integriert werden. Erstellte Bildsynthesen können mithilfe des integrierten Compositors und Videoschnittprogramms nachbearbeitet werden. Für die Entwicklung dieses Programms werden C und C++ als Programmiersprachen genutzt, Python wird als Skriptsprache verwendet. Trotz seines Funktionsumfangs ist das Programm relativ klein und läuft auf den meisten gebräuchlichen Rechnersystemen. Um Änderungen an 3D-Objekten in dieser Software vornehmen zu können, müssen diese Objekte als *.blend*-Dateien heruntergeladen werden. Die Änderungen betreffen die Größe und Ausrichtung eines Objekts. Diese beiden Eigenschaften werden so eingestellt, dass sie ergonomisch gut in einen Raum passen, d.h. die Objekte werden verkleinert, um besser in den Raum zu passen. Die Ausrichtung von Objekten soll auch fest zur $-Z$ Achse gesetzt werden. Diese Ausrichtung ist festgelegt, damit alle benötigten Drehungen der Möbel erfolgreich durchgeführt werden können. So wird das Möbelstück am Ende richtig in Bezug auf die Wände des Gebäudes und andere Möbel im Raum ausgerichtet.

Die drei Teile der Software wurden nach der Entwicklung getestet, um zu überprüfen, ob alle Anforderungen erfüllt sind und das Ziel der Arbeit, Innenräume von Gebäuden zu modellieren, typisieren und mit den Möbeln zu füllen erreicht wurde. Die Tests wurden auf einem Laptop mit einem Intel(R) Core(TM) i5-9300H-Prozessor und einer NVIDIA GeForce RTX 2060-Grafikkarte durchgeführt.

[4] Bleder (https://www.blender.org/)

5 Implementierung

5.1 Raumpartitionierungsalgorithmus

In diesem Kapitel wird die Implementierung des Raumpartitionierungsalgorithmus vorgestellt, der im Rahmen der Masterarbeit als Konzept entwickelt wurde. Darüber hinaus werden auch Hilfsklassen sowie die Implementierung der Halbkanten-Datenstruktur präsentiert.

5.1.1 Halbkanten-Datenstruktur und Etagenerstellung

Für die Implementierung der Halbkanten-Datenstruktur wurden drei Java-Klassen entwickelt. Die erste Klasse repräsentiert die Halbkante, die zweite Klasse den Start- und Endpunkte der Kante und die dritte Klasse die gesamte Halbkanten-Datenstruktur. Die Halbkante enthält Verweise auf ihre Nachfolger, die gegenüberliegende Kante und den Anfangspunkt, der ein Punkt im 3D-Koordinatensystem ist. Die Klasse für die Datenstruktur besteht aus einer Liste aller Halbkanten, die mit dem Raumpartitionierungsalgorithmus erstellt wurden. Die Start- und Endpunkt der Halbkante werden als Objekte der Klasse *Vector3f* dargestellt. Diese Klasse kommt aus der JMonkeyEngine.

Die Implementierung dieser Datenstruktur ist notwendig, um Räume darstellen zu können. Der minimale Bestandteil eines Raums ist eine Wand, die durch eine Halbkante repräsentiert wird. Zusätzlich wird für jede Wand eine Variable hinzugefügt, die angibt, ob es sich um eine äußere oder innere Wand handelt. Es wird dann eine Entitätenklasse erstellt, die aus einer Halbkante und einem Flag für den Wandtyp besteht. Ein Raum besteht immer aus vier Wänden, die in der Halbkanten-Datenstruktur nacheinander aufgebaut sind. Jede Wand hat als Nachfolger eine weitere Wand, die zum gleichen Raum gehört. Hier wird auch eine Entitätenklasse erstellt, die einen Raum als Liste von vier Wänden darstellt.

Eine Etage besteht aus allen Räumen, die sich darauf befinden. Die Etage enthält auch allgemeine Informationen zu allen Räumen und zur Wandbreite. Um eine Etage zu erstellen, wird eine neue Operation in der bestehenden Grammatik entwickelt, die die Prozesse der Raumpartitionierung, der Halbkanten-Datenstrukturerstellung und der 3D-Modellerstellung für diese Etage durchführt. Diese Operation wird in weiteren Abschnitten genauer erläutert.

5.1.2 Aufbau vom BSP-Algorithmus

Für die Entwicklung des BSP-Algorithmus wird eine spezielle Datenstruktur benötigt, der BSP-Baum. Dieser Baum besteht aus Blättern. Zu Beginn besteht der Baum nur aus einem Blatt, der als Wurzel des Baums dient und eine Etage ohne Partitionierung repräsentiert. Dieser Raum kann als ein einziger Raum auf der Etage vorgestellt werden, der nur aus äußeren Wänden besteht. Der Vorgang der Raumteilung wird als *Split* bezeichnet. Jede Teilung kann als Erstellung einer inneren Wand betrachtet werden, wodurch insgesamt zwei Arten von Wänden entstehen: äußere und innere Wände. Aus diesen Wänden wird dann der Raum erstellt. Bevor die Teilung durchgeführt wird, werden die für die Teilung benötigten Halbkanten anhand der Teilungsrichtung (X-Achse oder Z-Achse) ermittelt. Dafür werden zwei Halbkanten, die perpendikular zur Teilungsrichtung ausgerichtet sind, gefunden. Für jede dieser Halbkanten wird ein neuer Punkt in der Mitte der Halbkante erstellt und dadurch wird die Halbkante in zwei neue, gleich große Halbkanten geteilt. Sobald die Teilung der Halbkante abgeschlossen ist und die neuen Halbkanten erstellt sind, werden sie korrekt mit anderen benachbarten Halbkanten verbunden. Um sicherzustellen, dass die neuen Kanten korrekt verbunden sind, muss ausgehend von einer neuen Halbkante ein geschlossener Zyklus gebildet werden. Der Zyklus wird geprüft, indem von einer Halbkante zu ihrem Nachfolger gegangen wird, und dieser Prozess wird wiederholt, bis die ursprüngliche Halbkante erreicht wird. Wenn die ursprüngliche Halbkante erreicht wird, ist der Zyklus erfolgreich abgeschlossen. Am Ende des Teilungsprozesses werden zwischen den neu erstellten Punkten zwei Halbkanten in entgegengesetzte Richtungen erstellt, die als innere Wand bezeichnet werden. Eine dieser Halbkanten steht der anderen gegenüber. Auch bei der Erstellung dieser Wand werden die Verbindungen zwischen den Halbkanten untersucht und korrigiert, damit die neuen Halbkanten korrekte Verweise auf ihre Nachfolger erhalten und am Ende auch die Zyklen zwischen den Halbkanten korrekt aufgebaut werden. Diese Zyklen dienen als Prüfung, ob die Teilung erfolgreich war und ein Raum erstellt wurde. Wenn eine Teilung erfolgreich durchgeführt wurde und eine neue Wand erstellt wurde, wird auf Basis dieser Wand ein neuer Raum erzeugt. Dieser Raum wird dann als neues Blatt zum Baum hinzugefügt. Bei jeder Teilung werden immer ein rechtes und ein linkes Blatt erzeugt und zum Baum hinzugefügt. Deswegen werden auch bei der Teilung immer zwei Halbkanten für die Innenwand erstellt, damit auch zwei Blätter definiert werden können. Wenn eine benutzerdefinierte Anzahl von Teilungen durchgeführt wird, repräsentieren alle Blätter im Baum, die keine Kinderblätter haben, Räume in der Etage. Mit diesem Algorithmus wird ein BSP-Baum mit Blättern erstellt, der die Partitionierung der Etage in Räume darstellt.

Um eine Operation für die generative Grammatik zu erstellen, die die Erzeugung von 3D-Modellen für Etagen ermöglicht, müssen alle Methoden zur Teilung, Halbkanten-Datenstruktur

und zum Aufbau des BSP-Baums in einer Sequenz ausgeführt werden. Zunächst wird eine Wurzel des BSP-Baums erstellt, die aus einem Raum besteht, der nur äußere Wände hat. Sobald die Wurzel erstellt wurde, wird der BSP-Baum mit einer benutzerdefinierten Anzahl von Teilungen erstellt. Nachdem der Prozess der Baumerstellung abgeschlossen ist, werden alle Blätter, die sich auf der untersten Ebene des Baums befinden und keine Kinderblätter haben, als separate Räume zu einer Etage hinzugefügt. Der Pseudocode für diesen Vorgang ist im Algorithmus 1 beschrieben.

Algorithm 1 generateShapesFrom

```
1:  function generateShapesFrom(shape, childIds)
2:      result ← empty list
3:      root ← RoomCreationUtils.createOuterWalls(shape)
4:      rootLeaf ← new Leaf(root, ShapeOperationSplit.Axis.X)
5:      tree ← new BSPTree(rootLeaf, split)
6:      tree.createBSPTree()
7:      leaves ← tree.getAllLeafs()
8:      roomsOnTheFloor ← empty list
9:      for l ∈ leaves do
10:         if l.getLeftChild() == null & l.getRightChild() == null then
11:             roomsOnTheFloor.add(l.getShape())
12:         end if
13:     end for
14:     adjustWallType(shape, roomsOnTheFloor, root)
15:     wallStartPoints ← empty list
16:     for w ∈ root.getWalls() do
17:         wallStartPoints.add(w.getShape().getStartVertex().getPosition())
18:     end for
19:     floorShape ← new FloorShape()
20:     result.add(floorShape)
21:     return result
22: end function
```

Diese Operation wurde für die verwendete Grammatik erstellt, damit der Prozess der Partitionierung über die Grammatik, die in der Software verwendet wird, gestartet werden konnte. Die Pseudocodes für Hilfsmethoden wie *createBSPTree()*, die für die erfolgreiche Ausführung dieses Algorithmus benötigt werden, sind im Anhang zu finden 8. Einige Algorithmen, die

selbsterklärende Namen haben, wie zum Beispiel *createOuterWalls*, werden nicht als separater Pseudocode erfasst.

Nach der Durchführung dieses Algorithmus wird aus einer gegebenen Form ein anderes Objekt (FloorShape) erzeugt, das die erstellte Etage mit den Räumen repräsentiert. Dieses Objekt enthält eine Liste von allen Räumen und wird später bei der Typisierung und Möbelplatzierung benötigt, um Informationen über die Räume mit anderen Prozessen zu teilen. Darüber hinaus ist dieses Objekt dasjenige, das in 3D modelliert und auf der Oberfläche dargestellt wird.

5.1.3 3D-Modellierung der Etage mit Räumen

Wenn eine Etage mit Räumen partitioniert wird, soll diese Etage auch in 3D modelliert werden. Um die Etage richtig zu modellieren, werden alle erstellten Wände separat in 3D-Meshes umgewandelt und anschließend werden alle in 3D dargestellten Wände zusammengefügt, um eine Etage zu modellieren. Dabei wird ein Dreiecksmesh benutzt. Zunächst werden die äußeren Wände betrachtet und mit Fenstern modelliert. Falls eine äußere Wand mehr als ein Fenster haben soll, wird die Wand in mehrere Teile geteilt, wobei die Anzahl der Teile von der Anzahl der Fenster abhängt. Es wird überprüft, ob eine Wand groß genug ist, um die erforderliche Anzahl von Fenstern aufzunehmen. Das wird anhand der Größe der Wand, der Anzahl von Fenstern und der Größe der Fenster geprüft. Wenn dies nicht der Fall ist und nicht alle benötigten Fenstern erstellt werden können, wird die Anzahl der Fenster reduziert, bis die maximale Anzahl von Fenstern ermittelt wird, die auf der Wand platziert werden können. Anschließend wird die Wand entsprechend unterteilt und jede Teilwand separat verarbeitet und modelliert.

Für die Modellierung wird ein Polygon mit Löchern benötigt, da ein geschlossenes Polygon keine Löcher modellieren kann. Für jede Teilwand werden zuerst die Basispunkte ermittelt, die die Eckpunkte der Wand beschreiben. Danach werden die Punkte ermittelt, die ein Loch repräsentieren, dafür werden die Maße der Fenster (Höhe und Breite) benötigt. Diese Punkte sind einfache 3D-Koordinaten, die sich auf der Wand befinden. Die Basispunkte sind dann die Grenzen eines Polygons und die Lochpunkte beschreiben die Ecken des Lochs, das modelliert werden soll. Die Triangulierung des Polygons erfolgt mithilfe des Ear-Cutting-Algorithmus [39]. Wenn das erstellte Polygon trianguliert wird, erfolgt die Erstellung eines 3D-Meshs mithilfe der Triangulierungspunkte, Basispunkte und Lochpunkte. In ein Mesh-Objekt werden zuerst alle diese Punkte hinzugefügt. Der nächste Schritt ist das Hinzufügen von Dreiecken in das Mesh, die diese Punkte als Ecken haben. Zunächst werden die obere und untere Seite der Wand modelliert und anschließend werden die Dreiecke in die Seiten der Wand hinzugefügt. Wenn

alle Dreiecke hinzugefügt sind, werden die Normalen für diese Dreiecke berechnet und das erstellte Mesh mit dem gesamten Mesh des Raums verknüpft.

Der gleiche Vorgang wird auch für die inneren Wände durchgeführt, wobei die inneren Wände nicht geteilt werden. Für die inneren Wände, die laut den Metadaten eine Tür haben müssen, wird ein Loch für die Tür in das Polygon hinzugefügt. Die Metadatenvergabe erfolgt während der Ausführung des Typisierungsalgorithmus. Diese Metadaten sind notwendig, um den 3D-Modellierungsprozess anzuweisen, ob eine Wand überhaupt modelliert werden soll oder ob sie zusammen mit einer Tür modelliert werden soll. Der Prozess der Metadatenvergabe wird im nächsten Abschnitt beschrieben. Wenn die Modellierung aller Wände abgeschlossen ist, werden alle erstellten Meshes zu einem großen Mesh zusammengefügt und somit ein 3D-Mesh des Gebäudes mit allen Räumen modelliert.

5.2 Raumtypisierungsvorgang

In diesem Kapitel wird die Implementierung des oben beschriebenen Konzepts der Raumtypisierung im Detail erklärt. Es werden alle Prozesse, die für die Typisierung zuständig sind, dargestellt. Wichtige Prozesse werden auch als Pseudocode präsentiert, um das Verständnis zu erleichtern.

5.2.1 Regelaufbau, Parser und Regel-Auswertung

Der erste Teil der Implementierung bezieht sich auf die Erstellung einer Konfigurationsdatei mit Regeln, die zur Typisierung von Räumen verwendet werden. Die Konfigurationsdatei ist eine einfache Datei im CSV-Format, die dem Parser übergeben wird. In der Abbildung 5.1 ist ein Beispiel der Konfigurationsdatei zu sehen.

```
Type;Size;Prob;Neighbours;isOutsideConnected;appearence;additionRoom
Kitchen;0.25;0.5;Livingroom;0;1;0
Livingroom;1;1;Bathroom,Livingroom,Badroom,Kitchen;true;1;0
Badroom;1;0.1;Bathroom,Livingroom,Badroom;0;1;0
Bathroom;0.25;0.4;Badroom,Livingroom,Bathroom;0;1;0
Laundryroom;0.1;0;Bathroom;0;1;1
Storageroom;0.1;0;Kitchen;0;1;1
Commonroom;0;0;Bathroom,Livingroom,Badroom,Kitchen;0;1;1
```

Abbildung 5.1: Beispiel einer Konfigdatei

Das Parsen einer solchen Datei erfolgt durch einfache String-Teilungsoperationen. Zunächst wird die Datei anhand von Zeilenumbrüchen in separate Regeln aufgeteilt und dann anhand von Semikolons in die einzelnen Informationen für jede Regel aufgeteilt und entsprechend gespeichert. Die erste Zeile mit den Spaltennamen wird dabei übersprungen.

Für die Erstellung einer Regel wird in Java eine Entitätsklasse mit dem Namen **TypeRule** erstellt. Die Attribute dieser Klasse werden aus den Elementen der ersten Zeile der Konfigurationsdatei abgeleitet. Der *Type* beschreibt einen Raumtypen, abhängig von der Funktionalität des Raums. Hierfür wird ein Enum erstellt, das alle möglichen Raumtypen enthält. Um neue Raumtypen hinzufügen zu können, muss dieses Enum um diese Typen erweitert werden. Das Attribut *Size* beschreibt die Größe des Raums in Quadratmetern. Da in dem Basisprojekt verwendete Maßstab 1:10 Meter ist, wird die Größe als Dezimalzahl dargestellt. Das Attribut *Prob* bezeichnet die Wahrscheinlichkeit, dass die Regel als erste Regel verwendet wird. Dieses Attribut ist wichtig, um den iterativen Typisierungsprozess erfolgreich zu starten. Es muss mindestens ein Raum typisiert sein, damit weitere Typen vergeben werden können. Der Grund hierfür ist das Attribut *Neighbours*. *Neighbours* gibt an, welche Räume mit diesem Raum benachbart sein können. Der Raumtyp aus einer Regel kann nur den Räumen zugewiesen werden, die sich neben den Räumen befinden, deren Typ in dem Attribut *Neighbours* angegeben ist. Das nächste Attribut *isOutsideConnected* beschreibt, ob ein Raum mit einer Eingangstür modelliert werden muss. Es darf nur eine Regel existieren, bei der dieser Parameter auf den Wert "true"gesetzt ist. Das Attribut *Appearance* gibt an, wie oft diese Regel erfüllt werden darf. Das letzte Attribut, *AdditionRoom*, teilt die Regeln in zwei Kategorien auf: obligatorische Regeln, die vergeben werden müssen, und optionale Regeln. Optionale Regeln werden nur dann erfüllt, wenn alle obligatorischen Regeln erfüllt sind und es noch Räume gibt, die nicht typisiert sind.

5.2.2 Iterativer Algorithmus für Typenvergabe und Regelkorrekturprozess

Nachdem die Regeln aus der Konfigurationsdatei gelesen und erstellt wurden, werden sie in einer Liste gesammelt und an einen iterativen Algorithmus übergeben, damit sie auf die Räume angewendet werden können. Das ist der zweite Teil des Typisierungsprozesses. Der Algorithmus wird weiter im Detail beschrieben.

Die Raumtypenzuweisung erfolgt in zwei Phasen. In der ersten Phase wird eine Regel sowie ein zufälliger Raum ausgewählt und der Typ aus der Regel wird diesem Raum zugewiesen. Die zufällig ausgewählte Zahl im Bereich zwischen 0 und 1 wird für jede Regel mit dem Attribut *Prob* verglichen. Wenn *Prob* größer als diese Zufallszahl ist, darf die Regel als erste angewendet werden. Wenn es mehrere Regeln gibt, die diesem Kriterium genügen, werden diese Regeln in eine Liste hinzugefügt. Die Regel, die zuerst erfüllt sein soll, wird zufällig aus dieser Liste ausgewählt. Wenn es noch weitere Regeln gibt, die nicht erfüllt sind oder es Räume gibt, die noch keinen Typ haben, wird mit dem eigentlichen Typisierungsprozess begonnen. Das ist die zweite Phase der Typisierung, und der Prozess wird so lange durchgeführt, bis alle Räume

typisiert sind oder alle Regeln erfüllt sind. Zuerst wird der Prozess für alle Regeln durchgeführt, die obligatorisch erfüllt werden müssen. Danach, wenn es noch nicht typisierte Räume gibt, werden die optionalen Regeln erfüllt.

Zu Beginn wird geprüft, ob eine Regel überhaupt erfüllt werden kann. Dafür wird geprüft, ob ein Raumtyp aus dem Regelattribut "Neighbours"bereits einem Raum auf der Etage zugewiesen ist. Wenn dies der Fall ist, wird die Regel weiter verarbeitet und anschließend an einen Raum vergeben, ansonsten wird mit der nächsten Regel fortgefahren.

Generell werden nur zwei Bedingungen geprüft, bevor der Typ gemäß der Regel einem Raum zugewiesen wird. Diese Bedingungen sind die Raumgröße und die Raumnachbarschaft. Der Typ wird nur einem Raum zugewiesen, der größer als oder gleich dem Regelattribut *Size* ist und mindestens einen Raum mit dem Typ aus dem Attribut *Neighbours* als Nachbarraum hat. Falls es einen solchen Raum gibt, wird der Typ gemäß der Regel diesem Raum zugewiesen und die Regel ist erfüllt. Der Typ, der diese Prüfung nicht benötigt, ist *Commonroom*. Dieser Typ wird am Ende allen Räumen zugewiesen, die noch keinen Typ erhalten haben. Sobald alle obligatorischen und anderen optionalen Regeln vergeben sind und es noch Räume ohne Typ gibt, werden diese Räume dem Typ *Commonroom* zugewiesen. Die erfüllten Regeln werden in einer separaten Map gespeichert, um möglicherweise später für den Korrekturprozess verwendet zu werden. Die Map stellt eine Verbindung zwischen der Regel und dem Raum her, zu dem diese Regel angewendet wurde. Um zu überprüfen, ob der Korrekturprozess erforderlich ist, wird geprüft, ob die Anzahl der Regeln in dieser Map mit der tatsächlichen Anzahl der erfüllten Regeln übereinstimmt. Wenn dies nicht der Fall ist, wird ein Korrekturprozess gestartet. Ein Pseudocode für den iterativen Typisierungsprozess ist im Algorithmus 2 zu finden.

Algorithm 2 processRules

```
1:  function processRules(rooms, rulesList)
2:      random ← new Random()
3:      firstRules ← new ArrayList<TypeRule>()
4:      chance ← random.nextDouble(0, 1)
5:      for all rule in rulesList.get("needed") do
6:          if chance <= rule.getAppearanceProbability() then
7:              firstRules.add(rule)
8:          end if
9:      end for
10:     firstRule ← firstRules.get(random.nextInt(0, firstRules.size()))
11:     r ← rooms.get(random.nextInt(0, rooms.size()))
12:     r.setType(firstRule.getNeededType())
13:     doneRulesRoomMap ← new HashMap<TypeRule, Room>()
14:     doneRulesRoomMap.put(firstRule,r)
15:     rulesList.get("needed").remove(firstRule)
16:     while ¬ allNeededAssigned(rulesList.get("needed")) & ¬allTyped(rooms) do
17:         assignRules(rooms, rulesList.get("needed"), doneRulesRoomMap)
18:     end while
19:     if rooms.size() > rulesList.get("needed").size() & ¬ allTyped(rooms) then
20:         while ¬ allTyped(rooms) do
21:             assignAdditionalRules(rooms, rulesList.get(ädditional"))
22:         end while
23:     end if
24:     rulesList.get("needed").add(firstRule)
25:     metaDateForFrontDoor(rooms, rulesList)
26:     return connectRooms(rooms, rulesList)
27: end function
```

Dieser Algorithmus besitzt eine Hilfsmethode in der Zeile 17. In der Zeile 21 wird dieselbe Methode verwendet, jedoch nur für optionale Regeln. Die Methode **assignRule** wird im Anhang im Algorithmus 11 beschrieben.

Im Korrekturprozess stehen eine Map mit erfüllten Regeln und die fehlgeschlagene Regel zur Verfügung. Die Regeln werden von hinten verarbeitet, d. h. zuerst werden die zuletzt erfüllten Regeln verarbeitet und korrigiert. Der Raumtyp, der zu der untersuchten Regel gehört, wird aus

dem Raum entfernt, wodurch die Regel nicht mehr erfüllt wird. Dieser Raum wird nicht mehr als Kandidat für die Erfüllung der Regel berücksichtigt. Bei allen anderen zu untersuchenden Räumen wird ein Typisierungsprozess durchgeführt. Zuerst werden Räume überprüft, die noch keine Typen haben. Falls ein Raum ohne Type einer untersuchten Regel entspricht, wird dieser Regel zugewiesen und die fehlerhafte Regel wird überprüft, ob nun eine erfolgreiche Zuweisung möglich ist. Wenn dies der Fall ist, ist die Korrektur abgeschlossen. Andernfalls werden weitere Regeln verarbeitet. Falls kein Raum gefunden wird, der zu der untersuchten Regel passt, werden bereits typisierte Räume untersucht. Wenn ein bereits typisierter Raum einen anderen Typen bekommt, wird der Typ dieses Raums verändert. Anschließend wird in der Liste der bereits erfüllten Regeln eine Regel gefunden, die den alten Typen des Raums im Attribut *Type* hat. Diese Regel wird erneut versucht zu erfüllen. Das Ziel dieses Vorgangs ist, die bereits erfüllten Regeln auch nach den Änderungen erfüllt zu halten. Andernfalls würden nicht alle Regeln erfüllt werden, und der Korrekturprozess könnte unendlich fortgesetzt werden. Sobald die untersuchte Regel erneut erfüllt ist und ein neuer Raum für diese Regel gefunden wurde, wird überprüft, ob die ursprünglich fehlgeschlagene Regel nun erfüllt werden kann. Falls ja, wird die Untersuchung abgebrochen. Andernfalls wird die nächste Regel untersucht und neu erfüllt. Der Prozess läuft so lange, bis die Regel, die den Korrekturprozess ausgelöst hat, und alle anderen Regeln erfüllt sind. Wenn alle bereits erfüllten Regeln erneut verarbeitet wurden und die Regel, die den Korrekturprozess ausgelöst hat, immer noch nicht erfüllt ist, wird ein Fehler zurückgegeben, der signalisiert, dass für diese Konfiguration und Position der Räume keine Typisierung möglich ist. Um dieses Problem zu lösen, muss die Regelkonfigurationsdatei verändert werden. Das Ziel dieses Prozesses ist es, die Typenvergabe so lange zu korrigieren, bis alle bereits erfüllten Regeln und die fehlerhafte Regel erfüllt sind. Der Korrekturprozess wird nur für obligatorische Regeln durchgeführt. Optionale Regeln müssen nicht korrigiert werden, da sie nur bei Bedarf hinzugefügt werden. Der Typisierungsprozess wird trotzdem abgeschlossen, auch wenn nicht alle optionalen Regeln erfüllt sind, aber alle Räume Typen erhalten haben. Weiter ist ein detaillierter Pseudocode für den Korrekturprozess unter Algorithmus 3 zu finden.

Algorithm 3 fixTyping

```
1:  procedure FIXTYPING(ruleRoomMap, failedRule)
2:      usedRules ← an empty linked list of TypeRule
3:      nextTry ← usedRules.keySet()
4:      for i ← length of usedRules downto 0 do
5:          rule ← usedRules[i]
6:          roomsToCheck ← a copy of allRooms as a list
7:          Remove ruleRoomMap[rule] from allRooms
8:          Set the type of ruleRoomMap[rule] to null
9:          Add ruleRoomMap[rule] back to allRooms
10:         Remove ruleRoomMap[rule] from roomsToCheck
11:         notCheck ← CHECKROOMS(roomsToCheck, rule)
12:         if ¬ notCheck then
13:             allRooms.remove(ruleRoomMap.get(rule))
14:             ruleRoomMap.get(rule).setType(rule.getNeededType())
15:             allRooms.add(ruleRoomMap.get(rule))
16:         else
17:             finalRoomCheck ← newArrayList < Room > (allRooms)
18:             finalCheck ← CHECKROOMS(finalRoomCheck, failedRule)
19:             if finalCheck then
20:                 break
21:             end if
22:         end if
23:     end for
24: end procedure
```

Dieser Pseudocode nutzt die Methode *checkRooms*. Diese Methode prüft, ob die Regel für einen anderen Raum ausgeführt werden kann. Wenn dies der Fall ist, wird dieser Raum typisiert. Der vollständige Pseudocode für diese Methode ist im Anhang unter Algorithmus 12 zu finden.

5.2.3 Raumgraph, Raumverbindungsprozess, Typisierungsoperation

In diesem Abschnitt wird zunächst die Implementierung und die Struktur des Raumgraphen beschrieben. Dieser Graph dient zur Darstellung der Verbindungen zwischen den Räumen und besteht aus Knoten und Kanten. Eine Kante gibt an, ob zwei Knoten miteinander verbunden sind. In dieser Implementierung dienen die bereits typisierten Räume als Knoten, während die

Kanten als eine Map zwischen einem Knoten und einer Liste von anderen Knoten dargestellt sind. Dies ermöglicht, dass ein Raum mit mehr als einem anderen Raum verbunden werden kann.

Für den Aufbau eines solchen Graphen wird ein spezieller Algorithmus entwickelt. Dieser Algorithmus wird nach Abschluss des Typisierungsprozesses und der Zuweisung der Typen an alle Räume verwendet. Als Parameter werden eine Liste aller Räume und eine Map mit allen Regeln benötigt. Für jeden zu untersuchenden Raum wird zuerst ein Knoten erstellt. Anschließend wird die entsprechende Regel für den Raumtyp in der Map der Regeln gesucht, um die mögliche Nachbarschaft des Raums (Attribut *Neighbours*) zu erhalten. Wenn diese Information verfügbar ist, wird die Nachbarschaft des untersuchten Raums analysiert, um die Nachbarräume zu ermitteln. Hierfür werden die gemeinsamen Wände des untersuchten Raums und aller anderen Räume gefunden. Wenn eine gemeinsame Wand existiert, wird dieser Raum als Nachbarraum des untersuchten Raums betrachtet. Wenn der Nachbarraum dem Raumtyp in der Nachbarschaftsliste entspricht, wird auch für diesen Nachbarraum ein Knoten erstellt. Anschließend wird zwischen diesen beiden Knoten eine Kante erstellt, die dem Graphen hinzugefügt wird. Sobald alle Räume untersucht und alle möglichen Verbindungen erstellt sind, ist der Prozess abgeschlossen. Zusätzlich zum Graphen wird am Ende des Prozesses auch eine Adjazenzliste des Graphen erstellt, die den Zusammenhang zwischen jedem Knoten und allen anderen Knoten im Graphen zeigt. Die Adjazenzliste wird für die weitere Modellierung und Metadatenvergabe benötigt. Ein Pseudocode, der diesen Prozess beschreibt, ist in 4 zu finden.

Algorithm 4 connectRooms

```
1: function connectRooms(rooms, rulesList)
2:     edges ← new ArrayList<>()
3:     for r in rooms do
4:         node ← GraphNode(r)
5:         roomType ← r.getType
6:         neededRule ← null
7:         for rule in rulesList.get("needed") do
8:             if rule.getNeededType = roomType then
9:                 neededRule ← rule
10:            end if
11:        end for
12:        for rule in rulesList.get("additional") do
13:            if rule.getNeededType = roomType then
14:                neededRule ← rule
15:            end if
16:        end for
17:        if neededRule != null then
18:            roomNeighbours ← findNeighbour(r, rooms)
19:            for neighbour in roomNeighbours do
20:                neighbourNode ← new GraphNode(neighbour)
21:                for possibleConnection in neededRule.getPossibleNeighbours do
22:                    if neighbour.getType = possibleConnection then
23:                        edge ← new GraphEdge(node, neighbourNode)
24:                        if node.getRoom.getType ≠ neighbourNode.getRoom.getType
   then
25:                            edges.add(edge)
26:                        end if
27:                    end if
28:                end for
29:            end for
30:        end if
31:    end for
32:    return new RoomGraph(edges)
33: end function
```

Sobald der Graph vollständig erstellt ist, können die Metadaten für die Wände hinzugefügt werden. Die Metadaten geben an, ob eine Wand mit einer Tür modelliert werden soll oder ob die Wand überhaupt modelliert werden soll. Dazu wird jeder Knoten aus der Adjazenzliste durchgelaufen. Diese Liste ist in Form einer Map aufgebaut, in der ein Knoten des Graphen als Schlüssel, und eine Liste aller anderen Knoten, die mit diesem Knoten verbunden sind, als Wert eingetragen werden. Es werden die Wände der Räume im untersuchten Knoten analysiert, um gemeinsame Wände zwischen diesen Räumen zu finden. Zwei Wände gelten als gemeinsam, wenn sie dieselbe Orientierung haben (X-Achse oder Z-Achse) und ihre Anfangs- oder Endkoordinaten übereinstimmen. Es kann auch vorkommen, dass eine Wand größer als die andere ist. In diesem Fall wird geprüft, ob die kleinere Wand Teil der größeren Wand ist. Wenn dies der Fall ist, handelt es sich ebenfalls um eine gemeinsame Wand. Für diese Wände werden Metadaten vergeben, die angeben, dass sie ein Loch für eine Tür benötigen. Die Metadaten, die bestimmen, ob eine Wand modelliert werden soll, werden für alle Wände auf den Wert *true* gesetzt. Es kann jedoch Wände geben, die nicht modelliert werden sollen. Zum Beispiel, wenn eine längere Wand und eine kürzere Wand aufeinander positioniert sind und gemeinsame Wände sind und auf der kürzeren Wand eine Tür modelliert wird, soll die längere Wand nicht in 3D dargestellt werden, da sie die kürzere Wand mit der Tür blockieren würde. Diese Metadaten sind später für die 3D-Meshmodellierung erforderlich, damit nur benötigte Wände modelliert werden und die Tür auch nur auf den benötigten Wänden erscheint.

Um sicherzustellen, dass die Raumtypisierung und der Verbindungsaufbau nacheinander ausgeführt werden und in die bestehende Grammatik integriert werden können, wird eine Grammatikoperation erstellt. Sobald die Räume erstellt wurden, wird diese Operation hinzugefügt und sowohl die Typisierung als auch der Verbindungsaufbau werden ausgeführt. Diese Operation kann nur auf ein Objekt der Klasse **FloorShape** angewendet werden. Bei der Ausführung dieser Operation wird zuerst die Konfigurationsdatei gelesen und die Regeln werden erstellt. Anschließend wird der iterative Typisierungsprozess gestartet und die typisierten Räume werden in einen Graphen eingefügt und entsprechend verbunden.

5.3 Möbelpositionierung

In diesem Teil werden die Algorithmen und Prozesse erklärt, die für die Positionierung der Möbel im Raum benötigt werden.

5.3.1 Regelauswertung bei der Möbelpositionierung

Genau wie der Typisierungsprozess ist auch der Prozess der Möbelpositionierung regelbasiert. Die Regeln, die definiert werden, beschreiben die Position von Möbeln im Raum und werden in einer Konfigurationsdatei festgelegt. Wie zuvor handelt es sich dabei um eine CSV-Datei. In der Abbildung 6.13 ist ein Beispiel einer Konfigurationsdatei mit Regeln dargestellt.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17
1	Room	Name	isInCorner	isAlongWall	hasPaar	PaarWith	minPaarDistance	maxPaarDistance	needInitTurn	clearanceSide	clearanceFront	clearanceBack	canCloseWindow	canBeAround	nonCollision	isDetail	isOnTheSide
2	Livingroom	window	false	false	false	<null>	0	0	false	0	0	0	false	false	false	true	false
3	Livingroom	door	false	false	false	<null>	0	0	false	0	0	0	false	false	false	true	false
4	Livingroom	tableBig	false	true	false	<null>	0	0	false	0.3	0.3	0.3	false	false	false	false	false
5	Livingroom	chair	false	false	true	tableBig	0	0	false	0.0	0	0.0	false	true	false	false	false
6	Livingroom	chair1	false	false	true	tableBig	0	0	false	0.0	0.0	0.1	false	true	false	false	false
7	Livingroom	chair2	false	false	true	tableBig	0	0	false	0.0	0.0	0.1	false	true	false	false	false
8	Livingroom	Sofa	false	true	false	<null>	0	0	false	0.1	0.1	0.0	false	false	false	false	false
9	Livingroom	tvboard	false	true	true	Sofa	0.4	0.5	false	0.1	0.0	0.0	false	false	false	false	false
10	Livingroom	FloorLamp	true	false	false	<null>	0	0	false	0.0	0.0	0.0	false	false	false	false	false
11	Livingroom	bookshelf	false	true	false	<null>	0	0	false	0.1	0.0	0.0	true	false	false	false	false
12	Livingroom	bookshelf1	false	true	false	<null>	0	0	false	0.1	0.0	0.0	true	false	false	false	false
13	Livingroom	SofaTable	false	false	true	Sofa	0.2	0.4	false	0.1	0	0	false	false	false	false	false

Abbildung 5.2: Konfigdatei mit Regeln für die Möbelplatzierung

Wenn die Konfigurationsdatei mit Regeln befüllt ist, soll sie gelesen werden. Der Parsing-Prozess erfolgt mittels einer einfachen String *split*-Operation, durch die die einzelnen Informationen aus der Datei extrahiert werden. Diese Informationen werden dann einer Regel zugeordnet.

Um eine Regel zu erstellen, wird eine Entitätsklasse namens **FurnitureRule** erstellt. Die Elemente, die in der ersten Zeile der Konfigurationsdatei aufgeführt sind, werden als Attribute dieser Klasse verwendet. Der *Room* gibt den Raumtyp an, in dem sich das Möbelstück befinden soll. Dieser Typ entspricht dem Enum, das zur Typisierung der Räume erstellt wurde. Das Attribut *Name* ist der Name des Objekts und wird benötigt, um das gleiche Möbelstück mehrmals im Raum hinzuzufügen. So können beispielsweise mehrere Stühle im Raum platziert werden, indem die Namen wie folgt definiert werden: **chair**, **chair1**, **chair2**. Der Name sollte auch mit dem Namen der Datei übereinstimmen, in der die Informationen zum 3D-Modell des Möbelstücks gespeichert sind. Andernfalls kann das Möbelstück nicht in 3D dargestellt werden. Die Elemente *isInCorner* und *isAlongWall* beschreiben die Position des Möbelstücks im Raum, ob es sich in der Ecke oder entlang einer Wand befinden soll. Wenn das Element *hasPaar* gesetzt ist, wird das Möbelstück als Kind eines anderen Möbelstücks bezeichnet. Das Kindermöbelstück wird einem Elternmöbelstück zugeordnet. Der Name des Elternmöbelstücks wird im Element *paarWith* definiert. Wenn ein Möbelstück ein Elternmöbelstück hat, kann es vor dem Elternmöbelstück, neben dem Elternmöbelstück oder um das Elternmöbelstück herum platziert werden. Die nächsten Attribute in der Konfigurationsdatei, *minPaarDistance* und *maxPaarDistance*, beschreiben die minimale und maximale Distanz zwischen einem Elternmöbelstück und einem Kindermöbelstück. Die Algorithmen zur Platzierung von Möbel-

stücken in der Ecke, entlang der Wand und zur Platzierung von Kindermöbelstücken werden in weiteren Abschnitten beschrieben. Die nächsten drei Elemente, *clearanceSide*, *clearanceFront* und *clearanceBack*, beschreiben den freien Abstand, den ein Möbelstück in die entsprechende Richtung haben soll. Das Element *canCloseWindow* beschreibt, ob ein Möbelstück ein Fenster blockieren kann. Dieses Attribut ist für große Objekte wie Schränke sinnvoll, die nicht direkt vor dem Fenster platziert werden sollten, damit das Fenster nicht blockiert wird. Das Attribut *canBeAround* bezieht sich auf Objekte, die in Paaren stehen. Es gibt an, ob ein Kindermöbelstück um das Elternmöbelstück herum platziert werden soll. Mit dem Attribut *nonCollision* können die Möbelstücke gekennzeichnet werden, die nicht mit anderen Möbelstücken kollidieren können, wie z. B. Teppiche. Das Element *isDetail* bezieht sich auf die Objekte im Raum, die zwar keine Möbel sind, aber dennoch platziert werden müssen. Dieses Element wird beispielsweise für Türen und Fenster im Gebäude verwendet. Das letzte Element *isOnTheSide* sollte nur bei Möbeln mit Paaren verwendet werden. Es gibt an, dass das Möbelstück direkt neben dem Elternmöbelstück platziert werden soll. So können beispielsweise Nachttische neben dem Bett platziert werden.

Sobald die Regeln erstellt und ausgewertet sind, werden sie in einem iterativen Algorithmus abgearbeitet. Zunächst werden jedoch die benötigten 3D-Modelle der Möbel ermittelt. Nur die Modelle, die in den Regelnamen erwähnt wurden, werden für die Platzierung benötigt, die werden aus bereitgestellten *.obj*-Dateien gelesen, als Dreiecksmeshes dargestellt und in eine Map gepackt. Anschließend werden die Ecken aller Räume gefunden und in separate Listen gespeichert. Es werden auch einige Hilfsvariablen erstellt, die bei der Ausführung des Prozesses benötigt werden. Insgesamt kann die Vorbereitung der Ausführung in folgenden Pseudocode zusammengefasst werden:

Algorithm 5 placeFurniture

```
1: function placeFurniture(rules)
2:     mapOfObjects ← readObjectsDir(<PATH-TO-MODELS>, rules)
3:     placedObjects ← []
4:     placedSize ← 0
5:     floorRooms ← shape.getRooms()
6:     mapOfRoomVertices ← fillMapWithCorners(floorRooms, mapOfRoomVertices)
7:     neededObjects ← getNeededObjects(rules, mapOfObjects, mapOfRoomVertices)
8:     detailObjects ← []
9:     allObjects ← []
```

Wenn diese Information verfügbar ist, werden den benötigten Möbeln zufällige Positionen im Raum zugewiesen. Sobald alle Möbel im Raum zufällige Positionen haben, werden diese Positionen mithilfe des Separation Steering Algorithmus optimiert. Dafür werden nur zwei Phasen dieses Algorithmus benötigt: der Zusammenhalt und die Trennung. Für beide Phasen werden separat Vektoren ermittelt, die zeigen, wie nah oder weit auseinander die Möbel verschoben werden sollen. Sobald diese Vektoren berechnet sind, wird die Position der entsprechenden Möbel auf die Summe dieser beiden Vektoren verschoben. Die Optimierung der Position erfolgt in einer Schleife mit einer benutzerdefinierten Anzahl von Durchläufen. In dieser Schleife wird wiederum durch alle Möbel iteriert und die Möbel werden entsprechend den berechneten Vektoren verschoben. Die Möbel, die laut einer Regel in einer Ecke stehen sollen, nehmen jedoch generell nicht am Optimierungsprozess teil. Die Algorithmen, die für die zufällige Platzierung und Optimierung zuständig sind, sind im Anhang unter den Algorithmen 14 und 15 als Pseudocode dargestellt. Bei diesen Prozessen und auch bei der späteren Bewegung von Möbeln anhand der Informationen aus den Regeln werden die Möbel nur in X- oder Z-Richtung bewegt.

Algorithm 6 placeFurniture II

```
10:  for r ∈ rules do
11:      placedSize ← placedObjects.size()
12:      neededFurniture ← neededObjects.get(r.getName())
13:      if r.isDetailObject() then
14:          detailObjects.add(neededFurniture) and continue
15:      end if
16:      neededFurniture.setNoCollision(r.isCreatesNoCollision())
17:      neededWalls ← getNeededRoom(floorRooms, r)
18:      corners ← mapOfRoomVertices.get(r.getRoomType())
19:      if r.isCreatesNoCollision() then
20:          center ← FurniturePlacementUtils.getRoomCenter(corners)
21:          transformFurniture(neededFurniture, center)
22:      end if
```

Bevor die Positionen der Möbel nach den Regeln festgestellt werden, wird geprüft, ob es sich um ein Detail-Objekt oder um ein Objekt ohne Kollisionen handelt. Zusätzlich werden hier die Variablen für benötigte Wände und Ecken des Raums befüllt (Zeilen 17-18). Der Anfang des iterativen Prozesses ist im Algorithmus 6 als Pseudocode dargestellt.

Algorithm 7 placeFurniture III

```
23:         if r.isPaar() then
24:             if r.getPaarWith().isEmpty() then
25:                 print("Pair rule with Parent Furniture")
26:                 placeFurnitureInCornerOrAllongWall(rules, placedObjects, r, neededFurniture, neededWalls, boundaries, corners)
27:             end if
28:             parent ← canPairBePlaced(r, placedObjects)
29:             if parent ≠ null then
30:                 placeNotParentFurniture(rules, placedObjects, r, neededFurniture, parent)
31:             else
32:                 continue
33:             end if
34:         else
35:             placeFurnitureInCornerOrAllongWall(rules, placedObjects, r, neededFurniture, neededWalls, boundaries, corners)
36:         end if
37:         if placedObjects.size() == placedSize + 1 then
38:             updateExtent(neededFurniture, r)
39:             r.setDone(true)
40:         else
41:             print("Could not place "+ neededFurniture.getName())
42:         end if
43:     end for
44:     placedObjects.addAll(addDoorAndWindows(detailObjects, floorRooms, wallWidth, windowWidth, windowHeight, doorWidth, doorHeight))
45:     return placedObjects
46: end function
```

Dieser Pseudocode enthält Hilfsmethoden, die Teilprozesse wie die Platzierung von Eltern- und Kindermöbelstücken steuern. Für diese Algorithmen wurde aus Gründen der Übersichtlichkeit kein Pseudocode erstellt. In weiteren Abschnitten werden diese Algorithmen und ihr Ablauf detailliert erklärt. Im iterativen Prozess werden alle Regeln abgearbeitet, sie müssen jedoch nicht zwingend alle erfüllt werden, damit der Algorithmus abgeschlossen werden kann. Wenn bei einer beliebigen Regel ein Problem auftritt und die Möbel wegen Platzmangels oder

Abstandsproblemen nicht platziert werden können, wird diese Information an den Benutzer über die Konsole ausgegeben, damit er entsprechend reagieren kann.

5.3.2 Platzierung des Elternmöbelstücks und Abstandflächeberechnung

Wenn die Möbelpositionen randomisiert und optimiert sind, werden sie auch entsprechend den Informationen aus jeder Regel angepasst. Dafür wird der iterative Prozess gestartet, der durch jede Regel separat läuft und versucht, sie zu erfüllen. Zuerst wird festgestellt, ob ein Objekt ein Paar hat. Wenn dies der Fall ist, wird geprüft, ob das Attribut *paarWith* nicht leer ist. Ist es leer, handelt es sich um ein Elternmöbelstück und das Möbelstück wird als Elternteil behandelt.

Um Möbel erfolgreich platzieren zu können, dürfen keine Kollisionen mit anderen Möbeln auftreten und die Abstandsflächen müssen gewährleistet sein. Diese Prüfung der Abstandsfläche erfolgt immer dann, wenn ein Möbelstück positioniert wurde, ohne dass eine Kollision aufgetreten ist. Sie dient dazu sicherzustellen, dass ausreichend Platz um die Möbel herum vorhanden ist, um sie erfolgreich nutzen zu können. Um die Abstandsflächen zu berechnen, werden die Regelelemente *clearanceBack*, *clearanceSide* und *clearanceFront* benötigt. Die Prüfung der Abstandsflächen erfolgt in einer Schleife, die Anzahl der benötigten Versuche kann vom Benutzer festgelegt werden. Zunächst werden vier Klone des Möbelstücks erstellt, mit denen die Prüfung durchgeführt wird, damit die tatsächliche Möbelposition zunächst beibehalten werden kann.

Um die Abstandsflächen zu prüfen, wird der Klon in die entsprechende Richtung verschoben. Die Entfernung dieser Verschiebung ist in dem entsprechenden Element der Regel angegeben. Wenn beispielsweise überprüft werden soll, ob vor dem Möbelstück ausreichend Platz frei ist, wird der Klon nach vorne verschoben. Die Bewegung erfolgt mit der fest definierten Schrittgröße, bis eine Kollision mit anderen Möbeln auftritt oder die vorgegebene Entfernung der Bewegung erreicht ist. Wenn keine Kollision auftritt, ist die Abstandsfläche gewährleistet. Im Falle einer Kollision wird versucht, die Position der Möbel abhängig von den Regeln anzupassen, um die Abstandsfläche zu gewährleisten. Wenn das betrachtete Möbelstück nur in einer Ecke platziert werden darf, wird das Möbelstück, das die Kollision verursacht hat, Schritt von dem betrachteten Möbelstück wegbewegt. Wenn die Bewegung von dem anderen Möbelstück nicht möglich ist, wird versucht, das betrachtete Möbelstück in eine andere Ecke zu bewegen. Wenn das andere Möbelstück bewegt wurde, wird der Prozess für dieses Möbelstück wiederholt, um die Abstandsfläche von diesem Möbelstück noch mal zu prüfen. Wenn das betrachtete Möbelstück entlang einer Wand platziert werden muss, wird es entlang der Wand verschoben, bis die Abstandsfläche passt. Nachdem ein Möbelstück verschoben wurde, wird die Prüfung erneut

durchgeführt, um zu prüfen, ob die Abstandsfläche gewährleistet ist. Die gleiche Prüfung wird auch für die hintere, linke und rechte Seite des Möbelstücks durchgeführt, um sicherzustellen, dass alle Abstandsflächen von allen Seiten gewährleistet sind. Wenn die Abstandsflächen nicht gewährleistet werden können und die Anzahl der Versuche erreicht ist, kann das Möbelstück nicht platziert werden. In diesem Fall wird dem Benutzer eine entsprechende Meldung auf der Konsole ausgegeben.

Es gibt noch zwei Sonderfälle, bei denen die Position der Möbel optimiert wird. Der erste Sonderfall ist, wenn das Möbelstück neben einer Tür platziert werden soll. Um den Abstand zwischen dem Möbelstück und der Tür zu berechnen, muss zuerst die Position der Tür gefunden werden. Dafür werden die Koordinaten der Ecken der Türöffnung als Metadaten der Wand übermittelt. Diese Koordinaten werden in Form einer Map von dem Lochtyp (Tür oder Fenster) und einer Liste von Eckkoordinaten dargestellt. Anhand dieser Koordinaten wird das Zentrum der Tür oder des Fensters berechnet. Danach wird der Abstand zwischen dem Zentrum des Möbelstücks und der Tür oder dem Fenster berechnet. Wenn dieser Wert kleiner als der vom Benutzer definierte Abstand ist, wird das Möbelstück weiter von der Tür oder dem Fenster entfernt positioniert. Falls das Möbelstück nur entlang der Wand positioniert werden soll, wird es auch nur entlang der Wand verschoben. Wenn es nicht möglich ist, genügend Abstand zur Tür einzuhalten, wird das Möbelstück entweder gar nicht platziert oder an eine andere Wand verschoben. Gleiches gilt für den zweiten Sonderfall, bei dem das Möbelstück ein Fenster überdecken kann. Dies wird durch das Regelelement *canCloseWindow* beschrieben. Wenn dieses Element den Wert *true* hat und das Möbelstück nicht genügend Abstand zum Fenster hat, wird es so verschoben, dass der Abstand gewährleistet ist.

Für ein Elternmöbelstück oder ein Möbelstück ohne Paar werden zwei weitere Attribute geprüft, die die Position der Möbel im Raum beschreiben: *isInCorner* und *isAlongWall*. Wenn ein Möbelstück in einer Ecke platziert werden soll, wird zuerst versucht, es in die erste Ecke zu platzieren. Falls das nicht klappt, wird das Möbelstück um 90 Grad um die y-Achse gedreht und der Prozess wird wiederholt. Wenn auch das nicht funktioniert, wird die nächste Ecke ausprobiert und der Vorgang beginnt von vorne.

Um das Möbelstück genau in der Ecke zu platzieren, muss die richtige Position berechnet werden. Dafür werden das minimal umgebende Rechteck des Möbelstücks und die genauen Koordinaten der Ecke des Raums benötigt. Das minimal umgebende Rechteck hilft dabei, die Größe des Objekts zu berechnen, sodass das Zentrum des Möbelstücks so verschoben werden kann, dass keine Teile des Objekts außerhalb des Raums stehen. Um die Position zu berechnen, werden auch die Wände benötigt, die die betrachtete Ecke gemeinsam haben. Es wird geprüft, in welche Richtung jede Wand orientiert ist, und die Position wird mithilfe

dieser Richtungen und Möbelgröße entsprechend ermittelt. Als Basis dienen die Koordinaten der Raumecke, die dann entsprechend den Wandrichtungen angepasst werden. Nach der Anpassung wird das Zentrum des Möbelstücks zu diesen Koordinaten verschoben. Wenn eine Ecke ohne Kollisionen gefunden wurde, wird überprüft, ob der Abstand zwischen dem Möbelstück und anderen Objekten gewährleistet ist, dann wird das Möbelstück erfolgreich platziert. Wenn das Möbelstück in die freie Ecke platziert wird, wird erneut überprüft, ob es richtig ausgerichtet ist und der Richtungsvektor des Möbelstücks nicht in die Richtung der Wand orientiert ist. Dazu wird eine Kopie des Möbelstücks in Richtung des Richtungsvektors bewegt. Wenn das nicht möglich ist, weil das Möbelstück sonst außerhalb des Raums platziert wäre, wird es um 180 Grad um die Y-Achse gedreht.

Wenn die Möbel laut Regel nur entlang der Wand stehen sollen, muss ein anderer Ablauf durchgeführt werden. Zunächst muss die nächstgelegene Wand in Bezug auf die aktuelle Position des Möbelstücks ermittelt werden. Dazu wird eine Gerade zwischen dem Zentrum des Möbelstücks und jeder Wand gezogen und die kürzeste Gerade ermittelt, die das Zentrum mit der nächsten Wand verbindet. Hierfür wird in der Implementierung die Klasse *Line* aus JMonkeyEngine genutzt. Sobald die Wand gefunden wurde, wird das Möbelstück zu dieser Wand bewegt. Zunächst wird bestimmt, in welche Richtung das Möbelstück bewegt werden soll. Hierfür wird geprüft, ob der Richtungsvektor des Möbelstücks in die gleiche Richtung wie die Wand ausgerichtet ist. Ist dies der Fall, wird die Bewegungsrichtung um 90 Grad gedreht, basierend auf dem Richtungsvektor des Möbelstücks. Zusätzlich dazu wird das Möbelstück um 90 Grad gedreht, damit es entsprechend ausgerichtet ist. Anschließend wird das Möbelstück in Richtung der Wand bewegt. Wenn die Bewegung nicht mehr möglich ist, weil die Grenze des Raums erreicht wurde, wird das Möbelstück neben der Wand positioniert. Ist diese Position bereits besetzt, wird versucht, das Möbelstück entlang der Wand weiterzubewegen, bis ein freier Platz gefunden wurde. Wenn entlang der gesamten Wand kein Platz mehr vorhanden ist, wird die Wand als besetzt markiert und der gleiche Vorgang wird für die zweite nächstgelegene Wand wiederholt. Sobald ein freier Platz entlang der Wand gefunden wurde, wird geprüft, ob die vorgeschriebene Abstandsfläche eingehalten wurde. Ist dies der Fall, ist die Regel erfüllt. Der Prozess der Positionierung des Elternmöbelstücks wird im Algorithmus 7 in den Zeilen 26 und 35 aufgerufen.

5.3.3 Platzierung des Kindermöbelstücks

Die oben beschriebenen Algorithmen werden zur Platzierung des Elternmöbelstücks verwendet. Kindermöbelstücke haben jedoch in der Regel noch weitere Eigenschaften, die berücksichtigt werden müssen. Um sicherzustellen, dass es sich um ein Kindermöbelstück handelt, werden

im Platzierungsprozess zwei Elemente überprüft: „hasPaar“ und „PaarWith“. Wenn das erste Element den Wert „true“ hat und das zweite Element nicht leer ist, handelt es sich um ein Kindermöbelstück, das mit einem Elternmöbelstück mit dem Namen aus dem zweiten Element verbunden ist.

Vor dem Platzieren des Kindermöbelstücks, wird überprüft, ob bereits ein entsprechendes Elternmöbelstück erfolgreich platziert wurde. Das Kindermöbelstück wird zurück zum Nullpunkt des Koordinatensystems bewegt, um die Position des Zentrums des Kindermöbelstücks einfacher bestimmen zu können. Das Kindermöbelstück wird dann mit einer Transformation an die richtige Stelle platziert. Anschließend werden weitere Informationen aus den Regeln abgerufen, die die genaue Position des Kindermöbelstücks beschreiben.

Wenn das Kindermöbelstück um das Elternmöbelstück herum platziert werden soll, müssen zunächst die freien Seiten des Elternmöbelstücks gefunden werden. Dazu werden Klone des Elternmöbelstücks in verschiedene Richtungen bewegt. Wenn die Bewegung nicht möglich ist, weil das Möbelstück außerhalb des Raums wäre, wird diese Seite als nicht frei markiert. Alle anderen Seiten können für die Platzierung des Kindermöbelstücks verwendet werden.

Sobald die freien Seiten gefunden wurden, wird über diese Seiten iteriert und versucht, das Kindermöbelstück zu platzieren. Die Position des Zentrums des Kindermöbelstücks hängt von der Größe, Richtung der freien Seite des Elternmöbelstücks und der Position des Elternmöbelstücks ab. Die Größe des Kindermöbelstücks wird wieder mithilfe des minimal umgebenden Rechtecks bestimmt.

Wenn die freie Seite in Richtung der Z-Achse orientiert ist, werden die X- und Y-Koordinaten des Kindermöbelstücks auf dieselben Koordinaten wie beim Elternmöbelstück gesetzt und die Z-Koordinate für das Kindermöbelstück wird um die Größe des Elternmöbelstücks in der Z-Richtung erweitert. Der entsprechende Vorgang wird auch bei einer zur X-Achse orientierten Seite durchgeführt.

Nachdem das Kindermöbelstück platziert wurde, wird überprüft, ob das Elternmöbelstück mehr als ein Kindermöbelstück auf dieser Seite haben kann. Dazu werden die Größen des Elternmöbelstücks in X- und Z-Richtungen verglichen. Wenn mehr als ein Kindermöbelstück hineinpasst, wird das neue Zentrum des Kindermöbelstücks nach links oder rechts in die entsprechende Richtung verschoben.

Wenn die Koordinate für das Zentrum des Kindermöbelstücks bestimmt ist, muss auch die Ausrichtung des Möbelstücks angepasst werden. Die Anforderung ist, dass das Kindermöbelstück in Richtung des Elternmöbelstücks ausgerichtet sein soll. Um dies zu berechnen, werden drei Werte benötigt: die Richtungsvektoren des Kinder- und Elternmöbelstücks sowie die Orientierung der betrachteten freien Seite des Elternmöbelstücks. Wenn der Richtungsvektor

des Elternmöbelstücks und die Orientierung der freien Seite gleich sind, wird geprüft, ob das Kindermöbelstück um 180 Grad gedreht werden muss. Es wird nur dann gedreht, wenn der Richtungsvektor des Elternmöbelstücks entgegengesetzt zur Richtung des Kindermöbelstücks orientiert ist. Wenn der Richtungsvektor des Elternmöbelstücks und die Orientierung der freien Seite unterschiedlich sind, soll das Kindermöbelstück nur um 90 Grad gedreht werden. Diese Drehung hängt allein von der Orientierung der freien Seite und dem Richtungsvektor des Elternmöbelstücks ab. Der Algorithmus für die Drehung des Kindermöbelstücks ist im Anhang 13 als Pseudocode zu sehen.

Eine weitere Möglichkeit, wo sich das Kindermöbelstück befinden kann, ist an der Seite des Elternmöbelstücks. Dafür wird das Attribut *isOnTheSide* im entsprechenden Regel auf den Wert „true“ gesetzt. Wie im ersten Fall wird das Kindermöbelstück zum Nullpunkt des Koordinatensystems verschoben. Danach wird versucht, das Kindermöbelstück richtig zu rotieren, damit es in die gleiche Richtung wie das Elternmöbelstück orientiert ist. Dazu wird ein Winkel zwischen den Richtungsvektoren des Eltern- und des Kindermöbelstücks berechnet, und abhängig von diesem Winkel wird das Kindermöbelstück gedreht. Sobald das Kindermöbelstück richtig ausgerichtet ist, wird es zum Zentrum des Elternmöbelstücks platziert.

Im nächsten Schritt werden der Bewegungsvektor und die Grenzen der Bewegung berechnet. Die Bewegungsrichtung ist senkrecht zum Richtungsvektor orientiert, damit das Kindermöbelstück an der Seite des Elternmöbelstücks positioniert werden kann. Um die Bewegungsrichtung zu ermitteln, wird der Richtungsvektor 90 Grad um die Y-Achse gedreht. Die Grenzen der Bewegung sind notwendig, damit das Kindermöbelstück nicht zu weit vom Elternmöbelstück entfernt wird und am Ende genau neben dem Elternmöbelstück steht. Zunächst wird das Kindermöbelstück entlang des Richtungsvektors bewegt, bis die obere Grenze des Elternmöbelstücks erreicht ist. Um die obere Grenze zu bestimmen, wird das minimal umgebende Rechteck des Elternmöbelstücks verwendet. Bei einer Bewegung in positiver Z-Richtung ergibt sich die obere Grenze als Summe der Z-Koordinate des Zentrums des Elternmöbelstücks und der Hälfte des minimalen umgebenden Rechtecks in Z-Richtung. Wenn die Grenze erreicht wird, wird die Bewegung in einer Schleife ausgeführt, und das Kindermöbelstück bewegt sich in die Richtung des berechneten Bewegungsvektors. Sobald das Kindermöbelstück keine Kollision mit dem Elternmöbelstück verursacht, ist es an der Seite des Elternmöbelstücks positioniert. Falls eine Seite des Elternmöbelstücks bereits besetzt ist, wird das Kindermöbelstück auf die andere Seite des Elternmöbelstücks bewegt. Wenn auch dort kein Platz ist, wird dieses Möbelstück nicht platziert und die entsprechende Fehlermeldung wird auf der Konsole ausgegeben.

Wenn in der Regel nicht spezifiziert ist, wo das Kindermöbelstück in Bezug auf das Elternmöbelstück platziert werden soll, wird es gegenüber dem Elternmöbelstück platziert, wenn die

Werte von *isOnTheSide* und *canBeAround* auf „false“ gesetzt sind. Wie bei anderen Vorgängen wird zunächst die Orientierung des Kindermöbelstücks an die Orientierung des Elternmöbelstücks angepasst. Wenn die Orientierung gesetzt ist und das Kindermöbelstück richtig ausgerichtet ist, wird geprüft, ob das Kindermöbelstück entlang einer Wand platziert werden soll. Falls das Kindermöbelstück entlang der Wand platziert werden soll, wird es zunächst auf die Position des Elternmöbelstücks verschoben. Anschließend wird das Kindermöbelstück entlang des Richtungsvektors verschoben, bis es neben der Wand platziert ist. Falls das Kindermöbelstück auf dem Weg eine Kollision mit anderen Möbeln hat, wird es senkrecht zum Richtungsvektor verschoben, bis die Kollision behoben ist, und dann weiter in die Richtung des Richtungsvektors zur Wand geschoben. Wenn das Kindermöbelstück ohne Kollision neben der Wand platziert wird, wird die Abstandsfläche überprüft. Wenn die Abstandsfläche eingehalten wird, ist das Kindermöbelstück entlang der Wand platziert.

Falls das Kindermöbelstück nicht entlang der Wand platziert werden soll, wird es einfach gegenüber dem Elternmöbelstück platziert. Dabei müssen die minimalen und maximalen Abstände zwischen dem Elternteil und dem Kind berücksichtigt werden. Diese können aus den Regeln in den Elementen *minPaarDistance* und *maxPaarDistance* abgelesen werden. Wenn das Kindermöbelstück zu nah am Elternmöbelstück ist, wird es in kleinen Schritten weiter vom Elternmöbelstück entfernt, bis der in den Regeln definierte Abstand erreicht ist. Genauso verfährt man, wenn das Kindermöbelstück zu weit vom Elternmöbelstück entfernt ist. Falls es bei der Bewegung eine Kollision mit anderen Möbeln gibt, wird das Kindermöbelstück wieder senkrecht zum Bewegungsvektor verschoben, bis die Kollision behoben ist. Am Ende wird auch hier die Abstandsfläche überprüft, und wenn sie eingehalten wird, wird das Kindermöbelstück platziert.

Wenn ein Kindermöbelstück am Ende nicht gegenüber eines Elternmöbelstücks steht, sollte laut Konzept versucht werden das Elternmöbelstück so zu bewegen, damit es dem Kindermöbelstück gegenübersteht. Zunächst wird die Bewegungsrichtung des Elternmöbelstücks berechnet, indem der Richtungsvektor des Elternmöbelstücks um 90 Grad entlang der Y-Achse gedreht wird. Abhängig von der Position des Kindermöbelstücks wird die Koordinate bestimmt, an der das Elternmöbelstück bewegt werden soll. Dies ist entweder die X- oder die Z-Koordinate des Zentrums des Kindermöbelstücks. Sobald diese Koordinate festgelegt ist, wird das Elternmöbelstück in Richtung der Bewegung verschoben, bis die X- oder Z-Koordinate des Zentrums des Elternmöbelstücks mit der berechneten Koordinate übereinstimmt. Wenn das Elternmöbelstück zu diesem Zeitpunkt keine Kollisionen aufweist und der Abstand zwischen den Möbeln gewährleistet ist, wird das Elternmöbelstück verschoben. Wenn es jedoch aufgrund von Kollisionen oder Platzmangel nicht möglich ist, das Elternmöbelstück zu platzieren, wird

keine Verschiebung durchgeführt. Der Prozess der Kindermöbelplatzierung wird in Zeile 32 des Algorithmus 7 aufgerufen.

5.3.4 Möbelplatzierungsoperation

Damit alle oben beschriebenen Algorithmen ausgeführt werden können und die Möbelplatzierung stattfindet, wird in der bestehenden Grammatik eine neue Operation entwickelt, die den Möbelplatzierungsprozess startet und am Ende die fertige Etage mit Möbeln an den Meshgenerator übergibt. Genau wie beim Typisierungsprozess kann die Operation nur auf das Objekt der Klasse **FloorShape** angewendet werden. Bei der Ausführung dieser Operation wird zuerst die Konfigurationsdatei gelesen und die Regeln werden erstellt. Danach wird das Objekt der Klasse **PlacementProcess** erstellt und die Methode **placeFurniture** ausgeführt. Die Methode erhält das FloorShape-Objekt, die Größen von Fenstern und Türen sowie alle anderen benötigten Parameter (Anzahl der Optimierungsschritte, Anzahl der Schritte bei der Abstandsflächenprüfung), die der Benutzer in die Grammatik hinzugefügt hat. Alle benötigten Parameter zusammen mit dem Pfad zur Konfigurationsdatei müssen vom Benutzer beim Aufrufen der Operation als Parameter angegeben werden. Ein Beispiel für eine solche Grammatik wird im Kapitel 6 präsentiert.

6 Auswertung

In diesem Kapitel wird die Auswertung der entworfenen Software gezeigt. Für jeden Teil der Implementierung, Raumpartitionierung, Raumtypisierung und Möbelplatzierung. Zusätzlich gibt es zu jeder Etappe ein Teilkapitel, welches die detaillierte Auswertung der Etappe beschreibt.

6.1 Auswertung der Raumpartitionierungsmethode mit dem BSP-Algorithmus

In Abbildung 6.1 ist eine Grammatik zu sehen, die eine Raumgenerierungsoperation beinhaltet und für die Evaluation verwendet wird. Zunächst wird mit der Operation **repeated_split** das Gebäude in Etagen unterteilt (Anforderung 1.1). Danach wird auf jeder Etage eine Operation **make_floor** angewendet (Anforderung 1.12). Diese ist auf das Element BaseFloorFacade angewendet, das eine Etage des Gebäudes repräsentiert (Anforderung 1.13). Die Operation führt die Erstellung der Räume mit dem BSP-Algorithmus aus. Als Parameter erhält sie eine Anzahl von Teilungen (split) (Anforderung 1.16) und die Breite der Wände (wallWidth) (Anforderung 1.9). Als Ergebnis dieser Modellierung erhält man ein 3D-Objekt, wie in der Abbildung 6.2 gezeigt.

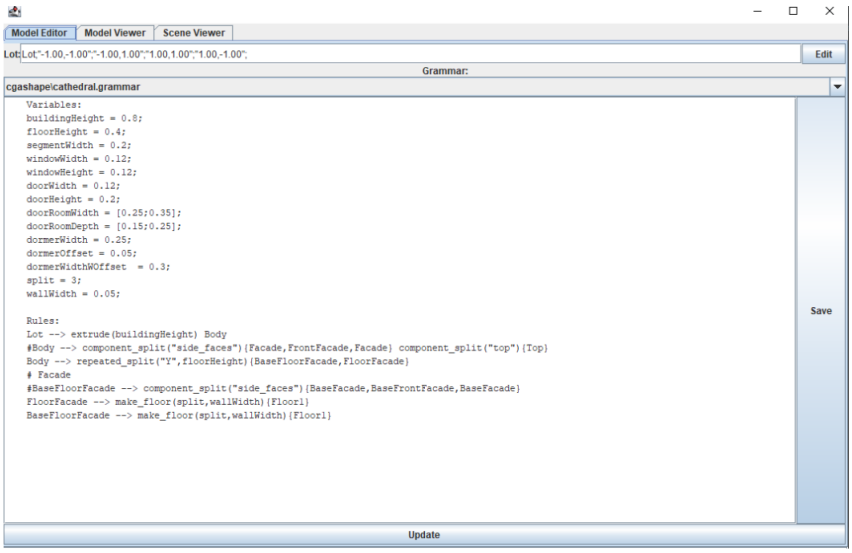


Abbildung 6.1: Grammatik, die für den Test benutzt wurde

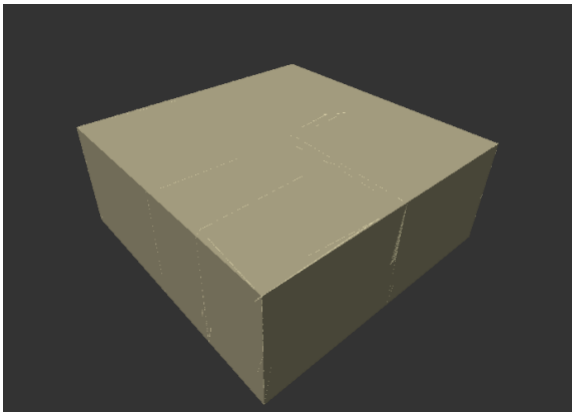


Abbildung 6.2: Resultat der Modellierung

In der Abbildung 6.3 wird nur das Model der erstellten Etage gezeigt.

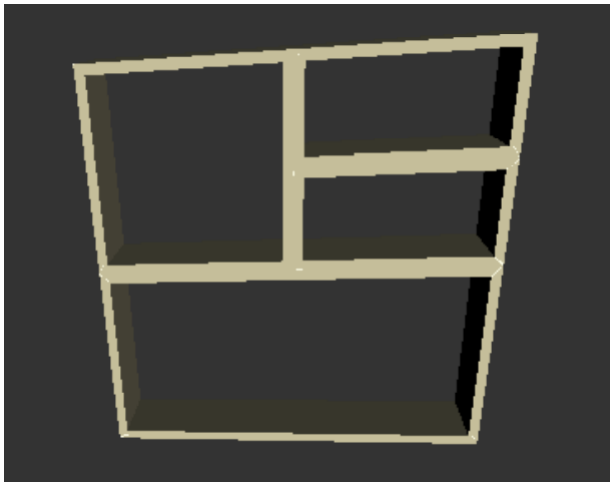


Abbildung 6.3: Erstellte Etage

Es ist zu sehen, dass genau drei innere Wände erstellt wurden, was der Anzahl von Teilungen in der Grammatik entspricht. Die Wände haben eine angemessene Breite und sind nicht zu dünn. Wenn die Anzahl der Teilungen in der Grammatik verändert wird, entstehen entsprechend weitere innere Wände.

In der Grammatik kann auch die Anzahl der Etagen durch die Anpassung der Variablen **buildingHeight** und **floorHeight** verändert werden. Dies hat jedoch keine Auswirkung auf die Raumverteilung auf jeder Etage, da die Räume auf jeder Etage immer die gleiche Anordnung haben. Zusätzlich wurde auch durch die Tests geprüft, ob die Erstellung von Räumen, Wänden und der Halbkanten-Datenstruktur erfolgreich war. Für diesen Test wurde auch die gleiche Grammatik verwendet. Es wurde geprüft, ob jeder Raum genau aus vier Wänden besteht und ob jede Wand eine Halbkante zugeordnet bekommen hat. Es wurde auch sichergestellt, dass jeder Raum ein geschlossener Zyklus von Halbkanten ist. Mit allen durchgeführten Tests wurde festgestellt, dass die Raumverteilung mithilfe des BSP-Algorithmus erfolgreich funktioniert und auch in 3D modelliert werden kann. In der folgenden Abbildung ist eine Ausgabe der Tests zu sehen:

```
Leaf Nummer 0 : projects.cgashape.room_creation.bsp.Leaf@779aed0e
Room Nummer 1 : projects.cgashape.room_creation.entity.Room@44dcd28c
Count of Walls: 4
Wall: HalfEdge{next=HalfEdgeVertex{pos=(2.0, 0.0, 0.0)}, startVertex=HalfEdgeVertex{pos=(2.0, 0.0, 2.0)}}
Wall: HalfEdge{next=HalfEdgeVertex{pos=(1.0, 0.0, 0.0)}, startVertex=HalfEdgeVertex{pos=(2.0, 0.0, 0.0)}}
Wall: HalfEdge{next=HalfEdgeVertex{pos=(1.0, 0.0, 2.0)}, startVertex=HalfEdgeVertex{pos=(1.0, 0.0, 0.0)}}
All Walls are connected
IsOuterWall false
IsOuterWall true
IsOuterWall true
IsOuterWall true
Room Nummer 2 : projects.cgashape.room_creation.entity.Room@156221ad
Count of Walls: 4
Wall: HalfEdge{next=HalfEdgeVertex{pos=(0.0, 0.0, 2.0)}, startVertex=HalfEdgeVertex{pos=(0.0, 0.0, 0.0)}}
Wall: HalfEdge{next=HalfEdgeVertex{pos=(1.0, 0.0, 2.0)}, startVertex=HalfEdgeVertex{pos=(0.0, 0.0, 2.0)}}
Wall: HalfEdge{next=HalfEdgeVertex{pos=(1.0, 0.0, 0.0)}, startVertex=HalfEdgeVertex{pos=(1.0, 0.0, 2.0)}}
All Walls are connected
IsOuterWall false
IsOuterWall true
IsOuterWall true
IsOuterWall true
```

Abbildung 6.4: Ausgabe vom Test des BSP-Baums und Raumerstellung

Mit diesem Test wurde auch bestätigt, dass der BSP-Baum im Projekt erfolgreich umgesetzt und für die Raumpartitionierung verwendet wurde (Anforderung 1.2). Es ist zu erkennen, dass bei einer Teilung genau drei Blätter im Baum entstehen (Anforderung 1.3). Davon wurden zwei Objekte der Klasse „Room“ erzeugt und somit nur zwei Räume erstellt (Anforderung 1.4). Diese Objekte sind auch Blätter des Baums, haben jedoch keine Unterblätter (Anforderung 1.5). Somit wurde die Etage erfolgreich in Räume unterteilt (Anforderung 1.6). Aus der Ausgabe ist auch ersichtlich, dass Räume genau aus vier Wänden bestehen und die Verbindungen von Wänden richtig aufgebaut wurden (Anforderung 1.7). Insgesamt gibt es acht Wände gegenüber den vier, die ursprünglich mit „Root“ erstellt wurden, was beweist, dass die Teilungsoperation erfolgreich funktioniert und auf die Wände angewendet werden kann (Anforderung 1.15). In der Ausgabe ist auch ersichtlich, dass jede Wand aus Halbkanten besteht (Anforderung 1.8) und das Attribut *isOuterWall* unterschiedliche Werte aufweist, damit existieren zwei Typen von Wänden: äußere und innere Wände (Anforderung 1.14). Die unterteilte Etage wurde erfolgreich in 3D modelliert, somit wurden die Anforderungen 1.10 und 1.11 erfolgreich umgesetzt.

6.2 Typisierung der erstellen Räumen

In diesem Abschnitt wird die Auswertung von Algorithmen zur Raumtypisierung durchgeführt und die Ergebnisse der Typisierung präsentiert. Damit die Tests erfolgreich sind, muss eine entsprechende Grammatik geschrieben werden, die die Erstellung der Räume und die Typisierung ermöglicht. In der Abbildung 6.5 ist die für den Test verwendete Grammatik dargestellt.

```
Variables:
buildingHeight = 0.8;
floorHeight = 0.8;
segmentWidth = 0.2;
doorRoomWidth = [0.25;0.35];
doorRoomDepth = [0.15;0.25];
dormerWidth = 0.25;
dormerOffset = 0.05;
dormerWidthWOffset  = 0.3;
split = 4;
wallWidth = 0.05;
windowHigh = 0.1;
windowWidth = 0.2;
numWindows = 1;
doorWidth = 0.09;
doorHeight = 0.5;


Rules:
Lot --> extrude(buildingHeight) Body
#Body --> component_split("side_faces"){Facade,FrontFacade,Facade} component_split("top"){Top}
Body --> repeated_split("Y",floorHeight){BaseFloorFacade,FloorFacade}
# Facade
#BaseFloorFacade --> component_split("side_faces"){BaseFacade,BaseFrontFacade,BaseFacade}
FloorFacade --> make_floor(split,wallWidth,numWindows,windowWidth,windowHigh,doorWidth,doorHeight) Floor1
BaseFloorFacade --> make_floor(split,wallWidth,numWindows,windowWidth,windowHigh,doorWidth,doorHeight) Floor1
Floor1 --> type_rooms("D:\\RoomsProjekt\\pcgofrooms\\src\\main\\java\\projects\\cgashape\\room_creation\\processor\\TestConfig"){FloorTyped1}
```

Abbildung 6.5: Grammatik für den Test

In dieser Grammatik wird zuerst der Körper des Gebäudes erstellt (Zeile 20), danach wird in Zeile 22 das Gebäude in Etagen unterteilt und mithilfe der Operation **make_floor** werden die einzelnen Etagen in Räume unterteilt. Sobald die Räume erstellt sind, wird die Operation **type_rooms** ausgeführt, welche den erstellten Räumen unterschiedliche Typen zuweist. Hierfür erhält die Operation als Parameter einen Pfad zur Konfigurationsdatei, die die Beschreibung der Regeln enthält. Insgesamt werden mit dieser Grammatik fünf Räume erstellt und typisiert. Die Grammatik enthält auch Eingaben zur Größe und Anzahl von Fenstern und Türen, die für die Modellierung verwendet werden (Anforderungen 2.14, 2.15). In der Abbildung 6.6 wird die Konfigurationsdatei dargestellt, die für die Typisierung verwendet wird.

	A	B	C	D	E	F	G
1	Type	Size	Prob	Neighbours	isOutsideConnected	appearence	additionRoom
2	Kitchen	0.25	0.5	Livingroom	false	1	0
3	Livingroom	1	1	Bathroom,Livingroom,Badroom,Kitchen	true	1	0
4	Badroom	1	0.1	Bathroom,Livingroom,Badroom	false	1	0
5	Bathroom	0.25	0.4	Badroom,Livingroom,Bathroom	false	1	0
6	Laundryroom	0.1	0	Bathroom	false	1	1
7	Storageroom	0.1	0	Kitchen	false	1	1
8	Commonroo	0	0	Bathroom,Livingroom,Badroom,Kitchen	false	1	1
9							

Abbildung 6.6: Konfigdatei für den Test

In der Abbildung 6.7 ist die Ausgabe des Parser-Tests dargestellt. Bei einem Vergleich mit der Konfigurationsdatei (Abbildung 6.6) fällt auf, dass die Anzahl der erstellten Regeln der Anzahl der in der Konfigurationsdatei definierten Regeln entspricht. Die Ausgabe zeigt auch die Attribute der ersten obligatorischen Regel. Es ist ersichtlich, dass der Parser alle Angaben

korrekt interpretiert und zur Regel hinzugefügt hat. Dieser Test bestätigt die Erfüllung der Anforderungen 2.1 bis 2.4.

```
Number of Additional Rules: 3
Number of needed Rules: 4
Room type: Kitchen
Room size: 0.25
Room prob: 0.5
Room Neighbours: [Livingroom]
Room is Connected to outside: false
```

Abbildung 6.7: Konsolenausgabe von dem Parsertest

Aus der Konfigdatei ist zu erkennen, dass es vier obligatorische und drei optionale Regeln gibt. Da genau fünf Räume typisiert werden müssen, werden bei diesem Test alle obligatorischen und einige optionale Regeln benötigt. In der Abbildung 6.9 ist ein Ergebnis der Typisierung dargestellt.

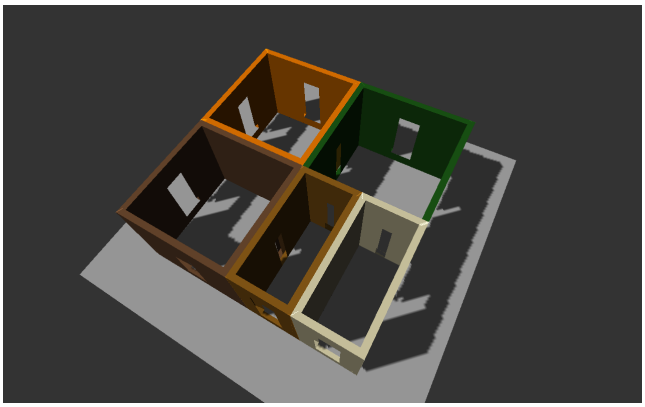


Abbildung 6.8: Ergebnis der Typisierung von fünf Räumen

Damit die Ergebnisse nachvollziehbar sind, wurden die Räume mit unterschiedlichen Farben versehen. Es ist erkennbar, dass jedem Raum der entsprechende Typ zugewiesen wurde. Das Wohnzimmer ist grün, das Schlafzimmer orange, das Badezimmer hellbraun, die Küche weiß und der Waschraum braun. Es ist auch zu erkennen, dass die Fenster und Türen modelliert wurden und dass sich die Türen nicht immer an jeder Wand befinden. Das bedeutet, dass für die Platzierung der Türen Metadaten verwendet wurden (Anforderungen 2.10 und 2.11). Mit der „Neighbours“-Spalte aus der Konfigurationsdatei ist zu erkennen, dass alle Verbindungen korrekt dargestellt sind. Da für den Verbindungsaufbau ein Graph verwendet wurde,

kann bestätigt werden, dass Anforderung 2.9 ebenfalls erfüllt ist. Die Küche ist nur mit dem Wohnzimmer verbunden. Da sich das Badezimmer nicht neben dem Schlafzimmer befindet, hat das Schlafzimmer nur eine Tür, die es mit dem Wohnzimmer verbindet. Das Badezimmer hingegen hat zwei Türen, eine zum Wohnzimmer und eine weitere, die das Badezimmer mit dem Waschraum verbindet. Alle Außenwände sind mit Löchern für die Fenster modelliert (Anforderung 2.12). Nur eine Außenwand verfügt über eine Eingangstür (Anforderung 2.13). Der Typisierungsprozess wurde so lange durchgeführt, bis alle obligatorischen Regeln erfüllt und die Räume erfolgreich typisiert wurden (Anforderungen 2.5-2.6). Wenn die Regeln aus der Konfigurationsdatei mit dem Ergebnis in 3D verglichen werden, wird dies ebenfalls bewiesen.

In der folgenden Abbildung ist noch ein Test gezeigt: hier wurden nur vier Räume modelliert und dadurch sind für diese Typisierung nur obligatorische Regeln verwendet.

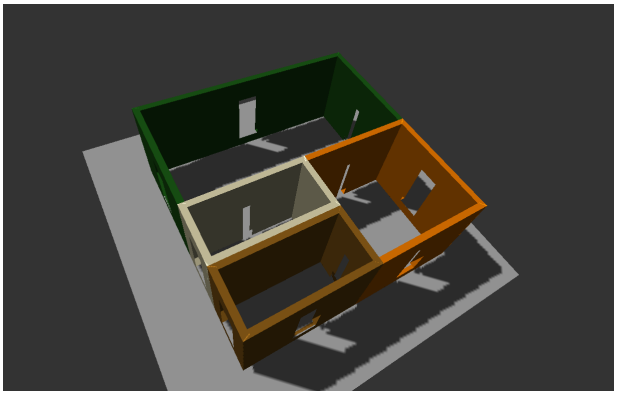


Abbildung 6.9: Ergebnis der Typisierung von vier Räumen

In diesem Fall gibt es keinen Waschraum mehr, da die Anzahl der obligatorischen Regeln ausreichend ist, um alle Räume zu typisieren. Die Position des Schlafzimmers wurde geändert, sodass es jetzt zwei Türen hat, eine zum Badezimmer und eine zum Wohnzimmer. Bei der Ausführung von dem Typisierungsprozess erscheint folgende Meldung in der Konsole:

```
Rule can not be assigned because Room size is not matching to Rooms without Types!! Try to change Room size for Badroom_0
Need Correction for Badroom_0!!!!
```

Abbildung 6.10: Meldung, dass ein Raum nicht typisiert werden konnte und Korrektur nötig ist

Diese Meldung besagt, dass die Regel für das Schlafzimmer nicht sofort erfüllt werden konnte und ein Korrekturprozess erforderlich ist. Die Typisierung wurde jedoch erfolgreich abgeschlossen, einschließlich des Schlafzimmers. Somit wird die Anforderung 2.7-2.8 umgesetzt,

da der Korrekturprozess erfolgreich durchgelaufen ist und die benötigte Regeln korrigiert wurden, um anschließend das Schlafzimmer erfolgreich zu typisieren.

In dieser Abbildung ist die Ausgabe der Adjazenzliste zu sehen, die erstellt wurde, als ein Graph für die Verbindung zwischen den Räumen erstellt wurde:

```
GraphNode: Room: Livingroom
    GraphNode: Room: Kitchen
    GraphNode: Room: Bathroom
    GraphNode: Room: Badroom
GraphNode: Room: Bathroom
    GraphNode: Room: Livingroom
GraphNode: Room: Badroom
    GraphNode: Room: Livingroom
GraphNode: Room: Storageroom
    GraphNode: Room: Kitchen
GraphNode: Room: Kitchen
    GraphNode: Room: Livingroom
```

Abbildung 6.11: Adjazenzliste von dem Graphen

Diese Liste zeigt die Verbindungen zwischen den Graphknoten. Die Knoten, die eingerückt sind, sind die Kindknoten der Knoten ohne Einrückung. Es ist ersichtlich, dass der Graph tatsächlich für den Verbindungsaufbau genutzt wurde (Anforderung 2.9).

6.3 Möbelplatzierung in Räumen

In diesem Teil werden Tests für die Algorithmen durchgeführt, die für die Platzierung von Möbeln in den Räumen zuständig sind. Um diese Algorithmen anzusprechen und auszuführen, musste die Grammatik aus dem vorherigen Kapitel angepasst werden, um die Operation der Möbelplatzierung auszuführen.

```
Variables:
buildingHeight = 0.8;
floorHeight = 0.8;
segmentWidth = 0.2;
doorRoomWidth = [0.25;0.35];
doorRoomDepth = [0.15;0.25];
dormerWidth = 0.25;
dormerOffset = 0.05;
dormerWidthWOffset  = 0.3;
split = 0;
wallWidth = 0.05;
windowHigh = 0.3;
windowWidth = 0.2;
numWindows = 2;
doorWidth = 0.15;
doorHeight = 0.4;
boidIteration = 10;
clearanceTries = 20;


Rules:
Lot --> extrude(buildingHeight) Body
#Body --> component_split("side_faces"){Facade,FrontFacade,Facade} component_split("top"){Top}
Body --> repeated_split("Y",floorHeight){BaseFloorFacade,FloorFacade}
# Facade
#BaseFloorFacade --> component_split("side_faces"){BaseFacade,BaseFrontFacade,BaseFacade}
#FloorFacade --> make_floor(split,wallWidth,numWindows,windowWidth,windowHigh,doorWidth,doorHeight) Floor1
BaseFloorFacade --> make_floor(split,wallWidth,numWindows,windowWidth,windowHigh,doorWidth,doorHeight) Floor1
Floor1 --> type_rooms("D:\RoomsProjekt\pcgRoom\src\main\java\projects\cgashape\room_creation\processor\TestConfig.csv") FloorTyped1
FloorTyped1 --> add_furniture("D:\RoomsProjekt\pcgRoom\src\main\java\projects\cgashape\room_creation\furnitureplacement\TestConfig.csv",wallWidth,windowWidth,windowHigh,doorWidth,doorHeight,boidIteration,clearanceTries) FurnitureFloor
```

Abbildung 6.12: Grammatik für die Erstellung von einem Raum mit Typ und Möbel

In der Abbildung 6.12 ist eine Grammatik dargestellt, die für die Tests des Möbelplatzierungsprozesses verwendet wird. Diese Grammatik enthält eine neue Operation namens **add_furniture**, die den Prozess startet. Damit die Operation ausgeführt werden kann, muss bereits eine Etage mit mindestens einem typisierten Raum erstellt worden sein. Vor der **add_furniture** Operation müssen daher die Operationen **type_rooms** zur Typisierung und **make_floor** zur Erstellung der Etage ausgeführt werden. Zusätzlich wurden zwei Variablen hinzugefügt: **boidIteration** und **clearanceTries**, die zur Optimierung und Abstandsflächenprüfung verwendet werden. Zusammen mit dem Pfad zu einer Konfigurationsdatei für Möbelregeln werden alle benötigten Variablen der **add_furniture**-Operation hinzugefügt. Durch diesen Aufbau der Grammatik können alle in dieser Arbeit entwickelten Prozesse gemeinsam ausgeführt und die Ergebnisse dieser Prozesse dargestellt werden.

In der Abbildung 6.13 ist eine Konfigdatei im Form von einer *csv* Datei dargestellt (Anforderung 3.1), diese Datei wurde für die Tests vom Möbelpositionierungsprozess verwendet.

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17
1	Roon	Name	isInCorner	isAlongWall	hasPaar	PaarWith	minPaarDistance	maxPaarDistance	needInitTurn	clearanceSide	clearanceFront	clearanceBack	canCloseWindow	canBeAround	nonCollision	isDetail	isOnTheSide
2	Livingroom	window	false	false	false	<null>	0	0	false	0	0	0	false	false	false	true	false
3	Livingroom	door	false	false	false	<null>	0	0	false	0	0	0	false	false	false	true	false
4	Livingroom	tableBig	false	true	false	<null>	0	0	false	0.3	0.3	0.3	false	false	false	false	false
5	Livingroom	chair	false	false	true	tableBig	0	0	false	0.0	0	0.0	false	true	false	false	false
6	Livingroom	chair1	false	false	true	tableBig	0	0	false	0.0	0.0	0.1	false	true	false	false	false
7	Livingroom	chair2	false	false	true	tableBig	0	0	false	0.0	0.0	0.1	false	true	false	false	false
8	Livingroom	Sofa	false	true	false	<null>	0	0	false	0.1	0.1	0.0	false	false	false	false	false
9	Livingroom	tvboard	false	true	true	Sofa	0.4	0.5	false	0.1	0.0	0.0	false	false	false	false	false
10	Livingroom	FloorLamp	true	false	false	<null>	0	0	false	0.0	0.0	0.0	false	false	false	false	false
11	Livingroom	bookshelf	false	true	false	<null>	0	0	false	0.1	0.0	0.0	true	false	false	false	false
12	Livingroom	bookshelf1	false	true	false	<null>	0	0	false	0.1	0.0	0.0	true	false	false	false	false
13	Livingroom	SofaTable	false	false	true	Sofa	0.2	0.4	false	0.1	0	0	false	false	false	false	false

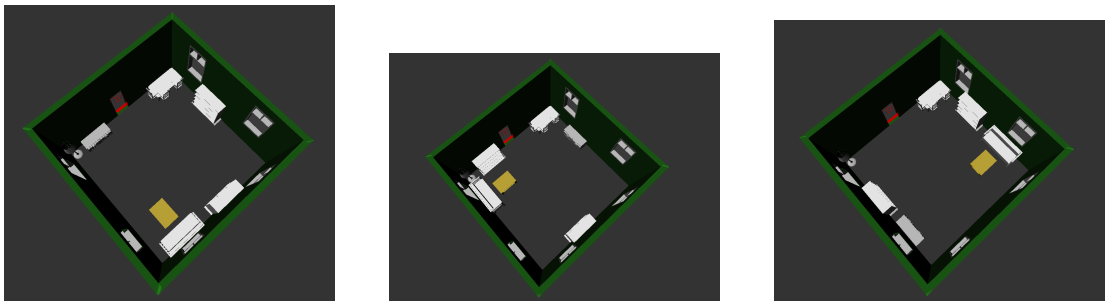
Abbildung 6.13: Konfigdatei, um die Positionen der Möbel zu konfigurieren

Die derzeitige Implementierung des Möbelplatzierungsprozesses ist ausschließlich auf die Platzierung von Möbeln in einem Raum ausgelegt. Daher wird in der Konfiguration für das Element *Room* ein einziger Raum mit dem Typ *Livingroom* erstellt, in dem insgesamt zwölf Möbelstücke platziert werden sollen. Fünf dieser Möbelstücke sind als Kindermöbelstücke

vorgesehen (Anforderung 3.4), während die restlichen Möbelstücke entweder als Elternmöbelstücke (Sofa, Tisch) oder als einzelne Möbelstücke, die nicht mit anderen Möbeln verbunden sind, konfiguriert sind. Die Kindermöbelstücke stehen auch unterschiedlich zu anderen Elternmöbelstücken. Die Möbelstücke mit dem Namen *chair* sollen um das Möbelstück mit der Name *tableBig* herumstehen. Da die Möbelstücke *tvboard* und *SofaTable* keine besondere Eingabe bezüglich der Position haben, sollen die vor dem Elternmöbelstück *Sofa* platziert werden. Das heißt, dass die Anforderung 3.5 erfüllt ist. Zwei der Möbelstücke sind als Detailobjekte definiert und werden bei der Platzierung und Optimierung nicht berücksichtigt. Diese Möbelstücke werden am Ende des Prozesses hinzugefügt.

Einige Möbelstücke, wie zum Beispiel das *bookshelf*, sind so konfiguriert, dass sie das Fenster überdecken können, weshalb sie Regeln zur angemessenen Platzierung in Bezug auf das Fenster haben. Die Abstandsflächen in der Konfigurationsdatei sind so eingestellt, dass die Abstände zwischen den Möbelstücken realistisch sind und alle Möbelstücke zugänglich sind.

Eine Voraussetzung für die Möbelplatzierung ist das Vorhandensein aller 3D-Modelle für die Möbelstücke, die als Ressourcen im Verzeichnis *Models* abgelegt werden sollen. Wenn für ein Möbelstück kein Modell vorhanden ist, wird ein Fehler verursacht und das Möbelstück kann nicht platziert werden. Die Größe des Möbelstücks und seine ursprüngliche Ausrichtung spielen auch eine wichtige Rolle im Platzierungsprozess. Initial sollen alle Möbelstücke in die -*Z* (0,0,-1) Richtung orientiert werden.



(a) Layoutvariante a (b) Layoutvariante b (c) Layoutvariante c

Abbildung 6.14: Resultate der Möbelplatzierung mit der beschriebenen Konfigdatei

In der Abbildung 6.14 werden drei Layoutvarianten dargestellt, die mit dieser Konfiguration erstellt werden können. Unterschiedliche Varianten mit gleicher Konfiguration können erstellt werden, weil zu Beginn der Möbelplatzierung zufällige Positionen der Möbel ermittelt werden. Durch diese unterschiedlichen Startpositionen werden auch die Möbelpositionen unterschiedlich optimiert, wodurch verschiedene Endwerte entstehen können. Es waren auch alle Möbel,

die in der Konfigdatei vorgekommen sind in 3D dargestellt und richtig lautet Regelbeschreibungen platziert. Diese Beobachtung bedeutet, dass die Konfigdatei richtig gelesen wurde und alle Regeln richtig interpretiert wurden (Anforderungen 3.2, 3.3). Es ist auch zu sehen, dass alle Möbelstücke erfolgreich platziert wurden und dadurch die Anforderung 3.6 auch erfüllt ist.

```
Start to place bookshelf1
Can't move from window!!
Could not move bookshelf1 to wall. COLLISION Number of Wall 1!!
Moved bookshelf1 to wall. NO COLLISSION! Check Clearance!!Number of Wall 2!!
Check clearance for rule bookshelf1
Try 0
Check clearance for rule bookshelf1 direction Side
Try 1
Check clearance for rule bookshelf1 direction Side
Try 2
Check clearance for rule bookshelf1 direction Side
Try 3
Check clearance for rule bookshelf1 direction Side
Could not place bookshelf1 along wall! Change Configuration or Distance to Window
Problem can be in the Configuration of clearance!!Number of Wall 2!!
Moved bookshelf1 to wall. NO COLLISSION! Check Clearance!!Number of Wall 3!!
Check clearance for rule bookshelf1
Try 0
Successfully Placed bookshelf1
Start to place SofaTable
Pair rule with Child Furniture
Check clearance for rule SofaTable
Try 0
Successfully Placed SofaTable
```

Abbildung 6.15: Ausgabenausschnitt mit möglichen Meldungen bei der Platzierung von *bookshelf1*

In der Abbildung 6.15 ist ein Ausschnitt dargestellt, der zeigt, welche Meldungen bei Kollisionen und fehlenden Abstandsflächen auftreten können. Das Beispiel zeigt die Platzierung des Möbelstücks *bookshelf1*. Es ist zu sehen, dass es bei der Platzierung neben der ersten Wand zu einem Kollisionsproblem kam (Anforderung 3.8). Daraufhin wurde versucht, das Möbelstück an eine andere Wand zu bewegen (Anforderung 3.9) und an der *Wand 2* traten keine Kollisionen mehr auf. Nachdem das Möbelstück ohne Kollisionen platziert wurde, wurde die Abstandsfläche überprüft (Anforderung 3.10). Die Informationen zur Abstandsfläche sind in der Konfigurationsdatei unter den Spalten *clearanceFront*, *clearanceSide* und *clearanceBack* zu finden (Anforderung 3.11). In der Ausgabe ist zu sehen, dass die Abstandsfläche von der Seite des Möbelstücks nicht gewährleistet werden konnte. Um dies zu bestätigen, wurden alle

angegebenen Versuche der Abstandsflächenprüfung durchgeführt (Anforderung 3.12). Nach jedem Versuch wurde das Möbelstück leicht verschoben und die Prüfung wiederholt. Dennoch konnte die Abstandsfläche nicht gewährleistet werden. An der *Wand 3* traten keine Kollisionen auf und die Abstandsflächenprüfung war erfolgreich, sodass *bookshelf1* dort platziert werden konnte. Die letzte bearbeitete Regel war die *SofaTable*, die auch die letzte Regel in der Konfigurationsdatei war. Danach war der Prozess der Möbelplatzierung abgeschlossen (Anforderung 3.13).

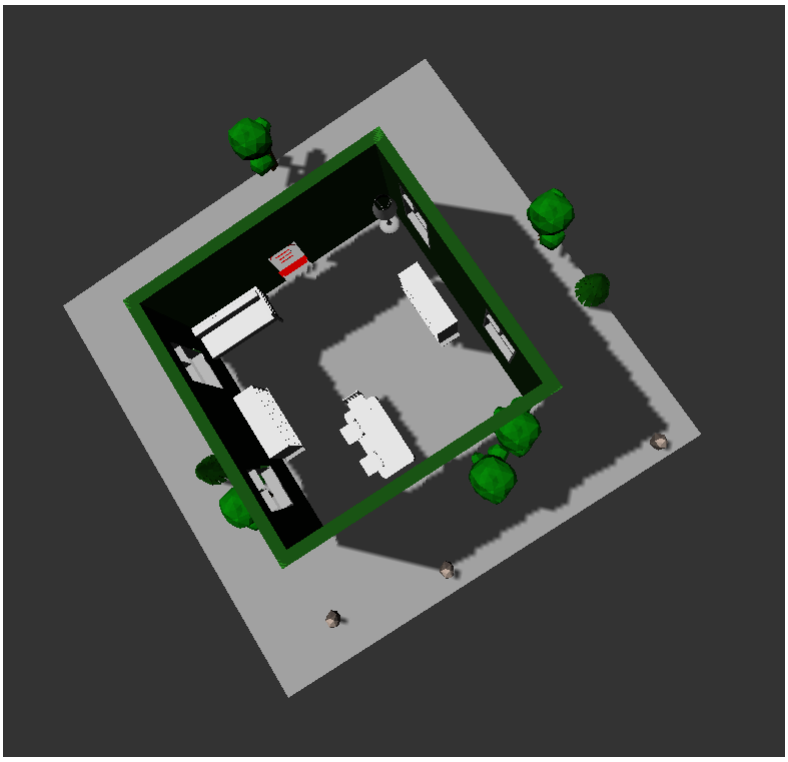


Abbildung 6.16: Kleine Etage mit keinem Platz für alle Möbelstücke

In der Abbildung 6.16 wird ein weiteres Ergebnis dargestellt, das zeigt, dass die Modellierung erfolgreich sein kann, selbst wenn nicht alle Regeln für Möbelstücke erfüllt wurden und nicht alle in der Konfigurationsdatei beschriebenen Möbelstücke platziert wurden. Um dies zu erreichen, wurde die Größe der Etage so verkleinert, dass nicht alle Möbelstücke hineinpassen können. Dieser Test bestätigt die Aussage aus Anforderung 3.14. In der Abbildung 6.17 wird eine Etage präsentiert, wo die Wände nicht visualisiert wurden, um zu zeigen, dass die Möbelstücke nicht in Richtung der Y-Achse verschoben wurden, sondern nur in den Richtungen X und Z

bewegt wurden. Hier kann man sehen, dass nach der erfolgreichen Platzierung der Möbelstücke alle auf der gleichen Ebene liegen (Anforderung 3.7).

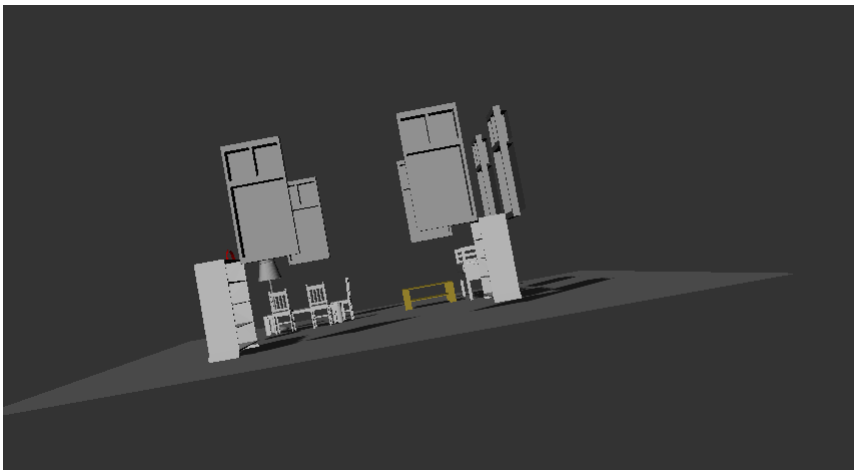


Abbildung 6.17: Etage ohne Wände, um die Richtigkeit der Möbelbewegung zu prüfen

Es wurde auch ein Leistungstest durchgeführt, bei dem die Dauer des Platzierungsprozesses gemessen wurde. Der Test wurde auf einem Laptop mit Intel(R) Core(TM) i5-9300H CPU und der Grafikkarte Nvidia GeForce RTX 2060 durchgeführt. Der Prozess wurde genau zehnmal ausgeführt und danach wurde der Durchschnittswert der Zeit in Millisekunden berechnet. Die Ergebnisse dieses Tests für die oben beschriebene Grammatik sind in der folgenden Tabelle zu sehen:

Zeit pro Durchlauf in Millisekunden
207,00
136,00
133,00
159,00
132,00
181,00
128,00
123,00
121,00
145,00
Durchschnitt
146,5

Tabelle 6.1: Durchlaufzeit des Möbelplatzierungsprozesses für Konfiguration aus 6.13

Es ist zu sehen, dass mit der oben beschriebenen Konfiguration der Möbelplatzierungsprozess im Durchschnitt 146,5 Millisekunden braucht, um die Platzierung komplett durchzuführen.

	Room	Name	isInCorner	isAlongWall	hasPaar	PaarWith	minPaarDistance	maxPaarDistance	needInitTurn	clearanceSide	clearanceFront	clearanceBack	canCloseWindow	canBeAround	nonCollision	isDetail	isOnTheSide
2	Livingroom	window	false	false	false	<null>	0	0	false	0	0	0	false	false	false	true	false
3	Livingroom	door	false	false	false	<null>	0	0	false	0	0	0	false	false	false	true	false
4	Livingroom	tableBig	false	true	false	<null>	0	0	false	0.3	0.3	0.3	false	false	false	false	false
5	Livingroom	chair	false	false	true	tableBig	0	0	false	0.0	0	0.0	false	true	false	false	false
6	Livingroom	chair1	false	false	true	tableBig	0	0	false	0.0	0.0	0.1	false	true	false	false	false
7	Livingroom	chair2	false	false	true	tableBig	0	0	false	0.0	0.0	0.1	false	true	false	false	false
8	Livingroom	Sofa	false	true	false	<null>	0	0	false	0.1	0.1	0.0	false	false	false	false	false
9	Livingroom	tvboard	false	true	true	Sofa	0.4	0.5	false	0.1	0.0	0.0	false	false	false	false	false

Abbildung 6.18: Konfigdatei für den Test mit acht Möbel

In dem nächsten Test wurde die Anzahl von Möbel, die platziert werden müssen, von 12 auf 8 reduziert und folgendes Ergebnis ist entstanden:

Zeit pro Durchlauf in Millisekunden
73,00
104,00
69,00
74,00
74,00
92,00
82,00
85,00
75,00
72,00
Durchschnitt
80

Tabelle 6.2: Durchlaufzeit des Möbelplatzierungsprozesses für Konfiguration aus 6.18

Die Konfigdatei, die für diesen Test benutzt wurde, ist in der Abbildung 6.18 zu sehen. Hier ist die durchschnittliche Laufzeit nur 80 Millisekunden. Das hängt mit der Anzahl der Objekte zusammen, die als Dreiecksmesh geladen und platziert werden müssen. Je mehr Objekte in einem Raum platziert werden sollen, desto länger dauert auch der Prozess der Möbelplatzierung. Die Laufzeit des Möbelplatzierungsprozesses kann auch durch das Hinzufügen von komplizierten und großen Objekten beeinflusst werden. Das wird im nächsten Beispiel gezeigt.

Es können auch weitere Möbelstücke zu der Konfigdatei hinzugefügt werden. Zum Beispiel, wenn ein Teppich zu der Datei hinzugefügt wird, wird das Möbellayout wie folgt aussehen.

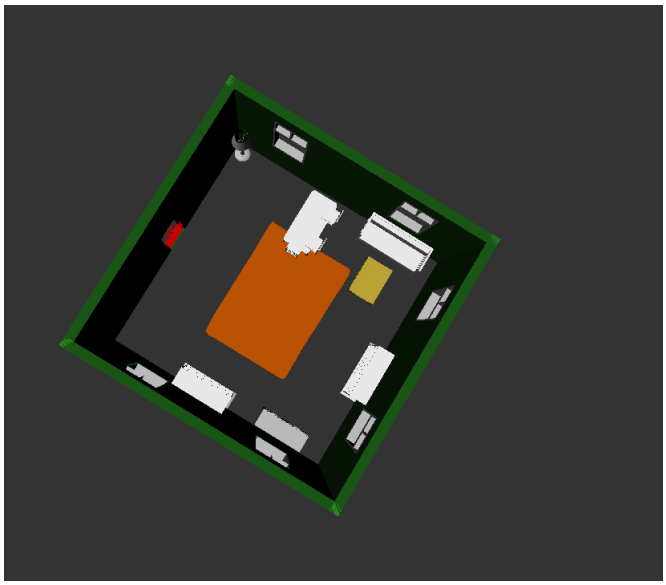


Abbildung 6.19: Möbellayout mit dem Teppich

Es ist zu erkennen, dass der Teppich mit einem kleinen Tisch kollidiert. Um dies zu ermöglichen, wurde in der Konfigurationsdatei für den Teppich das Element *nonCollision* auf den Wert *true* gesetzt, was besagt, dass dieses Objekt keine Kollisionen mit anderen Objekten verursachen kann.

Für die Konfiguration mit dem Teppich wurde auch ein Leistungstest durchgeführt, die Resultate dieses Tests sehen wie folgt aus:

Zeit pro Durchlauf in Millisekunden
381,00
334,00
452,00
385,00
374,00
377,00
391,00
409,00
370,00
365,00
Durchschnitt
383,8

Tabelle 6.3: Durchlaufzeit des Möbelplatzierungsprozesses für Konfiguration mit Teppich Objekt

Es ist zu erkennen, dass das Hinzufügen neuer Möbelstücke im Raum, die aus relativ vielen Polygonen erstellt werden können, die Leistung des Prozesses beeinflusst. Das passiert, weil nachdem der Platzierungsprozess der Möbel stattfindet, alle Modelle, die in der Konfigurationsdatei erwähnt wurden, als Dreieckmeshes aus den *.obj*-Dateien erstellt werden müssen. Je mehr unterschiedliche Möbelstücke geladen werden müssen, desto länger dauert der Platzierungsprozess. Die *.obj*-Dateien enthalten die Positionen und die Beschreibung von Punkten und Polygonen, aus denen das Objekt erstellt wird. Je komplizierter und größer das Objekt ist, desto länger dauert die Umwandlung dieses Objekts in das Dreieckmesh. Für den Test und allgemein bei der Entwicklung der beschriebenen Software wurden sogenannte „Low-Poly“ Objekte benutzt. Das sind vereinfachte Objekte mit einer geringeren Anzahl an Punkten und Polygonen. Solche Objekte werden schneller geladen und der Prozess der Möbelpositionierung kann schneller abgeschlossen werden. Zusätzlich werden solche Objekte schneller in 3D dargestellt und das gemeinsame Dreieckmesh von den Etagen mit Möbeln kann schneller generiert werden.

Aus dieser Beobachtung kann man verstehen, dass für eine performante und schnelle Durchführung der Möbelplatzierung mit der geschriebenen Software bestenfalls „Low-Poly“ Objekte benutzt werden sollten. Zusätzlich soll bei der Erstellung von Konfigurationsdateien eine angemessene Anzahl an unterschiedlichen Möbelstücken verwendet werden.

7 Zusammenfassung

Die Prozedurale Generierung umfasst Methoden, mit denen unterschiedliche Programminhalte wie beispielsweise Texturen, 3D-Objekte oder Töne generiert werden können. Sie eignet sich auch sehr gut für die Erstellung von Innenräumen. Im Rahmen der Masterarbeit wurde das Ziel verfolgt, eine Software zu entwickeln, die mittels Prozeduraler Generierung Räume innerhalb von Gebäuden erstellen, typisieren und mit Möbeln ausstatten kann. Die Implementierung dieser Software stellt eine Erweiterung eines bestehenden Projekts dar. Vor der Erweiterung konnte die Software nur die Gebäudehüllen erzeugen, die lediglich die Fassade der Gebäude darstellten. Durch die Erweiterung ist es nun möglich, die Innenräume von Gebäuden vollständig und benutzerdefiniert in 3D zu modellieren.

Die Arbeit wurde in drei Teile unterteilt, wobei jeder Teil ein Stück der Software enthält, das zur Erreichung des Ziels erforderlich ist. Im ersten Teil wurde die Partitionierung der Etage in verschiedene Räume realisiert. Für die Raumpartitionierung auf jeder Etage des Gebäudes wurde der BSP-Algorithmus implementiert. Dieser Algorithmus basiert auf einer bestimmten Anzahl von Teilungen und teilt die Etage in X- und Z-Richtung auf. Die resultierenden Räume nach der Teilung werden in Blätter des sogenannten BSP-Baums gespeichert. Jede Teilung, die durchgeführt wird, erzeugt eine neue innere Wand, die die Räume voneinander trennt. Die Wände entstehen aus den Halbkanten, die wiederum eine Halbkanten-Datenstruktur der gegebenen Etage bilden. Diese Halbkanten wurden auch bei der 3D-Modellierung der Etage verwendet.

Sobald der Partitionierungsprozess durchlaufen ist, wird im zweiten Teil der Arbeit ein weiterer Prozess gestartet, der die Möglichkeit bietet, die generierten Räume mithilfe eines regelbasierten Algorithmus erfolgreich zu typisieren. Für jeden Typ kann eine Regel erstellt und modifiziert werden, die Informationen für jeden Typ enthält. Die Regel kann obligatorisch oder optional sein. Alle Regelbeschreibungen werden in eine Konfigurationsdatei geschrieben, die vom entwickelten Algorithmus gelesen wird, um die Regeln zu definieren.

Sobald die Regeln definiert sind, wird mit dem Typisierungsprozess begonnen. Dabei wird jede Regel bearbeitet und versucht, den entsprechenden Typ dem passenden Raum zuzuweisen. Die Größe und Nachbarschaft jedes untersuchten Raums werden mit den Vorgaben der Regel

verglichen. Nur Räume, die die passende Größe und Nachbarschaft aufweisen, können den jeweiligen Typ erhalten. Der Algorithmus beginnt mit der Erfüllung aller obligatorischen Regeln und bearbeitet erst danach die optionalen Regeln. Wenn es eine obligatorische Regel gibt, die zu keinem Raum passt, wird ein Korrekturprozess gestartet. Im Rahmen dieses Prozesses werden bereits erfüllte Regeln neu geprüft, um Typen den neuen Räumen zuzuweisen und die fehlerhafte Regel zu erfüllen. Der Korrekturprozess geht so lange weiter, bis alle betroffenen Regeln erfüllt sind und die fehlerhafte Regel auch erfüllt wurde. Sobald alle Regeln erfüllt sind und jedem Raum im Gebäude ein Typ zugewiesen wurde, ist der Typisierungsprozess abgeschlossen.

Der nächste Schritt besteht darin, die typisierten Räume richtig zu verbinden. Dafür werden die erstellten Regeln und ein Graph, der die Räume als Knoten hat und die Verbindung zwischen zwei Räumen durch Kanten darstellt, benötigt.

Sobald beide Algorithmen abgeschlossen sind, erfolgt die Modellierung der Räume. Hierbei wird das Polygon verwendet, um die Löcher im Mesh darzustellen. Jede Wand in einem Raum muss entsprechende Metadaten für die Darstellung von Löchern haben, insbesondere für Türen. Wenn alle Metadaten vergeben sind, können die Räume modelliert werden.

Sobald der Typisierungsprozess abgeschlossen ist und die Räume ihre Typen erhalten haben, wird die dritte Phase der Generierung gestartet. In dieser Phase werden die Räume mit verschiedenen Möbelstücken ausgestattet. Auch dieser Prozess basiert auf einem regelbasierten Algorithmus. Die Informationen zu einzelnen Regeln werden in einer Konfigurationsdatei geschrieben und mit einem Parser eingelesen, um die Regeln zu erstellen. Für jedes Möbelstück werden separate Regeln erstellt, die Informationen über die Position des Möbelstücks im Raum enthalten. Die Möbelstücke haben zwei Kategorien: Elternmöbelstücke und Kindermöbelstücke. Elternmöbelstücke werden wie alle anderen Möbelstücke behandelt und können in den Ecken, entlang der Wand und frei im Raum platziert werden. Die Kindermöbelstücke werden dann den Elternmöbelstücken zugeordnet und je nach Regelbeschreibung unterschiedlich zu ihren Elternmöbelstücken orientiert. Die Regeln enthalten auch Informationen über Abstandsflächen, die benötigt werden, um genügend Abstand zwischen den Möbelstücken zu halten. Bevor die Regeln angewendet werden, erhalten alle Möbelstücke eine zufällige Position im Raum, die dann mithilfe des Separation-Steering-Algorithmus optimiert wird.

Im Gegensatz zu den Regeln im Typisierungsprozess müssen nicht alle Möbelregeln erfüllt sein, damit der Prozess der Möbelplatzierung abgeschlossen werden kann. Falls ein bestimmtes Möbelstück nicht hinzugefügt werden kann, wird das Programm eine entsprechende Fehlermeldung in der Konsole anzeigen und dem Benutzer signalisieren, dass ein Fehler aufgetreten ist.

Damit jeder dieser Teile innerhalb von dem Basisprojekt ausgeführt werden kann, wurden für jeden Teil die entsprechenden Operationen in die Basisprojekt-Grammatik aufgenommen. Diese Operationen wurden nacheinander ausgeführt, um die vollständige Innenraumgenerierung zu ermöglichen. Im Rahmen der Arbeit wurden die Prozesse auch in einem separaten Kapitel ausgewertet und getestet. Die Implementierung der benötigten Algorithmen zeigte, dass die für jeden Teil der Arbeit definierten Anforderungen erfolgreich umgesetzt wurden und das Ziel der Arbeit, eine erfolgreiche Innenraumgenerierung mit Typisierung und Möbelplatzierung zu erreichen, erfüllt wurde.

7.1 Ausblick

Im Rahmen der Auswertung der Konzepte und Prototypen wurden unterschiedliche kleine Probleme festgestellt, die im weiteren Verlauf des Projekts behoben werden können. Bei der Möbelplatzierung wurde festgestellt, dass die Größe des Möbelstücks von der Raumgröße abhängt. Aus diesem Grund wurde in diesem Prototyp der Möbelplatzierungsprozess nur mit einem Raum implementiert und vorgestellt. Dadurch sollen Verbesserungen hinzugefügt werden, die zum Beispiel eine dynamische Änderung der Größe des Möbelstücks ermöglichen. Eine weitere Besonderheit, die beachtet werden muss, betrifft die Nutzung von Materialien und Texturen für Möbelstücke. Derzeit werden die Dreiecksmeshes von Möbelstücken ohne Materialien und Texturen modelliert, obwohl die dafür benötigten Dateien auch im *Models*-Ordner vorhanden sind. Dies hat keine Auswirkungen auf die Leistung des Programms, sondern würde die visuelle Darstellung der erstellten Etage verbessern und kann bei der Weiterentwicklung der Software implementiert werden. Die Raumpartitionierung funktioniert momentan mit strikter Teilung von Räumen genau in der Mitte, und Räume auf jeder Etage werden jetzt genau gleich geteilt. An dieser Stelle könnte die Partitionierung verbessert werden, damit sie auf unterschiedlichen Etagen auch unterschiedlich vorgehen kann und dadurch mehr verschiedene Layouts entstehen können. Der Prozess der Teilung kann auch kontrollierter gemacht werden, indem benutzerdefinierte Werte für mögliche Offsets bei der Teilung vergeben werden, wie zum Beispiel wie weit in X- oder Z-Richtung die Teilung verschoben werden kann. Das System von Fehlermeldungen bei der Typisierung und Möbelplatzierung kann auch verbessert werden. Bei diesen Meldungen kann auch spezifische Information ausgegeben werden, die zum Beispiel beschreibt, wie viel größer oder kleiner ein Raum sein muss, um einen Typ zu bekommen, bei dem ein Fehler aufgetreten ist. Gleiches gilt für die Nachbarschaft. Es können Informationen darüber ausgegeben werden, welche Nachbartypen noch in der Nachbarschaftsliste fehlen. Eine Verbesserung, die noch hinzugefügt werden kann, ist die Integration der Konfigurationsdateien

direkt in die Benutzeroberfläche des Programms. Das wird eine bessere Benutzererfahrung bieten und eine bequemere Nutzung des Programms gewährleisten. Die Änderungen an diesen Konfigurationsdateien sollen auch direkt im 3D-Modell angezeigt werden.

Literaturverzeichnis

[1] L. M. Bond, "Procedural generation: An algorithmic analysis of video game design and level creation," 2017.

[2] J. Baron, "Procedural dungeon generation analysis and adaptation," pp. 168–171, 04 2017.

[3] R. Lopes, T. Tutenel, R. Smelik, K. J. de Kraker, and R. Bidarra, "A constrained growth method for procedural floor plan generation," 01 2010.

[4] E. Hahn, P. Bose, and A. Whitehead, "Persistent realtime building interior generation," pp. 179–186, 07 2006.

[5] P. Merrell, E. Schkufza, and V. Koltun, "Computer-generated residential building layouts," *ACM Transactions on Graphics*, vol. 29, 12 2010.

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. USA: Cambridge University Press, 3 ed., 2007.

[7] R. E. Neapolitan, *Learning Bayesian Networks*. USA: Prentice-Hall, Inc., 2003.

[8] R. Koenig and K. Knecht, "Comparing two evolutionary algorithm based methods for layout generation: Dense packing versus subdivision," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 28, pp. 285–299, 07 2014.

[9] S. Arvin and D. House, "Modeling architectural design objectives in physically based space planning," *Automation in Construction*, vol. 11, pp. 213–225, 02 2002.

[10] Elinder, Tobias, "General Methods for the Generation of Seamless Procedural Cities," 2017. Student Paper.

[11] M. Mirahmadi and A. Shami, "A novel algorithm for real-time procedural generation of building floor plans," *CoRR*, vol. abs/1211.5842, 2012.

[12] J. Martin, "Procedural house generation : A method for dynamically generating floor plans," 2006.

[13] F. Marson and S. Musse, "Automatic real-time generation of floor plans based on squarified treemaps algorithm," *International Journal of Computer Games Technology*, vol. 2010, 01 2010.

[14] M. Bruls, K. Huizing, and J. J. v. Wijk, "Squarified Treemaps," in *Eurographics / IEEE VGTC Symposium on Visualization* (W. de Leeuw and R. van Liere, eds.), The Eurographics Association, 2000.

[15] T. Germer and M. Schwarz, "Procedural Arrangement of Furniture for Real-Time Walkthroughs," *Computer Graphics Forum*, 2009.

[16] K. A. Kjolaas, "Automatic furniture population of large architectural models," 2000.

[17] P. Merrell, E. Schkufza, Z. Li, M. Agrawala, and V. Koltun, "Interactive furniture layout using interior design guidelines," SIGGRAPH '11, (New York, NY, USA), Association for Computing Machinery, 2011.

[18] M. Fisher, D. Ritchie, M. Savva, T. Funkhouser, and P. Hanrahan, "Example-based synthesis of 3d object arrangements," in *ACM SIGGRAPH Asia 2012 papers*, SIGGRAPH Asia '12, 2012.

[19] P. Guerrero, S. Jeschke, M. Wimmer, and P. Wonka, "Learning shape placements by example," *ACM Trans. Graph.*, vol. 34, jul 2015.

[20] Y. Akazawa, Y. Okada, and K. Niijima, "Automatic 3d scene generation based on contact constraints," *Intelligenza Artificiale - IA*, vol. 8, 01 2005.

[21] K. Xu, J. Stewart, and E. Fiume, "Constraint-based automatic placement for scene composition," in *Proceedings of the Graphics Interface 2002 Conference, May 27-29, 2002, Calgary, Alberta, Canada*, pp. 25–34, May 2002.

[22] S. Chojnacki, "Scoring functions for automatic arrangement of business interiors," in *SIGGRAPH Asia 2012 Technical Briefs*, SA '12, (New York, NY, USA), Association for Computing Machinery, 2012.

[23] P. Kán and H. Kaufmann, "Automatic furniture arrangement using greedy cost minimization," *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 491–498, 2018.

[24] P. Kán, A. Kurtic, M. Radwan, and J. Rodríguez, "Automatic interior design in augmented reality based on hierarchical tree of procedural rules," *Electronics*, vol. 10, p. 245, 01 2021.

[25] G. Stiny and J. Gips, "'shape grammars and the generative specification of painting and sculpture'," vol. 71, pp. 1460–1465, 01 1971.

[26] W. J. Mitchell, *The Logic of Architecture: Design, Computation, and Cognition.* Cambridge, MA, USA: MIT Press, 1st ed., 1990.

[27] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," vol. 25, p. 614–623, jul 2006.

[28] J. Halatsch, A. Kunze, and G. Schmitt, *Using Shape Grammars for Master Planning*, pp. 655–673. 01 2008.

[29] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," SIGGRAPH '01, (New York, NY, USA), p. 301–308, Association for Computing Machinery, 2001.

[30] H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On visible surface generation by a priori tree structures," SIGGRAPH '80, (New York, NY, USA), p. 124–133, Association for Computing Machinery, 1980.

[31] Z. Chen, A. Tagliasacchi, and H. Zhang, "Bsp-net: Generating compact meshes via binary space partitioning," 2020.

[32] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications.* Springer-Verlag, second ed., 2000.

[33] A. Banks, J. Vincent, and C. Anyakoha, "A review of particle swarm optimization. part i: background and development," *Natural Computing*, vol. 6, pp. 467–484, 2007.

[34] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *SIGGRAPH Comput. Graph.*, vol. 21, p. 25–34, aug 1987.

[35] "Haw pcg basisprojekt."

[36] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," SIGGRAPH '01, (New York, NY, USA), p. 301–308, Association for Computing Machinery, 2001.

[37] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," *ACM Transaction on Graphics*, vol. 22, 07 2003.

[38] D. H. Eberly, *3D Game Engine Design, Second Edition: A Practical Approach to Real-Time Computer Graphics (The Morgan Kaufmann Series in Interactive 3D Technology).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[39] D. Eberly, “Triangulation by ear clipping,” 01 2002.

Anhang

Algorithm 8 createBSPTree

```
1:  function createBSPTree
2:      startAxis ← getRoot().lastSplit
3:      for i ← 0 to splitCount do
4:          currentLeaves ← newArrayList(getAllLeafs())
5:          for Leaf l in currentLeaves do
6:              if l.leftChild == null & l.rightChild == null then
7:                  if Axis.X.equals(l.lastSplit) & ¬ startAxis.equals(Axis.Z) then
8:                      if l.splitLeaf(Axis.Z) then
9:                          leaves.add(l.leftChild)
10:                         leaves.add(l.rightChild)
11:                         startAxis ← Axis.Z
12:                         break
13:                     end if
14:                 else if Axis.Z.equals(l.lastSplit) & ¬ startAxis.equals(Axis.X) then
15:                     if l.splitLeaf(Axis.X) then
16:                         leaves.add(l.leftChild)
17:                         leaves.add(l.rightChild)
18:                         startAxis ← Axis.X
19:                         break
20:                     end if
21:                 end if
22:             end if
23:         end for
24:     end for
25: end function
```

Dieser Algorithmus beschreibt die Erstellung vom BSP-Baum für die Raumpartitionierungsprozess. In den Zeilen 8 und 15 ist die Methode **splitLeaf** verwendet, die in dem Algorithmus 9 als Pseudocode dargestellt ist.

Algorithm 9 splitLeaf

```
1:  function splitLeaf(splitDirection)
2:      if leftChild ≠ null ∨ rightChild ≠ null ∨ splitDirection = null then
3:          return false
4:      end if
5:      if splitDirection.equals(Axis.X) ∨ splitDirection.equals(Axis.Z) then
6:          listOfWalls ← shape.getWalls()
7:          vektorMap ← createVektorAxisRelation(listOfWalls)
8:          if splitDirection.equals(Axis.X) then
9:              wallsToSplit ← vektorMap.get(Axis.X)
10:             newWall ← HalfEdgeOperations.split(wallsToSplit, shape.getHalfEdgeDS())
11:             rightChild ← new Leaf(newRoomCreation(newWall.getShape()), Axis.X)
12:             leftChild ← new Leaf(newRoomCreation(newWall.getShape().getOpposite()),
    Axis.X)
13:         else
14:             wallsToSplit ← vektorMap.get(Axis.Z)
15:             newWall ← HalfEdgeOperations.split(wallsToSplit, shape.getHalfEdgeDS())
16:             rightChild ← new Leaf(newRoomCreation(newWall.getShape()), Axis.Z)
17:             leftChild ← new Leaf(newRoomCreation(newWall.getShape().getOpposite()),
    Axis.Z)
18:         end if
19:         lastSplit ← splitDirection
20:         return true
21:     else
22:         throw new IllegalArgumentException("Leaf can not be split in this Direction")
23:     end if
24: end function
```

Die Methode *splitLeaf* beinhaltent die Hilfsmethode **split** Algorithmus 10 damit die Teilung gemacht werden kann. Als Ergebnis liefert diese Methode einen booleschen Wert, der angibt, ob die Teilung erfolgreich durchgeführt werden konnte.

Algorithm 10 split

function SPLIT($edgesToSplit, dataStructure$)
  $newVertices \leftarrow$ empty List of HalfEdgeVertex
  $newEdges \leftarrow$ empty List of HalfEdge
  $newEdge \leftarrow$ null
  $newEdge2 \leftarrow$ null
  for $w \in edgesToSplit$ **do**
    $he \leftarrow w$.getShape()
    $va_{\leftarrow}$createVertexHalfEdgeSplit(he)
    $newEdge \leftarrow$ createNewHalfEdge(va,he.getNext())
    $newEdge2 \leftarrow$ createNewHalfEdge(he.getStartVertex(), $newEdge$)
    $dataStructure$.add($newEdge$)
    $dataStructure$.add($newEdge2$)
    $newEdges$.add($newEdge$)
    $newEdges$.add($newEdge2$)
    $newVertices$.add($va_{)}$
    correctHalfEdgeConnection($newEdge2, he$)
    $dataStructure$.remove(he)
  end for
  $newInnerWall \leftarrow$ createInnerWall($newVertices, newEdges, dataStructure$)
  return $newInnerWall$
end function

Die oben beschriebene Methode **split** teilt die gegebenen Kanten und erstellt eine neue Kante an der Stelle der Teilung. Dadurch wird ein Raum in zwei neue Räume aufgeteilt.

Algorithm 11 AssignRules

```
1:  procedure AssignRules(rooms, rulesList, doneRulesRoomMap)
2:      for rule in rulesList do
3:          canDo ← False
4:          neighbourTypes ← rule.getPossibleNeighbours()
5:          for r in rooms do
6:              if r.getType() ≠ Null & neighbourTypes.contains(r.getType()) then
7:                  canDo ← True
8:                  break
9:              end if
10:         end for
11:         if allTyped(rooms) then
12:             break
13:         end if
14:         doneSize ← doneRulesRoomMap.size()+1
15:         neighbourCheck ← False
16:         sizeCheck ← False
17:         list ← rule.getPossibleNeighbours()
18:         if canDo & !rule.isDoneRule() then
19:             checkRoomsForRule(rooms, doneRulesRoomMap, rule, neighbourCheck, sizeCheck, list)
20:             if doneSize ≠ doneRulesRoomMap.size() then
21:                 correctionProcessor ← CorrectionProcessor(rooms, rulesList)
22:                 correctionProcessor.fixTyping(doneRulesRoomMap, rule)
23:             end if
24:         end if
25:     end for
26: end procedure
```

Diese Methode ist eine Hilfsmethode aus dem Pseudocode 2. Hier werden die Regeln verarbeitet und den Räumen zugewiesen. In diesem Algorithmus gibt es auch eine Hilfsmethode in der Zeile 19. Diese Methode prüft, ob eine Regel für irgendwelchen Raum passt, hier werden Prüfungen für die Raumgröße und die Nachbarschaft gemacht.

Algorithm 12 checkRooms($roomsToCheck, rule$)

```
1:  function checkRooms(roomsToCheck, rule)
2:      check ← false
3:      nullCheck ← true
4:      typedCheck ← true
5:      for room in roomsToCheck do
6:          if room.type is null then
7:              check ← rulesChecker(room, roomsToCheck, rule)
8:              if check then
9:                  remove(allRooms, room)
10:                 room.type ← rule.neededType
11:                 add(allRooms, room)
12:                 break
13:             else
14:                 nullCheck ← false
15:             end if
16:         end if
17:     end for
18:     if !nullCheck then
19:         for typedRoom in roomsToCheck do
20:             if typedRoom.type is not null then
21:                 check ← rulesChecker(typedRoom, roomsToCheck, rule)
22:                 if check then
23:                     typedCheck ← true
24:                     for removeType in allRooms do
25:                         if rule.neededType is equal to removeType.type then
26:                             removeType.type ← null
27:                         end if
28:                     end for
29:                     oldType ← typedRoom.type
30:                     remove(allRooms, typedRoom)
31:                     typedRoom.type ← rule.neededType
32:                     add(allRooms, typedRoom)
33:                     roomsToRecheck ← copy(allRooms)
34:                     neededRule ← null
```

```
35:                     for typeRule in doneRuleList do
36:                         if typeRule.neededType is equal to oldType then
37:                             neededRule ← typeRule
38:                             break
39:                         end if
40:                     end for
41:                     checkRooms(roomsToRecheck, neededRule)
42:                     break
43:                 else
44:                     typedCheck ← false
45:                 end if
46:             end if
47:         end for
48:     end if
49:     return typedCheck
50: end function
```

Die oben beschriebene Methode ist eine Hilfsmethode von dem Korrekturprozess bei der Raumtypisierung. Sie prüft, ob es bei dem Korrekturprozess einen Raum gibt, der zu einer entsprechenden Regel passt. Wenn ein solcher Raum existiert, wird er in dieser Methode entsprechend typisiert. Diese Methode ist eine Hilfsmethode für *fixTyping* (Pseudocode 3). Hier wird versucht die ausgewählte Regel dem anderen Raum zuzuweisen. In der Zeile 21 wird noch eine Methode aufgerufen *rulesChecker*. Diese Methode prüft, ob eine Regel zum ausgewählten Raum passt.

Algorithm 13 rotateChild(neededFurniture, parentDir, sideDir, rotation)

```
1: function rotateChild(neededFurniture, parentDir, sideDir, rotation)
2:     if checkDir(parentDir, neededFurniture.getDirVector()) then
3:         if parentDir.getZ() < 0 & (sideDir.getZ() > 0 or sideDir.getX() > 0) then
4:             rotation.fromAngleAxis(FastMath.HALF_PI, Vector3f.UNIT_Y)
5:         else if parentDir.getZ() > 0 & (sideDir.getZ() < 0 or sideDir.getX() < 0) then
6:             rotation.fromAngleAxis(FastMath.HALF_PI, Vector3f.UNIT_Y)
7:         else if parentDir.getX() > 0 & (sideDir.getX() < 0 or sideDir.getZ() < 0) then
8:             rotation.fromAngleAxis(-FastMath.HALF_PI, Vector3f.UNIT_Y)
9:         else if parentDir.getX() > 0 & (sideDir.getX() > 0 or sideDir.getZ() > 0) then
10:            rotation.fromAngleAxis(FastMath.HALF_PI, Vector3f.UNIT_Y)
11:        else
12:            rotation.fromAngleAxis(-FastMath.HALF_PI, Vector3f.UNIT_Y)
13:        end if
14:    else
15:        if checkAngleValue(parentDir, sideDir) == FastMath.HALF_PI & sideDir.getX()
    > 0 then
16:            rotation.fromAngleAxis(-FastMath.HALF_PI, Vector3f(0, 1, 0))
17:        else if checkAngleValue(parentDir, sideDir) == FastMath.HALF_PI & side-
    Dir.getX() < 0 then
18:            rotation.fromAngleAxis(FastMath.HALF_PI, Vector3f(0, 1, 0))
19:        end if
20:        if checkAngleValue(parentDir, sideDir) == FastMath.HALF_PI & sideDir.getZ()
    > 0 then
21:            rotation.fromAngleAxis(-FastMath.HALF_PI, Vector3f(0, 1, 0))
22:        else if checkAngleValue(parentDir, sideDir) == FastMath.HALF_PI & side-
    Dir.getZ() < 0 then
23:            rotation.fromAngleAxis(FastMath.HALF_PI, Vector3f(0, 1, 0))
24:        end if
25:    end if
26: end function
```

Die Methode **rotateChild** zeigt, wie das Kindermöbelstück rotiert werden soll, wenn es um ein Elternmöbelstück herum platziert wird und in die Richtung des Elternmöbelstücks ausgerichtet werden soll.

Algorithm 14 separationSteeringOptimization

```
1:  function SEPARATIONSTEERINGOPTIMIZATION(rules, allObjects)
2:      for i ← 0 to boidIteration do
3:          for each object in allObjects do
4:              r ← GETRULEBYOBJECT(object, rules)
5:              if !r.isInCorner() then
6:                  cohesion ← COHESION(object, allObjects)
7:                  separation ← SEPARATION(object, allObjects)
8:                  boidVector ← cohesion + separation
9:                  SEEK(object, boidVector, r, boundaries)
10:             end if
11:         end for
12:     end for
13: end function
```

Die oben beschriebene Methode zeigt den Prozess der Positionsoptimierung der Möbel. In der Zeilen 6 und 7 werden die Vektoren für Zusammenhalt und Trennung berechnet. Die Summe diese beiden Vektoren wird dann benutzt, um die Möbeln näher zu entsprechend optimierten Position zu bewegen (Zeile 9 Methode Seek).

Algorithm 15 placeNeededObjectsRandomly

```
1:  procedure PLACENEEDEDOBJECTSRANDOMLY(rules, floorRooms, mapOfRoomVertices, neededObjects, allObjects)
2:      for rule in rules do
3:          boundaries ← getAllBoundaries(getNeededRoom(floorRooms, rule))
4:          if !rule.isDetailObject() then
5:              neededFurniture ← neededObjects.get(rule.getName())
6:              if rule.isCreatesNoCollision() then
7:                  neededFurniture.setNoCollision(true)
8:              end if
9:              if !rule.isCreatesNoCollision() & !rule.isDetailObject() & !rule.isPaar() then
10:                 room ← getNeededRoom(floorRooms, rule)
11:                 randomPlacement(rules, allObjects, rule, neededFurniture, getAllBoundaries(room), mapOfRoomVertices.get(rule.getRoomType()))
12:             end if
13:         end if
14:     end for
15: end procedure
```

Diese Methode ermöglicht die zufällige Platzierung der Möbel am Anfang. Dafür wird die Hilfsmethode **randomPlacement** verwendet, die das Möbelstück in eine zufällige Richtung bewegt, bis es keine Kollisionen mehr gibt.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

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

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

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

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

Ein Prozedurales Regelsystem zur Generierung und Positionierung von Räumen und Möbeln in Gebäudehüllen

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

_____	_____	_____
Ort	Datum	Unterschrift im Original