HAW HAMBURG

BACHELOR THESIS
Julius Held

Prozedurale Generierung von texturierten 3D-Modellen mit einer Form-Grammatik

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Julius Held

Prozedurale Generierung von texturierten 3D-Modellen mit einer Form-Grammatik

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

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr.-Ing. Lars Hamann

Eingereicht am: 13. Dezember 2024

Julius Held

Thema der Arbeit

Prozedurale Generierung von texturierten 3D-Modellen mit einer Form-Grammatik

Stichworte

Prozedurale Generierung, Form-Grammatik, Texturen

Kurzzusammenfassung

In dieser Arbeit wird eine für Forschungszwecke entwickelte Form-Grammatik zur prozeduralen Generierung von Gebäuden so erweitert, dass es möglich ist, jeder Fassade Texturen zuzuweisen, welche ebenfalls Prozedural generiert werden. Hierfür werden mögliche Nutzeranforderungen und eine hierfür geeignete Syntax ermittelt. Des Weiteren werden die für die Generierung notwendigen Algorithmen entwickelt und es wird sich mit den Möglichkeiten der Hardwarebeschleunigung (GPU) auseinandergesetzt.

Julius Held

Title of Thesis

Procedural generation of textured 3d models with a shape-grammar

Keywords

Procedural generation, shape-grammar, textures

Abstract

In this thesis we further develop a shape-grammar (designed for research purposes) which can procedurally generate buildings, to support the assignment of procedurally generated textures to individual facades. We determine the requirements of potential users and develop a syntax to satisfy those. Furthermore we develop the required algorithms for our generation tasks and we explore the possibilities of hardware acceleration (GPU).

Inhaltsverzeichnis

Tabellenverzeichnis

1 Einleitung

Ob bei Computerspielen, in Simulationen oder für die Präsentation bei der Stadtplanung. Realistische und authentisch modellierte Gebäude spielen in vielen Bereichen eine große Rolle. Die Erstellung dieser ist in der Regel mit viel Aufwand verbunden, da sie zum einen mit fortschreitender Technik immer detailreicher werden und es für ein authentisches Gesamtbild eine große Vielfalt an Gebäuden geben muss. Da viele dieser Gebäude einen ähnlichen Aufbau haben, ist es naheliegend, sie von einem Algorithmus generieren zu lassen. Ein beliebter Ansatz hier ist die Generierung von Formen aus einer Form-Grammatik, welche rekursiv den Aufbau von Gebäuden und ihren Fassaden mit beliebigem Detailgrad beschreiben kann. Hierbei ist es möglich, durch Verwendung von Pseudozufallszahlen die generierten Gebäude in ihren Eigenschaften zu randomisieren und somit die gewünschte Vielfalt zu erzeugen.

In dem cgashape-Projekt des „Procedural Content Generation"-Repositorys wird eine solche Formgrammatik und ein 3D-Gebäudegenerator, welcher diese auswerten kann, fortlaufend weiterentwickelt. Bislang wurden hier 3D-Modelle mit schlichten Materialien und ohne Texturen generiert. Zudem erfolgte die Materialvergabe nach fest einprogrammierten Regeln und ließ sich nicht beeinflussen. Als Resultat sahen die generierten Modelle sehr eintönig aus und es ließ sich teilweise beim Betrachten nur schwer die Größe der Gebäude einschätzen.

In dieser Arbeit wird das bestehende System so erweitert, dass bei der Eingabe der Grammatik angegeben werden kann, welche Materialien welcher Fassade zugewiesen werden und welche Texturen hierbei generiert und verwendet werden sollen. Als Resultat entsteht ein Gebäudemodell mit keiner, einer oder mehreren Texturen. Hierfür entwickeln wir die Syntax der Grammatik weiter und entwickeln ebenfalls einen neuen Algorithmus, welcher das generierte Modell in texturierbare Fassaden aufteilt. Zudem werden Shader zum effizienten Generieren der Texturen verwendet, wobei die Einschränkungen dieses Vorgehens beschrieben und für Diese Lösungen gefunden werden. Der Aufbau der Arbeit orientiert sich (nach den Grundlagen) an den Phasen des Softwareentwicklungszyklus.

2 Grundlagen und Stand der Technik

2.1 Form-Grammatik

Eine Form-Grammatik nach G. Stiny 1980 [12] ist eine formale Grammatik, bestehend aus:

- einer endlichen Menge an Formen S
- einer endlichen Menge an Symbolen L
- einer endlichen Menge an Form-Regeln der Form $\alpha \rightarrow \beta$
  - α und β sind Formen aus S mit Punkten *im nachfolgenden als Marker-Punkte bezeichnet* denen jeweils ein Symbol aus L zugeordnet ist
- einer Startform I aus S mit Marker-Punkten denen jeweils ein Symbol aus L zugeordnet ist

Kernelemente der Grammatik sind mit Marker-Punkten markierte Formen (siehe oben). Eine solche Form f_1 wird als Teilform von f_2 bezeichnet, wenn alle Linien von f_1 jeweils auf einer Linie von f_2 liegen (die Eckpunkte müssen nicht aufeinander liegen) und die Menge der Marker-Punkte von f_1 eine Teilmenge der Marker-Punkte von f_2 ist. *Anmerkung: Damit 2 Marker-Punkte identisch sind, müssen sie relativ zu ihrer jeweiligen Form dieselbe Position und dasselbe zugewiesene Symbol haben.*

Um aus einer Form-Grammatik eine Form zu generieren, wird ausgehend von der Startform eine Form-Regel $\alpha \rightarrow \beta$ ausgewählt für die gilt, dass die markierte Form auf ihrer linken Seite (α), transformiert mit einer affinen Transformation τ (Marker-Punkte werden mit derselben transformiert), eine Teilform der aktuellen markierten Form γ (anfangs die Startform) ist. Diese Form-Regel wird nun unter Berücksichtigung von τ angewendet, indem alle mit $\tau(\alpha)$ überlappenden Teillinien aus γ und alle mit τ transformierten

Marker-Punkte von α aus der Menge der Marker-Punkte von γ entfernt werden. Anschließend wird die mit τ transformierte Form β (mit all ihren mit τ transformierten Marker-Punkten) der Form γ hinzugefügt. Dies lässt sich mit folgender Formel ausdrücken:

$\gamma_i + 1 = [\gamma_i - \tau(\alpha)] + \gamma(\beta)$

Auf die produzierte markierte Form können nun nach demselben Prinzip weitere Form-Regeln rekursiv angewendet werden, sofern diese Form markierte Punkte hat. Ist dies nicht der Fall, terminiert die Formgenerierung und die Form ist Teil der von der Grammatik definierten Sprache.

Ergänzung: In ihrer ursprünglichen Definition nach G. Stiny 1971 [14] gibt es keine Formen mit einer Menge von Marker-Punkten, stattdessen wird unterschieden zwischen Terminal-Formen und Marker-Formen, welche nicht aus Terminal-Formen bestehen dürfen. In den Form-Regeln ist jeweils eine von 3 Operationen erlaubt: Eine Marker-Form entfernen, eine Marker-Form ändern, eine Marker-Form ändern und zusätzlich ein oder mehrere neue Terminal-Formen hinzufügen. Das heißt, um auf eine Form eine Form-Regel anwenden zu können, muss diese eine Marker-Form beinhalten. Beinhaltet die Form keine Marker-Form, terminiert die Formgenerierung.

2.1.1 Split-Grammar

Die von Wonka et al. 2003 [16] vorgestellte Split-Grammar ist eine spezialisierte Form der Set-Grammar [13], in welcher die Kernelemente Mengen an Formen sind, auf welchen ausschließlich mit der Mengenvereinigung, der Mengendifferenz, der Teilmengenrelation und der Transformation aller Elemente gearbeitet wird. Elemente dieser Mengen sind sogenannte Standardformen. Diese sind konvexe, dreidimensionale, am Koordinatenursprung zentrierte Formen mit einem zugewiesenen Symbol und Marker-Punkten an den Schnittpunkten der 3 positiven Koordinatenachsen mit der Formoberfläche (siehe Abbildung 2.1).

Bei den Formregel wird unterschieden zwischen (möglicherweise kontextsensitiven) Zerteilungs- und Umwandlungsregeln. Beim Anwenden von Zerteilungsregeln wird die gematchte Form in kleinere Bestandteile (Standardformen) zerlegt, sodass diese das exakt gleiche Volumen wie vor dem Anwenden ausfüllen. In Umwandlungsregeln wird die gematchte Form lediglich in eine andere Standardform umgewandelt, welche nicht zwingend das

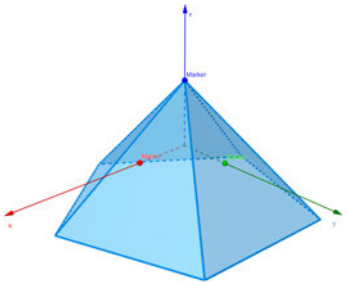

Abbildung 2.1: Beispiel einer Standardform mit Marker-Punkten für jede Achse

gleiche Volumen ausfüllen muss. Das zugewiesene Symbol bleibt in beiden Fällen erhalten.

Anmerkung: In dem oben genannten Paper wurde ebenfalls eine Control-Grammar definiert, welche für unsere Zwecke jedoch nebensächlich ist.

Eine solche Split-Grammar wurde im Rahmen der Bachelorarbeit von Thorben Watzl [15] im Cg-Research-Project (inzwischen transferiert in das „Procedural Content Generation"- [frontend] und das „CG Algorithms Datastructures"- [backend] Repository) implementiert und dient als Grundlage für diese Arbeit. Sie wird angepasst und genutzt, um das texturierte Gebäude zu erstellen.

2.2 Texturen und Texturkoordinaten

Texturen in der Computergrafik sind sehr allgemein gefasst 1-3-dimensionale Hyperquader (meist 2-dimensional also ein Rechteck) bestehend aus einer endlichen Anzahl an sogenannten Texeln (**Tex**ture-**El**ement), in welchen numerische Informationen in einem einheitlichen Format (1-4-dimensionale Vektoren aus Fließkomma- oder Ganzzahlen) gespeichert sind. Aufgrund dieser Eigenschaften ist es sehr effizient, auf Informationen für eine beliebige Textur-Koordinate zuzugreifen und Koordinaten zu identifizieren, welche nicht Teil der Textur sind, um diese Fälle entsprechend zu händeln. Dies ist für Grafikkarten ein großer Vorteil, da alle Threads, welche auf die gleichen Texturen zugreifen, die gleichen Operationen mit unterschiedlichen Daten durchführen und somit der Durchsatz erhöht werden kann.

In der Regel werden Texturen genutzt, um Informationen über die Oberflächenbeschaffenheit, z.B. Farbe und Glanz eines Models zu speichern. Diese Informationen können

dann bei der Darstellung des Modells, mit Hilfe von in den Eckpunkten gespeicherten Texturkoordinaten, abgefragt und für die Lichtberechnung bzw. Ermittlung des darzustellenden Pixel-Farbwerts verwendet werden.

Die Berechnung der Texturkoordinaten für jeden Pixel aus den Texturkoordinaten der Eckpunkte erfolgt über die baryzentrischen Koordinaten innerhalb des zu berechnenden Dreiecks.

Die Texturkoordinaten sind in der Regel „normiert“, das heißt, die einzelnen Komponenten werden durch das Ausmaß der Textur in der jeweiligen Dimension geteilt (für 2-Dimensionen (x/breite, y/höhe)) (siehe Abbildung 2.2).

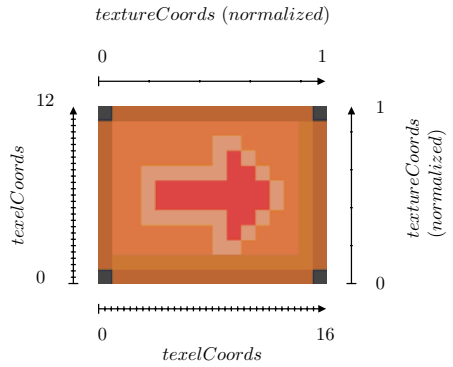

Abbildung 2.2: Verhältnis von normalisierten Texturkoordinaten zu Texel-Koordinaten

Diese „normierten“ Texturkoordinaten liegen jedoch nicht zwingend im Bereich 0-1. Für Werte, die außerhalb dieses Bereiches fallen, gibt es beim Zugriff auf die Textur (dem sog. Sampeln) in der Regel 2 Möglichkeiten: 1. Alle Werte unter 0 werden als 0 betrachtet und alle Werte größer als 1 werden als 1 betrachtet (Clamp). 2. Von allen Werten im Definitionsbereich wird die Differenz zum jeweils abgerundeten Wert berechnet, sodass sich im Wertebereich die Zahlenspanne von 0-1 unendlich wiederholt (Repeat). [10] Je nach Grafikengine bzw. API ist es auch möglich, die Texturkoordinaten so zu wiederholen, dass jede 2. Spanne gespiegelt ist (Mirror repeat) oder dass an dem Koordinatenursprung gespiegelt und bei 1 und -1 geclampt wird (Mirror once) (siehe Abbildung 2.3).

In dieser Arbeit spielen Texturen eine zentrale Rolle, insbesondere Albedo-Maps und Normal-Maps, da eben Diese für alle Fassaden (oder ausgewählte) generiert werden sollen. Albedo/Diffuse-Maps geben die grundlegenden Farben an, welche von der Oberfläche reflektiert werden und Normal-Maps verändern die Normalen-Information (die Richtung der Oberfläche, welche für die Lichtberechnung verwendet wird) so, dass ein Eindruck

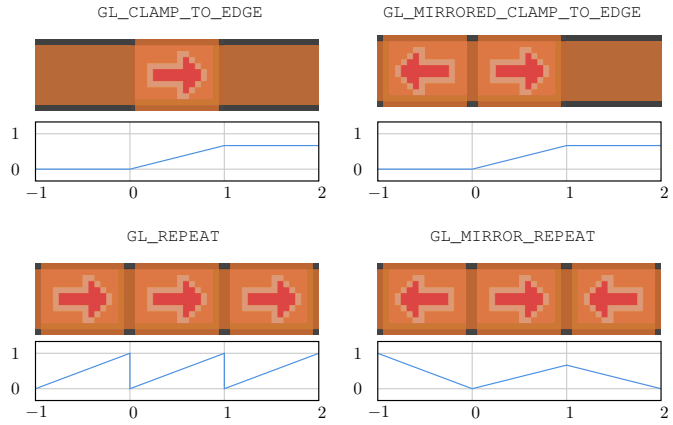

Abbildung 2.3: Veranschaulichung der Texture-Wrap-Modes ab OpenGL 4.4 (`GL_CLAMP_TO_BORDER` wird der Einfachheit halber weggelassen).

von Erhöhung und Vertiefung entsteht. Sie erfüllen somit dieselbe Rolle wie Bump-Maps, jedoch enkodieren sie direkt die Normalen-Information (relativ zur Oberflächennormale).

Ein Verständnis von normierten Texturkoordinaten ist ebenfalls sinnvoll, um den Code des Texturier-Algorithmus zu verstehen, jedoch nicht zwingend, da die Normalisierung erst kurz vor der Generierung der Texturen stattfindet. Bei unserem Vorgehen sind wir nicht auf Wrap-Modes angewiesen, trotzdem erschien es sinnvoll diese im Zuge des Texture-Sampling vorzustellen.

2.3 Shader und die Graphics-Pipeline

Shader sind vorkompilierte Programme, welche auf der Grafikkarte ausgeführt werden. Diese werden unter anderem in der sogenannten Graphics-Pipeline (siehe Abbildung 2.4) genutzt, um einzelne Schritte wie die Positions-/Attribut-Berechnung der Eckpunkte (mit einem Vertex-Shader) und Berechnung des darzustellenden Farbwerts (mit einem Fragment/Pixel-Shader) programmieren zu können.

Die Graphics-Pipeline modelliert alle Schritte zur Darstellung von Geometrie. Vom Einlesen der im Grafikspeicher gespeicherten Daten, bis hin zum Ändern der einzelnen Pixel.

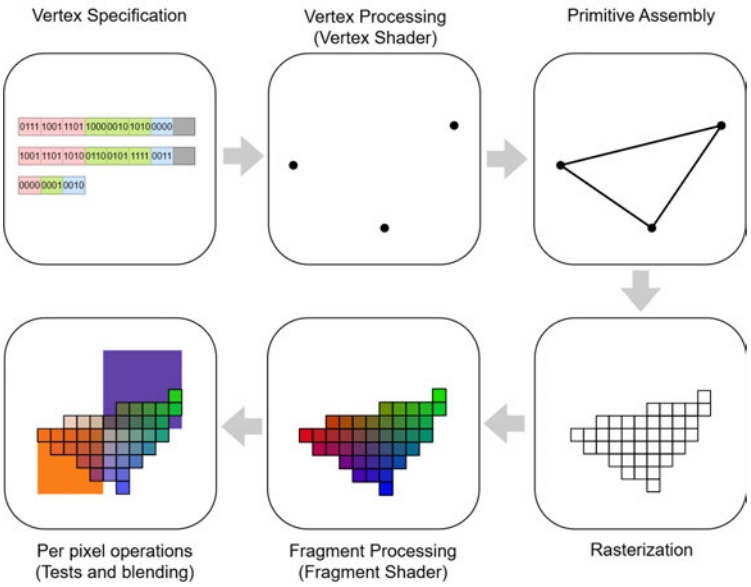

Abbildung 2.4: Die OpenGL Rendering-Pipeline. *Alle optionalen Stages wurden weggelassen.*

Grafik nachempfunden von Mladen Korunoski [9]

In modernen Grafik-APIs muss für jeden Aufruf zum Zeichnen von Geometrie mindestens ein Vertex- und ein Fragment/Pixel-Shader angegeben werden. Diese sind pure Funktionen, welche gedanklich einen Eckpunkt mit Attributen zu einem neuen Eckpunkt mit Attributen bzw. einen Pixel mit den interpolierten Attributen zu einem Pixel mit ein oder mehreren Farbwerten (und Tiefeninformation) transformieren. Für die Programmierung der Shader sind einem sehr viele Freiheiten geboten und es kann mehr oder weniger frei auf Ressourcen wie Texturen und spezielle Puffer zugegriffen werden. Da diese Puffer theoretisch in jedem Frame mit neuen Informationen gefüllt werden können und sich fast alles innerhalb eines Shaders berechnen lässt, ist eine Vielzahl an interessanten visuellen Effekten realisierbar. [7]

Anmerkung: In aktuellen Grafik-APIs ist es zum Teil möglich mit Mesh-Shadern zu arbeiten, mit welchen die Verwendung eines Vertex-Shaders tatsächlich nicht notwendig und auch gar nicht erst möglich ist. [5]

Shader spielen in dieser Arbeit eine tragende Rolle, da sie aufgrund ihrer Parallelität für die spätere Generierung der Texturdetails pro Texel zuständig sind. Es wird pro Fassade mindestens ein Polygon mit einem speziellen Shader auf eine Textur gerendert, welche anschließend ausgelesen wird, um ein Abbild dieser Textur zu erstellen, welches in beiden

Frontends unabhängig von der Grafik-API/-Engine genutzt und später exportiert werden kann.

2.4 Ansätze zur Texturierung von generierten Gebäuden

2.4.1 City-Engine

In der ArcGIS City-Engine [3], welche als Inspiration für die in dieser Arbeit zu erweiternde Form-Grammatik dient, hat jede Fassade ein Material Attribut mit vielen Eigenschaften, darunter auch die Texture-Maps für alle 10 Texturlayer (Albedo-Map, Normal-Map, Roughness-Map etc.). Alle Eigenschaften des Materials können für jede Fassade mittels der „set"-Operation geändert werden. Den Texture-Maps kann eine Bilddatei und den korrespondierenden UV-Sets der Texturelayer 0-9 eine Projektionsmatrix zugewiesen werden. Um eine Projektionsmatrix zuzuweisen wird die „setupProjection"-Operation verwendet in welcher die Ausrichtung und lokale Transformation der projizierten Textur angegeben werden können. Die Textur wird jedoch nur auf Fassaden im generierten Modell angewendet, für welche die Textur-Koordinaten mit Hilfe der „projectUV"-Operation berechnet und dem jeweiligen UV-Set zugewiesen wurden. Es liegt nahe, dass diese Textur-Koordinaten durch die Transformation der Fassaden-Eckpunkte im Worldspace mit der gesetzten Projektionsmatrix des UV-Sets berechnet werden.

Informationen entnommen aus der offiziellen Dokumentation der ArcGIS City-Engine [4]

2.4.2 Neuronale-Netze

Viele moderne Ansätze zur Generierung von texturierten Gebäuden nutzen Neuronale-Netze wie GANs und CNNs, um die Farbinformation der Gebäudetexturen generieren zu lassen.

FrankenGAN

Das von Kelly et al. 2018 [8] vorgestellte FrankenGAN-Framework nutzt eine Vielzahl von Generative Adversarial Networks (GANs), um für alle Fassaden und Dächer eines detailarmen Gebäude-/Stadtmodells jeweils eine hochauflösende Textur und eine detaillierte Geometrie zu generieren. Diese GANs sind jeweils entweder darauf trainiert a) aus

einer Form-Maske (region mask) und einem Außenkanten-Distanzfeld (context info) eine Segmentierung (Label map) zu generieren oder b) aus einer Segmentierung eine hochauflösende Textur zu generieren. Jeder dieser GANs kann mit Style-Parametern parametrisiert werden, welche den Gebäudestil aus Referenzbildern enkodieren. Das Framework ist so aufgebaut, dass teilweise mehrere GANs in Reihe geschaltet werden, um gleichzeitig ein oder mehr Segmentierungen und eine detaillierte Textur für den nächsten Schritt zu generieren (siehe Abbildung 2.5).

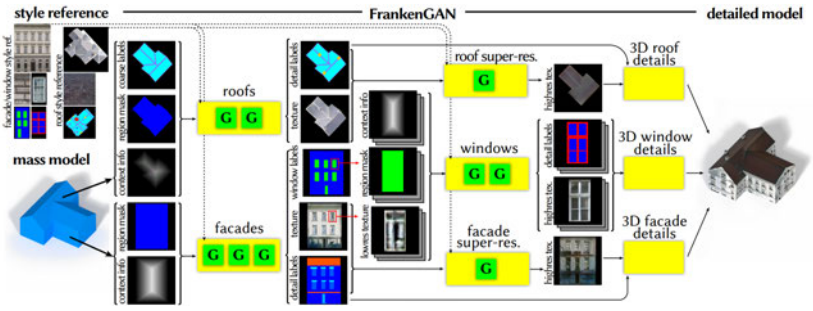

Abbildung 2.5: Skizzierung des Aufbaus des FrankenGAN Frameworks. Die individuellen GANs sind mit G markiert. GAN-Ketten und Geometrie-Generatoren sind als gelbe Boxen dargestellt.

Grafik entnommen aus dem Paper von Kelly et al. 2018 [8]

Anmerkung: Da die generierten Label-Maps Generierungsartefakte aufweisen, müssen diese erst in einem zusätzlichen Schritt mit Domänenspezifischen Algorithmen „normalisiert“ werden.

Projective Urban Texturing

Das von Georgiou et al. 2021 [6] vorgestellte Projective Urban Texturing verfolgt einen anderen Ansatz. Hier wird ein Faltungsnetz (**C**onvolutional **N**eural **N**etwork) verwendet, welches mit 360° Street-View Panoramen der Stadt trainiert werden muss, deren Stil nachgeahmt werden soll. Das Faltungsnetz ist so aufgebaut, dass es in der Lage ist, ein teilweise oder auch gar nicht texturiertes Straßenpanorama vollständig in dem trainierten Stil zu texturieren.

Im vorgestellten Algorithmus wird mit Hilfe dieses Faltungsnetzes ein Stadtmodell mit einem Texturatlas iterativ texturiert. Hierbei wird zuerst im Modellierprogramm blender eine Szene mit einem (bislang untexturierten) Modell der Stadt geladen. In dieser

wird nun eine Panoramakamera mit einer äquirektangulären Projektion an dem ersten Standpunkt (siehe unten) platziert und anschließend mit dieser mittels Pathtracing ein Panoramabild gerendert. Das Faltungsnetz bekommt dieses teilweise texturierte Bild als Eingabe und erstellt hieraus ein vollständig texturiertes Bild. Im nächsten Schritt wird für jeden Pixel im Texturatlas die gemappte Position in der Stadtgeometrie berechnet, welche anschließend mit Hilfe derselben Projektion wie die der Kamera in blender projiziert wird, um so die Pixelposition im gerenderten Bild zu ermitteln. Der Farbwert an dieser Position in dem von der KI texturierten Bild wird ausgelesen und mit einem von der Entfernung zur Kamera abhängigen Faktor multipliziert auf den vorhandenen Farbwert im Atlas addiert. Dieser nun Atlas wird in die Szene geladen, die Kamera wird zum nächsten Standpunkt (siehe unten) bewegt und alle Schritte ab dem Rendern des Panoramabilds (nun mit einem teilweise texturierten Stadtmodell) werden wiederholt. Der Algorithmus terminiert sobald alle Standpunkte (siehe unten) durchgegangen wurden.

Anmerkung: Die Standpunkte werden (entlang der Mittellinie) in jeder Straße in einem Abstand von 5m zueinander und in einer Höhe von 2,5m (ungefähre Höhe der Street-View Kamera) platziert.

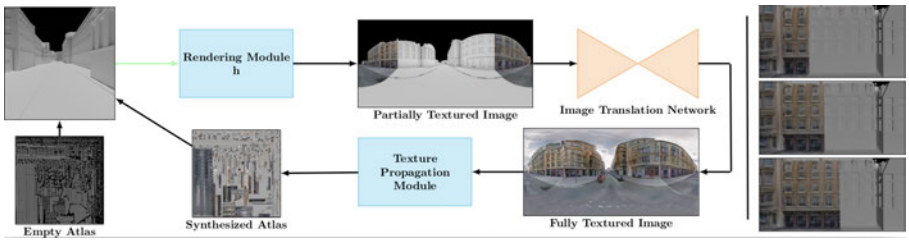

Abbildung 2.6: Aufbau der Pipeline des Projective Urban Texturing
Grafik entnommen aus dem Paper von Georgiou et al. 2021[6]

3 Anforderungsanalyse und Spezifikation

3.1 Analyse des bestehenden Systems

Bei dem bestehenden System handelt es sich um eine Anwendung mit einer schlichten grafischen Benutzeroberfläche bestehend aus einem 3D-Viewport und einem Seitenbereich zum Einstellen der Parameter. Der/die Nutzer/in hat die Möglichkeit, eine Grundform für das zu generierende Gebäude anzugeben (entweder als Text oder mit einem grafischen Editor, welcher sich auf Knopfdruck öffnen lässt). Des Weiteren kann er/sie die Form-Grammatik (ausschließlich in textueller Form), sowie einen optionalen Startwert (seed) für den Zufallsgenerator angeben. Nach dem Einstellen der Parameter kann der/die Nutzer/in mit Klicken des „Update"-Buttons sich ein Gebäudemodell aus der angegebenen Form-Grammatik generieren und anzeigen lassen. Das generierte Modell hat keine Texturen und im Allgemeinen wenig Farbvariation.

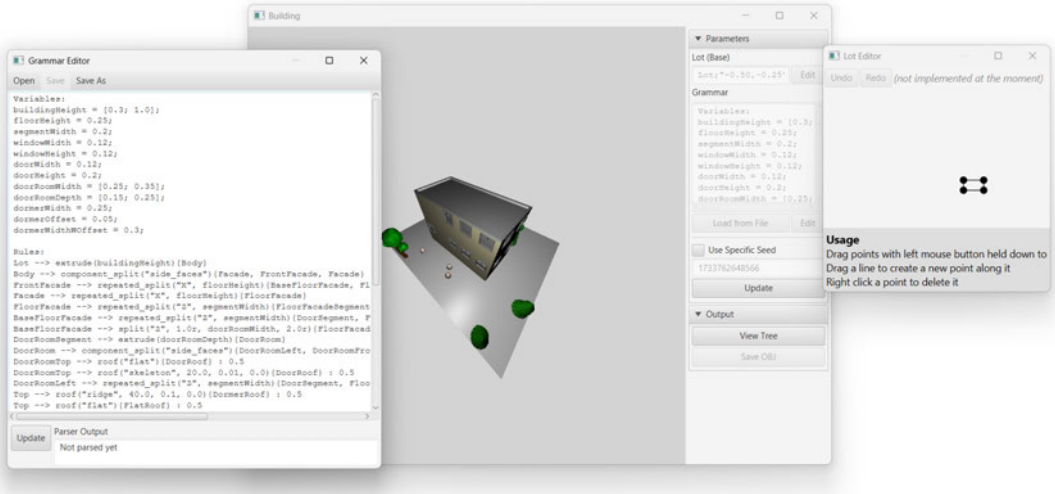

Abbildung 3.1: Screenshot der Benutzeroberfläche des bestehenden Systems

3.1.1 Analyse der Form-Grammatik

In der Form-Grammatik ist es möglich, eine Liste von Variablen zu definieren und diese jeweils mit einer festen Zahl oder einem Zufallswert zu initialisieren. Für Letzteres gibt der/die Nutzer/in eine Unter- und Obergrenze an. Darauf folgend werden die Ableitungsregeln definiert, bestehend aus einem Symbol und einer Reihe an Operation, welche nacheinander auf die Formen angewendet werden, welchen dieses Symbol in einer Ableitung zugewiesen wurde. In diesen Operationen ist es möglich, eine Zuweisung der Symbole für die neu entstandenen Formen anzugeben. Des Weiteren können in den Ableitungsregeln Aktionen, wie das Erhöhen des Wertes einer Variable, durchgeführt werden. Es ist möglich, mehr als eine Ableitungsregel anzugeben, diese müssen jedoch jeweils mit einem Wahrscheinlichkeitsfaktor versehen werden (in Summe 1). Als Startsymbol/Axiom wird der Name verwendet, welchen man in der textuellen Form der Gebäudegrundform angeben hat.

3.2 Anforderungsanalyse

In der ersten Phase des Softwareentwicklungszyklus, der Anforderungsanalyse, ermitteln wir, welche Anforderungen potentielle Nutzer/innen, bezogen auf die Iteration (Texturierte Gebäude), an das System haben. Hierfür formulieren wir User-Stories, welche die Wünsche und Ziele der Nutzer/innen offenlegen, um eine konkrete Vorstellung zu haben, welche Features und Änderungen für die Nutzer/innen relevant sind. Die nachfolgenden User-Stories wurden nicht von Nutzer/innen verfasst und dienen lediglich dazu die Richtung dieser Arbeit zu lenken. *Der komplette Umfang der Arbeit stand zu diesem Zeitpunkt bereits fest.* Trotzdem halte ich sie für authentische Anforderungen von potentiellen Nutzern, welche diese Anwendung zum Erstellen von Gebäuden in einem Kreativen Bereich, wie beispielsweise Videospielentwicklung, nutzen wollen:

US-1: Als Nutzer kann ich Gebäude mit Texturen generieren.
US-2: Als Nutzer kann ich in der Form-Grammatik jeder Fassade eine eigene Textur zuweisen
US-3: Als Nutzer kann ich die Farben, die Ausmaße und ggf. weitere Eigenschaften der Texturdetails (z.B. der Backsteine und des Mörtels) nach Belieben meinen Anforderungen anpassen.
US-4: Die Texturierung des generierten Modells zeigt keine auffälligen Wiederholungen oder Nahten.
US-5: Das generierte Modell ist in einem gängigen Modell-Format exportierbar, um es in anderen Applikationen verwenden zu können. Akzeptanztest: Ein generiertes texturiertes Modell der angepassten Grammatik house.grammar kann in Blender importiert werden und ist optisch, bis auf die Beleuchtung, identisch mit der Vorschau im Softwaresystem.
US-6: Texturen sind optional und können inkrementell in bestehende Grammatiken integriert werden **Akzeptanztest:** Die Vorschau für ein generiertes Modell der Grammatik house.grammar mit dem gleichen Startwert und gleicher Grundform ist identisch in beiden Versionen des Systems (vor und nach der Iteration)

Aus **US-3** und **US-4** geht hervor, dass wir keine vordefinierten Texturen verwenden können, da diese nur bedingt anpassbar sind und auffällige Kachelung und Nahtstellen sich nur mit großem Aufwand vermeiden lassen. Somit werden wir die Texturen für jedes Modell zur Laufzeit generieren.

3.3 Spezifikation - Anwendungsfälle

Aus den oben genannten User-Stories leiten wir nun einen oder mehrere Anwendungsfälle ab, welche diese Wünsche und Ziele so gut wie möglich erfüllen. Grob lassen sich die User-Stories in 3 Kategorien aufteilen: Parametrisierung (US-2, US-3), Endergebnis (US-1, US-4, US-5) und Rückwärtskompatibilität (US-6). Man könnte nun Argumentieren, dass die Generierung, also die Erzeugung des Endergebnisses, unabhängig von der Parametrisierung jederzeit durchgeführt werden kann und ohne Angabe eines Startwerts

(random seed) stochastisch immer zu einem anderen Ergebnis führen wird (sofern die Form-Grammatik es zulässt), sodass es sich bei Parametrisierung und Generierung um 2 Anwendungsfälle handelt. Umgekehrt würde ich jedoch argumentieren, dass die Parametrisierung ohne ein Endergebnis keinen Mehrwert für den/die Nutzer/in bietet und somit nicht als Anwendungsfall gesondert betrachtet werden sollte. Zu guter Letzt ist wichtig anzumerken, dass sich hinter US-5 ein weiterer Anwendungsfall verbirgt: Das Exportieren des Modells muss ebenfalls von dem System übernommen werden, mit dem/der Nutzer/in als Initiator. Somit gibt es in dieser Iteration genau 2 Anwendungsfälle:

UC-1 - Ein texturiertes Modell generieren lassen

Akteur: Nutzer

Ziel: Es soll ein texturiertes Modell von einem Gebäude generiert werden

Vorbedingung: Keine

Nachbedingung: Die Graphische Benutzeroberfläche zeigt ein 3D-Modell an, welches mit wenig Aufwand exportiert werden kann

Erfolgsszenario:

1. Der Nutzer gibt die Grundform des Gebäudes an (textuell oder mit dem Grafischen Form-Editor)
2. Der Nutzer gibt die Grammatik an. In dieser gibt er zusätzlich die zu verwenden Texturen/Materialien und ihre Parameter an
3. Der Nutzer drückt den Update-Knopf
4. Das System interpretiert die angegebene Grammatik (inklusive der Parameter der zu verwendenden Texturen)
5. Das System erstellt das Modell (untexturiert)
6. Das System analysiert das Modell und generiert der Modell-Oberfläche angepasste Texturen mit den angegebenen Parametern
7. Das System wendet die Texturen auf das Modell an und präsentiert das fertige Modell im Viewport

Zugehörige Anforderungen: US-1, US-2, US-3

Die Möglichkeit, eine existierende Grammatik zu laden, wird hier der Einfachheit halber nicht betrachtet, man würde dies jedoch als Erweiterungsfall aufschreiben. Die Fehlerfälle (ungültige Grundform-Syntax, ungültige Grammatik) sind bereits von dem existierenden System abgedeckt und werden somit ebenfalls nicht aufgeführt.

UC-2 - Das generierte Modell exportieren

Akteur: Nutzer

Ziel: Das generierte Modell soll in einem externen Programm nutzbar sein

Vorbedingung: Es wurde ein Modell generiert (in derselben Sitzung) und dieses wurde noch nicht durch ein Anderes überschrieben

Nachbedingung: Im gewählten Verzeichnis (auf dem gewählten Datenträger) existiert eine Modell-Datei zusammen mit allen notwendigen Texturen als Bilddateien, welches sich in dem Modellierprogramm Blender importieren lässt (siehe Akzeptanztest von US-5)

Erfolgsszenario:

1. Der Nutzer drückt auf den Exportieren-Knopf
2. Der Nutzer wählt in einem Systemdialog das Verzeichnis zum Speichern
3. Das System schreibt/kopiert die Modell-Datei und die Texturen in das angegebene Verzeichnis

Zugehörige Anforderung: US-5

3.4 Spezifikation - Textur-Generatoren

Um dem/der Nutzer/in eine sinnvolle Auswahl zu bieten, werden in dieser Iteration 3 Textur-Generatoren spezifiziert und implementiert. Diese haben zwar einen vorgesehenen Verwendungszweck, bieten jedoch auch allesamt umfangreiche Anpassungsmöglichkeiten:

3.4.1 Bricks (Backsteine)

Die Backsteintextur besteht aus mehreren Reihen an zufällig platzierten Backsteinen. Bei geraden Reihen wird als erstes ein Backstein mit der Breite A platziert, bei allen

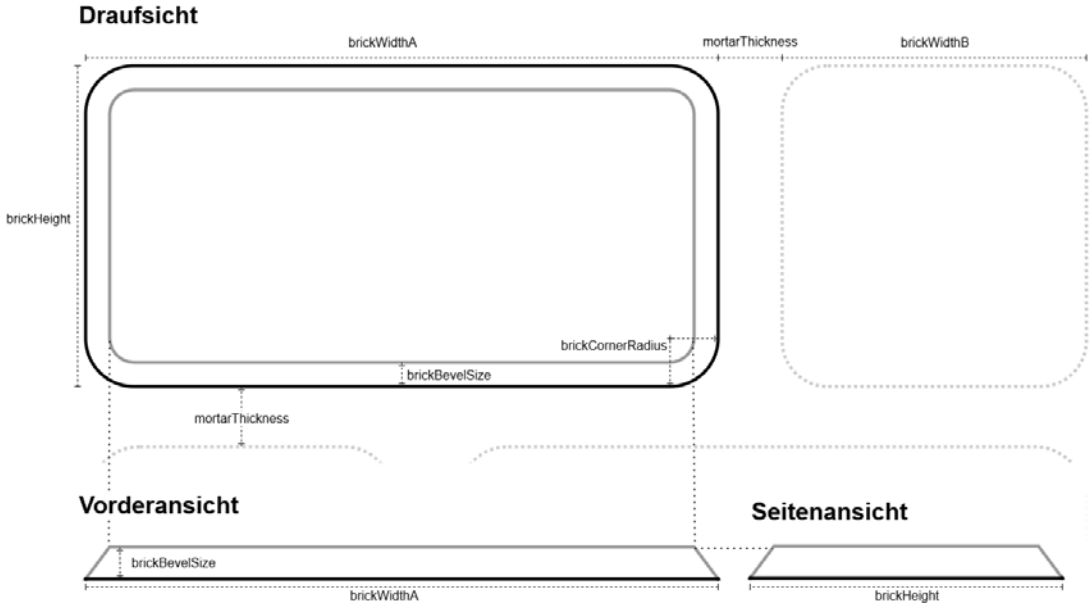

Abbildung 3.2: Spezifikation des Aussehens einer Backstein-Textur

ungeraden ein Backstein der Breite (A+B)/2. Alle darauf folgenden Backsteine in der Reihe haben jeweils entweder Breite A oder Breite B. Die Backsteine werden mit einem festen Abstand, der Fugendicke, zueinander platziert. Der letzte Stein in der Reihe wird ab der vorgegebenen Länge der Reihe abgeschnitten. Zwischen den Reihen wird ebenfalls eine Fugendicke Platz gelassen. Jeder Backstein ist an den Ecken abgerundet und hat eine Abschrägung an der Kante. Die Farbe jedes Backsteins ist zufällig bis zu 20% dunkler als die angegebene Farbe. Der Bereich zwischen den Backsteinen wird mit der angegebenen Fugen-Farbe gefüllt. Am Rand der Textur werden die Backsteine abgeschrägt.

Parameter	Standardwert	Beschreibung
brickWidthA	0.1	Die minimale Breite die ein Backstein einnehmen kann
brickWidthB	0.2	Die maximale Breite die ein Backstein einnehmen kann
brickHeight	0.1	Die Höhe der einzelnen Backsteine
brickCornerRadius	0.01	Radius der abgerundeten Ecken der Backsteine
brickBevelWidth	0.008	Breite der Abschrägung am Rand der Backsteine
brickColor	rgb(0.75, 0.35, 0.35)	Farbe der Backsteine
mortarColor	rgb(0.6, 0.6, 0.6)	Farbe der Fugen
mortarThickness	0.008	Dicke der Fugen/Abstand zwischen Backsteinen

3.4.2 RoofTiles (Dachziegel)

Die Dachpfannen-Textur besteht aus mehreren in einem Gitter angeordneten angewinkelten Dachpfannen ohne eine sichtbare Naht, mit jeweils einer Erhöhung rechts und einer Abrundung am unteren Ende. Am Rand der Textur werden die Dachpfannen abgeschrägt.

Parameter	Standardwert	Beschreibung
tileWidth	0.2	Breite der Dachpfannen
tileHeight	0.2	Höhe/Länge der Dachpfannen
tileThickness	0.08	Die Dicke der Dachpfannen (beeinflusst den Neigewinkel)
color	rgb(0.3, 0.3, 0.3)	Farbe der Dachpfannen
bumpWidth	0.08	Breite der wahrnehmbaren Erhöhung
bumpHeight	0.04	Höhe der Erhöhung
bumpFilletWidth	0.04	Breite der (konkaven) Kurve am Fuß der Erhöhung

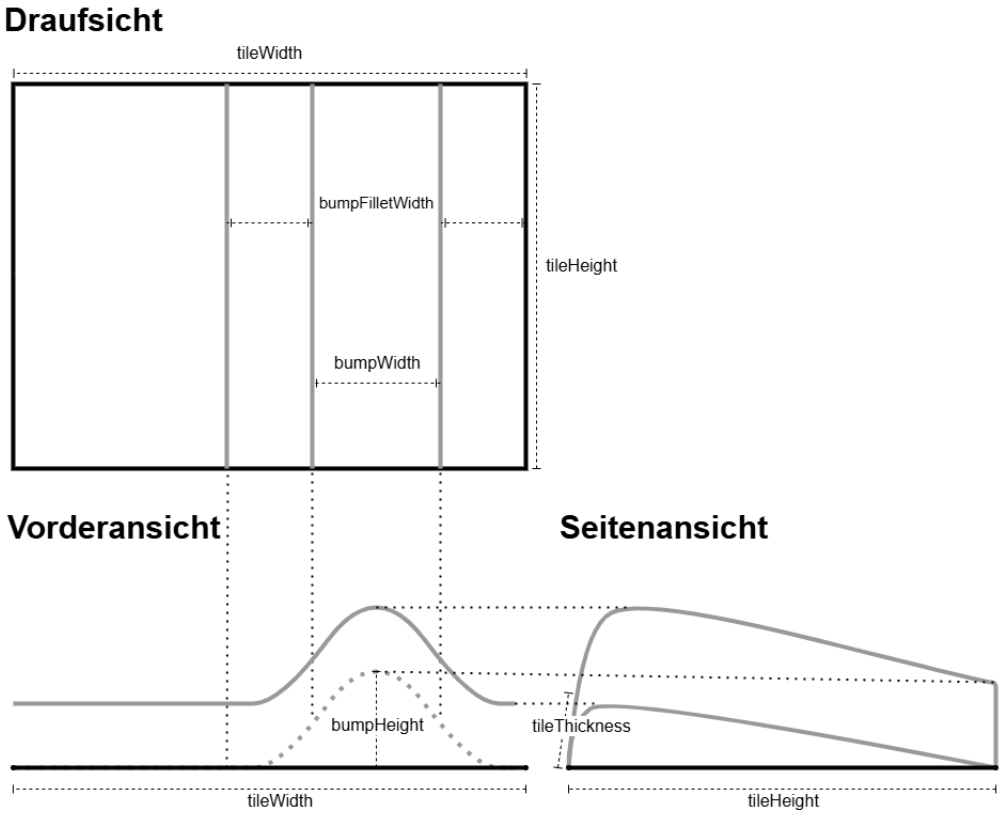

Abbildung 3.3: Spezifikation des Aussehens einer Dachziegel-Textur

3.4.3 Pebbles (Kieselsteine)

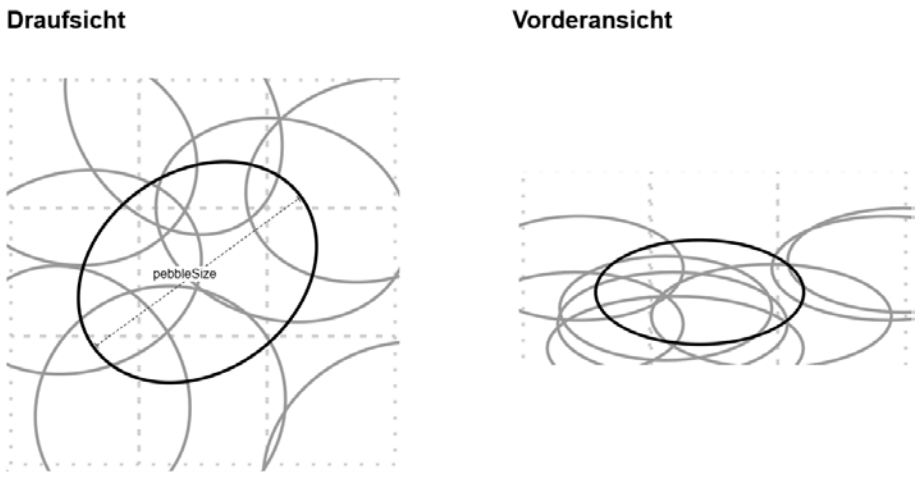

Abbildung 3.4: Best-effort Veranschaulichung des Konzepts der Kieselstein-Textur

Die Kieselstein-Textur besteht aus einem gedanklichen Gitternetz, in welchem pro Zelle ein Kieselstein mit einer zufälligen Breite und Rotation in einer zufälligen Tiefe platziert wurde. Der Kieselstein selbst ist ein Ellipsoid mit einer angegebenen Höhe und Länge und einer Breite, die zwischen 80% und 100% der Länge liegt. Die Größe der Gitterzellen entspricht der angegebenen Kieselsteingröße, sodass ein Kieselstein nie mit 2 Zellen überlappen kann, welche nicht benachbart sind. Die Farbe der Kieselsteine variiert zwischen den angegebenen Farben und wird zudem dunkler, je tiefer der Stein liegt, um Beschattung nachzuahmen. Am Rand der Textur werden die Kieselsteine, welche nicht in der „Tiefe“ liegen, minimal abgeschrägt und es wird eine weiche, schwarze Umrandung gezeichnet, welche Ambient Occlusion nachahmt.

Parameter	**Standardwert**	**Beschreibung**
colorA	rgb(0.5, 0.5, 0.5)	Farbe der Kieselsteine
colorB	rgb(0.5, 0.5, 0.5)	Farbe der Kieselsteine [Variation]
pebbleSize	0.04	Kieselsteingröße
pebbleThickness	0.03	Die „Dicke“ der Kieselsteine (Der Wert wird standardmäßig an die Kieselsteingröße angepasst)
maxDepth	0.08	Maximale Tiefe, in welcher die Kieselsteine platziert sein können (Der Wert wird standardmäßig an die Kieselsteingröße angepasst)
rotateScaleVariation	1	Variation in der Rotation und Skalierung der einzelnen Kieselsteine (Wertebereich: 0-1)
positionVariation	1	Variation in der Positionierung der Kieselsteine relativ zu einem Gitter (Wertebereich: 0-1)

3.4.4 SolidColor

Zusätzlich zu den oben definierten Texturgeneratoren macht es ebenfalls Sinn einen Texturgenerator zu implementieren, welcher uns eine einfarbige Textur liefert um Rückwärts

kompatibel zu bleiben und um dem/der Nutzer/in die Möglichkeit zu geben, Teile des Modells untexturiert zu lassen, wenn eine aufwendige Texturierung nicht notwendig ist.

3.5 Spezifikation - Anpassung der Form-Grammatik

Für diese Iteration sind keine Änderungen der GUI vorgesehen, sodass keine Notwendigkeit besteht, Wireframes zu entwerfen. Jedoch stellt auch die Form-Grammatik in dieser Anwendung eine nicht unbedeutende Benutzerschnittstelle dar. Es ist also wichtig, die Änderungen dieser ebenfalls zu spezifizieren und hierbei die Anforderungen der Nutzer/innen zu berücksichtigen

3.5.1 Wahl der Syntax

Der Use-Case UC-1 gibt den Nutzer/innen die Möglichkeit, in der Form-Grammatik die Texturen und ihre jeweiligen Parameter anzugeben. Hierfür gibt es mehrere mögliche Umsetzungen:

1. Der/die Nutzer/in definiert, in einem gesonderten Bereich der Form-Grammatik, Materialien mit Angabe der Parameter und der Art der Textur, welche dann in Form eines Operators in den Produktionen zugewiesen werden können
2. Das System definiert eine Vielzahl an voreingestellten Materialien, welche wie bei 1. in Form eines Operators zugewiesen werden können, in welchem man zusätzlich bei Bedarf die Parameter überschreiben kann
3. Kombination aus 1 und 2: Der/die Nutzer/in gibt definiert die Materialien und kann Parameter bei Bedarf in den Operatoren überschreiben
4. Der/die Nutzer/in gibt alles in den Produktionen direkt an (als Operator oder mit eigener Syntax)

Anmerkung: Bei allen Umsetzungen, in welchen der/die Nutzer/in die Materialien selbst konfigurieren muss, legt das System Standardwerte für alle Parameter fest und berechnet diese, wenn möglich, aus angegebenen Parametern neu, um Mehraufwand für den/die Nutzer/in und Redundanzen zu vermeiden und „fast prototyping“ zu ermöglichen.

Die Möglichkeiten 2, 3 und 4 haben den Vorteil, dass der/die Nutzer/in direkt die Parameter angeben und sie auch auf einen Blick sehen kann, im Gegensatz zu Möglichkeit 1, bei welcher die Parameter ausschließlich im Variablen-Bereich definiert werden, welcher getrennt von den Ableitungsregeln ist und separat betrachtet und bei Bedarf um neue Materialien erweitert werden muss.

Möglichkeit 4 gibt dem/der Nutzer/in eine Freiheit, welche in den meisten Fällen eher wenig von Nutzen ist und eher zur Last fällt (unter anderem durch mehr Schreibaufwand und eine Verschlechterung der ohnehin schon geringen Überschaubarkeit der Grammatik). Zudem kann es auch zu einer ungewollten Inkonsistenz der verwendeten Materialien kommen, was ein unstimmiges Endergebnis zur Folge haben kann.

Der Erfolg von Möglichkeit 2 ist stark von der Wahl der Standard Materialien abhängig. So sollten diese ein weites Spektrum abdecken und hauptsächlich Konfigurationen beinhalten, welche sich stark an den Vorstellungen der Nutzer/innen orientieren. Für ein sinnvolles Ergebnis in einer möglichen offiziellen Implementierung, lässt sich also eine Befragung der Nutzer/innen nicht vermeiden, sodass dieser Ansatz für diese Iteration ausgeschlossen werden kann.

Möglichkeit 3 erscheint für den/die Nutzer/in als die beste Wahl, da hier selbst bestimmt werden kann, wie viel Freiheit für die individuelle Gestaltung der Fassaden benötigt wird. Jedoch ist diese im Gegenzug mit dem höchsten Entwicklungsaufwand verbunden, welcher darin besteht, eine Syntax und die entsprechende Interpretation, sowohl für die Definition der Materialien als auch für die Überschreibung der Parameter in dem Material-Zuweisen-Operator, zu entwickeln.

Am sinnvollsten ist für diese Iteration also Möglichkeit 1, da diese einen verhältnismäßig geringen Aufwand für den Entwickler und den Endnutzer darstellt. Zudem lässt sich dieser Ansatz in einer späteren Iteration ohne große Komplikationen zu dem Ansatz aus Möglichkeit 3 erweitern.

3.5.2 Spezifikation der Syntax

Es muss also eine Syntax spezifiziert werden, mit welcher der/die Nutzer/in die zu verwendenden Materialien angeben kann. Die vorgestellte Syntax ist inspiriert durch die Objekterstellungs-Syntax in den Programmiersprachen go und rust.

```
VARIABLE_NAME = TEXTURE_GENERATOR_NAME {
 PARAMETER_NAME: [COLOR_]EXPRESSION,
 PARAMETER_NAME: [COLOR_]EXPRESSION,
 [...]
 PARAMETER_NAME: [COLOR_]EXPRESSION[,]
};
```

Der VARIABLE_NAME ist ein gültiger eindeutiger Identifier, welcher angibt, welcher Variable das erstellte Material zugewiesen werden soll. Diese Variable kann im Material-Zuweisungs-Operator genutzt werden, um das jeweilige Material zuzuweisen. Der TEXTURE_GENERATOR_NAME gibt an, welcher Texturengenerator für das entsprechende Material verwendet werden soll. Dieser wird in der darauffolgenden Parameter-Zuweisungs-Liste konfiguriert. Hierbei gibt der PARAMETER_NAME an, welcher Parameter zugewiesen werden soll und die EXPRESSION bzw. COLOR_EXPRESSION wird evaluiert und der resultierende Wert wird dem Parameter zugewiesen.

Dies könnte in der Praxis so aussehen:

```
RedBricks = Bricks {
 BrickWidth: BrickWidthVar,
 BrickHeight: 0.22,
 BrickColor: mix(Color("Red"), Color("#b32"), [0;1])
};
```

Nach der letzten Parameterzuweisung in einer Materialdefinition kann ein optionales Komma gesetzt werden. Dies wird ignoriert.

Für die Angabe und Berechnung der zu verwendenden Farben sollen dem/der Nutzer/in viele Möglichkeiten geboten werden, welche sich an CSS, als eine der bekanntesten Sprachen für Design, orientieren, hierbei jedoch Uneindeutigkeiten vermeiden. Es werden folgende Funktionen definiert:

`Color(HTML_COLOR_NAME)`
Hier versucht das System, die angegebene HTML-Farbe zu finden und den Farbwert dieser zurückzugeben.

`Color(RBG_HEX_CODE)`
Hier versucht das System, den angegebenen Hexadezimal Code als Farbwert zu interpretieren und diesen zurückzugeben.

`rgb(EXPRESSION(Red), EXPRESSION(Green), EXPRESSION(Blue))`
Hier evaluiert das System die angegebenen Ausdrücke und berechnet aus den Ergebnissen als Rot-, Grün- und Blauwert (jeweils von 0 bis 1) einen Farbwert welcher zurückgegeben wird. Jeder Wert wird zwischen 0 und 1 „geclampt“.

`hsb(EXPRESSION(Hue), EXPRESSION(Saturation), EXPRESSION(Brightness))`
Hier evaluiert das System die angegebenen Ausdrücke und berechnet aus den Ergebnissen als Farbton, Sättigung und Helligkeit (jeweils von 0 bis 1) einen Farbwert, welcher zurückgegeben wird. Bei dem Farbton wird die Differenz zum abgerundeten Wert genommen. Jeder andere Wert wird zwischen 0 und 1 „geclampt“.

`mix(COLOR_EXPRESSION, COLOR_EXPRESSION, EXPRESSION(Percentage))`
Hier evaluiert das System die angegebenen Ausdrücke und interpoliert die beiden resultierenden Farben zu dem resultierenden Prozentanteil und gibt das Ergebnis zurück.

Diese Farbwerte können auch Variablen zugewiesen werden, um diese in `COLOR_EXPRESSION`s zu verwenden.

Für die Zuweisung des Materials in einer Produktion wird ein neuer Operator definiert, welcher allen eingehenden Flächen das angegebene Material zuweist. Dieser hat folgende Syntax:
`material(USER_VARIABLE_NAME)`

Anmerkung: Die Operation kann nur mit Variablen aufgerufen werden, denen ein Material zugewiesen wurde.

Die Angepasste Form-Grammatik Syntax sieht nun folgendermaßen aus:

```
Variables:
USER_VARIABLE_NAME=[COLOR_]EXPRESSION | MATERIAL_CONSTRUCTOR
[...]
Rules:
RULE\_DEFINITION
[...]
```

Mit diesen Anpassungen ist der/die Nutzer/in in der Lage, in der Form-Grammatik Materialien, nach eigenen Anforderungen und auf Wunsch mit randomisierten Parametern, zu definieren und sie den gewünschten Fassaden zuzuweisen.

3.6 Spezifikation - Exportiertes Modell & Texturen

Genau wie die Form-Grammatik stellt auch das generierte texturierte Modell eine Benutzerschnittstelle dar. Für die Darstellung ist es für den/die Nutzer/in zwar irrelevant wie das 3D-Modell im Speicher aussieht und ob es überhaupt Texturen nutzt (Stichwort: Prozedurale Shader), jedoch ist für die Exportierung und weiterverwendung des Modells eine genaue Spezifikation notwendig:

Das Modell wird als eine Wavefront-OBJ-Datei exportiert, da hierfür bereits ein Exporter implementiert ist. Dieser muss lediglich erweitert werden, um Materialien mit Texturen zu unterstützen. Die Eckpunkte der Facetten des generierten Modells werden mit ihrer Position, den Texturkoordinaten und den Normalen gespeichert. Die Zuweisung der Materialien und ihre Namen werden aus der Form-Grammatik und dem generierten Modell übernommen. Da zur Anwendung eines Material in der Grammatik oftmals mehr als ein Material generiert werden muss, wird der Materialname um einen Index ergänzt:

```
Bricks_00
Bricks_01
Bricks_02
DarkerBricks_00
Roof_00
Roof_01
```

Die Texturen werden als Portable-Network-Graphics(PNG)-Dateien exportiert, da dies das aktuell gängigste Dateiformat für Grafiken ist.

4 Entwurf

4.1 Der Texturier-Algorithmus

Eine der umfangreichsten Aufgaben, welche das System nach dieser Iteration übernimmt, ist die Generierung und das Mappen der Texturen. Der Algorithmus hierfür soll die Textur so generieren und mappen, dass nach Möglichkeit keine auffälligen Nahtstellen zu sehen sind und die Texturen idealerweise am Boden ausgerichtet sind. Des Weiteren soll es möglich sein, (in einer späteren Iteration) einen Texturatlas zu generieren, sodass möglichst wenig Speicherplatz verschwendet wird.

Der Algorithmus wird für jedes Material ausgeführt. Die Eingabe ist ein 2er Tupel aus einem Material und einem Dreiecksnetz, welches die Facetten beinhaltet, denen dieses Material zugewiesen wurde. Das Ergebnis ist eine Liste an 3er Tupeln, jeweils bestehend aus einer Referenz zum generierten Albedo-Textur- und Normalen-Textur-Atlas und einem Dreiecksnetz, welches die Facetten beinhaltet, denen diese Texturen zugewiesen sind. Auf diese Weise unterstützen wir die Erstellung von mehreren Textur-Atlanten.

Einen Textur-Atlas aus Polygonen zu erstellen nicht trivial ist und mit der Anpassung der Formgrammatik (siehe Abschnitt 4.3.2), dem Implementieren der Algorithmen, sowie den notwendigen Anpassungen in den Frontends (siehe Abschnitt 5) besteht bereits ein großer Entwicklungsaufwand. Somit erhält in dieser Iteration jede Fassade einen eigenen Texturatlas.

Anmerkung: Um Rechenleistung zu sparen, wird als tatsächliche Eingabe ein Dreiecksnetz mit allen Facetten genutzt und es wird angegeben, welcher Farb-/Materialindex zu nutzen ist. Und anstatt für jedes benötigte Material ein neues Dreiecksnetz zu erstellen, werden die Facetten lediglich an ein übergebenes (anfangs leeres) Netz angefügt wobei den angefügten Facetten für jeden Textur-Atlas ein eigener Material-Index zugeordnet wird, aus welchem sich trivial auf die zu verwendenden Texturen in der zurückgegebenen Liste schließen lässt.

Im folgenden gehe ich die einzelnen Schritte des Algorithmus durch auf und erläutere meine Überlegungen und Entscheidungen.

4.1.1 Schritt 0: Modell in eine geeignete Datenstruktur überführen

Für den Algorithmus ist es wichtig, dass wir effizient durch die Kanten des Models iterieren und Außenkanten finden und „verfolgen“ können. Außerdem müssen wir den Winkel zwischen den 2 angrenzenden Facetten (wenn anwendbar) berechnen können.

Somit übertragen wir das Dreiecksnetz in eine Halbkanten-Datenstruktur, in welcher alle Halbkanten in einer Liste gespeichert werden und die Halbkanten die Referenz zu ihrer angrenzenden Facette speichern. Die Facetten selbst speichern lediglich ihren Normalen-Vektor und den Punkt im Zentrum (auf welchem dieser Normalen-Vektor gedanklich steht).

Pro Eckpunkt speichern wir die Position im 3D-Raum und pro Halbkante speichern wir den Index des zugehörigen Eckpunkts im Dreiecksnetz, sowie einen Index in ein Array der Koordinaten im 2D-Fassaden-Raum und der Texturkoordinaten.

4.1.2 Schritt 1: Modell in texturierbare Teilnetze aufspalten

Anmerkung: In diesem Schritt ist mit „aufspalten“ lediglich gemeint, dass wir bei 2 Halbkanten die Referenz auf die jeweils gegenüberliegende Halbkante entfernen. Es werden in keinem Fall Eckpunkte hinzugefügt.

Im ersten richtigen Schritt wollen wir die zusammenhängenden Netze des generierten Modells so zerteilen und aufklappen, dass wir eine kontinuierliche unverzerrte Parametrisierung der Oberfläche erreichen (im oben genannten 2D-Fassaden-Raum). Diese werden wir in den folgenden Schritten nutzen, um so viele zusammenhängende Facetten wie möglich zu texturieren, sodass die Textur immer (wenn anwendbar) am Boden ausgerichtet ist.

Schritt 1 (Teil 1/2): Ungewollte Kantenübergänge entfernen

Abhängig von dem Texturgenerator und seiner Parametrisierung wird das Dreiecksnetz so aufgespalten, dass alle Kanten zwischen 2 zusammenhängenden Fassaden einen gewissen Winkel nicht überschreiten (vgl. Edge-Split-Modifier in blender). Des Weiteren werden alle Kanten gespalten, welche die Orientierung der Textur zwischen 2 Facetten ändern würden (wenn die Textur senkrecht zum Boden ausgerichtet wird). Diese sind daran zu erkennen, dass sie weder horizontal noch vertikal orientiert sind und die Normalen der anliegenden Facetten nicht identisch sind. Ebenfalls dürfen die anliegenden Facetten einer horizontalen Kante nicht in unterschiedliche Himmelsrichtungen zeigen (Stichwort: Dachgiebel). Letzteres schließt auch Facetten mit ein, die gerade nach oben und somit in keine Himmelsrichtung zeigen.

Zu guter Letzt müssen wir auch präventiv alle Kanten spalten, welche nach innen gerichtet sind, da diese Kanten, wenn sie direkt oder indirekt an eine Dachkante angrenzen, zu einer einer Überlappung beim Ausklappen (siehe Abbildung 4.1) führen. Für eine Veranschaulichung welche Kanten gespalten werden siehe Abbildung 4.2.

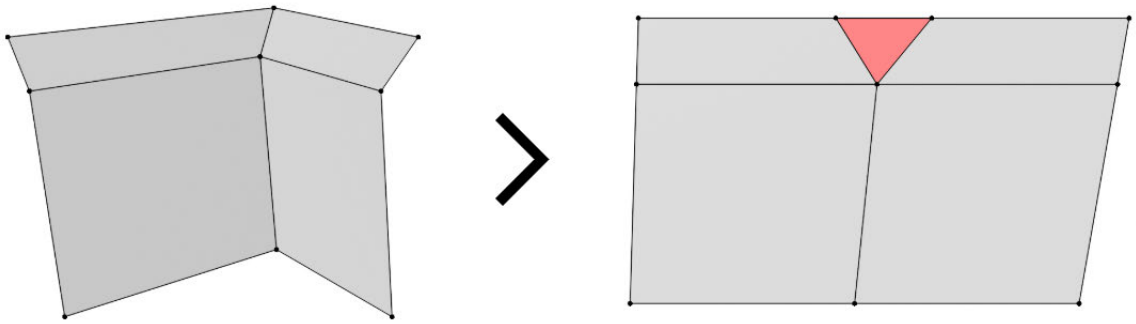

Abbildung 4.1: Nach innen gerichtete Kante grenzt an ein Schrägdach an. Ausklappen führt zu Überlappung (rot makiert)

Schritt 1 (Teil 2/2): Teilnetze ausklappen/an XZ-Achse ausrichten

Zudem sollen die resultierenden Netze ohne Probleme „ausklappbar“ sein.

Hierfür werden die Netze so lange an der Kante mit dem größten Winkel aufgespalten, bis eine Parametrisierung der Facetten-Eckpunkte gefunden wird, für welche alle Punkte

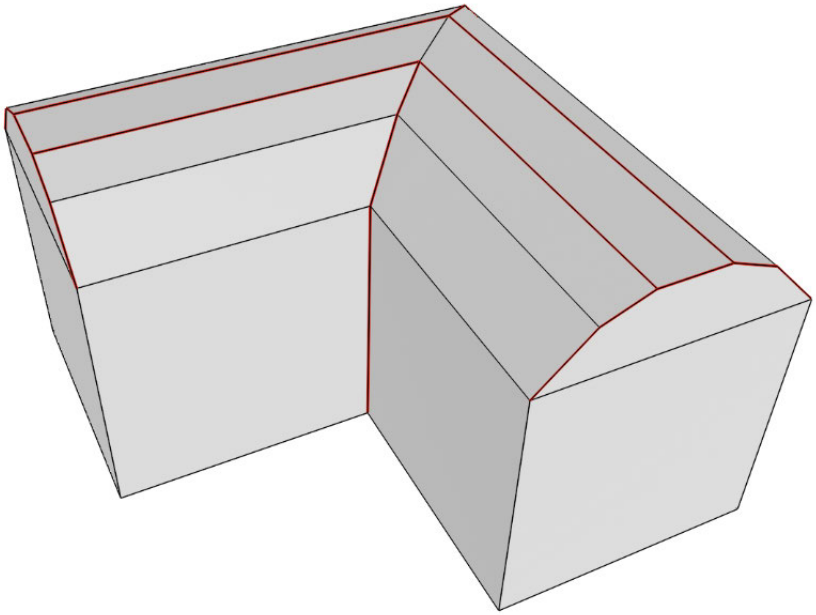

Abbildung 4.2: Das Netz wird in jedem Fall an den rot markierten Kanten gespalten

in der XZ-Ebene liegen und der Flächeninhalt sowie die Winkel der Facetten-Polygone erhalten bleiben (unverzerrt und ausgeklappt):

Als Startfacette wird eine beliebige Facette des jeweiligen Teilnetzes gewählt. Diese wird so ausgerichtet, dass sie in der XZ-Ebene liegt und die Ausrichtung im 2D-Raum mit der vertikalen Ausrichtung im 3D Raum (wenn anwendbar) übereinstimmt (ein hypothetischer gerade nach oben zeigender Vektor, welcher auf der Facette steht, zeigt nach der Ausrichtung der Facette, im 2D-Raum betrachtet, weiterhin gerade nach oben).

Das verfahren zur Ausrichtung wird später in diesem Abschnitt erläutert.

Zum Ausklappen der Fassaden-Netze wird ausgehend von der gewählten Startfacette Breadth-First durch alle Facetten und ihre Halbkanten iteriert. Für jede besuchte Halbkante wird eine Transformation berechnet welche die zugehörige Facette im 2D-Fassaden-Raum an der Vorgänger-Facette ausgerichtet (die Facette zeigt nach oben, ist unverzerrt und teilt sich eine Kante mit der Vorgänger-Facette). Wenn die Facette bereits als ausgerichtet markiert ist, aber die transformierte Halbkante nicht mit den bereits gespeicherten Werten übereinstimmt (innerhalb eines Toleranzbereichs), so terminiert der Algorithmus mit dem Ergebnis: Nicht ausklappbar. Ansonsten wird die berechnete Transformation auf die Facette angewandt und die resultierenden Positionen werden gespeichert.

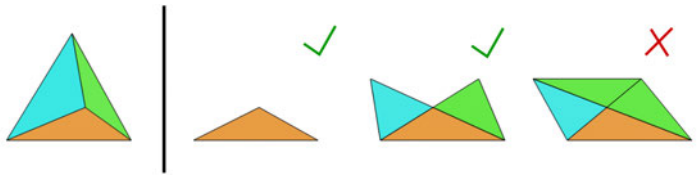

Abbildung 4.3: Dieses Netz ist nicht ausklappbar. Im letzten Schritt wird versucht die grüne Facette zu platzieren, sie wurde jedoch schon an einer anderen Stelle platziert.

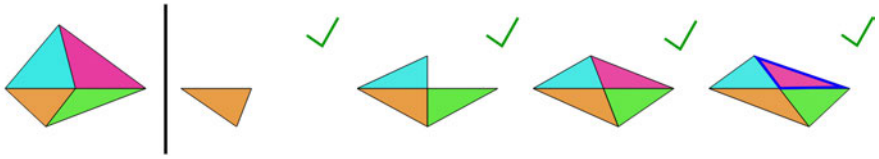

Abbildung 4.4: Dieses Netz ist ausklappbar. Facetten wurden entweder nur einmal platziert oder ihre Platzierung ist identisch (dunkelblaue Umrandung)

Wenn das Teilnetz als nicht ausklappbar erkannt wurde, wird es an der Kante mit dem größten Winkel aufgespalten und der Algorithmus versucht erneut es auszuklappen. Der Algorithmus zum vollständigen ausklappen und ggf. aufspalten aller Teilnetze befindet sich aus Platzgründen im Anhang A.2.

Um ein beliebiges Dreieck an der XZ-Ebene auszurichten, muss zuerst der Arkuskosinus des Y-Werts und der Atan2 aus dem X- und Z-Wert der Normale berechnet werden. Mit dem berechneten Atan2 wird das Dreieck so um die Y-Achse gedreht, dass es parallel zur X-Achse ausgerichtet ist. Anschließend wird der berechnete Arkuskosinus verwendet, um das Dreieck so um die X-Achse zu drehen, dass die Normale gerade nach oben zeigt (siehe Abbildung 4.5).

Um ein beliebiges Dreieck an einem anderen Dreieck in der XZ-Ebene auszurichten, müssen wir zuerst die minimale Rotation zwischen den aneinander auszurichtenden Kanten berechnen und diese auf das Dreieck anwenden. Anschließend berechnen wir die Rotation, betrachtet aus der Kantenrichtung, zwischen den Normalen der beiden Dreiecke. Hierfür Transformieren wir die Normale des unausgerichteten Dreiecks mit der Rotationsmatrix
$N' = (Kantenrichtung \times (0,1,0), (0,1,0), Kantenrichtung)^{-1} \cdot N$
und berechnen den Atan2 aus dem X- und Z-Wert des resultierenden Vektors. Mit dieser Rotieren wir das Dreieck um die Kantenrichtungs-Achse, sodass die Normale nach oben zeigt (siehe Abbildung 4.6).

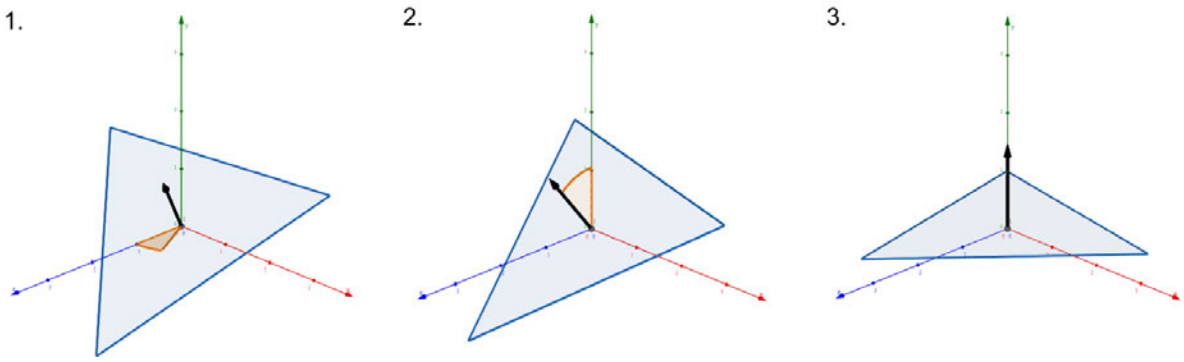

Abbildung 4.5: Visualisierung der Schritte um ein Dreieck an der XZ-Ebene auszurichten

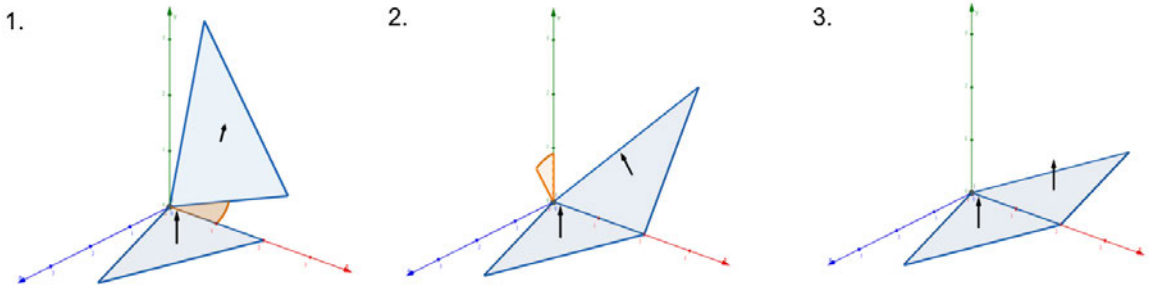

Abbildung 4.6: Visualisierung der Schritte um ein Dreieck an einem anderen Dreieck in der XZ-Ebene auszurichten

Als Ergebnis dieses Schrittes haben wir nun zusammenhängende Teilnetze, welchen eine kontinuierliche Textur zugewiesen werden kann. Überlappungen sind bei diesem Vorgehen zwar nicht ausgeschlossen, jedoch mit der aktuellen Form-Grammatik und dem vorherigen Zerteilen nur sehr schwer, wenn nicht sogar unmöglich herbeizuführen.

4.1.3 Schritt 2: Fassaden Netze positionieren und analysieren

Die Fassaden-Netze werden im 2D-Fassaden-Raum so positioniert, dass die untere linke Ecke der Bounding-Box auf den Koordinaten (0,0) liegt. Wenn dies von dem Texturgenerator verlangt ist, werden die Netze im 2D-Fassaden-Raum vermessen, um eine finale Parametrisierung vorzunehmen (z.B. die ideale Höhe eines Ziegelsteins) und über die Außenkanten der Netze wird jeweils ein geschlossener 2D-Polygonzug erstellt.

4.1.4 Schritt 3: Außenkanten-Polygone berechnen

Für alle Texturgeneratoren ist es wichtig, die Distanz zur dichtesten Außenkante berechnen zu können, damit es keine unschönen Übergänge zwischen zwei Fassaden gibt, die entweder nicht das gleiche Material teilen oder in Schritt 1 getrennt wurden und somit die Textur eine andere Orientierung hat. Diese Distanz soll sich trivial pro Pixel berechnen lassen und im Idealfall effizient in einem Shader durchgeführt werden können. Um die GPU hierbei optimal zu nutzen müssen wir dafür sorgen, dass für jeden Pixel mit teilweise unterschiedlichen Daten die gleichen Berechnungen durchgeführt werden, stichwort SIMD (Single Instruction Multiple Data) und die execution pointer der einzelnen Workgroups nicht divergieren stichwort SIMT (Single Instruction Mutiple Thread) [11]. Ebenfalls sollten alle komplizierten Berechnungen im Vornherein durchgeführt werden, um den Overhead pro Pixel zu minimieren.

Die kürzeste Distanz zu einem Liniensegment berechnet man, wie bei einer Linie, mit dem Lotfußpunkt. Der Unterschied ist: Wenn der Lotfußpunkt außerhalb der Endpunkte liegt, wird stattdessen die Distanz des Punkts zu dem dichtesten Endpunkt berechnet (siehe Abbildung 4.7).

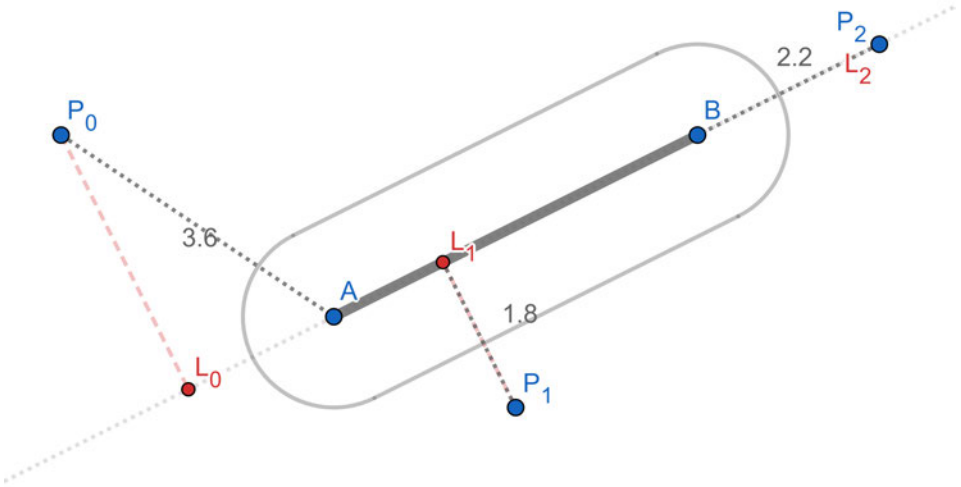

Abbildung 4.7: Veranschaulichung der Berechnung der Distanz von mehreren Punkten (p_0, p_1, p_2) zu einem Liniensegment (mit Lot und Lotfußpunkt, rot dargestellt)]

Zur Berechnung der Distanz zu der Umrandung eines Polygons müssen wir schlicht das Minimum der Entfernungen zu allen Kanten berechnen. Hierbei kommen wir innerhalb

und außerhalb eines Polygons immer zu einem gültigen Ergebnis, sofern es keine Selbst-Überlappungen gibt (siehe Abbildung 4.8).

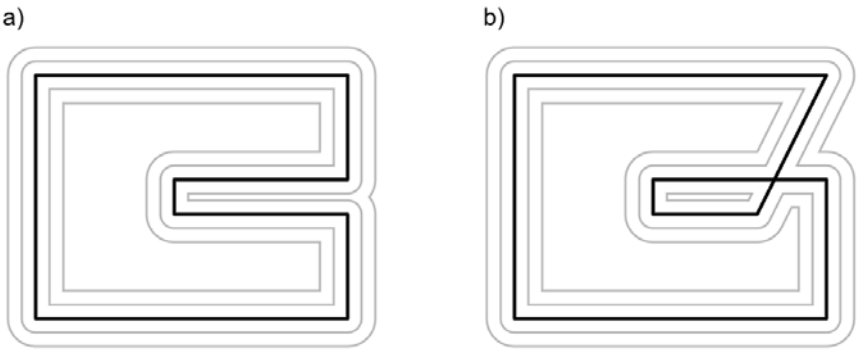

Abbildung 4.8: a) Polygon ohne Selbst-Überlappung: Distanz immer korrekt. b) Polygon mit Selbst-Überlappung: Ergebnis wird verfälscht

Bei dieser Methode fällt jedoch auf, dass an den nach innen gerichteten Ecken innerhalb des Polygons Abrundungen entstehen, die für Gebäudearchitektur eher ungewöhnlich sind und nicht zu den nach außen gerichteten Kanten passen, bei welchen diese Abrundungen innerhalb des Polygons fehlen.

In der Vektorgrafik gibt es drei gängige Optionen für die Verbindung der Außenlinien, welche im SVG Standard [2] als die Stroke-Joins: Round, Bevel und Miter definiert wurden (siehe Abbildung 4.9).

Abbildung 4.9: Darstellung der Stroke-Join-Optionen des SVG-Standards
Grafik entnommen aus der offiziellen SVG 1.1 Dokumentation [2]

Die Option „Miter“ erzeugt hierbei den gewünschten Effekt. Dies ist wenig überraschend, da der „Miter“ (auf Deutsch Gehrung) im Bauwesen häufig zur Anwendung kommt. Die Gehrung ist in der Regel die Winkelhalbierende an einer nach innen oder außen gerichteten Ecke und zeichnet sich dadurch aus, dass 2 Leisten, welche entlang dieser angesägt wurden, perfekt aneinander und an der Ecke anliegen (siehe Abbildung 4.10).

Dies bedeutet: Wenn wir pro Kante mit jeweils dem gleichen Abstand eine Gerade parallel zur Kante ziehen und diese an den anliegenden Gehrungen schneiden, sodass ein

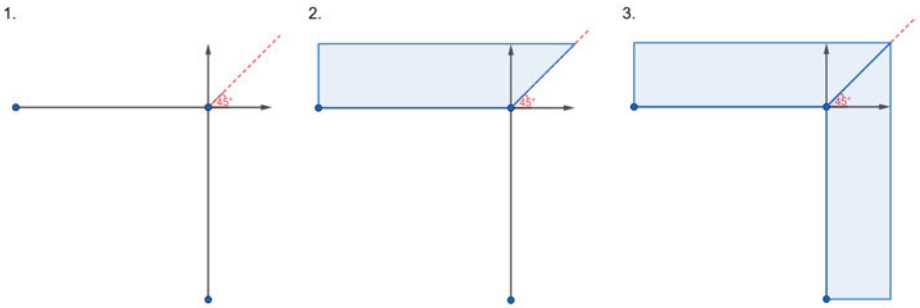

Abbildung 4.10: Veranschaulichung des Konzepts der Gehrung: Gehrung wird ermittelt und beide Leisten werden angesägt und angelegt.

Liniensegment entsteht, dann erhalten wir einen geschlossenen Polygonzug mit einem konstanten Abstand zur Umrandung.

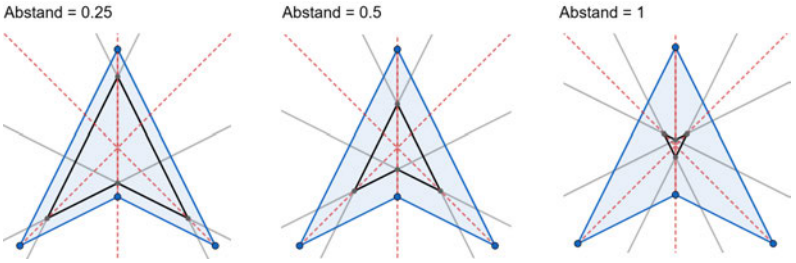

Abbildung 4.11: „Schrumpfen“ eines Polygons mit dem oben beschriebenen Vorgehen. Gehrungen sind rot gestrichelt, Parallelgeraden und Schnittpunkte sind grau. Anmerkung: Bei einem Abstand von 1 kehrt sich hier die Windung um, also ist dieses Ergebnis ungültig.

Dies können wir mit beliebigen Abständen durchführen (siehe Abbildung 4.11) , sofern es keine Überschneidungen der Liniensegmente gibt oder sich die Windung umkehrt, sodass sich ein (Signed)-Distance-Field ableiten lässt. Die dazugehörige Distanzfunktion ist das Minimum der Distanzen zu allen Kanten (betrachtet als Gerade statt als Segment), für welche gilt, dass der abgefragte Punkt von den anliegenden Gehrungen „eingeschlossen“ wird und lokal betrachtet „über“ der Kante liegt. Mathematisch notiert:

$$f_d(E,p) = \min_{(a,b,n,m_a,m_b)\in E} \left(\begin{cases} \frac{(p-a)\cdot n^0}{|n|} & \text{wenn } inside(a,b,n,m_a,m_b,p), \\ \infty & \text{andernfalls.} \end{cases} \right)$$

$$\begin{aligned} inside(a,b,n,m_a,m_b,p) \quad &= \quad (p-a)\cdot(R_{90}\ m_a) \ \geqslant\ 0 \ \wedge \\ &\qquad (p-b)\cdot(R_{90}\ m_b) \ \leqslant\ 0 \wedge (p-a)\cdot n \ \geqslant 0 \end{aligned}$$

E ist hierbei eine Menge von 5er-Tupel, welche sich nahezu trivial aus den Außenkanten ableiten lässt. Die Tupel bestehen aus: Endpunkt A, Endpunkt B, Projektionsvektor (welcher orthogonal nach innen gerichtet auf dem Außenkantensegment steht), Gehrungs-Vektor an Punkt A und Gehrungs-Vektor an Punkt B. Über jedes dieser Tupel lässt sich gedanklich ein zu einer Seite hin offenes Trapez bzw. Dreieck definieren, sodass diese im Nachfolgenden als Außenkantenpolygone bezeichnet werden. $R90$ ist eine 2x2 Matrix, welche eine Rotation um 90° im Uhrzeigersinn repräsentiert.

Anmerkung: Der Projektionsvektor ist nicht zwingend normiert, sodass eine minimale lokale Skalierung des Distanzfeldes möglich ist. Dies ist beabsichtigt.

Diese Funktion gibt uns die Distanz zur „dichtesten“ Kante und lässt sich trivial so umformulieren, dass wir den Normalenvektor n der dichtesten Kante ermitteln können.

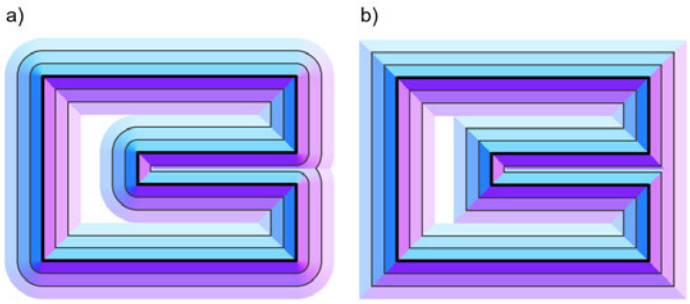

Abbildung 4.12: Distanz- und Richtungsfelder desselben Polygons a) euklidischen Distanz b) Distanz zur Liniengerade unter Einschluss der Gehrungen

Wenn wir die Gehrung nutzen, stoßen wir jedoch auf ein Problem: Je größer der Innenwinkel, desto weiter ist der Schnittpunkt der Parallellinien von der jeweiligen Ecke entfernt (im Verhältnis zur Distanz der Parallellinien zu den anliegenden Kanten) (siehe Abbildung 4.13).

Der SVG Standard löst dieses Problem mit einer Limitierung der „Miter-Length“, welche nichts anderes ist als die Distanz zwischen dem Eckpunkt und dem oben genannten Schnittpunkt. Wenn dieses Limit überschritten wird, wird die Außenlinie abgeschnitten, und es wird ein Dreieck in der entstandenen Lücke platziert. Wir betrachten also lediglich den Eckpunkt zwischen beiden Kanten als neue Kante mit einer Länge von 0 und einem nach innen zeigenden Normalenvektor, welcher in Richtung der vorher berechneten Gehrung zeigt und berechnen die neuen Gehrungsvektoren so, dass sie das einzufügende Dreieck aufspannen. (Hierzu verwenden wir lediglich die Richtungsvektoren der anliegenden Kanten (siehe Abbildung 4.14))

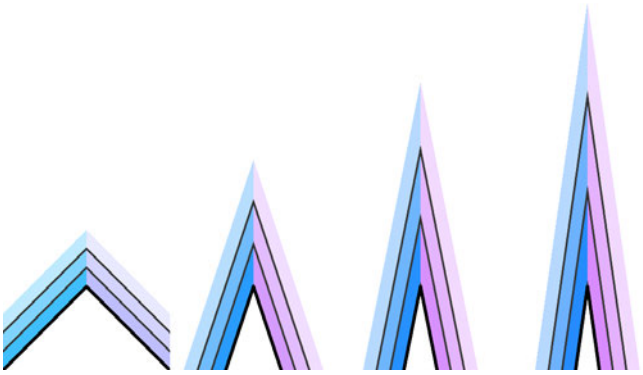

Abbildung 4.13: Die Gehrungs-„länge“ ist bei größeren Winkeln deutlich länger

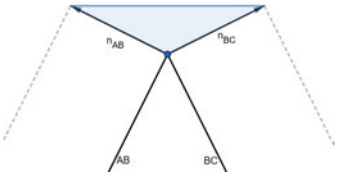

Abbildung 4.14: Darstellung der Gehrungs- bzw. Normalvektoren und dem von ihnen aufgespannten Dreieck

Dieses Vorgehen bringt jedoch bezogen auf Distanzberechnung ein neues Problem mit sich: Je spitzer eine nach innen gerichtete Kante ist, desto schmaler wird das eingefügte Dreieck. Hierdurch wird die Distanzberechnung bedeutend ungenauer, da das Distanzfeld lokal gestaucht wird und es entsteht eine starke Diskontinuität, wenn wir versuchen dieser entgegenzuwirken (siehe Abbildung 4.15). Schlimmer noch, wenn die Kanten antiparallel verlaufen, verschwindet dieses Dreieck, sodass eine noch stärkere Diskontinuität entsteht.

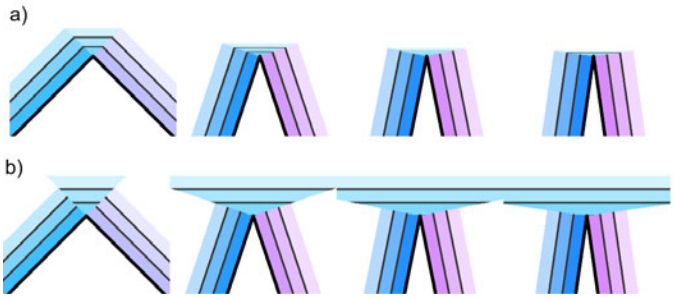

Abbildung 4.15: Die Höhe des „Fülldreieck“ wird bei größeren Winkeln geringer. Bei a) wird das Distanzfeld wird lokal zunehmend gestaucht. Bei b) wirken wir dem entgegen sodass eine starke Diskontinuität entsteht

Eine einfache Lösung ist, für die Berechnung der aufspannenden Vektoren des „Fülldreiecks“ die anliegenden Normalvektoren mit dem parallel bzw. antiparallel verlaufenden Richtungsvektor der jeweiligen Kante zu addieren (siehe Abbildung 4.16).

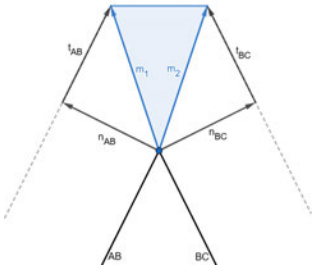

Abbildung 4.16: Darstellung der Normal- und Tangentenvektoren, den daraus berechneten Gehrungsvektoren und dem von ihnen aufgespannten Dreieck

Für Winkel größer als 90° definieren diese Gehrungsvektoren ein Dreieck mit einem positiven Flächeninhalt, für kleinere Winkel nutzen wir die vorher berechnete Gehrung und fügen keine zusätzliche Kante hinzu.

Abbildung 4.17: „Fülldreieck“ variiert nur noch leicht in der Höhe. Die resultierende Streckung des Distanzfeldes ist zu vernachlässigen.

Der Pseude Code um für einen beliebigen Außenkanten-Polygonzug die notwendigen Außenkantenpolygone zu berechnen sieht folgendermaßen aus:

```
$Polygone := []
$Neues_Polygon := (Nil, Nil, Nil, Nil, Nil)

// pro Iteration wird das Polygon aus der vorherigen Iteration
// (wenn verfügbar)
```

```
// vervollständigt und zur Liste hinzugefügt und
// es wird das Polygon für die nächste Iteration vorbereitet

[außenkanten_schleife]
für $i von 0 bis $(Anzahl der Außenkanten):
   // Die Kanten sind zyklisch nummeriert:
   // Kante 1 ist gleichzeitig Kante n+1
   $Gehrung := Gehrung zwischen Kante $i und $i+1
   ($A, $B) := Positionen der Endpunkte von Kante $i+1
   $Normale0 := Normale von Kante $i
   $Normale1 := Normale von Kante $(i+1)
   $Richtung0 := Richtung von Kante $i
   $Richtung1 := Richtung von Kante $(i+1)

   wenn $Richtung0 = $Richtung1:
      springe zur nächsten iteration von [außenkanten_schleife]

   //# heißt der Wert im Tupel wird nicht überschrieben
   $Neues_Polygon := (#,#,#,#,$Gehrung)

   // Wenn die Kanten nach innen gerichtet sind ist der Winkel negativ
   // Wenn die Kanten in die gleiche Richtung zeigen ist der Winkel 0°
   $Winkel := Vorzeichenbehafteter Winkel zwischen Kante $i

   wenn $Winkel < -90°:
      $GehrungL0 := $Normale0 + $Richtung0
      $GehrungL1 := $Normale1 - $Richtung1

      $ProjektionsVektor := (GehrungL0 + GehrungL1)/2
      $Polygone += ($A, $A, $ProjektionsVektor, $GehrungL0, $GehrungL1)

      $Neues_Polygon := (#,#,#,#,$GehrungL0)
      $Gehrung := $GehrungL1

   // bei nach außen gerichteten Kanten ist eine Limitierung
   // an der Gehrung nicht sinnvoll und sorgt teilweise
```

```
   // für zu früh abgeschnittene Kanten, daher limitieren wir an der
   // jeweils vom Eckpunkt aus gegenüberliegenden Kante
   wenn $Winkel > 0:
      $Neues_Polygon = (#,#,#,#,Richtung1)

   wenn kein Nil in $Neues_Polygon:
      $Polygone += $Neues_Polygon

   wenn $Winkel > 0:
      $Gehrung := -Richtung0

   $Neues_Polygon := ($A, $B, $Normale1, $Gehrung, Nil)

[/außenkanten_schleife]
```

Die Gehrung wird berechnet, wie oben beschrieben. Eine Normierung ist nicht notwendig, da für die Formel der Distanzberechnung lediglich die Richtung der Gehrungsvektoren entscheidend ist. Das Ergebnis dieses Schritts ist eine Liste aus 5er Tupeln, welche die Außenkantenpolygone (zu einer Seite geöffnete Trapeze und Dreiecke) repräsentieren und von einem Shader genutzt werden können, um sehr effizient die Entfernung und Richtung zur „dichtesten“ Kante zu berechnen.

Nach diesem Schritt wird das Halbkanten-Netz in ein Dreiecksnetz mit Texturkoordinaten überführt.

4.1.5 Schritt 4: Die Texturen generieren

Je nach Texturgenerator wird ein Positions-Buffer, ein 2D-Fassaden-Positions-Buffer und ein Texturkoordinaten-Buffer erstellt. Danach wird aus der Höhe und Breite der jeweiligen Bounding-Box mit einer globalen Texel-pro-Meter-Konstante die Größe der Texturen für jede Fassade einzeln berechnet. Es werden jeweils 2 Texturen erstellt: Eine Albedo-Map und eine Normal-Map. Anschließend werden alle Polygone des Dreiecksnetzes mit ihren Texturkoordinaten in die 2 Texture-Maps (Albedo-Map und Normal-Map) als Color-Target gerendert. Hierfür wird ein Shader verwendet, welcher die tatsächliche Oberflächentextur generiert. Dieser Shader hat Zugriff auf die Liste der Außenkanten-Polygone aus Schritt 3 und die Parameter des Textur-Generators in Form von Shader-Parametern.

Zudem kann er auf die Daten der zuvor erstellten Buffer in Form von Vertex-Attributen zugreifen. Die resultierenden Farb- und Normalwerte werden als Shader-Output in die jeweilige Textur geschrieben. Die generierten Texturen werden in-memory temporär abgespeichert, um sie im Frontend laden und später beim Exportieren im Dateisystem speichern zu können.

4.2 Die Textur-Generatoren

4.2.1 Von der Spezifikation zum Shader

In der Spezifikation der Texturen ist ein 3-Dimensionaler Aufbau angegeben. Für die zu generierenden Texturen müssen nun jedoch jeweils eine 2 Dimensionale Albedo-Map und Normal-Map erstellt werden. Hier ist Kreativität gefragt.

Für die Berechnung der Normalen haben wir im Allgemeinen 3 Optionen:

1. Wir berechnen den Tiefenwert des Texels, vergleichen ihn mit den umliegenden Werten, ermitteln daraus den Gradienten und Normieren diesen.
2. Wir berechnen 2 Vektoren im Tangentialraum, welche möglichst orthogonal zueinander stehen, berechnen das Kreuzprodukt und normieren dieses.
3. Wir berechnen unser Objekt so, dass wir die Normale direkt mit berechnen können.

Die erste Option ist mit Hilfe der partiellen Differentiale, welche der Shader automatisch berechnet, trivial umzusetzen, jedoch hängt die Genauigkeit der berechneten Normale stark von der Auflösung der Textur ab, somit ist es eher als Notlösung anzusehen.

Für die Backsteine zeigt die Normale überall in die gleiche Richtung, abgesehen von der Abschrägung an der Kante (und an der Umrandung des Fassaden-Polygons) für diese gibt es jedoch einfach zu implementierende Funktionen welche uns die Richtung der Kante liefern, aus welcher sich trivial der Normalen-Vektor an der Abschrägung berechnen lässt.

Für die Dachpfannen sind das Front- und Seitenprofil als Mathematische Funktionen definiert (siehe Abschnitt 4.2.2), welche sich ableiten lassen. Somit können wir an jedem Punkt auf der durch die 2 Profile definierten Mannigfaltigkeit 2 nahezu orthogonale Vektoren im Tangentialraum berechnen und somit die Normale ermitteln.

Für die Kieselsteine transformieren wir die Texelkoordinaten in den lokalen Vektorraum des Kieselsteins. In diesem Vektorraum ist der Kieselstein eine Einheitskugel, von welcher sich an jeder Stelle der Schnittpunkt mit einem von oben kommenden Strahl und somit auch die Normale berechnen lässt. Diese muss lediglich anschließend mit der Transponierten Matrix zurück transformiert werden. Die Kieselsteine haben zusätzlich die Besonderheit, dass sie sich gegenseitig bedecken können. Um dies im Shader zu beachten, müssen wir den Tiefenwert berechnen (sehr ähnlich zur Berechnung der Normale) und diesen für jeden Stein, welcher potentiell mit dem Texel überlappen könnte, vergleichen. (Der niedrigste Wert bestimmt die finale Tiefe, Normale und Farbe des Texels.)

4.2.2 Formeln für die Dachpfannen Oberfläche

Die Oberfläche der Dachpfannen ist durch eine Mannigfaltigkeit definiert, welche wiederum von 2 Profilkurven definiert wird (siehe Abbildung 4.18).

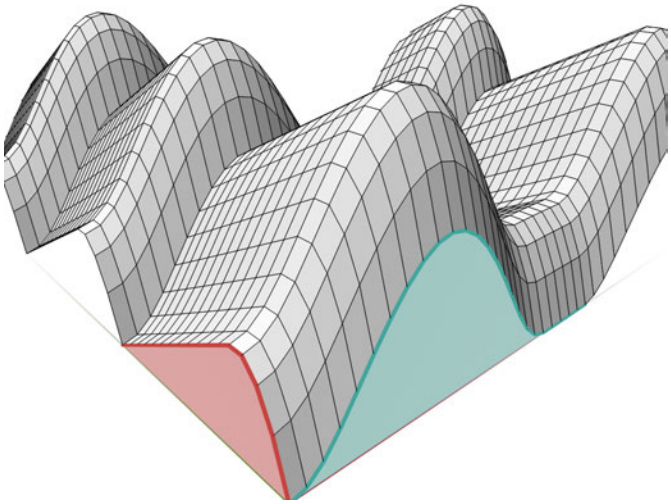

Abbildung 4.18: Visualisierung der Dachpfannen-Mannigfaltigkeit

Für die Normalberechnung (siehe Abschnitt 4.2.1) müssen wir den lokalen Gradienten der Oberfläche berechnen können. Also müssen beide Profilkurven ableitbar sein.

Das Frontprofil der Dachpfannen ist eine Kurve zusammengesetzt aus einer Konstante, einer Sinus-Funktion für die Erhöhung und 2 Kosinus-Funktionen für die konkaven Kurven am Fuß der Erhöhung. Die Sinus-/Kosinus-Kurven werden in Amplitude und Frequenz skaliert und teilweise verschoben, sodass sich ihre Wendepunkte (in der 0. und 1. Ableitung) jeweils bei $x = a$ und $x = 1 - a$ berühren (siehe Abbildung 4.19). Das

Seitenprofil der Dachpfannen ist eine Kurve bestehend aus einer Parabel welche den Ursprung durchläuft und ab dem Punkt $(i; j)$ in eine Gerade übergeht, welche die x Achse bei $x = k$ schneidet. Hierbei haben Parabel und Gerade an Punkt (i, j) dieselbe Steigung (siehe Abbildung 4.20). Die Funktionen der Profilkurven (Frontprofil: $f(x)$, Seitenprofil: $g(x)$) setzen sich wie folgt zusammen:

$$f_0(x) = \frac{2a}{\pi}\left(-\cos\left(\frac{\pi}{2a}(x)\right) + 1\right)$$

$$f_1(x) = \frac{1-2a}{\pi}\sin\left(\frac{\pi}{1-2a}(x-a)\right) + f_0(a)$$

$$f_2(x) = \frac{2a}{\pi}\left(-\cos\left(\frac{\pi}{2a}(x-1)\right) + 1\right)$$

$$f(x) = \frac{b}{f_1(0.5)}\begin{cases} f_0(x) & \text{wenn } 0 < x < a \\ f_1(x) & \text{wenn } a < x < 1-a \\ f_2(x) & \text{wenn } 1-a < x < 1 \\ 0 & \text{andernfalls.} \end{cases}$$

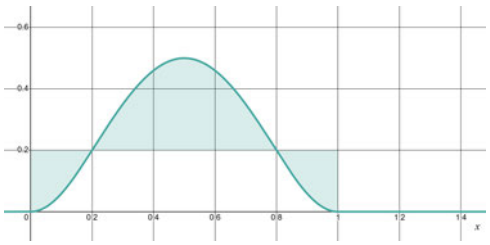

Abbildung 4.19: Visualisierung der Zusammensetzung des Frontprofils mit $a = 0.2$ und $b = 0.5$

Hier ist a die Breite der Abrundung am Fuß der Erhöhung und b ist die Höhe der Erhöhung im Frontprofil.

$$g_0(x) = \frac{i\frac{j}{i-k} - j}{i^2}x^2 + \left(\frac{2j}{i} - \frac{j}{i-k}\right)x$$

$$g_1(x) = \frac{j}{i-k} \cdot (x-i) + j g_1(x) = \frac{j}{i-k} \cdot (x-i) + j$$

$$g(x) = \begin{cases} g_0(x) & \text{wenn } 0 < x < i \\ g_1(x) & \text{wenn } i < x < l \end{cases}$$

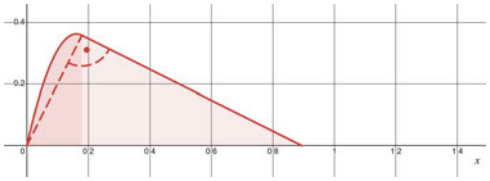

Abbildung 4.20: Visualisierung der Zusammensetzung des Seitenprofils mit $i = 0.18$, $j = 0.36$, $k = 0.89$

Hier sind die Parameter i, j und k so zu wählen, dass die Punkte $(0;0)$, $(i;k)$, $(k;0)$ ein Rechtwinkliges-Dreieck ergeben (siehe Abbildung 4.20), in welchem die Länge der linken Ankathete der Dachpfannendicke und die Länge der rechten Kathete der Dachpfannenhöhe/-länge entspricht.

4.2.3 Platzierung der Backsteine

Die Generatoren für Dachziegel und Kieselsteine sind mehr oder weniger trivial in einem Shader implementierbar, da sie sich an einem festen Gitter ausrichten und im Fall von Kieseln lediglich Informationen aus den direkten Nachbarzellen brauchen. Eine realistische Backsteintextur hat jedoch kein festes Gitter und ist zudem abhängig davon, wie viele kleine und große Backsteine links von einem Texel platziert wurden. Zwar lässt sich dies in der Theorie berechnen, bringt jedoch einiges an Aufwand und Limitationen mit sich. Ein Shader ist schlicht nicht dafür geeignet, große und kleine Backsteine zufällig in mehreren Reihen zu platzieren. Daher müssen wir vor dem Rendern auf der GPU ein 2D-Modell auf der CPU erstellen, in welchem alle Backsteine bereits als Rechtecke platziert wurden und jeder Eckpunkt die Position sowie Höhe und Breite des jeweiligen Rechtecks speichert. Um Bandbreite zwischen CPU und GPU zu sparen, wäre es zudem möglich, mit GPU-Instancing zu arbeiten, so dass die Daten nur noch einmal pro Rechteck zur GPU geschickt werden müssen. Die Rechtecke werden (mit den Dreiecken des 2D-Fassaden-Modells maskiert) gerendert und im Shader kann nun mit den genauen Positionen des zu renderndem Rechtecks weiter gerechnet werden.

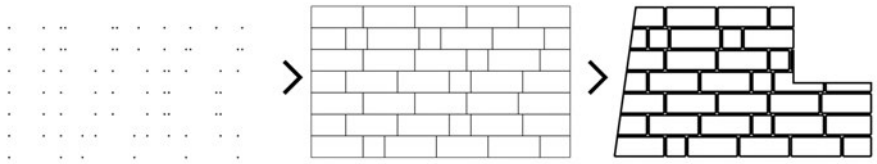

Abbildung 4.21: Backsteine auf der GPU: Punkte > Quads > Maskierung und Shader

4.3 Die Projektstruktur

4.3.1 Analyse des bestehenden Systems

Für eine sinnvolle Erweiterung des bestehenden Systems ist es wichtig, dass wir uns einen Überblick von der Organisation und den Zusammenhängen der einzelnen Komponenten in dem bestehenden System verschaffen. Hierfür habe ich jede Komponente des Procedural-Content-Generation (pcg)-Projekts, sowie die Unterkomponenten cgashape und cgashape2d mit Beschreibung ihrer Funktion und Abhängigkeiten in Abbildung 4.22 dokumentiert.

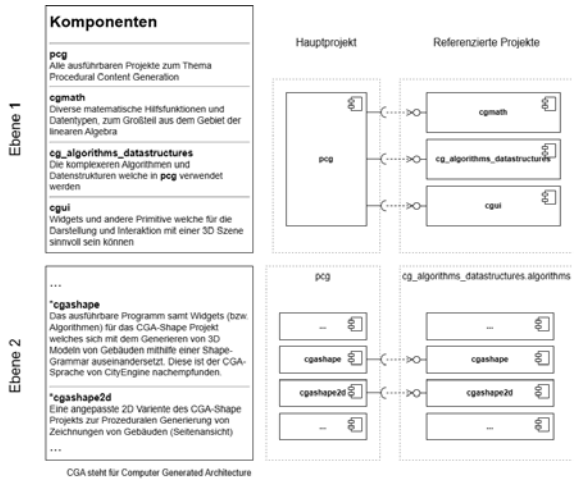

Abbildung 4.22: Komponenten-Diagramm für das „pcg“-Projekt Ebenen 1 und 2, Fokus auf cgashape in Ebene 2

Zusammenspiel der einzelnen Klassen in cgashape

Ebenfalls habe ich mich bemüht, die Abhängigkeiten der Klassen des cgashape-package in Abbildung 4.23 zu dokumentieren. Hierbei lege ich keinen Anspruch auf Vollständigkeit, jedoch konnte ich keine weiteren Abhängigkeiten identifizieren.

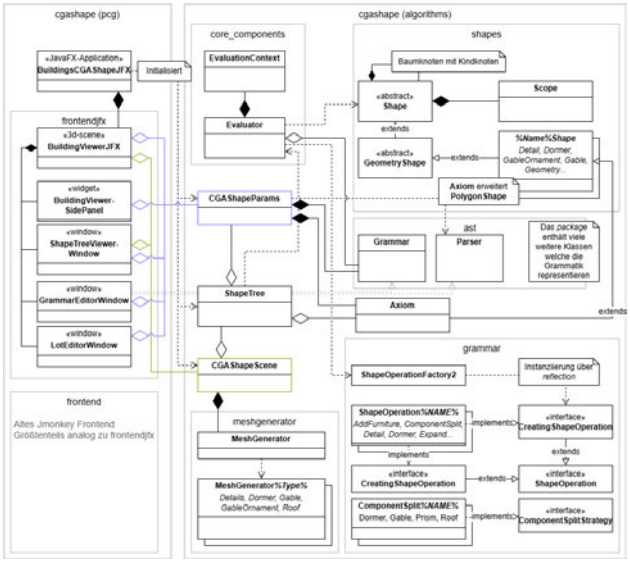

Abbildung 4.23: Klassendiagramm, Übersicht der Abhängigkeiten im cgashape-package (Frontend und Backend)

Parameters, Model, Scene

Viele Unterprojekte in dem „pcg“-Projekt implementieren bzw. erweitern die Klassen `Parameters`, `Model` und `Scene`. Diese stellen die Verarbeitungsschritte der einzelnen Algorithmen dar:

Die `Parameters`-Implementation nimmt die Eingabeparameter des Nutzers entgegen und validiert und interpretiert diese.

Die `Model`-Implementation initialisiert den Algorithmus mit den angegebenen Parametern und wendet ihn an.

Die `Scene`-Implementation wandelt das Ergebnis in eine für den/die Nutzer/in leicht verständliche Darstellung um, sodass es ohne weitere Anpassungen direkt in der GUI angezeigt werden kann.

Konkret bezogen auf das cgashape-Projekt sieht dies wie folgt aus:

Die `CGAShapeParams` stellen das, mit Hilfe des `Parsers` validierte, Axiom und die Grammatik dar.

Der `ShapeTree` repräsentiert den Form-Baum, welcher durch Ableiten des Axioms mit der angegebenen Form-Grammatik entsteht. Das Ableiten selbst wird von der Klasse `Evaluator` durchgeführt.

Die `CGAShapeScene` beinhaltet das eingefärbte 3D Modell, welches der `MeshGenerator` aus dem `ShapeTree` generiert hat. Dieses Mesh wird letztendlich in der GUI angezeigt.

Die Klassen sind so implementiert, dass eine Änderung der Eingabeparameter die Grammatik evaluiert und die darauffolgende Aktualisierung des `ShapeTrees` das 3D-Modell generiert.

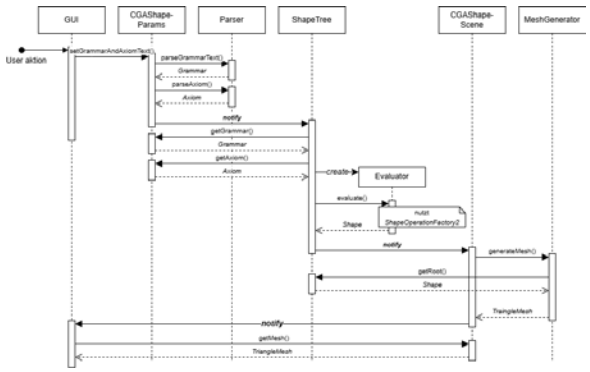

Abbildung 4.24: Sequenz-Diagramm User-Interaktion bis Aktualisierung der GUI

Die Definition der Form-Grammatik

Die Form-Grammatik wird in mehreren Teilen im Projekt definiert:

Die Syntax der Grammatik wird in der Datei *ShapeGrammar.g4* in einem für ANTLR [1] lesbaren Format definiert. Hier ist definiert, aus welcher textlichen Zusammensetzung alle Syntax Bausteine der Form-Grammatik (e.g. `EXPRESSION`) bestehen bzw. bestehen können. Die Definition ist rekursiv, sodass ein Syntax Bausteinen aus weiteren Syntaxbausteinen oder Literalen besteht.

Der Aufbau der vom `Parser` eingelesenen Grammatik wird mithilfe von Objekten der Klassen in `cg_algorithms_datastructures.cgashape.ast` modelliert. Jede Klasse repräsentiert einen Syntaxbaustein mit all seinen Eigenschaften. Zusammengesetzt ergeben sie eine Baumstruktur, ein Abbild des Abstract-Syntax-Tree.

Die Funktionsweise der verschiedenen Operationen in der Grammatik wird in den Klassen in `cg_algorithms_datastructures.cgashape.grammar` definiert (siehe Abbildung 4.25). Jede Klasse (abgesehen von `ShapeOperationFactory2` und `ShapeGrammarConstants`) „implementiert“ das Interface `ShapeOperation`.

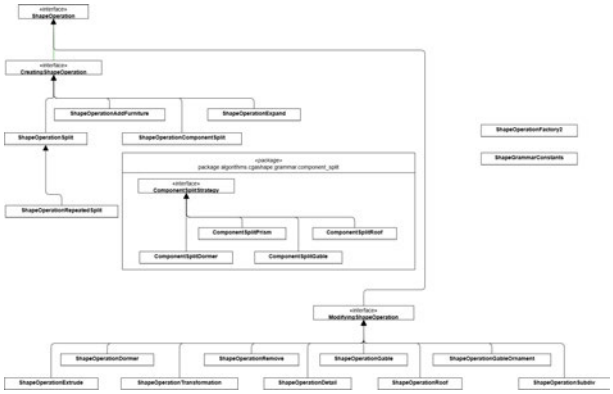

Abbildung 4.25: Klassendiagramm des cg_algorithms_datastructures.cgashape.grammar-package generiert von Intellij mit minimalen Anpassungen

Des Weiteren werden die Formen, welche in dem resultierenden Form-Baum verwendet werden können, in `cg_algorithms_datastructures.cgashape.shapes` definiert (siehe Abbildung 4.26). Neben Allgemeinen Formen wie Prismen werden auch Gebäudespzezifische Formen wie z.B. Dachgiebel definiert.

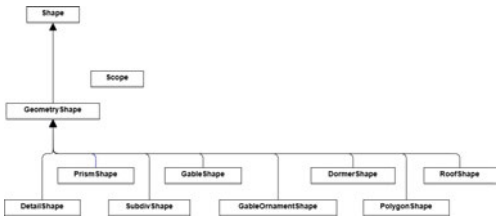

Abbildung 4.26: Klassendiagramm des cg_algorithms_datastructures.cgashape.shapes-package generiert von Intellij

4.3.2 Notwendige Änderungen

antlr / ast-package

Die *ShapeGrammar.g4* wird um die in der Spezifikation spezifizierten Material Syntaxbausteine erweitert und das `cgashape.ast`-package erhält eine neue Klasse `MaterialConstructor`, welche die Definition des Material mit seinen Parametern (jeweils Name und `Expression`-Objekt) modelliert. Der `MaterialConstructor` kann ausschließlich alternativ zu einem Ausdruck bei der Variablenzuweisung genutzt werden.

Bei der Auswertung der Variablen in der Grammatik werden mittels einer neuen `MaterialFactory`-Klasse und ihrer statischen Methode `createMaterial(String, Map<String,Object>) -> Material` für jeden `MaterialConstructor` anhand des Namen und ausgewerteten Parameter ein neues `Material`-Objekt erstellt und der Variable zugewiesen. Dieses `Material`-Objekt wird beim Auswerten einer material-Operation oder einem entsprechenden Overload einer bereits existierenden Shape-Operation dem abgeleiteten `Shape`-Objekt zugewiesen.

Anmerkung: `MaterialFactory` befindet sich im `core_components`-package shapes-/grammar-package

Die `Shape`-Klasse wird um das Feld `material` (inklusive getter/setter) erweitert. In welchem das zugewiesene Material gespeichert wird. Die Klassen Roof-, Dormer- und GableShape bekommen ein neues Feld `roofMaterial` (inklusive getter) in welchem das für's Dach zugewiesene Material gespeichert wird. Dieses wird im Konstruktor zugewiesen.

Die Operationen roof, dormer und gable bekommen Überladungen, in welchem ein Material für das Dach angegeben werden kann.

Beim Ableiten eines `Shape`-Objekts in der `Evaluator`-Klasse wird das zugewiesene Material übernommen. (`roofMaterial` wird jedoch vorerst nicht beachtet)

Anmerkung: `Evaluator` befindet sich im `core_components`-package

meshgenerator-/texturing-package

Die `generateMesh`-Methode der `ModelGenerator`-Klasse (und das `meshgenerator`-package allgemein) muss so angepasst werden, dass über die Farbindizes der Facetten des zurückgegebenen `TriangleMesh`-Objekts und einer zurückgegebenen Liste an `MaterialTextures`-Objekten ermittelt werden kann, welches Material und welche Texturen welchen Fassaden zugeordnet wurden.

Neu dazu kommt die Klasse `TexturingAlgorithm` (in dem neuen `cgashape.texturing`-package) mit der statischen Methode

```
List<MaterialTextures> execute(TriangleMesh inputMesh,
      int inputMaterialIdx, TextureGenerator textureGenerator,
      TriangleMesh outputMesh)
```

diese Implementiert die Schritte 1-3 des Texturier-Algorithmus. Ebenfalls nutzt sie die angegebene `TextureGenerator`-Instanz, um die Texturen generieren zu lassen (Schritt 4). Die zurückgegebenen `MaterialTextures`-Objekte repräsentieren alle erstellten Texturen/Materialien, in der Reihenfolge, in der sie zum `outputMesh` hinzugefügt wurden. Die `execute`-Methode wird innerhalb der `generateMesh`-Methode der `MeshGenerator`-Klasse für jedes in der Grammatik zugewiesenes Material aufgerufen.

texturing_generation-package

Da sowohl Texturen als auch Shader meiner Meinung nicht ins Backend gehören, müssen wir das Rendern der Texturen in einer separaten Klasse im Frontend implementieren und diese in die jeweilige `TexturGenerator`-Klasse zur Laufzeit (als einen effektiven Singleton, also nur eine Instanz in der gesamten Applikation) injecten. Das dafür pro Textur-Art zu implementierende Interface `TextureRenderer<T>` hat die Methode `renderTexture(T) -> MaterialTextures` wobei `T` das Datenobjekt repräsentiert. Die Idee hierbei ist, dass jeder Texturgenerator sein `TextureData`-Interface selbst implementiert (siehe Abbildung 4.27), um den Datenaustausch so minimal wie möglich zu halten: Jeder `TextureRenderer` fragt nur die Daten ab, die gebraucht werden. Das resultierende `MaterialTextures`-Objekt ist ein Tupel bestehend aus dem ursprünglichen

`Material`-Objekt und der generierten Albedo-Textur und Normalen-Textur. Diese Texturen können nun im `MeshGenerator` den generierten `TriangleMeshes` zugewiesen werden.

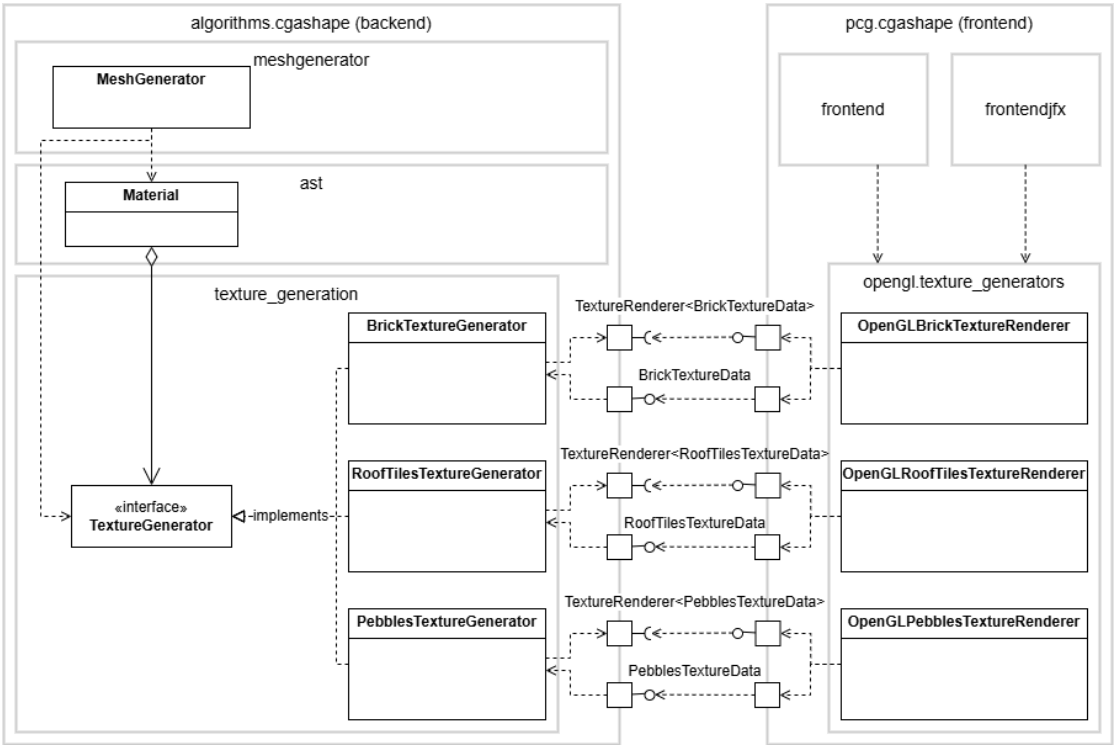

Abbildung 4.27: Klassendiagramm, Übersicht der Abhängigkeiten der neuen Klassen im texture_generation- und opengl.texturing-package zueinander und zu bzw. von umliegenden Klassen/packages

5 Implementierung

In diesem Kapitel will ich ein paar interessante Aspekte der Implementierung beleuchten. Hierbei spielt der im vorherigen Kapitel entworfene Algorithmus nur eine nebensächliche Rolle.

5.1 Das neue Typsystem

In dieser Iteration können Variablen nicht nur Fließkommazahlen, sondern auch Farben und Materialien beinhalten. Um dies zu realisieren wird aus der Klasse `Value` ein Marker-Interface, welches von den Klassen `FloatValue`, `ColorValue` und `Material` implementiert wird. Es muss also zur Laufzeit nur noch zum erwarteten Typ gecasted werden. Hierbei würden jedoch für den Nutzer kryptische Fehler entstehen, welche sich auch teilweise nicht ohne extremen Aufwand durch für den Nutzer lesbare Fehler ersetzen lassen. Die Lösung: Wir führen einen `Validator` ein, welcher die komplette Grammatik auf Typen und Variablen Nutzung soweit wie möglich überprüft, ohne die Grammatik abzuleiten. Hierfür implementiert jede `RuleAction`, jede `Condition` und der `MaterialConstructor` die Methode `validate`, welche den `ValidationContext` nutzt, um rekursiv alle Ausdrücke zu validieren und den resultierenden Typen zu ermitteln, um diesen für weitere Validierung zu nutzen.

5.2 Die neue Modelbuilder-Klasse

Die `TriangleMesh` Datenstruktur bietet zwar keine direkte Möglichkeit den Facetten Materialien zuzuweisen, es ist jedoch möglich die Facettenfarben, genauer gesagt die Farbindizes, als Materialindizes zu interpretieren und somit eine indirekte Zuweisung zu ermöglichen.

Der `MeshGenerator` verkompliziert dieses Vorhaben jedoch, da dieser zur Laufzeit für jede Form ein `TriangleMesh` erstellt und dieses mit den Netzen der Kinder-Formen vereint. Hierdurch kommt es nicht nur zu einer übermäßigen Nutzung des Arbeitsspeichers und der CPU. Es macht auch die Nutzung der Farb-/Materialindizes für das genannte Vorhaben unmöglich, da beim Vereinigen der Netze keine Farbindizes als Duplikate erkannt werden können. Heißt, es muss stets eine Liste mitgeführt werden, welche die zugewiesenen Materialien in korrekter Reihenfolge inklusive Duplikate beinhaltet. Das bedeutet: Noch größerer Aufwand und hohes Fehlerpotential ohne Single Point of Error.

Die Lösung ist hier die Einführung des `Modelbuilders`, einer Wrapper-Klasse für die `TriangleMesh` Datenstruktur, welche die Materialindizes und Materialzuweisungen verwaltet und zudem ein paar weitere Funktionen wie beispielsweise ein push/pop-transform System bietet, welches es ermöglicht, beliebig viele Transformationen zu schachteln und mit einzigartigen Tokens dafür sorgt, dass es immer korrekt genutzt wird. Somit ist es möglich, das alte Verhalten des `MeshGenerators` mit minimaler Anpassung des Codes perfekt zu emulieren und zusätzlich eine Materialverwaltung mit Single Point of Error zu realisieren. Am Ende wird das erstellte `TriangleMesh` finalisiert, der `Modelbuilder` wird „unbrauchbar“ gemacht, um nachträgliche versehentliche Änderungen zu unterbinden und es wird das `TriangleMesh` zusammen mit der Liste an zugewiesenen Materialien zurückgegeben.

5.3 Die OpenGL Einbindung

Für das Generieren der Texturen sollen Shader verwendet werden. Es wird somit eine Grafik-Schnittstelle gebraucht, welche diese unterstützt. Das JMonkey Frontend nutzt bereits OpenGL und JavaFX ist Hardware agnostic und bietet keine offizielle Möglichkeit Shader zu implementieren. Somit erscheint es sinnvoll, OpenGL für beide Frontends zu verwenden.

5.3.1 Offscreen Rendering

Für das Rendern der Texturen ist es sinnvoll, sogenannte Framebuffer zu verwenden, welche es ermöglichen, direkt in eine oder mehrere Texturen zu rendern. Somit können die Albedo- und die Normal-Map in einem Durchgang generiert werden und anschließend mit

der Methode `glReadPixels` alle Pixel aus dem Framebuffer ausgelesen und in einem BufferedImage gespeichert werden. Das Erstellen und Auslesen des Framebuffers wird von der Klasse `MaterialTextureRenderTarget` implementiert: Zwischen Erstellung und Finalisierung rendern alle OpenGL Befehle in den erstellten Framebuffer und die Methode `finalizeAndReadBackTextures` ließt den Framebuffer aus und gibt die Texturen als `BufferedImages` zurück.

Für das beschriebene Vorgehen wird an sich kein Fenster gebraucht. In OpenGL ist es zwar nicht möglich, einen Kontext ohne ein Fenster zu erstellen, auf welchem das gezeichnete Bild dargestellt werden kann, jedoch genügt hier bereits eine Größe von 1x1 Pixeln und das erstellte Fenster kann ohne Probleme so konfiguriert werden, dass es für den/die Nutzer/in unsichtbar ist. Die Erstellung des Fensters samt Kontexts erfolgt mit Hilfe des Graphics Library-Frameworks. In der neuen Klasse `GLFWOffscreenContext` werden zunächst die OpenGL Funktionen geladen und anschließend ein unsichtbares 1x1 Fenster mit einem OpenGL 4.5 Kontext erstellt. Die öffentliche Schnittstelle der Klasse ermöglicht es, den erstellten Kontext zu jeder Zeit zu „aktivieren“, sodass alle weiteren OpenGL Befehle auf diesem Kontext ausgeführt werden.

5.3.2 Einbindung in das JavaFX Frontend

Wenn die JavaFX Applikation mit der Funktion launch gestartet wird, blockiert der Main-Thread und die Anwendung wird auf einem separaten UI-Thread weitergeführt. Während der Entwicklung hatte ich angenommen, dass OpenGL Funktionen nur von dem Main-Thread aus aufgerufen werden, dies stellte sich jedoch letztendlich als Fehlinterpretation heraus. Die Funktionen müssen von dem Thread aus aufgerufen werden, auf welchem der OpenGL-Kontext initialisiert wurde. Da ich zu jenem Zeitpunkt mir dessen nicht bewusst war, beschloss ich einen Executerservice zu schreiben (bzw. von einem LLM generieren zu lassen), welcher alle Tasks auf dem Thread ausführt, auf welchem seine ProcessTask Methode aufgerufen wird. Diese wird nach dem Starten des Programms solange wiederholt aufgerufen, bis der Thread nach dem Schließen des Hauptfensters von einem Event-Listener aus interrupted wird.

5.3.3 Einbindung in das JMonkey-Frontend

Die JMonkey-Engine nutzt bereits OpenGL, jedoch ist die Engine nicht darauf ausgelegt, dass ihr OpenGL-Kontext von außerhalb mitbenutzt wird, so dass es hier zu einigen

Fehlern kommt. Es muss also ein neuer Kontext erstellt werden, um diesen für die Generierung der Texturen nutzen zu können. In jedem Fall sind die Textur-Generatoren jedoch auf die Funktionen angewiesen, welche die JMonkey-Engine bei der Erstellung ihres Kontexts geladen hat, da das nachträgliche Laden von weiteren Funktionen nicht von GLFW unterstützt wird. Das heißt, es muss eine Möglichkeit implementiert werden das Laden der Funktionen bei der Erstellung des `GLFWOffscreenContext` zu unterbinden (wurde realisiert durch einen flag Parameter). Zudem muss bei der Initialisierung von JMonkey eine OpenGL Version gewählt werden, welche alle Funktionen unterstützt, die in den Generatoren gebraucht werden (in unserem Fall OpenGL 3.3).

5.4 Texturen laden ohne Dateisystem

In JMonkey verwendet man normalerweise einen `AssetManager`, welcher Bilddateien aus dem Dateisystem lädt und diese in Texturen konvertiert. Es ist jedoch möglich, mit dem `AWTLoader` eine Textur direkt aus einem `BufferedImage` zu erstellen, sodass wir hierfür nicht auf das Dateisystem angewiesen sind.

In JavaFX kann ein `Image` entweder aus dem Dateisystem oder direkt aus einem `InputStream` geladen werden. Um uns hier den Umweg über das Dateisystem zu sparen verwenden wir `ImageIO` um das `BufferedImage` als eine PNG-Datei enkodiert in einen `OutputStream` zu schreiben, welchen wir direkt in einen `InputStream` pipen aus welchem JavaFX das Bild auslesen kann.

5.5 Entfernen der NaN-Vertices

Der bisherige Algorithmus, um in einem Dreiecksnetz überlappende Eckpunkte zu einem zusammenzuführen, ist aus Performancegründen so implementiert, dass keine Eckpunkte tatsächlich gelöscht werden. Stattdessen wird ihnen eine Position zugewiesen, welche nur aus NaN-Werten besteht. Normalerweise ist dies kein Problem, da auf die Eckpunkte immer nur indirekt über einen Index zugegriffen wird. Auch beim Senden der Daten an die Grafikkarte besteht hier kein Problem, da NaN an gültiger IEEE754-Float-Wert ist, welcher in allen gängigen Grafikkarten unterstützt wird, zudem wird der Wert nicht indiziert und somit auch nie gelesen bzw. interpretiert. Beim Exportieren müssen jedoch

alle Eckpunkte gültig sein (zumindest weigert sich blender, eine Wavefront-OBJ-Datei mit NaN-Werten einzulesen).

Es muss also eine Methode implementiert werden, welche alle Eckpunkte mit NaN-Werten aus der Liste der Eckpunkte entfernt und alle Indizes entsprechend anpasst. Hierfür werden die Indizes aller zu löschenden Eckpunkte sortiert in ein Array gespeichert, die jeweiligen Eckpunkte werden gelöscht und anschließend wird durch alle Facetten iteriert. Für jede Facette wird für jeden Eckpunkt-Index (mit einer Abwandlung des Binary-Search-Algorithmus) ermittelt, wie viele Eckpunkte vor Diesem gelöscht wurden und der Index wird entsprechend verringert. Der Algorithmus hat somit eine Laufzeitkomplexität von $n \cdot log(n)$, wobei n die Anzahl der Dreiecke ist.

6 Evaluation

Abbildung 6.1: Ein texturiertes Haus nach Ende der Iteration

6.1 Abgleich mit der Spezifikation

6.1.1 UC-1 - Ein texturiertes Modell generieren lassen

Nach dem Ende der Iteration ist es nun möglich, in der Form-Grammatik Materialien zu definieren und diesen parametrisierte Textur-Generatoren zuzuweisen. Die Parameter sind hierbei optional und haben sinnvolle Standardwerte, welche teilweise aus angegebenen Parametern neu berechnet werden. Die Materialien können jeder Fassade des zu generierenden Gebäudes innerhalb der Regeln der Form-Grammatik zugewiesen werden. Bei Anwendung der Regeln werden die Materialien, sofern sie nicht überschrieben werden, übernommen. Für jede Fassade, welcher ein Material zugewiesen wurde, wird bei der Generierung des Modells eine Textur generiert (sofern der Textur-Generator

nicht SolidColor ist), welche perfekt auf die Fassade abgestimmt ist und hierbei die für den Textur-Generator angegebenen Parameter so gut wie möglich berücksichtigt. Hierfür nutzt das System die Grafikkarte, welche auf das schnelle Rendern vieler Pixel mit gleichen Parametern optimiert ist und verwendet Formeln und Vorgehensweisen, welche für die Arbeitsweise der Grafikkarte optimiert wurden. (Performance-Tests wurden jedoch nicht durchgeführt) Vorraussetzung ist hierfür im aktuellen System OpenGL 3.3. Das JavaFX-Frontend nutzt aktuell OpenGL 4.5, um mit neuerer Hardware auch zukünftig kompatibel zu sein, jedoch ist fraglich ob dies benötigt wird.

In der Abbildung 6.1 wurde mit einer angepassten *house.grammar* (siehe Anhang A.3) ein Gebäude erstellt, welches zum einen alle verfügbaren Texturgeneratoren nutzt und zum anderen, mit einem angepassten Material im äußeren Eingangsbereich, exemplarisch die Anpassungsmöglichkeiten der Backsteine zeigt. Die anderen Textur-Generatoren sind bezüglich der Anpassungsmöglichkeiten vergleichbar.

6.1.2 UC-2 - Das generierte Modell exportieren

Das generierte Modell lässt sich im weit verbreiteten Wavefront(OBJ)-Format mitsamt allen Texturen im Portable-Network-Graphic(PNG)-Format exportieren. Das exportierte Modell ist ohne Probleme in blender importierbar und sieht, nachdem in jedem Material Backface-Culling aktiviert wurde, nahezu identisch zur Vorschau im JavaFX-Frontend aus (siehe Abbildung 6.2).

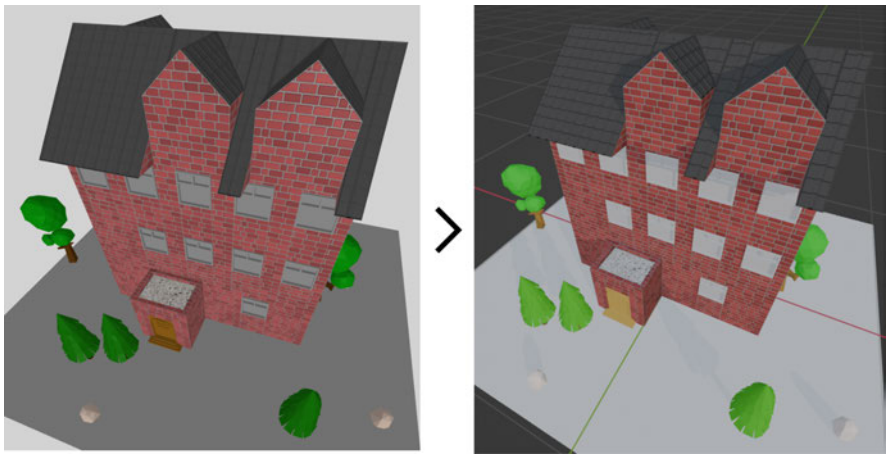

Abbildung 6.2: Texturiertes Haus mit Startwert 1 - Im Programm, In Blender

Hierbei werden jedoch einige Texturen erstellt, welche die Anzahl der Fassaden zum Teil weit überschreiten (siehe Abbildung 6.3). Grund hierfür ist eine ungünstige Geometrie der generierten Gebäude, welche es dem Algorithmus erschwert Fassaden korrekt zu erkennen (siehe Abschnitt 6.2.2).

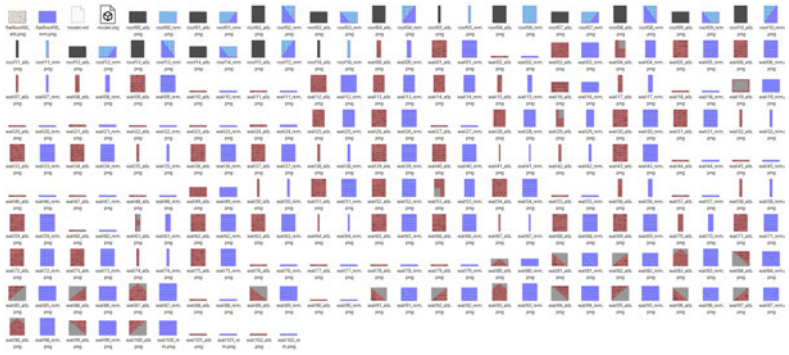

Abbildung 6.3: Alle exportierten Dateien für ein Haus mit 13 Fassaden (inklusive Dach)

6.2 Abgleich mit weiteren Anforderungen

6.2.1 Rückwärtskompatibilität

Die nicht funktionale Anforderung US-6, welche sich mit der Rückwärtskompatibilität befasst, besagt, dass Texturen optional sind und somit, wie im Akzeptanztest beschrieben, die Grammatik *house.grammar* mit dem gleichen Startwert vor und nach der Iteration zu dem gleichen Ergebnis und somit der selben Vorschau führen sollte. In Abbildung 6.4 und Abbildung 6.6 wird die Grammatik mit Startwert 0 in beiden Frontends ausgewertet und die Vorschau verglichen.

JMonkey-Frontend

Im JMonkey-Frontend sind deutliche Farbunterschiede zu erkennen. Diese sind darauf zurückzuführen, dass nach der Iteration alle Flächen (inklusive der einfarbigen) eine Textur verwenden, wohingegen vor der Iteration für die Einfärbung der Modelle ausschließlich Eckpunktfarben (Vertex-Colors) verwendet wurden. In der JMonkey-Engine werden Texturen gamma-korrigiert, dies ist jedoch bei Vertex-Colors nicht der Fall (siehe Abbildung

6.5 links). Wenn wir manuell die Vertex-Colors gamma-korrigieren (siehe Abbildung 6.5 rechts), sehen die texturierte und die eingefärbte Fläche nahezu identisch aus.

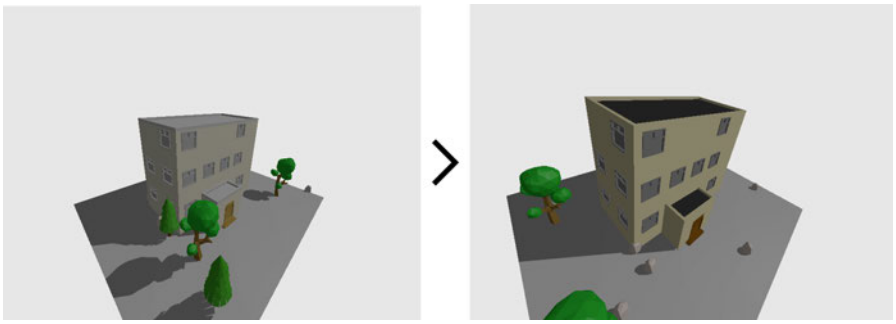

Abbildung 6.4: Haus mit Startwert 0 im JMonkey-Fontend - Vorher, Nachher

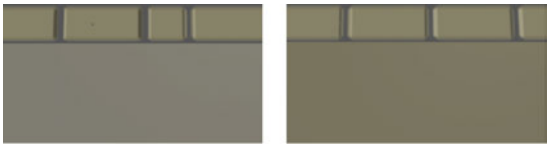

Abbildung 6.5: Gammakorrektur in JMonkey: Textur (oben) und mit Vertex-Colors eingefärbte Fläche (unten) haben die gleiche Farbe.

JavaFX-Frontend

Im JavaFX Frontend gibt es große Unterschiede in der Beleuchtung, da versucht wurde das JavaFX-Frontend so weit wie möglich an das JMonkey-Frontend optisch anzupassen. Des Weiteren ist das Model entlang der Z-Achse gespiegelt. Dies ist jedoch lediglich eine Fehlerkorrektur, da JavaFX eine andere Orientierung als die JMonkey-Engine hat und dies bei der Konvertierung vor dieser Iteration nicht korrekt gehändelt wurde.

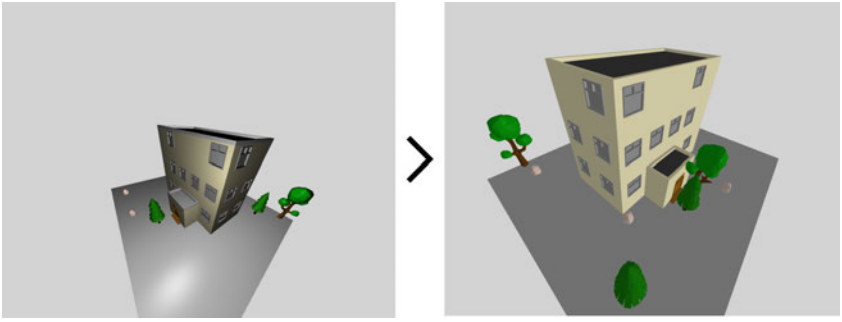

Abbildung 6.6: Haus mit Startwert 0 im JavaFX-Frontend - Vorher, Nachher

Das Modell

Unabhängig vom Frontend weißt das Modell einen Unterschied bei den Materialien der Flachdachform auf. Statt einer eigenen Farbe nimmt die Umrandung die Farbe der Wand ein. Des Weiteren wird der Dachfläche ein Material zugewiesen, welches dunkler ist als vor der Iteration. Hier wurden bei der Implementation kreative Entscheidungen getroffen, welche nicht spezifiziert wurden und sich auch aus keiner Anforderung ableiten lassen. Auch wenn sich das vorherige Ergebnis mit wenig Aufwand (durch das Verwenden der material-Operation kurz vor der roof-Operation mit dem Material-Overload) nachahmen lässt, so wurde hier dennoch falsch implementiert.

6.2.2 Keine Nahten

Die Anforderung US-4 besagt, dass es nicht zu auffälligen Wiederholungen in der Textur oder Nahten zwischen oder innerhalb der Fassaden kommen darf. Wiederholungen gibt es nur bei der Dachpfannen-Textur und hier sind sie explizit erwünscht. Nahtstellen werden von den implementierten Textur-Generatoren durch realistische Umrandungen/Abgrenzungen vermieden.

Bei genauerem Betrachten der generierten Modelle fällt jedoch auf, dass entlang vieler Fassaden eindeutig ungewollte Abgrenzungen zu sehen sind (siehe Abbildung 6.7), welche zwar keine direkten Nahten aufweisen, jedoch trotzdem einen ähnlich negativen Einfluss auf das Gesamtbild haben. Diese sind darauf zurückzuführen, dass die einzelnen Formen zwar aneinander liegen, jedoch in vielen Fällen aufgrund der vorangehenden Zerteilung nur ein Eckpunkt oder auch gar kein Eckpunkt überlappt, sodass der Texturier-Algorithmus die Fassade nicht korrekt als zusammenhängend erkennen kann.

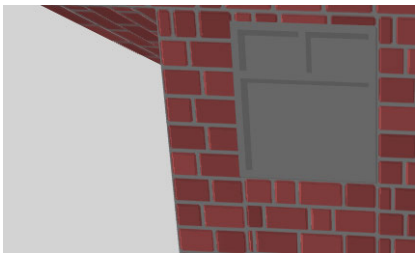

Abbildung 6.7: Auffällige Naht unter dem Fenster

7 Fazit und Ausblick

In dieser Arbeit wurde nach den Phasen der Softwareentwicklung ein Inkrement für das cgashape-Projekt in dem „Procedural Content Genration“ Repositittory entwickelt, welches es ermöglicht, aus einer Form-Grammatik texturierte Gebäude generieren zu lassen.

In der Anforderungsanalyse wurde zuerst das bestehende System analysiert und anschließend wurden aus der Leitfrage dieser Bachelorarbeit plausible Nutzeranforderungen extrapoliert und festgehalten. Für diese wurde in der Spezifikation ein MVP ermittelt und es wurde festgelegt, welche Anpassungen an der Form-Grammatik vorgenommen werden sollen, welche Texturarten es geben soll, welche Einstellungen der Nutzer verändern können soll und wie diese generierten Texturen auszusehen haben. Hierbei wurde zwischen einer guten Nutzererfahrung und einem tragbaren Entwicklungsaufwand abgewägt. Ebenfalls wurde festgelegt, in welcher Form das Modell exportiert werden kann.

In dem Entwurf wurde ein Algorithmus zur Isolation, Ausrichtung, Analyse und Texturierung einzelner Fassaden in einem 3D-Modellnetz mit zugewiesenen Materialien von grund auf entwickelt. Ebenfalls wurden Formeln für die Generierung von Texturdetails in der Albedo- und Normal-Map für alle 3 Texturarten gefunden, um diese in Shadern zu implementieren. Als Grundlage hierfür wurde eine Formel mit zugehöriger Datenstruktur entwickelt, welche die minimale Distanz zur Außenkante eines Polygons berechnet (mit einer gesonderten Behandlung der nach innen gerichteten Kanten). *Dies war wichtig für ein realistisches Aussehen der Fassadenränder.* Zusätzlich wurde ein Algorithmus entwickelt, welcher diese Datenstruktur aus den Außenkanten des Polygons aufbaut. Zu guter Letzt wurden alle notwendigen Änderungen im Projekt ermittelt und dokumentiert.

In der Implementation wurde die für die Textur-Generierung zu verwendende Grafik-Schnittstelle entschieden und die zusätzlich notwendigen Änderungen wurden implementiert. Als Nebenprodukt ist ein Validierungsystem entstanden, welches die Grammatik vor der Evaluierung vollständig validiert, anstatt nur dann einen Fehler zu werfen wenn

bei der Generierung des Modells eine Regel angewendet wird, welche aufgrund einer nicht anwendbaren Form-Operation fehlschlägt.

7.1 Mögliche Verbesserungen und Optimierungen

7.1.1 Abgrenzungen innerhalb der Fassaden vermeiden

In einer zukünftigen Iteration wäre es sinnvoll eine Lösung für die ungewollten Abgrenzungen innerhalb der Fassaden (siehe Abschnitt 6.2.2) zu finden. Eine mögliche Lösung wäre, bereits in den Mesh-Generatoren der einzelnen Formen zusammenhängende Fassaden und ihre Außenkanten zu identifizieren. (Hierfür bräuchte man einen Kontext welcher dem jeweiligen Mesh-Generator mitteilt an welchen Seiten welche Fassaden-Formen mit welchem Material in welchem Winkel anliegen. Zudem bräuchte man geeignete Algorithmen und Datenstrukturen zum Sammeln der Fassaden und Außenkanten und zur vorzeitigen Eliminierung von entgegengesetzten deckungsgleichen Kanten.) Im Texturier-Algorithmus würde man anschließend die Außenkanten zu einem Polygonzug zusammensetzen und bei Bedarf für die Texturgenerierung triangulieren.

7.1.2 Realistischere Materialvergabe bei Detail-Modellen

Die Detail-Modelle des cgashape-Projekts besitzen jeweils nur ein Material, dessen Zuweisung hartkodiert ist. Zudem wurden sie so modelliert, dass sie ein rechteckiges Loch ausfüllen. Bei Detail-Modellen, welche abgerundet oder abgeschrägt sind wie beispielsweise *door.obj* führt dies jedoch zu einem Problem, da die hierfür ins Modell eingebaute Wandgeometrie das gleiche Material zugewiesen bekommt wie das Modell selbst (siehe Abbilding 7.1). Hier wäre es sinnvoll den Detail-Modellen beim Modellieren eigene Materialien zuzuweisen und die Farben und Texturen dieser von dem System auslesen und konvertieren zu lassen. (Hierfür muss das der Code der `MeshGeneratorDetails`-Klasse angepasst werden. Zudem muss es im MeshGenerator eine Möglichkeit geben Materialien mit Texturen zuzuweisen, welche von dem Texturier-Algorithmus ignoriert werden) Eins dieser Materialien könnte automatisch anhand des Namens als Wand-Material erkannt und mit dem der Detail-Form zugewiesenen Material ersetzt werden.

Abbildung 7.1: Im generierten Modell platziertes Türmodell

7.1.3 Texturatlas implementieren

In der aktuellen Implementation werden sehr viele Texturen erstellt, welche sich gut zu einem Atlas zusammenfassen lassen. Zhang et al. 2021 [17] stellt einen Ansatz vor, mit welchem sich aus einem Texturierten Gebäudemodell effizient ein Texturatlas generieren und die zugehörige Transformation der Texturkoordinaten im Modell berechnen lässt. Diesen könnte man in einer nachfolgenden Iteration implementieren. (Der Code in dieser Iteration wurde so geschrieben, dass der Texturier-Algorithmus hierfür nur leicht angepasst werden muss)

Literaturverzeichnis

[1] : *ANTLR*. – URL https://www.antlr.org/. – Zugriffsdatum: 8.12.24

[2] : *Painting: Filling, Stroking and Marker Symbols – SVG 1.1 (Second Edition)*. – URL https://www.w3.org/TR/SVG11/painting.html. – Zugriffsdatum: 5.12.24

[3] : *Procedural City Generator | 3D City Maker | ArcGIS CityEngine*. – URL https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview. – Zugriffsdatum: 4.12.24

[4] : *Texturing: Essential knowledge—ArcGIS CityEngine Resources | Documentation*. – URL https://doc.arcgis.com/en/cityengine/2023.1/cga/cga-texturing-essential-knowledge.htm. – Zugriffsdatum: 4.12.24

[5] Elliott, Hamish R.: *Using Mesh Shaders for isosurface extraction*, The University of Waikato, Dissertation, 2022

[6] Georgiou, Yiangos ; Averkiou, Melinos ; Kelly, Tom ; Kalogerakis, Evangelos: Projective Urban Texturing. In: *2021 International Conference on 3D Vision (3DV)*, 2021, S. 1034–1043

[7] Gordon, V. S. ; Clevenger, John L.: *Computer Graphics Programming in OpenGL with C++*. Berlin, Boston : Mercury Learning and Information, 2021. – URL https://doi.org/10.1515/9781683926719. – ISBN 9781683926719

[8] Kelly, Tom ; Guerrero, Paul ; Steed, Anthony ; Wonka, Peter ; Mitra, Niloy J.: FrankenGAN: guided detail synthesis for building mass models using style-synchonized GANs. In: *ACM Trans. Graph.* 37 (2018), Dezember, Nr. 6. – URL https://doi.org/10.1145/3272127.3275065. – ISSN 0730-0301

[9] Korunoski, Mladen ; Gushev, Marjan ; Zdraveski, Vladimir: Programmable vs. Fixed-Function Pipeline in Real-Time Computer Graphics. In: *IEEE EUROCON 2019 -18th International Conference on Smart Technologies*, 2019, S. 1–5

[10] McReynolds, Tom ; Blythe, David ; Fowle, C ; Grantham, B ; Hui, S ; Womack, P: Programming with opengl: Advanced rendering. In: *SIGGRAPH* Bd. 97 Citeseer (Veranst.), 1997, S. 144–153

[11] Rosenfeld, Viktor ; Bress, Sebastian ; Markl, Volker: Query Processing on Heterogeneous CPU/GPU Systems. In: *ACM Comput. Surv.* 55 (2022), Januar, Nr. 1. – URL https://doi.org/10.1145/3485126. – ISSN 0360-0300

[12] Stiny, George: Introduction to shape and shape grammars. In: *Environment and planning B: planning and design* 7 (1980), Nr. 3, S. 343–351

[13] Stiny, George: Spatial relations and grammars. In: *Environment and Planning B: Planning and Design* 9 (1982), Nr. 1, S. 113–114

[14] Stiny, George ; Gips, James: Shape grammars and the generative specification of painting and sculpture. In: *IFIP congress (2)* Bd. 2 Citeseer (Veranst.), 1971, S. 125–135

[15] Watzl, Thorben: *Prozedurale Modellierung von Gebäuden anhand von Mustern in Form von Shape-Grammar*. 2015

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

[17] Zhang, Xuequan ; Liu, Wei ; Liu, Bing ; Zhao, Xin ; Hu, Zihe: Size-Adaptive Texture Atlas Generation and Remapping for 3D Urban Building Models. In: *ISPRS International Journal of Geo-Information* 10 (2021), Nr. 12. – URL https://www.mdpi.com/2220-9964/10/12/798. – ISSN 2220-9964

A Anhang

A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet. *Alle gelisteten Programme sind Kostenlos und zu einem Großteil Open-Source*

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	Verwendung
LaTeX	Textsatz- und Layout-Werkzeug verwendet zur Erstellung dieses Dokuments
TeXstudio	LaTeX-Entwicklungsumgebung verwendet zur Erstellung dieses Dokuments
Mathcha	(*Online*) Mathe-Notations-Programm (wysiwyg) verwendet zur visuellen Eingabe der Formeln für dieses Dokument *Die Offline-Version benötigt eine kostenpflichtige Lizenz*
desmos	*Online* Grafikrechner verwendet zur Entwicklung einiger Formeln und Erstellung weniger Grafiken
GeoGebra	(Online) Grafikrechner verwendet zur Erstellung einiger Grafiken
draw.io	*Online* Diagramm-Erstellungs-Porgramm verwendet zur Erstellung einiger Diagramme
blender	3D-Modellierprogramm verwendet zur Erstellung einiger Grafiken
Shadertoy	Minamistische *Online* Shader-IDE und -Plattform verwendet zur Entwicklung der Textur Shader
ChatGPT	AI-Chatbot verwendet zur Generierung des Codes einer nebensächlichen (und letztlich irrelevanten) Klasse

A.2 Listing - Ausklapp-Algorithmus

```
$queue := CREATE_QUEUE()
für jedes $teilnetz in $(alle zusammenhängenden Teilnetze):
   $facette := beliebige Facette in $teilnetz
   ENQUEUE($queue, ($facette, nil))

[fassaden_teilnetze_schleife]
solange $queue nicht leer:
   ($start_facette, $abgespaltete_facette) := DEQUEUE($queue)
   $kante_mit_größtem_Winkel := nil
   $größter_Kantenwinkel := -1
   > makiere alle Facetten als nicht ausgerichtet
   > makiere alle Halbkanten als nicht besucht
   > richte $start_facette an XZ-Ebene aus
   > schreibe berechnete Positionen als
      Positionen im 2D-Fassadenraum
   > makiere $start_facette als ausgerichtet

   $halbkanten_queue := CREATE_QUEUE()
   für jede $halbkante in $start_facette:
      wenn $halbkante hat gegenüberliegende Halbkante:
         $gegenüber := Halbkante gegenüber von $halbkante
         ENQUEUE($halbkanten_queue, $gegenüber)
         > makiere $halbkante als besucht
         > makiere $gegenüber als besucht
   $ist_gescheitert := False

   [teilnetz_halbkanten_schleife]
   solange $halbkanten_queue nicht leer:
      $halbkante := DEQUEUE($halbkanten_queue)
      $gegenüber := Halbkante gegenüber von $halbkante
      $winkel := Winkel zwischen Normalen der anliegenden Facetten
                 von $halbkante und $gegenüber
      wenn $winkel > $größter_Kantenwinkel:
         $kante_mit_größtem_Winkel := $halbkante
```

```
        $größter_Kantenwinkel := $winkel
    $facette := anliegende Facette von $halbkante
    richte $facette an XZ-Ebene und $gegenüber im 2D-Fassadenraum aus

    wenn $facette als ausgerichtet makiert und
          berechnete Positionen stimmen nicht mit
          bereits geschriebenen Positionen überein:
       $ist_gescheitert := True
       breche [teilnetz_halbkanten_schleife] ab

    > schreibe berechnete Positionen als
       Positionen im 2D-Fassadenraum
    > makiere $facette als ausgerichtet
    für jede $nächste in (Halbkanten von $facette)  $halbkante:

       wenn $nächste hat gegenüberliegende Halbkante und
             $nächste nicht als besucht markiert:

          $gegenüber := Halbkante gegenüber von $halbkante
          ENQUEUE($halbkanten queue, $gegenüber)
          > makiere $halbkante als besucht
          > makiere $gegenüber als besucht

[/teilnetz_halbkanten_schleife]

wenn $ist_gescheitert:
   ($facette_a, $facette_b) := spalte $kante_mit_größtem_Winkel

   // wir wissen nicht ob wir soeben unser Teilnetz
   // entzwei gespalten haben, somit müssen wir eine
   // Referenz auf beide Facetten einreihen
   // und später überprüfen ob beide Facetten weiterhin
   // zum gleichen Teilnetz gehören (siehe unten)
   ENQUEUE($queue, ($facette_a, $facette_b))

// wenn beide Facetten nicht zum gleichen Teilnetz gehören
```

```
   // (bzw. wir nicht das Gegenteil beweisen können)
   // müssen wir die 2. unbedingt einreihen
   wenn $abgespaltete_facette != nil und $abgespaltete_facette
         ist nicht als ausgerichtet makiert:
      ENQUEUE($queue, ($abgespaltete_facette, nil))

   // wenn zu diesem Zeitpunkt $ist_gescheitert nicht gesetzt ist,
   // ist das zu $start_facette gehörende Teilnetz
   // vollständig und ohne Konflikte aufgeklappt.

[/fassaden_teilnetze_schleife]
```

A.3 Modifizierte house.grammar

```
Variables:
buildingHeight = [0.3;1];
floorHeight = 0.25;
segmentWidth = 0.2;
windowWidth = 0.12;
windowHeight = 0.12;
doorWidth = 0.12;
doorHeight = 0.2;
doorRoomWidth = [0.25;0.35];
doorRoomDepth = [0.15;0.25];
dormerWidth = 0.25;
dormerOffset = 0.05;
dormerWidthWOffset = 0.3;
wall = Bricks{};
wall2 = Bricks{
   brickWidthA: 0.4,
   brickWidthB: 0.4,
   brickHeight: 0.4,
   brickColor: Color("#249"),
   brickCornerRadius: 0.1,
   mortarColor: Color("white"),
```

```
};
roof = RoofTiles{};
flatRoof = Pebbles{};
flatRoofWall = SolidColor{color: Color("white")};

Rules:
Lot --> material(wall) extrude(buildingHeight) Body
Body --> component_split("side_faces"){Facade,FrontFacade,Facade} component_sp

# Facade
FrontFacade --> repeated_split("X",floorHeight){BaseFloorFacade,FloorFacade}
Facade --> repeated_split("X",floorHeight){FloorFacade}
FloorFacade --> repeated_split("Z",segmentWidth){FloorFacadeSegment}

# Door or Door Room
BaseFloorFacade --> repeated_split("Z",segmentWidth){DoorSegment,FloorFacadeSe
BaseFloorFacade --> split("Z",1r,doorRoomWidth,2r){FloorFacadeSegment,DoorRoom
DoorRoomSegment --> extrude(doorRoomDepth) DoorRoom
DoorRoom --> material (wall2)
   component_split("side_faces"){DoorRoomLeft,DoorRoomFront,DoorRoomRight}
   component_split("top"){DoorRoomTop}
DoorRoomTop --> material(flatRoofWall) roof("flat", flatRoof) DoorRoof:0.5
DoorRoomTop --> roof("skeleton",20,0.01,0) DoorRoof:0.5
DoorRoomLeft --> repeated_split("Z",segmentWidth){DoorSegment,FloorFacadeSegme

# Roofs
Top --> roof("ridge",40,0.1,0, roof) DormerRoof:0.5
Top --> roof("flat") FlatRoof:0.5
DormerRoof --> component_split("side_faces"){Back,Side,Front,Side,Down2,Down1
Down1 -->  split("Z",1r,0.25,1r){Wall,Invisible,Wall}
Side --> split("Z",dormerOffset,1r){Wall,SideWOOffset}
SideWOOffset --> repeated_split("Z",dormerWidthWOffset){DormerSegmentWidthOffs
DormerSegmentWidthOffset --> split("Z",dormerWidth,1r){DormerS,Wall}
#Invisible --> remove()
Front --> split("X",floorHeight,1r){Floor1,Floor2}
Floor1--> split("Z",1r,segmentWidth,1r){Wall,FloorFacadeSegment,Wall}
```

```
DormerS --> dormer("wall",0.1, roof) Dormer:0.3
DormerS --> split("X",1r){Wall}:0.7
#Dormer --> component_split("side_faces"){DFront,DSide,DTopFront,DSide}
#DFront --> split("Z",1r,segmentWidth,1r){Wall,WindowSegment,Wall}

# Windows
FloorFacadeSegment --> split("Z",1r){Dummy}:0.3
FloorFacadeSegment --> split("Z",1r){WindowSegment}:0.7
WindowSegment --> split("Z",1r,windowWidth,1r){Wall,WindowSlice,Wall}
WindowSlice --> split("X",1r,windowHeight,0.01){Wall,Window,Wall}
Window --> detail("window") Unused

# Door
DoorSegment --> split("Z",1r,doorWidth,1r){Wall,DoorSlice,Wall}
DoorSlice --> split("X",doorHeight,1r){Door,Wall}
Door --> detail("door") Unused
```

Erklärung zur selbständigen Bearbeitung

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