HAW HAMBURG

BACHELOR THESIS
Benjamin Schröder

Beispiel-basierte inverse prozedurale Generierung für zweidimensionale Szenen

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

Benjamin Schröder

Beispiel-basierte inverse prozedurale Generierung für zweidimensionale Szenen

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. Peer Stelldinger

Eingereicht am: 11. Juli 2024

Benjamin Schröder

Thema der Arbeit

Beispiel-basierte inverse prozedurale Generierung für zweidimensionale Szenen

Stichworte

2D, Prozeduale Generierung, Inverse prozedurale Generierung, Ableiten einer Graphgrammatik, Polygone

Kurzzusammenfassung

Die Erstellung von immersiven fiktiven Welten in Videospielen und anderen Simulationen erfordert einen großen manuellen Aufwand der Entwickler, welcher oftmals mit viel monotoner Modellierungsarbeit einhergeht. Um diesen Aufwand zu minimieren, wurden über die Jahre viele Verfahren entwickelt, die durch das Automatisieren entsprechender Aufgaben Abhilfe schaffen sollen. Solche Verfahren fallen in den Bereich der prozeduralen Generierung. Hierbei gibt es eine Vielzahl verschiedene Ansätze, von denen einige in dieser Arbeit betrachtet werden. Viele dieser Verfahren sind jedoch nur stark eingeschränkt anwendbar oder erfordern das manuelle Bereitstellen von Regeln, wofür oft ein tiefes Verständnis des entsprechenden Verfahrens vorausgesetzt wird. In dieser Arbeit wird eine bestimmte Art von prozeduralen Generierungsverfahren genauer beleuchtet, welche auch diese Mängel beseitigen soll, indem solche Regeln automatisch aus einer gegebenen Beispielstruktur abgeleitet werden. Nach einem kurzen Vergleich wird eines solcher Verfahren im Detail erläutert und prototypisch umgesetzt.

Benjamin Schröder

Title of Thesis

Example-based inverse procedural generation for two-dimensional scenes

Keywords

2D, procedural generation, inverse procedural generation, deriving a graph grammar, polygons

Abstract

The creation of immersive fictional worlds in video games and other simulations requires a great deal of manual effort on the part of developers, often accompanied by a lot of monotonous modelling work. To minimize this effort, many methods have been developed over the years to automate these tasks. Such methods fall under the umbrella of procedural generation. There are many such approaches, some of which are discussed in this paper. However, many of these procedures are very limited in their applicability or require the manual provision of rules, which often requires a deep understanding of the corresponding procedure. This paper examines a specific type of procedural generation methods that aims to overcome these shortcomings by automatically deriving such rules from a given example structure. After a short comparison, one such procedure is explained in detail and implemented as a prototype.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

DPO Double Pushout.

GBS Graph Boundary String.

MVC Model-View-Controller.

PCG Prozedurale Content Generierung.

WFC Wave Function Collapse.

1 Einleitung

1.1 Motivation

Die Erstellung von fiktiven Welten spielt eine große Rolle in vielen Videospielen, Filmen, Virtual Reality Umgebungen und weiteren Bereichen der Simulation. Hierfür wird eine Vielzahl an verschiedenen Objekten und Strukturen benötigt, um ein nicht-repetitives und immersives Erlebnis für den Endnutzer zu schaffen. All dies manuell anzufertigen, stellt vor allem kleinere Indie-Entwicklerstudios vor eine große Herausforderung und kann die Entwicklungszeit signifikant in die Länge ziehen. Aber auch in größeren Teams mit einer Vielzahl von Designern nimmt die Erstellung von realistischen Welten einen Großteil der Entwicklungszeit in Anspruch und kann viele Monate dauern. [12] Hier kann an einigen Stellen nachgeholfen werden, indem man das Erstellen von Inhalten automatisiert. Entsprechende Prozesse lassen sich dem Bereich der prozeduralen Generierung zuordnen.

Mithilfe von verschiedensten Verfahren können so z.B. einzelne Dungeons oder sogar ganze Welten und darin enthaltene Gebilde automatisch erzeugt werden. Diese können eine Grundstruktur für ein komplexeres Design bilden, bei dem die Entwickler dann nur noch kleinere Details per Hand abändern oder hinzufügen müssen. [12] Andererseits existieren auch viele Videospiele, wie z.B. Minecraft[1] oder Terraria[2], die auf prozeduraler Generierung aufbauen, um ihr Spielkonzept umzusetzen. Konkret wird einem neuen Spieler hier eine komplett neue und einzigartige, aber dennoch logisch zusammenhängende Welt generiert; dies vollautomatisch und ohne zusätzlichen Aufwand für die Entwickler. Jeder Spieler bekommt so eine einzigartie Erfahrung geboten und kann das Spiel außerdem gewissermaßen unbegrenzt oft durchspielen, ohne dass es zu repetitiv wird. So etwas wäre ohne Automatisierung gar nicht erst umsetzbar.

[1] https://www.minecraft.net/ [Letzter Zugriff am 01.07.2024]
[2] https://terraria.org/ [Letzter Zugriff am 01.07.2024]

1.2 Problemstellung

Es gibt viele bekannte Verfahren, welche solche Ergebnisse unter der Verwendung von u.a. generativen Grammatiken oder Constraint-basierten Graphen erzielen können. [18] Ein Großteil dieser Verfahren erfordert jedoch menschliches Eingreifen in einige der Teilschritte, was in einigen Szenarien zu einem Problem werden kann. Hängt die Generierung von Inhalten eines Produkts z.B. von Entscheidungen des Endnutzers ab (z.B. in Spielen, in denen der Spieler dynamisch mit dem Terrain und anderen Strukturen interagiert), so kann der Entwickler keinen direkten Einfluss auf den Generierungsprozess nehmen und alles muss voll automatisiert sein. [7] Auch in Projekten, in denen dies nicht der Fall ist und der gesamte Inhalt im Voraus erstellt wird, kann das Voraussetzen von menschlicher Intervenierung als Teil des Prozesses zu einem Problem werden. Ein Beispiel hierfür wären Verfahren, die eine generative Grammatik nutzen und voraussetzen, dass dafür zunächst eine Menge an Regeln durch einen Menschen vorgegeben wird, bevor die automatische Generierung überhaupt beginnen kann (z.B. [3][3]). Das Erstellen solcher Regeln ist mit viel Arbeit verbunden und kann ohne ein ausgeprägtes Verständnis des angewandten Verfahrens sehr schwierig werden, wodurch der Einsatz eines solchen Verfahrens für viele Designer letztendlich doch nicht in Frage kommen wird. Hier setzt diese Arbeit an und untersucht Möglichkeiten zur Automatisierung des Erstellens solcher Regeln.

1.3 Ziele und Vorgehen

Spezifisch soll versucht werden, Muster in Beispielstrukturen zu identifizieren. Aus diesen Mustern sollen dann Regeln zum Zusammensetzen von Strukturen mit ähnlichen Eigenschaften abgeleitet werden. Gelingt dies, so muss ein Designer lediglich ein einziges Beispielmodell erstellen und kann damit eine kreative Vision vorgeben. Alle weiteren Schritte zum Ableiten von Variationen dieses Inputs laufen anschließend automatisch ab. Dies nennen wir *inverse* prozedurale Generierung, da der Prozess mit einem soweit fertigen Modell beginnt und daraus dann die Regeln ableitet, statt wie in den klassischen Verfahren zuerst mit der Erstellung der Regeln zu beginnen. Die Erstellung eines Beispielmodells erfordert zwar nach wie vor die Arbeit eines Designers, anschließend ist aber kein menschliches Eingreifen mehr nötig und das eigentliche Verfahren läuft vollautomatisch ab.

[3] https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=25020f8d955aee07b7dd49a3ec23b1f2a8cf1d06 [Letzter Zugriff am 01.07.2024]

Es gibt bereits verschiedene Verfahren, die einen solchen Ansatz verfolgen. Diese sind u.a. der Gitter-basierte Wave Function Collapse Algorithmus von Maxim Gumin [13] oder die nach Symmetrien suchende inverse prozedurale Modellierung von Bokeloh et al. [5].

Im Rahmen dieser Arbeit werden entsprechende Verfahren grob analysiert, deren Probleme aufgezeigt und anschließend ein neuer Ansatz vorgestellt, welcher die vorhandenen Probleme minimieren soll. Das Endergebnis der Arbeit soll dann sein, dass die Funktionsweise des neuen Konzepts ausführlich und verständlich dargestellt, und dieses anschließend prototypisch implementiert wird. Wir begrenzen uns dabei auf die Generierung von Strukturen im zweidimensionalen Raum. Der gleiche Ansatz kann auch auf den dreidimensionalen Raum erweitert werden, um so vielseitiger einsetzbar zu sein, jedoch reicht die vereinfachte Betrachtung zum Darstellen aller fundamentalen Konzepte.

2 Stand der Technik

Als Grundlage für das Verständnis des weiteren Inhalts dieser Arbeit machen wir zunächst einen kurzen Abstecher in den Bereich der prozeduralen Generierung allgemein. Wir stellen klar, was unter diesem Begriff zu verstehen ist und widmen uns außerdem kurz der zugrundeliegenden Geschichte. Dabei werden wir einige fundamentale Errungenschaften und Verfahren betrachten, die den Weg zum aktuellen Forschungsstand geprägt haben.

2.1 Prozedurale Generierung

Prozedurale Generierung, oder auch Prozedurale Content Generierung (PCG), beschreibt eine Menge von Verfahren zum algorithmischen Erstellen von Inhalten ("Content"). Dabei handelt es sich meist um Inhalte in Form von Texturen oder verschiedenen Gebilden im Kontext von Videospielen und anderen Simulationen, wie z.B. Landschaften, Flüsse, Straßennetze, Städte oder Höhlenstrukturen. [7] Auch Musik kann durch solche Verfahren generiert werden. [30]

Diese Definition ist absichtlich etwas allgemeiner gehalten, da das Aufstellen einer spezifischeren Definition nicht besonders trivial ist. Das Konzept von PCG wurde bereits aus vielen verschiedenen Blickwinkeln beleuchtet und ist für verschiedene Personen von unterschiedlicher Bedeutung. So hat z.B. ein Game Designer eine etwas andere Perspektive als ein Wissenschaftler, der sich lediglich in der Theorie mit der Thematik beschäftigt. [36] Verschiedene Definitionen unterscheiden sich in Bezug auf Zufälligkeit, die Bedeutung von "Content", oder darin, ob und in welchem Umfang menschliche Intervenierung eine Rolle in einem Verfahren spielen darf. Smelik et al. definieren "Content" als jegliche Art von automatisch generierten Inhalten, welche anhand von einer begrenzten Menge an Nutzer-definierten Parametern erzeugt werden können. [34] Timothy Roden und Ian Parberry beschreiben entsprechende Verfahren zur Erzeugung dieser Inhalte als *Vermehrungsalgorithmen* ("amplification algorithms"), da diese eine kleinere Menge von Inputparametern entgegennehmen und diese in eine größere Menge an Outputdaten transformieren. [31] Togelius et al. [36] versuchen den Bereich genauer abzugrenzen,

indem sie anhand von Gegenbeispielen aufzeigen, was *nicht* als PCG bezeichnet werden sollte. So zählt für Togelius et al. z.B. das Erstellen von Inhalten eines Videospiels mittels Level-Editor in keinem Fall als PCG, auch wenn dabei das Spiel indirekt durch z.B. automatisches Hinzufügen oder Anpassen von Strukturen beeinflusst wird. Generell wird sich in der Arbeit von Togelius et al. [36] ausführlich mit dem Problem der Definition von PCG befasst, weshalb dies hier nun nicht weiter thematisiert werden soll. Die oben genannte grobe Erklärung fasst die Kernaussage der verschiedenen Definitionen weitesgehend zusammen und sollte für unsere Zwecke ausreichen.

2.2 Klassische Verfahren

Schauen wir uns nun an, welche Verfahren über die Jahre hin entwickelt worden sind, um zu verstehen, wie wir zum momentanen Stand der Technik gekommen sind.

2.2.1 PRNG

Einer der simpelsten Ansätze für PCG beruht auf der Generation von Pseudozufallszahlen (“pseudo random number generation (PRNG)”). [14] Das ab 1982 entwickelte und 1984 veröffentlichte Weltall-Erkundungsspiel “Elite” benutzt einen solchen Ansatz, um Eigenschaften der dort zu erkundenen Planeten automatisch zu generieren. [35] Diese Idee enstand aufgrund der technischen Limitationen zur damaligen Zeit. Die Entwickler, David Braben und Ian Bell, wollten den Spielern eine Vielzahl an verschiedenen Planeten zum Erkunden bereitstellen. Da in gängiger Hardware jedoch zu wenig Speicherplatz vorhanden war, um alle geplanten Details für all diese Planeten unterzubringen, ist Braben und Bell die Idee gekommen, diese Details erst zur Laufzeit generieren zu lassen und somit das vorliegende Problem zu umgehen. Die Idee für das verwendete Verfahren beruht auf der Fibonacci-Folge. Diese beginnt mit den Ziffern 0 und 1, woraus anschließend eine unendliche Folge an weiteren Zahlen generiert werden kann, indem die letzten beiden Zahlen der Folge aufsummiert werden. So entsteht schließlich die Folge $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. Diese erzeugt natürlich keine zufällige Folge an Zahlen, hat Braben und Bell jedoch auf eine Idee gebracht. Nutzt man statt den Ziffern 0 und 1 einfach ein beliebiges Paar von Ziffern, so können viele verschiedene Sequenzen nach dem gleichen Prinzip generiert werden. Werden z.B. die Startziffern 3 und 6 gewählt, so erhält man im nächsten Schritt eine 9. Wird die Sequenz dann weitergeführt, so wird eine 15 erhalten. Statt die 15 jedoch unverändert als nächsten Element der Folge zu nutzen, kann auch lediglich die letzte Ziffer

aus dieser extrahiert werden, wodurch wir stattdessen eine 5 erhalten. Führt man die Sequenz nach diesem Prinzip weiter, erhält man eine Folge an Ziffern $(3, 6, 9, 5, 4, 9, 3, 2, 5, 7, 2, \ldots)$, die zwar nicht zufällig sind, aber zufällig wirken (daher "*Pseudo*zufallszahlen"). Diese Idee konnte anschließend fortgeführt und erweitert werden: statt nur Ziffern zu benutzen, können ebenfalls größere Zahlen genutzt werden. Statt in jedem Schritt die letzten beiden Ziffern einfach aufzusummieren, können weitere Transformationen vorgenommen werden, um die nächste Zahl der Sequenz zu bestimmen. Mittels dieses Vorgehens konnten Braben und Bell nun einen Algorithmus entwickeln, der quasi zufällige Zahlen erzeugt. Anschließend wurden diese Zahlen genutzt, um verschiedenste Eigenschaften der generierten Sternensysteme zu bestimmen, wie z.B. dessen Größe, dessen Position im Raum, die Anzahl an enthaltenen Planeten und die Preise von verschiedenen Items. Auch Namen und Beschreibungen von Planeten oder Gegenständen konnten generiert werden, indem die generierten Zahlen zum Auswählen von Wörtern in einer vordefinierten Tabelle an Adjektiven und Substantiven genutzt wurden. [35]

2.2.2 Fraktale

Etwa im gleichen Zeitraum, in dem "Elite" entwickelt wurde, haben sich weitere fundamentale Ansätze im Bereich von PCG angebahnt, welche auf die Erstellung einer anderen Art von prozeduralem "Content" abzielen: dem Erzeugen von natürlich wirkenden, organischen Strukturen. 1982 stellte Benoit Mandelbrot seine Erkenntnisse zum Zusammenhang zwischen dem Aufbau natürlicher Strukturen (wie z.B. Landschaften, Gebirge, ...) und der fraktalen Geometrie vor. [20]

Ein Fraktal ist ein geometrisches Objekt, das eine selbstähnliche Struktur aufweist. D.h. es sieht auf verschiedenen Skalen ähnlich oder identisch aus. Das bedeutet, dass ein Teil des Fraktals eine verkleinerte Kopie des gesamten Fraktals ist. [20] Ein sehr bekanntes Beispiel für eine solche Struktur ist das Sierpinski-Dreieck. Dieses kann aus einem gleichseitigen Dreieck konstruiert werden, indem man dieses mit drei kleineren gleichseitigen Dreiecken mit der halben Kantenlänge ersetzt, wobei in der Mitte eine Lücke entsteht. Ersetzt man die neu enstandenen Dreieck immer wieder bis in's Unendliche nach dem gleichen Vorgehen, so ensteht das Sierpinski-Dreieck. [33] Die ersten paar Schritte dieses Konstruktionsprozesses sind in Abbildung 2.1 zu erkennen. Eine simple Ersetzungsregel ermöglicht hier das Bilden einer komplexen Struktur und kann technisch recht trivial mittels eines rekursiven Algorithmus umgesetzt werden. Fraktale lassen sich häufig in der Natur wiederfinden, so z.B. in Schneeflocken, Küstenlinien, Wolken oder Pflanzen wie Brokkoli und Farnen, welche also algorithmisch erzeugt werden können. [20]

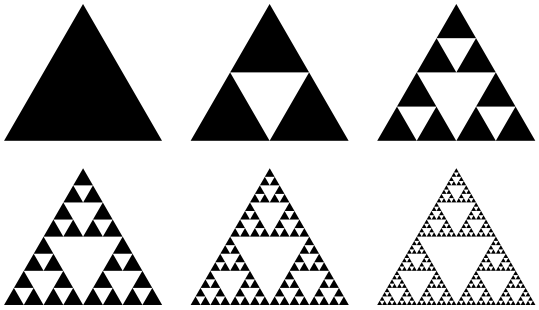

Abbildung 2.1: Konstruktion eines Sierpinski-Dreiecks.

Formal kann die Erzeugung eines solchen Fraktals mithilfe eines Lindenmayer-Systems (kurz: L-System) beschrieben werden. Dieses wurde bereits 1968 von dem Botaniker Aristid Lindenmayer vorgestellt [19], wurde zunächst allerdings nur zum Beschreiben von Wachstumsmustern in Pflanzen benutzt und erst später auf das Erstellen von geometrischen Strukturen angepasst. [29]

Ein L-System ist definiert durch das Tripel $L = (V, \omega, P)$, wobei V eine Menge an Symbolen, ω das sogennante Startwort, und P eine Menge an Produktionsregeln ist. Die Menge an Symbolen V, oder auch das *Alphabet*, besteht aus Variablen (Symbole, die ersetzt werden können) und Konstanten (Symbole, die nicht ersetzt werden können). ω ist eine nicht leere Zeichenkette bestehend aus Symbolen $\alpha \in V$, welche den Ursprungszustand des L-Systems darstellt. Die Produktionsregeln in P stellen alle möglichen Ersetzungsoperationen dar und beschreiben, womit die verschiedenen Variablen ersetzt werden können. Eine solche Produktionsregel besteht aus zwei Zeichenketten: dem Vorgänger α und dem Nachfolger β. Beim Anwenden dieser Produktion werden dann alle Symbole in der Vorgänger-Zeichenkette mit den Nachfolger-Symbolen ersetzt: $\alpha \rightarrow \beta$. Um nun verschiedene Zeichenketten aus diesem L-System ableiten zu können, wird zunächst das Startwort ω herangezogen, auf wessen Variablen dann iterativ die vorhandenen Produktionen angewandt werden können. In jeder Iteration werden dabei so viele Regeln wie möglich zur gleichen Zeit angewandt, was ein L-System ganz klar von formalen Grammatiken unterscheidet. [29]

Aus den erzeugten Zeichenketten eines L-Systems können nun fraktale geometrische Strukturen erzeugt werden, indem man die verwendeten Symbole z.B. als Kommandos einer Turtle-Grafik [26] interpretiert. Ein Beispiel hierfür ist die Erzeugung einer Koch-Kurve [15]. Das entsprechende L-System kann definiert werden durch $L = (V, \omega, P)$ mit $V = \{F, +, -\}$, $\omega = F + +F + +F$, und $P = \{F \rightarrow F - F + +F - F\}$, wobei F das Zeichnen einer Linie, $+$ das Drehen der Schildkröte um 60° nach rechts, und $-$ das Drehen der Schildkröte um

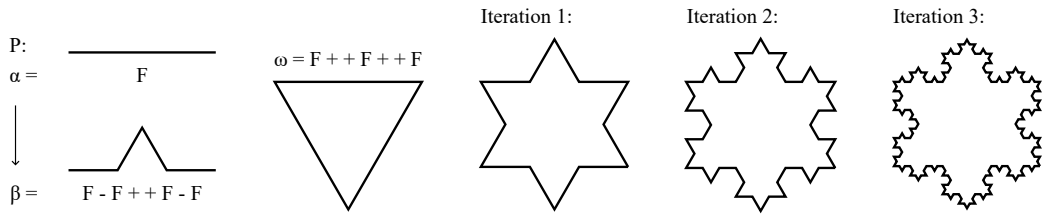

Abbildung 2.2: Konstruktion einer Koch-Kurve durch ein L-System.

60° nach links symbolisieren. Das entsprechende L-System und die ersten drei Iterationsschritte bei dessen Anwendung sind in Abbildung 2.2 dargestellt. [21]

2.2.3 Perlin Noise

Aufgrund der rekursiven Struktur solcher Fraktale wird deren Berechnung schnell sehr rechenintensiv. Dies ist und war vor allem damals für viele Anwendungsfälle ein nicht vernachlässigbares Problem. [4] Daher war eine nicht rekursive Alternative nötig, welche schließlich etwas später in Form von Rauschfunktionen bzw. *Noise* geliefert wurde. Der wohl bekannteste Vertreter dieses Konzepts ist das 1985 von Ken Perlin entwickelte [27] und 2002 verbesserte [28] *Perlin Noise*, welches seit dessen Veröffentlichung nicht mehr aus der Welt der Computergrafik wegzudenken ist. Mithilfe von Perlin Noise können eine Reihe von Zufallswerten erzeugt werden. Hierbei sind sich nah beieinander liegende Werte stets sehr ähnlich und es gibt keine starken Ausschläge, weshalb der entstehende Verlauf sehr organisch wirkt. Aufgrund von dieser Eigenschaft eignet sich Perlin Noise ebenfalls perfekt zum Erzeugen von natürlichen Strukturen. Generell ist der erzeugte Kurvenverlauf vielseitig einsetzbar und findet somit in vielen verschiedenen Anwendungen einen Nutzen, darunter bei der Synthese von Texturen oder auch im Bereich der Animation. [17]

Neben der vielseitigen Einsetzbarkeit von Perlin Noise gibt es außerdem den Vorteil, dass dieses Verfahren sehr günstig sowohl in Bezug auf die Berechnungszeit als auch in Bezug auf die Speicherverwendung ist. Einzelne Punkte im Verlauf lassen sich unabhängig voneinander berechnen, wodurch sich der Berechnungsprozess wunderbar parallelisieren lässt. Dies wird mit der immer weiter voranschreitenden Entwicklung von Grafikkarten und Prozessoren auch zu einem immer größeren Vorteil. [17]

Perlin Noise kann für eine beliebige Anzahl an Dimensionen berechnet werden. Dazu wird der n-dimensionale Raum in eine reguläre gitterartige Struktur aufgespalten. Die Punkte auf dem

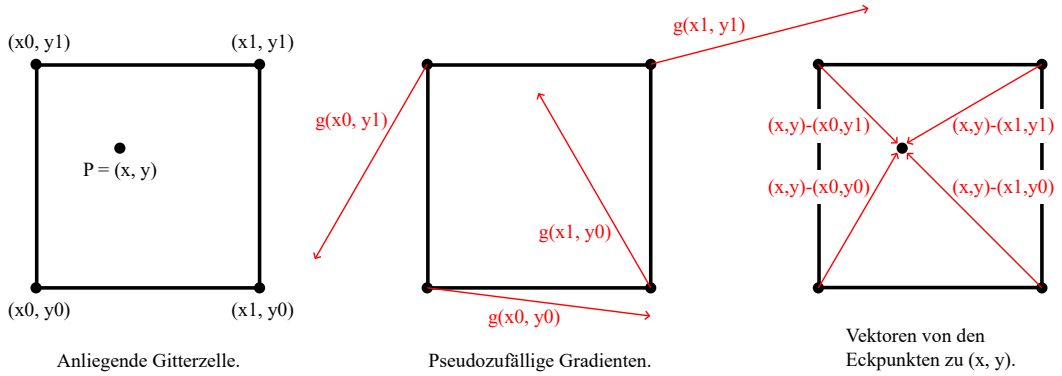

Abbildung 2.3: Perlin Noise: Gitterzelle und generierte Vektoren (in Anlehnung an [37]).

Gitter sind dabei all jene, die an ausschließlich ganzzahligen Koordinaten liegen. Im zweidimensionalen Raum wäre dies also die Menge an Punkten $\{(x, y) \parallel x, y \in \mathbb{N}\}$. Alle anderen Punkte im Raum befinden sich dann jeweils innerhalb einer der von den Gitterpunkten aufgespannten Zellen. Jeder der Gitterpunkte bekommt außerdem einen pseudozufälligen Gradienten (Richtungsvektor der Länge 1) zugeordnet. Soll jetzt der Noise-Wert für einen Punkt $P = (x, y)$ im Raum berechnet werden, werden zunächst die Eckpunkte der betroffenen Gitterzelle und deren zugeordnete Gradienten ermittelt. Außerdem werden die Vektoren berechnet, die von den Eckpunkten der Gitterzelle in Richtung des gewählten Punktes zeigen. Ein solche Zelle mitsamt der dazugehörigen Vektoren ist in Abbildung 2.3 dargestellt. Liegen all diese Werte vor, so kann der Noise-Wert berechnet werden, indem wir den Einfluss der pseudozufälligen Gradienten auf den ausgewählten Punkt berechnen und anschließend den gewichteten Mittelwert all dieser Einflüsse ermitteln. Der Einfluss eines einzelnen Gradienten g kann dabei berechnet werden, indem wir das Skalarprodukt zwischen g und dem Vektor, der vom entsprechenden Eckpunkt zu P zeigt, berechnen. Für den Gradienten $g(x0, y0)$ würde dies wie folgt aussehen: $g(x_0, y_0) \cdot ((x, y) - (x_0, y_0))$. Ist dies für jeden der Eckpunkte erfolgt, so wird nun der gewichtete Mittelwert daraus berechnet. Dabei erhalten die Eckpunkte, die dem Punkt P näher sind, eine stärkere Gewichtung. Dieser Mittelwert ist der gesuchte Noise-Wert für den Punkt P. [37]

2.3 Inverse Verfahren

Die bisher vorgestellten Verfahren haben alle eine Gemeinsamkeit: sie erzeugen neue Strukturen aus bereits vordefinierten Regeln. Im Anschluss wollen wir einige Verfahren vorstellen, die etwas anders vorgehen und solche Regeln selbst erzeugen. Dabei wird beispiel-basiert gearbeitet,

d.h. die entsprechenden Verfahren erhalten eine vorgefertigte Beispielstruktur als Input. Diese werden anschließend auf bestimmte Eigenschaften und Muster untersucht, um dann per Reverse Engineering einige Regeln ableiten zu können, mithilfe welcher neue Strukturen mit ähnlichen Eigenschaften automatisch erzeugt werden können. Aufgrund des umgekehrten Vorgehens hierbei bezeichnen wir solche Verfahren als *invers*.

2.3.1 Model Synthesis und Wave Function Collapse

Ein recht bekanntes und von vielen Spieleentwicklern genutztes Verfahren trägt den Namen *Wave Function Collapse (WFC)*. Dieses ist ein 2016 von Maxim Gumin entwickelter Algorithmus, welcher eine Bitmap als Input entgegen nehmen kann und aus dieser Variationen ableiten kann. Hier wird also mit gitterartigen Strukturen gearbeitet, weshalb sich perfekt zum Generieren von Pixel Art Texturen oder dem Erstellen von Leveln in Kachel-basierten Videospielen eignet. [13] Das Konzept hinter der Funktionsweise von WFC stammt allerdings nicht von Gumin selbst, sondern beruht auf einem von Paul Merrell in 2007 vorgestelltem Verfahren namens *Model Synthesis* [23]. Ein detaillierter Vergleich beider Verfahren wurde durch Paul Merrell selbst in 2021 vorgenommen [22] und soll hier nicht von weiterer Bedeutung sein. Was für uns jedoch wichtig ist, ist, dass Model Synthesis spezifisch auf das Erstellen von Modellen im dreidimensionalen Raum abzielt [23], während WFC in seiner ursprünglichen Form ausschließlich für zweidimensionale Strukturen genutzt werden konnte [13][22]. Da diese Arbeit ebenfalls nur auf das Generieren von Strukturen im 2D abzielt, betrachten wir im Folgenden also den WFC-Algorithmus etwas näher.

Grundlegend arbeitet WFC Gitter-basiert und mit einer Sammlung von Kacheln. Jede Kachel repräsentiert ein mögliches Muster oder eine mögliche Form, die in eine Zelle des Gitters platziert werden kann. Nutzt man den Algorithmus z.B. zum Erstellen von Texturen, so handelt es sich dabei um eine simple Farbe des entsprechenden Pixels. Nutzt man ihn zum Erstellen eines Levels in einem Kachel-basierten Videospiel, so kann es sich dabei auch um detaillierte Texturen für z.B. Wasserflächen oder Wälder handeln. Um zu bestimmen, welche Kacheln letztendlich in die Zellen platziert werden können, werden Constraints genutzt. Bevor eine neue Struktur generiert wird, werden für jede verschiedene Kachel Constraints abgeleitet, die beschreiben, neben welchen anderen Kacheln sich diese befinden darf. [13]

Nun zum Ablauf des eigentlichen Verfahrens: WFC beginnt mit einem leeren Gitter, welches ausgefüllt werden soll. Zu Beginn sind alle Positionen in diesem Gitter vollständig unbestimmt, d.h. in jede Zelle darf jede Kachel eingetragen werden. Anschließend werden die Zellen Schritt

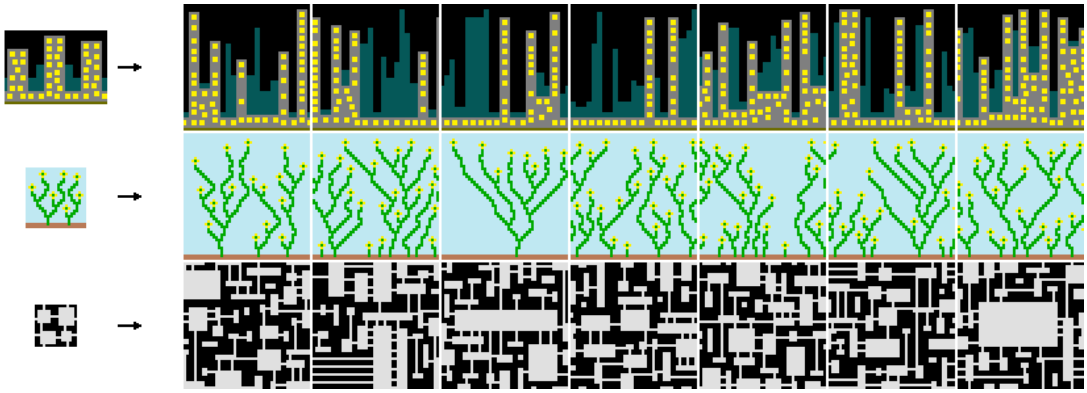

Abbildung 2.4: Wave Function Collapse: Beispielstrukturen und generierte Variationen [13].

für Schritt mit Kacheln gefüllt. Dazu wird in jedem Schritt ein Entropie-Wert für jede Zelle berechnet, welcher beschreibt, wie viele verschiedene Kacheln dort noch platziert werden dürfen. Es wird also geschaut, welche Kacheln sich bereits in der Nachbarschaft der ausgewählten Zelle befinden, woraufhin einige der verfügbaren Kacheln für diese Position ausgeschlossen werden können. Im Anschluss wird die Zelle mit der niedrigsten Entropie betrachtet und für diese zufällig eine der verfügbaren Kacheln ausgewählt und dort platziert. Dieser Vorgang wird als "Kollabieren" der Zelle bezeichnet (daher das "Collapse" im Namen des Algorithmus) und führt zu einer Anpassung der Entropie-Werte der umliegenden Zellen. Dieser Vorgang wird nun immer weiter wiederholt, bis schließlich alle Zellen nach diesem Prinzip kollabiert werden konnten oder es an irgendeiner Stelle zu einem Widerspruch kommt. Ein Widerspruch liegt vor, wenn für eine der noch offene Zellen ein Entropie-Wert von 0 berechnet wird und es somit keine Kacheln mehr gibt, die dort platziert werden können. Tritt dies auf, so muss der gesamte Vorgang wiederholt werden. Maxim Gumin zufolge tritt dieses Problem in der Praxis allerdings "überraschend selten" auf. [13] Einige Beispiele für Strukturen, die nach diesem Vorgehen generiert werden können, lassen sich in Abbildung 2.4 finden. Das dritte Beispiel hier könnte z.B. als Layout für einen Dungeon verwendet werden.

2.3.2 Nutzen von partieller Symmetrie

WFC kann zwar mit wenig Aufwand recht beeindruckende Ergebnisse erzeugen, ist jedoch stark limitiert in seinen möglichen Anwendungsszenarien, da dieser nur Gitter-basiert arbeiten kann. Ein etwas allgemeiner anwendbarer Ansatz wurde 2010 von Bokeloh et al. [5] vorgestellt. Auch hier werden wieder automatisch Regeln abgeleitet, die wir anschließend zum Erzeugen von Variationen des Inputs nutzen können. Die Voraussetzung für das Finden solcher Regeln besteht

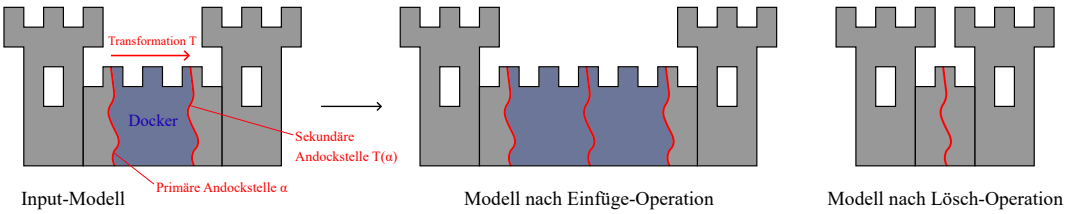

Abbildung 2.5: Nutzen von partieller Symmetrie zum Ableiten von Ersetzungsoperationen (in Anlehnung an [5]).

hierbei allerdings darin, dass es in dem gegebenen Input symmetrische Regionen gibt, d.h. einige Teilbereiche des Inputs müssen sich an anderen Stellen wiederfinden lassen. Dies klingt zunächst nicht unbedingt nach einem wirklichen Vorteil gegenüber den Limitation von WFC und Model Synthesis, jedoch besitzen etliche menschengemachte als auch natürlich vorkommende Strukturen einen gewissen Grad an Selbstähnlichkeit, wie bereits in Unterabschnitt 2.2.2 diskutiert wurde. In solchen selbstähnlichen Strukturen lassen sich stets symmetrische Regionen finden, was dieses Verfahren für eine Vielzahl an verschiedenen Strukturen einsetzbar macht. [5] In Abbildung 2.5 befindet sich ein entsprechendes Beispiel für ein Modell mit solchen partiellen Symmetrien, inklusive möglicher zu findender Ersetzungsoperationen.

Zum Finden möglicher Ersetzungsoperationen wird der Input auf sogenannte *Andockstellen* untersucht. Eine Andockstelle ist eine Kurve bzw. Sammlung von Punkten α entlang der Oberfläche S des gegebenen Modells ($\alpha \subseteq S$), welche das Modell in zwei vollständig abgetrennte Teile unterteilt. Eine Kurve die diese Bedingung nicht erfüllt, lässt sich in Abbildung 2.6b) finden. Eine solche Andockstelle ist nur von Nutzen, wenn sich dazu eine sekundäre Andockstelle finden lässt, die entsteht, wenn wir auf jeden Punkt der primären Andockstelle eine bestimmte Transformation T anwenden. Die transformierten Punkte müssen sich dabei alle ebenfalls auf der Oberfläche finden lassen ($T(\alpha) \subseteq S$). Außerdem muss die anliegende Topologie aller Punkte $x \in \alpha$ mit der Topologie der transformierten Punkte $T(x) \in T(\alpha)$ übereinstimmen, d.h. ist x ein Randpunkt, so muss $T(x)$ ebenfalls ein Randpunkt sein, oder ist x ein Punkt inmitten von S, so muss $T(x)$ ebenfalls ein Punkt inmitten von S sein. An dieser Stelle wird die Bedeutung der partiellen Symmetrie klar: ein solches Paar an primären und sekundären Andockstellen kann nur gefunden werden, wenn sich ein Teil des Modells (in diesem Fall alle Punkte in α) an einer anderen Stelle im Modell wiederfinden lässt. Die durch die Andockstellen vom Rest des Modells abgetrennten Bereiche nennen wir *Docker*. Diese können an andere Andockstellen mit dem gleichen Kurvenverlauf angedockt, aus dem Modell entfernt, oder durch andere Docker

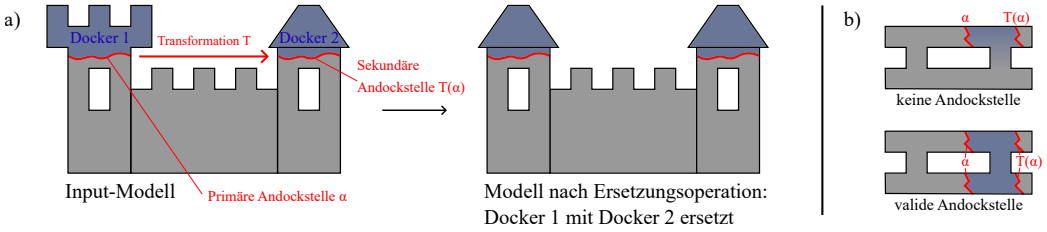

Abbildung 2.6: *a)* Ersetzungsoperation mit zwei verschiedenen Dockern. *b)* Valide und invalide Andockstellen. (in Anlehnung an [5]).

ersetzt werden, um Variationen des Modells zu erzeugen [5]. Beispiele hierfür lassen sich in den Abbildungen 2.5 und 2.6a) finden.

3 Konzept

Das Verfahren von Bokeloh et al. [5] ist zwar vielseitig einsetzbar und kann auf eine Vielzahl von verschiedenartigen Modellen angewandt werden, jedoch unterscheiden sich die erzeugten Strukturen in vielen Fällen kaum vom Input. Meist werden lediglich vorhandene Teilbereiche des Modells neu angeordnet. Die zugrundeliegenden Muster, die ein bestimmtes Modell ausmachen, werden jedoch nicht erkannt [24]. Genau hier setzt das nachfolgende Kapitel an und stellt ein Verfahren vor, welches versucht solche grundlegenden Muster zu erkennen. Zunächst wird ein grober Überblick zum gesamten Ablauf gegeben. Anschließend werden einige grundlegende Konzepte, die für das weitere Verständnis wichtig sind, vorgestellt, bevor dann letztendlich alle einzelnen Teilschritte im Detail erläutert werden. Die vorgestellten Konzepte beruhen dabei auf den Erkenntnissen von Paul Merrell in seiner Arbeit aus dem Jahr 2023 [24]. Alle Paragraphen in diesem Kapitel, die nicht explizit mit einer Referenz versehen sind, beziehen sich auf genau diese Quelle.

3.1 Überblick

Bevor es um die Einzelheiten und spezifischen Konzepte geht, wird zunächst ein grober Überblick zum Ablauf des umgesetzten Verfahrens geliefert. Das Ganze beginnt mit einer polygonalen Inputstruktur, d.h. einem Gebilde bestehend aus einem oder mehreren Polygonen (siehe Abbildung 3.1-I). Diese Inputstruktur wird anschließend umgewandelt in einen Graphen, in welchem die konkrete Geometrie des Inputs keine Rolle mehr spielt und sich auf die für das Verfahren wichtigen Eigenschaften des Inputs konzentriert werden kann.

Im nächsten Schritt wird der erstellte Graph nun in seine kleinstmöglichen Einzelteile zerlegt. Dazu werden alle Kanten in zwei Halbkanten aufgeteilt. Das Ergebnis sind viele Teilgraphen, welche jeweils nur noch aus einem Knoten und einigen Halbkanten bestehen. Einen solchen Teilgraphen nennen wir *Primitiv*. Diese Primitive werden dann Schritt für Schritt in allen möglichen

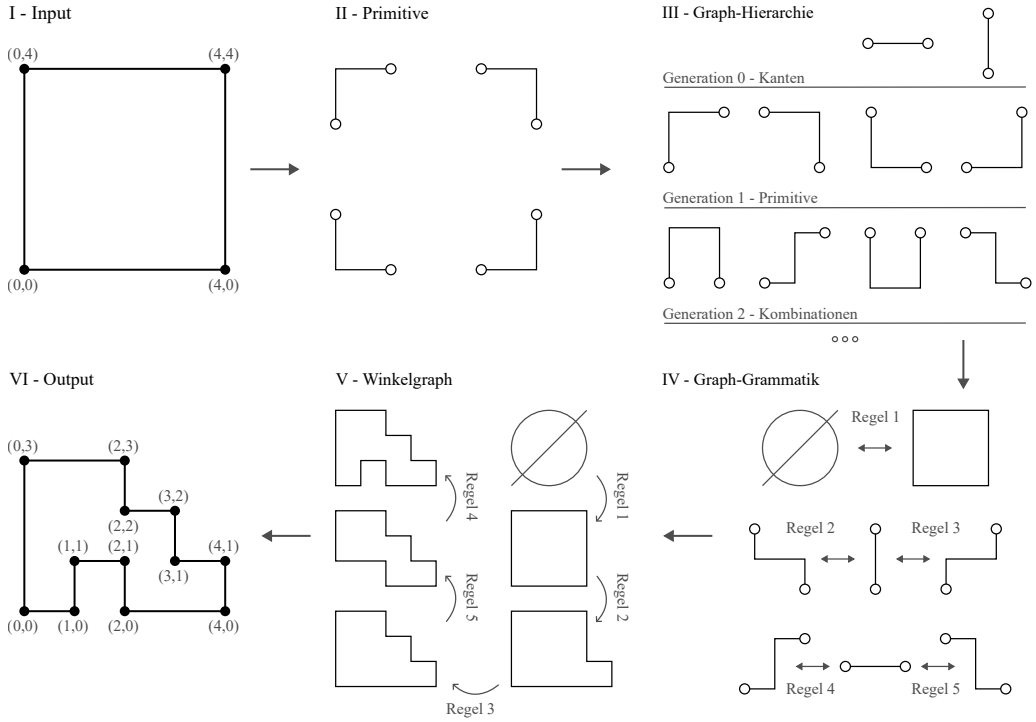

Abbildung 3.1: Überblick zum Ablauf des Verfahrens.

Kombinationen zusammengeklebt, was zum Entstehen einer Hierarchie an immer komplexer werdenden Graphen führt (siehe Abbildungen 3.1-II und 3.1-III). [24]

Beim Aufbau der Hierarchie werden die neu entstehenden Graphen auf bestimmte Eigenschaften überprüft, die es uns erlauben, daraus Regeln für eine Graphgrammatik abzuleiten (siehe Abbildung 3.1-IV). Das einfachste Beispiel hierfür sind vollständige Graphen, also Graphen, die nur noch aus in sich geschlossenen Kreisen bestehen und keine Halbkanten mehr besitzen. Aus diesen lässt sich eine sogenannte Startregel ableiten, welche den leeren Graphen mit dem gefundenen vollständigen Graphen ersetzt. Das Finden von weiteren Regeln ist deutlich komplizierter und wird später im Detail erläutert. [24]

Sobald nun eine Menge von Regeln für die Graphgrammatik gefunden wurde, kann man diese verwenden, um verschiedenste zum Inputgraphen ähnliche Graphen abzuleiten. Dazu werden die gefundenen nach und nach zufällig angewendet, was man in Abbildung 3.1-V sehen kann. Für einen solchen Graphen müssen dann nur noch konkrete Knotenpositionen und Kantenlängen bestimmt werden (siehe Abbildung 3.1-VI). [24]

3.2 Grundlagen

3.2.1 Input

Der Algorithmus kann mit beliebigen polygonalen Strukturen als Input arbeiten. Dies können einfache Rechtecke oder aber auch komplizierte Gebilde aus verschiedenen Häusern oder ähnlichem sein. Wichtig ist lediglich, dass der Input als Sammlung von Polygonen beschrieben werden kann.

Ein Polygon ist eine geometrische Figur, die vollständig durch ein Tupel P von n verschiedenen Punkten beschrieben werden kann:

$$P = (P_1, P_2, \ldots, P_n), P_i \in \mathbb{R}^2, 3 \leq i \leq n$$

Diese Punkte bezeichnen wir als *Eckpunkte* des Polygons. Verbindet man zwei aufeinanderfolgende Eckpunkte in Form einer Strecke $\overline{P_i P_{i+1}}$ (für $i = 1, \ldots, n-1$) bzw. $\overline{P_n P_1}$ miteinander, so erhält man eine *Seite* des Polygons. All diese Seiten zusammen spannen das Polygon auf. Eine Beschränkung der Anzahl an Eckpunkten nach oben gibt es dabei nicht, jedoch werden mindestens drei verschiedene Punkte für unsere Definition des Polygons vorausgesetzt. Mit weniger als drei Punkten können lediglich Figuren ohne Fläche (Punkte, Linien) erzeugt werden, welche für uns nicht von Nutzen sind. Ebenso sind Kreise oder andere Strukturen mit Rundungen nicht als Polygon darstellbar und können lediglich durch komplexe Polygone angenähert werden. Der Input kann somit keine Rundungen enthalten.

Die einzelnen Polygone können außerdem mit Farben versehen werden, um verschiedene Arten von abgegrenzte Bereichen im Input zu markieren.

3.2.2 Lokale Ähnlichkeit

Das Ziel, das durch dieses Verfahren erreicht werden soll, ist es, Variationen des Inputs zu erzeugen. Die erzeugten Outputstrukturen sollen dabei *lokal ähnlich* zum gegebenen Input sein. [24] Das Konzept der *lokalen Ähnlichkeit* wird im Folgenden vorgestellt. Das Verfahren gilt nur als erfolgreich, wenn diese zwischen Input und Output nachgewiesen werden kann.

Zwei Polygonstrukturen sind sich lokal ähnlich, wenn sich jeder Teil der einen Struktur zu einem Teil der anderen Struktur zuordnen lässt. Es müssen sich also alle verschiedenen Arten von Kanten und alle Polygonfarben irgendwo in beiden Strukturen finden lassen. Ein verwandtes

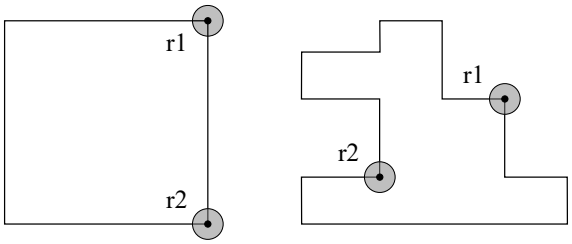

Abbildung 3.2: r-Ähnlichkeit.

Konzept, das zum Verständnis beitragen kann, ist das der *r-Ähnlichkeit*, welche im Paper von Bokeloh et al. [5] vorgestellt wurde. Zwei Strukturen sind hier *r*-ähnlich, wenn wir für jeden Punkt innerhalb der einen Struktur einen Kreis mit Radius *r* aufspannen können und sich der Inhalt dieses Kreises (r-Nachbarschaft des Punktes) genauso in der anderen Struktur wiederfinden lässt. Ein Beispiel hierfür befindet sich in Abbildung 3.2.

Die von uns verwendete lokale Ähnlichkeit funktioniert nach dem gleichen Konzept, mit der Ausnahme, dass der Radius so klein wie möglich gehalten wird. Wir schauen uns also lediglich an, welche Kanten und Polygone direkt an einem Punkt anliegen, während uns die restliche Nachbarschaft egal ist. So können die betrachteten Strukturen beliebig skaliert werden und trotzdem ihre lokale Ähnlichkeit zueinander bewahren, solang alle Kantenwinkel dabei beibehalten werden. [24]

3.2.3 Der Winkelgraph

Zur Verarbeitung des Inputs wird dieser in einen sogenannten Winkelgraphen umgewandelt, in welchem die spezifischen Positionen der Knoten keine Rolle spielen. Stattdessen wird nur abgebildet, welche Knoten es überhaupt gibt, welche der Knoten durch Kanten miteinander verbunden sind, und in welchem Winkel diese Kanten verlaufen. Die Kanten im Graphen werden mit einem *Kantenlabel* versehen, welches neben den Start- und Endknoten ebenfalls Informationen zum daraus ableitbaren Tangentenwinkel, sowie zu den Farben der links und rechts anliegenden Polygone enthält. Ein Kantenlabel besitzt die Form $\tilde{k} = (l, r, \theta)$, wobei $\tilde{k}$ die Bezeichnung der Kante, l und r die Farben der anliegenden Polygone, und θ der Tangentenwinkel der Kante sind. Nach der Umwandlung des Inputs in einen Winkelgraphen ist dieser zunächst *vollständig*, d.h. er besteht ausschließlich aus geschlossenen Kreisen. In späteren Verarbeitungsschritten wird dieser allerdings in unvollständige Teilgraphen zerlegt, welche dann außerdem *Halbkanten* enthalten können. Im Gegensatz zu den vorher erwähnten Kanten sind diese gerichtet, können

aber trotzdem durch ein gleichartiges Kantenlabel beschrieben werden. In späteren Abschnitten wird noch etwas genauer auf die Relevanz von Halbkanten und deren spezifische Notation eingegangen. [24]

Komplexität von Winkelgraphen

Später müssen einige der erstellten Winkelgraphen miteinander verglichen werden. Dabei muss bestimmt werden können, welcher von zwei Graphen komplexer bzw. simpler ist. Dies ist nicht schwierig, sollte jedoch einmal eindeutig definiert werden. Das ausschlaggebendste Kriterium hierbei ist die Anzahl der Halbkanten der verglichenen Graphen. Ein Graph mit weniger Halbkanten gilt direkt als simpler als ein Graph mit einer größeren Anzahl an Halbkanten. In vielen Fällen werden die verglichenen Graphen jedoch die gleiche Anzahl an Halbkanten vorzuweisen haben. Hier wird die Graph-Hierarchie wichtig. Wurde ein Graph früher in die Hierarchie eingefügt, so gilt dieser als simpler. Dies kann immer eindeutig bestimmt werden und es kommt zu keinen weiteren Konflikten. [24]

3.2.4 Planarität und der Graph Boundary String

Damit die erzeugten Winkelgraphen später auch ohne Überschneidungen der Kanten dargestellt werden können, muss deren Planarität sichergestellt werden. Die endgültig erzeugten vollständigen Winkelgraphen bestehen nur noch aus geschlossenen Kreisen. Diese können dann als Polygone dargestellt werden, vorausgesetzt der jeweilige Kreis war planar. [24]

Ein geschlossener Kreis ist planar, wenn wir uns beim Traversieren seiner Kanten exakt einmal um 360° gedreht haben. Wenn wir also iterativ alle Kanten eines solchen Kreises gegen den Uhrzeigersinn betrachten, jeweils die Differenz der Winkel berechnen und diese Differenzen aufsummieren, so erhalten wir einen Gesamtwinkel von 360°. Es kann allerdings auch vorkommen, dass wir beim Entlanglaufen eines Pfades um einen geschlossenen Kreis herum einen Gesamtwinkel von mehr als 360° erhalten, so z.B. 720°. Ist dies der Fall, so muss sich der Pfad zwingend selbst gekreuzt haben. Analog würde es bei der Darstellung der Kanten des entsprechenden Kreises mindestens eine unvermeidliche Überschneidung geben. Somit wäre der Winkelgraph, der diesen Kreis enthalten hat nicht mehr planar, was in Abbildung 3.3 visuell verdeutlicht wird. [24]

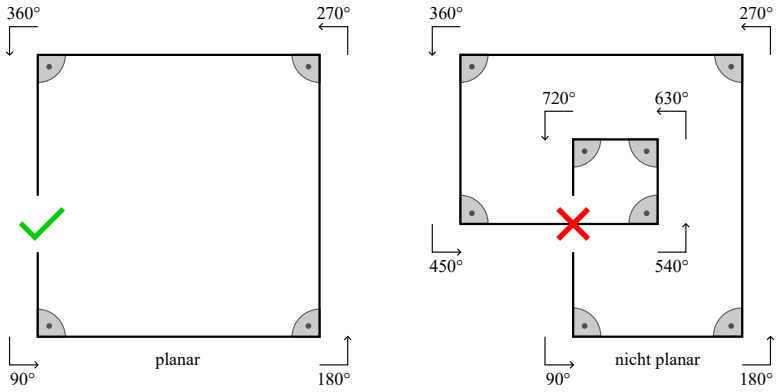

Abbildung 3.3: Der Zusammenhang zwischen Drehwinkel und Planarität.

Zum Vorbeugen dieses Problems definieren wir hier das Konzept der *Graph Boundary* und die dazugehörige Notation in Form vom *GBS*. Jeder Winkelgraph G besitzt eine solche Boundary ∂G. Diese beschreibt einen Pfad außen um den entsprechenden Winkelgraphen herum und enthält alle vorhandenen Halbkanten, sowie Informationen dazu, wie sich die Winkel entlang des Pfades ändern. Der Pfad verläuft gegen den Uhrzeigersinn und hat keinen festen Startpunkt. Wichtig ist lediglich die relative Anordnung der enthaltenen Elemente. Um dies abbilden zu können, muss sich der GBS nicht als Liste mit festem Start- und Endpunkt vorgestellt werden, sondern als Kreis mit einer zusätzlichen Verbindung zwischen Anfang und Ende. Angenommen $abc\wedge$ ist ein GBS, dann gilt $abc\wedge = bc\wedge a = c\wedge ab = \wedge abc$. [24]

Um den aktuellen Drehwinkel in Relation zum Startpunkt des Pfades zu ermitteln, können wir uns jeweils die Kantenlabel der traversierten Kanten anschauen und dort den Tangentenwinkel entnehmen. Dies wird allerdings problematisch, sobald sich der Pfad um mehr als 360° dreht, da die Tangentenwinkel bei 180° bzw. -180° umgebrochen werden. Berechnen wir den aktuellen Drehwinkel entlang der Graph Boundary mithilfe dieser Tangentenwinkel, so können wir nie eine Differenz von über 360° erhalten. Haben wir uns vom Startpunkt aus z.B. tatsächlich um 400° gedreht, so würde die hier berechnete Differenz lediglich $400° - 360° = 40°$ betragen. Wir wissen also nie, ob wir uns aktuell um den Winkel θ, $\theta + 360°$, $\theta + 720°$ oder noch mehr gedreht haben. Dieses Problem lässt sich durch Einführung des Konzepts der *positiven und negativen Drehungen* umgehen. [24]

Positive Drehung $\wedge$. Dreht sich der Pfad aktuell gegen den Uhrzeigersinn wird der Winkel mit jeder gefundenen Kante größer. Stoßen wir dabei allerdings auf den Schwellwert von 180°, so bricht der Winkel auf einmal in den negativen Bereich um. Diesen Umbruch bezeichnen wir als positive Drehung. Folgen wir beim Entlanglaufen des Pfades dem Verlauf einer positiven

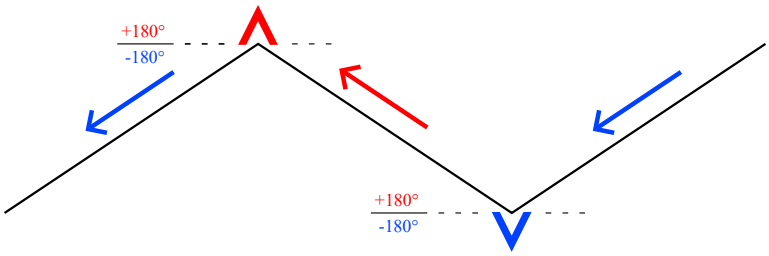

Abbildung 3.4: Positive und negative Drehungen.

Kanten und wechseln dann auf eine negativ verlaufende Kante, so werden wir dabei in einigen Fällen diesen Schwellwert überschreiten. Falls dies geschieht, so fügen wir eine positive Drehung in Form des Symbols $\wedge$ in den GBS ein. Die Boundary eines planaren Winkelgraphen muss zwingend eine solche positive Drehung enthalten. [24]

Negative Drehung $\vee$. Die negative Drehung stellt das Gegenteil zur positiven Drehung dar. Dreht sich der Pfad aktuell im Uhrzeigersinn, so nähern sich die gefunden Winkel nach und nach dem Schwellwert von -180°. Anschließend bricht der Winkel in den positiven Bereich um, was wir als negative Drehung bezeichnen und mit dem Symbol $\vee$ im GBS notieren. [24]

Die beiden in Abbildung 3.4 zu sehenden Drehungen heben sich gegenseitig auf. Befinden sich eine positive und eine negative Drehung nebeneinander im GBS, so können diese entfernt werden. Ebenfalls kann an jeder beliebigen Stelle ein "$\wedge\vee$" oder ein "$\vee\wedge$" eingefügt werden, ohne die Bedeutung des jeweiligen GBS zu verändern. Eine weitere Eigenschaft, die sich für den GBS für alle planaren Winkelgraphen ergibt ist, dass sich darin immer genau eine positive Drehung mehr befinden muss, als es negative Drehungen gibt. Dies liegt daran, dass sich der Pfad um einen planaren Graphen insgesamt exakt um 360° dreht und der Drehwinkel somit zumindest einmal irgendwo den Schwellwert überschreiten muss. Da wir die Graph Boundary entgegen des Uhrzeigersinns ablaufen, handelt es sich dabei um eine positive Drehung $\wedge$. [24]

3.2.5 Teil-Operation

Eine Kante $\tilde{k}$ kann in zwei Halbkanten k und $\overline{k}$ zerteilt werden. Im Gegensatz zu $\tilde{k}$ sind diese beiden Halbkanten gerichtet und zeigen in entgegengesetze Richtungen. Dabei zeigt k stets in positive Richtung und besitzt einen positiven Tangentenwinkel $\theta \in [0°, 180°)$, während $\overline{k}$ immer in negative Richtung zeigt und einen negativen Tangentenwinkel $\theta \in [-180°, 0°)$ besitzt. Ein Tangentenwinkel von 0° zählt hier als positiv. Der entgegengesetze Winkel von 180° gilt als

negativ, da dieser ebenfalls als -180° interpretiert werden kann. Die Teil-Operation ermöglicht das Zerlegen vom Input in seine Primitive. [24]

3.2.6 Klebe-Operation

Zwei entgegengesetze Halbkanten k und $\overline{k}$ können wieder zu einer vollständigen und ungerichteten Kante $\tilde{k}$ zusammengeklebt werden. Dies ermöglicht das Schließen von Kreisen innerhalb eines Graphen oder die Kombination von mehreren kleineren Graphen, vorausgesetzt diese besitzen passende Halbkanten. Hier ist erneut der GBS von Relevanz, da aus diesem alle möglichen Klebe-Operationen abgeleitet werden können, welche die Planarität der entstehenden Graphen bewahren. Grundlegend gibt es zwei verschiedene Arten von Klebe-Operationen: Loop Gluing und Branch Gluing. [24]

Loop Gluing beschreibt das Zusammenkleben zweier Kanten innerhalb eines einzigen Graphen. Das Anwenden einer solchen Operation führt zum Schließen eines Kreises innerhalb des Graphen. Ob ein Loop Gluing auf einen Graphen angewandt werden kann, lässt sich durch das Vorhandensein eines der zwei Teilstrings "$a\overline{a}$" oder "$\overline{a} \vee a\wedge$" innerhalb des GBS ermitteln. Werden die gefundenen Kanten dann zusammengeklebt, muss der GBS entsprechend angepasst werden. Für das Loop Gluing ist diese Anpassung besonders einfach und es muss lediglich der gefundene Teilstring entfernt werden. Die entsprechenden String-Ersetzungen besitzen die Form:

$$a\overline{a} \longrightarrow \epsilon \quad \text{bzw.} \quad \overline{a} \vee a\wedge \longrightarrow \epsilon$$

Branch Gluing beschreibt das Zusammenkleben zweier Kanten von verschiedenen Graphen. Dies führt zur Vereinigung der beiden betroffenen Graphen in einen neuen, größeren Graphen. Besitzt Graph G die Kante $\overline{a}$ und Graph H die Kante a, so kann ein Branch Gluing durchgeführt werden. Hierbei gibt es wieder zwei Optionen:

$$\overline{a}G \text{ an } a\text{: } a \longrightarrow G\vee \quad \text{bzw.} \quad aH \text{ an } \overline{a}\text{: } \overline{a} \longrightarrow \vee H$$

Die Großbuchstaben G und H stehen hier jeweils für den Rest des GBS der beiden Graphen. Der GBS des neu enstandenen Graphen stellt die Kombination der beiden kleineren GBS dar, allerdings ohne die zusammengeklebten Halbkanten und mit einer zusätzlichen negativen Drehung. [24]

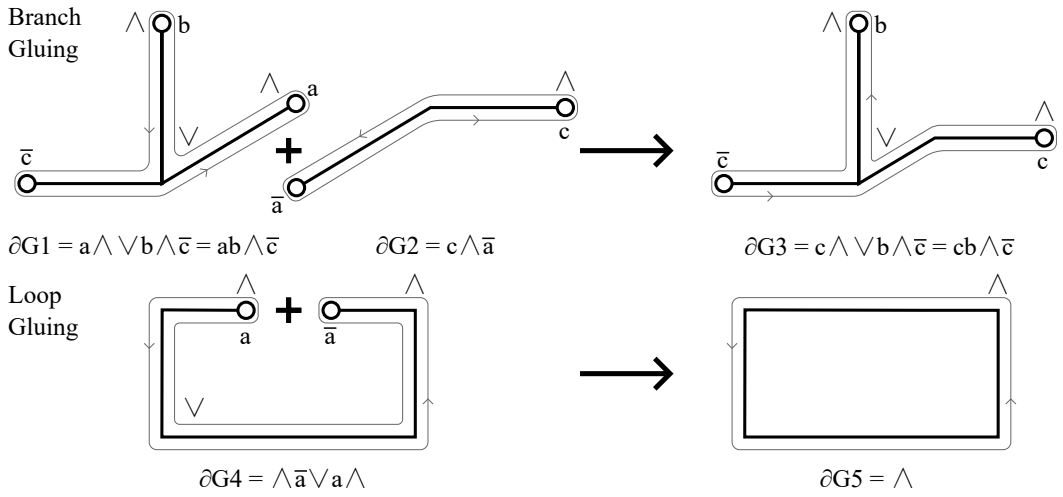

Abbildung 3.5: Branch und Loop Gluing.

3.3 Finden der Graphgrammatik

3.3.1 Anpassen des Inputs

Bevor wir mit dem Verfahren beginnen können, muss der Input an bestimmte Anforderungen angepasst werden. Die übergebene Polygonstruktur kann so nicht direkt verarbeitet werden und muss erst einmal in einen Winkelgraphen umgewandelt werden. Ohne diesen Schritt liegen uns keine Informationen zu den Kantenwinkeln vor, welche ausschlaggebend für die weiteren Schritte sind. Hierzu werden zunächst einfach alle Knoten und Kanten aus dem Input übernommen. Anschließend werden die Knotenpositionen genutzt, um den Verlauf der Kanten in Form eines Tangentenwinkels zu ermitteln. Sobald dies geschehen ist, können die Knotenpositionen dann ignoriert werden, da lediglich die Ausrichtung der Kanten eine Rolle für die weiteren Schritte spielt. Die restlichen Informationen zur Geometrie werden nicht benötigt und erst beim Erzeugen des finalen Outputs wieder festgelegt. [24]

3.3.2 Finden der Primitive

Ist nun der Winkelgraph ermittelt worden, können wir daraus die Primitive ableiten. Diese sind die fundamentalen Grundbausteine für das gesamte Verfahren. Aus ihnen werden alle weiteren Strukturen abgeleitet, weshalb es besonders wichtig ist, diese korrekt und vollständig zu ermitteln. Glücklicherweise wird dies durch die vorgestellte Teil-Operation recht trivial. Wenden wir

diese auf jede Kante des gegebenen Winkelgraphen an, so bleiben nur Teilgraphen übrig, welche nur aus einem einzelnen Knoten, sowie einigen Halbkanten bestehen. Diese Teilgraphen sind dann auch schon die gesuchten Primitive. Hier können allerdings einige identische Teilgraphen entstehen, falls Teile des Input eine ähnliche Struktur vorzuweisen hatten. Solche Duplikate sind nicht relevant für die weiteren Schritte und werden ignoriert. [24]

3.3.3 Aufbauen der Graph-Hierarchie

Die vorgestellten Klebe-Operationen ermöglichen es uns, die gefundenen Primitive nach und nach zu komplexeren Graphen zusammenzusetzen. Diese lassen sich in verschiedene Generationen einer Hierarchie einordnen. In Generation 0 befinden sich alle verschiedenartigen Kanten, also alle Kanten mit einem einzigartigen Kantenlabel. In Generation 1 befinden sich alle gefundenen Primitive. Anschließend können daraus die nachfolgenden Generationen automatisch generiert werden. Dazu wird durch alle Winkelgraphen der zuletzt generierten Generation iteriert und alle durchführbaren Klebe-Operationen auf diese angewandt. Gibt es in einem der iterierten Graphen einen noch offenen Kreis, der aber durch eine einfach Loop Gluing Operation geschlossen werden kann, so wird diese angwandt. Außerdem werden jeweils alle der gefundenen Primitive betrachtet und ein Branch Gluing mit diesen ausgeführt, vorausgesetzt deren GBS lässt dies zu. Wird durch eine dieser Operationen ein neuer Graph erzeugt, so gilt dieser als Kind des anderen Graphen. Jeder Graph wird also neben der Einordnung in eine Generation außerdem in eine Eltern-Kind-Beziehung gebracht. Die Struktur der Hierarchie selbst ähnelt somit fast der eines Baumes, allerdings kann ein und derselbe Kindsgraph durch verschiedene Elterngraphen erzeugt werden, wodurch wiederum Kreise innerhalb der Hierarchie entstehen. [24]

In der Theorie können alle Kombinationen an Primitiven erzeugt werden, wenn wir dieses Vorgehen bis in die Unendlichkeit weiterführen. Somit würden garantiert alle zum Input lokal ähnlichen Winkelgraphen erzeugt werden und man könnte einfach jeden beliebigen vollständigen Graphen aus der Hierarchie entnehmen, um jeden validen Output des Verfahrens erzeugen zu können. Praktisch gesehen ist dies natürlich leider nicht umsetzbar, da uns weder unendlich viele Ressourcen noch Zeit zur Verfügung stehen. Um diesen Ansatz also praktisch zu machen, müssen einige Anpassungen gemacht werden. [24]

3.3.4 Ableiten der Graphgrammatik

Statt zuerst eine "vollständige" Hierarchie zu erzeugen und aus dieser dann weitere Schritte abzuleiten, wird die Hierarchie inkrementell erzeugt. Jedes Mal, wenn ein neuer Graph erstellt wird, überprüfen wir diesen auf bestimmte Eigenschaften, die es uns erlauben daraus Regeln für eine Graphgrammatik abzuleiten. Ein solche Regel ermöglicht es uns bereits erzeugte Winkelgraphen zu reduzieren, was wiederum bedeutet, dass wir diese nicht mehr benötigen und aus der Hierarchie entfernen können. Ein detaillierter Einblick zu der Theorie dahinter wird erst in den folgenden Unterkapiteln gegeben, jedoch ist es genau diese Eigenschaft, die es uns ermöglicht, das Wachstum der Hierarchie einzugrenzen. Optimalerweise erreichen wir irgendwann einen Punkt, an dem alle bereits erzeugten Graphen durch eine der Regeln reduziert werden konnten. In diesem Fall können wir garantieren, dass sich aus den Regeln alle vollständigen und zum Inputgraphen lokal ähnlichen Winkelgraphen aus der erzeugten Grammatik ableiten lassen. Meistens werden wir allerdings auf Szenarien stoßen, in denen die Anzahl der neuen Graphen schneller wächst, als wir andere Graphen entfernen können. Kommt dies vor, so muss das Erstellen der Hierarchie irgendwann frühzeitig abgebrochen werden und die Graphgrammatik ist eventuell nicht in der Lage, alle lokal ähnlichen Graphen zu erzeugen. Trotzdem kann die Grammatik dann zum Ableiten einer Vielzahl von lokal ähnlichen Winkelgraphen genutzt werden. [24]

Graphgrammatiken

Bevor wir die Erzeugung dieser Datenstruktur genauer betrachten, soll erst einmal der Begriff der Graphgrammatik klar definiert werden. Eine Graphgrammatik ist ein formales System, welches spezifisch auf die Erstellung und Manipulation von Graphen in einem mathematisch präzisen Weg abzielt. Dazu wird eine Menge an Produktionsregeln definiert, welche verschiedene Operationen zum Ersetzen von Teilen eines Graphen beschreiben. Eine solche Produktion besteht üblicherweise aus drei Bestandteilen: zwei Graphen M und T ("Mutter" und "Tochter"), sowie einem Einbettungsmechanismus E. Diese Produktion kann nun auf jeden Graphen G angewandt werden, welcher M als Teilgraphen enthält. Um die Produktion anzuwenden, wird M aus G entfernt und mit T ersetzt. Dabei wird E genutzt, um zu definieren, wie genau T in G eingebettet werden kann. [10]

Das Konzept der Graphgrammatiken existiert bereits seit den frühen 70er Jahren und wurde mit dem Paper von Ehrig et al. [9] zum ersten Mal formal definiert. Seitdem haben sich sehr

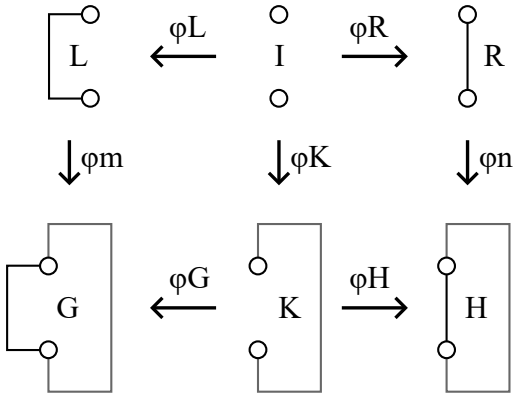

Abbildung 3.6: Double Pushout Produktionsregel.

viele verschiedene Herangehensweisen in diesem Kontext etabliert, was zu viel Verwirrung führen kann. [16] Wir beschränken uns hier auf den ursprünglich präsentierten algebraischen bzw. Gluing-Ansatz von Ehrig et al. [9]. Innerhalb dieses Ansatzes haben sich zwei verschiedene Vorgehensweisen etabliert: der Single Pushout Ansatz und der Double Pushout (DPO) Ansatz, wovon wir den letzteren verwenden. Der Begriff "Pushout" stammt aus der Kategorientheorie, welche ebenfalls zum Beschreiben von Graphgrammatiken genutzt werden kann. [24]

Die Produktionen im verwendeten DPO Ansatz bestehen aus drei Teilen. Einem linken und rechten Graphen (L und R), welche jeweils den Mutter- und Tochtergraphen darstellen, sowie einem Interface-Graphen I, welcher den Einbettungsmechanismus darstellt. Wie in Abbildung 3.6 zu sehen ist, können diese Graphen mithilfe der Homomorphismen φL und φR untereinander abbilden. [24]

Ein Homomorphismus von Graph A (mit der Knotenmenge $V(A)$ und der Kantenmenge $E(A)$) zu Graph B (mit $V(B)$ und $E(B)$) ist eine Abbildung $h : A \to B$, welche alle Knoten in $V(A)$ auf eine (meist echte) Teilmenge von $V(B)$ abbildet: $x, y \in V(A) \to h(x), h(y) \in V(B)$. Sind x und y in A adjazent, so müssen $h(x)$ und $h(y)$ in B ebenfalls adjazent sein: $(x, y) \in E(A) \to (h(x), h(y)) \in E(B)$. [32]

Ist L als Teilgraph in einem anderen Graphen G enthalten, d.h. gibt es einen Homomorphismus $\varphi m : L \to G$, so kann die entsprechende Produktion zum Transformieren von G verwendet werden. Dazu muss L zunächst aus G herausgeschnitten werden. Hierfür wird das Interface benötigt. Dieses beschreibt die Gemeinsamkeiten zwischen der linken und der rechten Seite der Produktion und ermöglicht das problemlose Austauschen der beiden Seiten miteinander. Wird L aus G entfernt, so erhalten wir den sogenannten Klebegraphen K. In diesen können wir nun

R hineinkleben, um den transformierten Graphen H zu erhalten. Das Anwenden einer solchen Produktionsregel ist aber auch in die andere Richtung möglich. Alternativ kann auch zuerst R aus H ausgeschnitten und dann dort L eingeklebt werden, um G zu produzieren, vorausgesetzt es gibt einen Homomorphismus $\varphi n : R \to H$. [9]

Theorie hinter der Funktionsweise

Die später aus der Hierarchie abzuleitenden Produktionsregeln werden so angeordnet, dass sich auf der linken Seite ein komplexerer Graph befindet, als auf der rechten Seiten. Wird die Regel von links nach rechts angwendet, so vereinfacht sie G. Dies bezeichnen wir als *destruktiv*. Wird sie von rechts nach links angewendet, so macht sie H komplexer, was wir wiederum als *konstruktiv* berzeichnen. [24]

Wie bereits erwähnt, versuchen wir nach dem Erstellen der einzelnen Graphen Regeln zu finden, die bereits erzeugte Graphen in der Hierarchie vereinfachen können und entfernen diese dann ggf. aus der Hierarchie. Hier werden die Regeln stets nur destruktiv genutzt, was zunächst unlogisch erscheint. An diesem Punkt können wir uns die Invertierbarkeit der Produktionen zunutze machen. Wenn wir genug Regeln finden können, um alle Graphen in der Hierarchie zum leeren Graphen reduzieren zu können, indem wir diese destruktiv verwenden, so können wir im Umkehrschluss genau die gleichen Regeln konstruktiv benutzen, um aus dem leeren Graphen alle anderen Graphen abzuleiten. [24]

Ableiten einer Produktionsregel

Kommen wir nun dazu, wie es möglich ist, solche Regeln automatisch ableiten zu können. Dazu schauen wir uns erneut kurz an, wie eine Produktionsregel angewandt wird. Wie oben beschrieben, involviert dies zwei Teilschritte. Zunächst wird die eine Seite der Regel aus einem größeren Graphen G ausgeschnitten, wodurch dieser zum Klebegraphen K wird, in welchem nun einige offene Halbkanten enthalten sind. Um K wieder in einen vollständigen Graphen H umwandeln zu können, müssen all diese Halbkanten anschließend durch Anwenden von entsprechenden Klebeoperationen vervollständig werden. Angenommen, wir haben für den vorherigen Schritt den Graphen auf der linken Seite der Regel, also L, aus G ausgeschnitten. Damit der rechte Graph R die entstandene Lücke in K schließen kann, muss dieser exakt die gleichen Halbkanten enthalten, wie L. Besitzt R weniger Halbkanten, so können einige der enstandenen Halbkanten in K nicht vervollständigt werden. Besitzt R mehr Halbkanten, so würde das Durchführen des

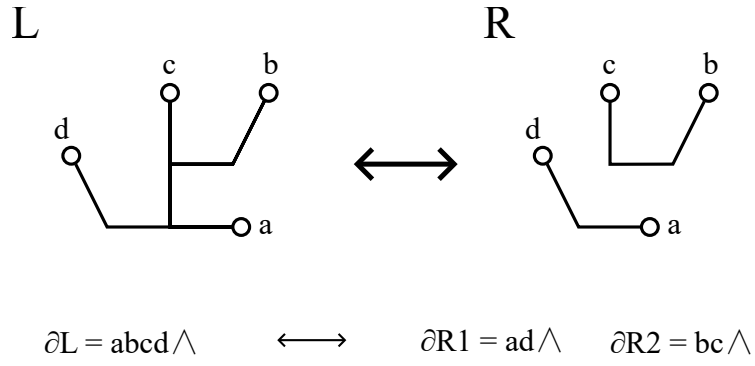

Abbildung 3.7: Ersetzen eines Graphen L mit mehreren Graphen $\{R_1, R_2\}$.

Branch Gluings zwischen K und R weitere offene Halbkanten in K einfügen. Stimmt die Anzahl der Halbkanten in L und R überein, aber die Kantenlabel unterscheiden sich, so kann K ebenfalls nicht vervollständigt werden. Nur bei hundertprozentiger Übereinstimmung der Halbkanten auf beiden Seiten der Regel kann diese also problemlos angewandt werden. [24]

Um aus der Hierarchie eine Regel ableiten zu können, müssen wir also jeweils zwei Graphen mit übereinstimmenden Halbkanten finden. Hier können wir uns den GBS zunutze machen, da dieser alle Halbkantenlabel eines Graphen enthält. Besitzen zwei Graphen einen identischen GBS, so besitzen sie zwingend auch die gleichen Halbkanten. Zusätzlich wird durch Abgleichen der Boundary sichergestellt, dass sich an der Planarität des enstehenden Graphen nach Anwendung einer Produktionsregel nichts ändert. Zwischen den Kanten befinden sich dann nach wie vor die gleichen Drehungen. [24]

Statt einen Graphen L immer nur mit genau einem anderen Graphen R zu ersetzen, können wir alternativ auch auf mehrere Graphen $\{R_1, R_2, \dots\}$ auf der rechten Seite abbilden, vorausgesetzt diese befinden sich in der Hierarchie. Lassen sich die GBS $\{\partial R_1, \partial R_2, \dots\}$ zu ∂L zusammensetzen, so sind ebenfalls alle Bedingungen erfüllt, um die Regel problemlos anwenden zu können. Durch dieses Vorgehen lassen sich sehr viele weitere Regeln ableiten und das Wachstum der Graph-Hierarchie wird zusätzlich eingeschränkt. [24]

Ein Beispiel für eine solche Produktionsregel ist in Abbildung 3.7 vereinfacht dargestellt. $\partial L = abcd\wedge$ lässt sich aus einer Kombination von $\partial R_1 = ad\wedge$ und $\partial R_2 = bc\wedge$ bilden. Das Kombinieren von zwei Boundary Strings ist jedoch etwas komplizierter als das Durchführen einer simplen Konkatenation. Würden wir ∂R_1 und ∂R_2 ohne weitere Anpassungen konkatenieren, so würden wir einen Boundary String mit zwei positiven Drehungen $\wedge$ und nicht einer einzigen

negativen Drehung $\vee$ erhalten (z.B. $ad \wedge bc\vee$). Dies würde den Bedingungen für die Planarität widersprechen, da die Differenz zwischen den positiven und negativen Drehungen nun 2 und nicht 1 betragen würde. Um dieses Problem zu umgehen, wird bei der Kombination zweier GBS eine weitere negative Drehung zwischen diesen eingefügt. Nehmen wir an A und B sind Boundary Strings. Die daraus resultierende Kombination hätte dann die Form $A \vee B$. Dies ist keine willkürliche Anpassung, die unsere vorher aufgestellten Regeln für einen GBS umgeht, sondern eine logische Konsequenz aus diesen. In Abbildung 3.8 ist dies noch einmal zusätzlich verdeutlicht. [24]

Widmen wir uns nun wieder dem Beispiel in Abbildung 3.7. Aufgrund der kreisförmigen Natur eines GBS können wir $\partial R_1 = ad\wedge$ auch als $d \wedge a$ und $\partial R_2 = bc\wedge$ als $\wedge bc$ darstellen. Werden diese mit einer zusätzlichen negativen Drehung in der Mitte kombiniert, so erhalten wir $\partial R_1 \vee \partial R_2 = d \wedge a \vee \wedge bc = d \wedge abc = abcd\wedge = \partial L$.

Mithilfe dieses Wissens können wir nun systematisch eine Vielzahl von Regeln ableiten: Nach der Erzeugung eines jeden neuen Graphen L in der Hierarchie betrachten wir stets alle vorher erzeugten Graphen R und vergleichen die Boundaries miteinander. Wird dabei eine hunderprozentige Übereinstimmung $\partial L = \partial R$ gefunden, so kann direkt eine neue Regel abgeleitet werden. Alternativ könnte ∂R auch nur ein Teil von ∂L sein, was einige Teil-Strings hinterlassen würde. Für diese Teil-Strings wird dann rekursiv nach weiteren Übereinstimmungen gesucht, bis sich ∂L entweder komplett aus den gefunden Teil-Strings zusammensetzen lässt oder es keine weiteren Graphen mehr gibt, die noch überprüft werden können. In letzterem Fall kann dann keine Produktionsregel abgeleitet werden. [24]

Betrachten wir erneut das Beispiel in Abbildung 3.7. Wir haben einen Graphen L mit der Boundary $\partial L = abcd\wedge$ erzeugt. Beim Iterieren durch die Hierarchie finden wir nun den Graphen R_1 mit $\partial R_1 = ad\wedge$. ∂R_1 ist als Teil in ∂L enthalten, wodurch R_1 also einen Teil von L ersetzen kann. Übrig bleibt der Teil-String bc. Um diesem einen Graphen zuordnen zu können, muss zunächst eine positive Drehung $\wedge$ an diesen angefügt werden. Die zusätzliche Drehung wird dann beim Kombinieren automatisch wieder negiert. Ein Graph R_2 mit $\partial R_2 = bc\wedge$ kann anschließend ebenfalls gefunden werden, wodurch alle Teile von ∂L nun komplett sind. Somit kann aus den gefundenen eine neue Produktionsregel erstellt und L aus der Hierarchie entfernt werden, da sich dieser nun nachweislich vereinfachen lässt. [24]

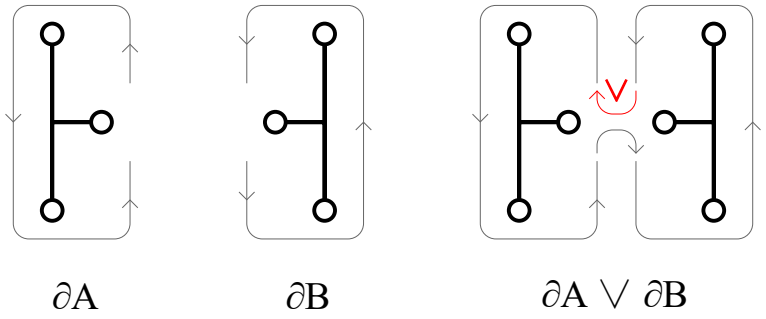

Abbildung 3.8: Kombinieren des GBS zweier Graphen mit zusätzlicher negativer Drehung.

Starter-Regeln

Eine Starter-Regel ist eine besondere Produktionsregel, welche den leeren Graphen $\varnothing$ mit einem vollständigen Graphen ersetzen kann. Davon kann es theoretisch beliebig viele Regeln geben, jedoch zumindest immer eine. Ohne eine Starter-Regel können keine Graphen aus der Grammatik abgeleitet werden, da der leere Graph keine Kanten besitzt, auf welche man die Graphen in den anderen Produktionsregeln abbilden könnte. Eine solche Starter-Regel kann durch das oben beschriebene Vorgehen niemals abgeleitet werden und stellt einen Sonderfall dar. Dieser tritt ein, sobald wir beim Erstellen der Hierarchie einen vollständigen Graphen L erzeugen. In diesem Fall wird nicht versucht, ∂L mit den GBS der anderen Graphen in der Hierarchie abzugleichen. Dies würde ohnehin zu keinem Erfolg führen, da in ∂L keinerlei Halbkanten enthalten sind. Stattdessen wird L sofort aus der Hierarchie entfernt und eine neue Produktionsregel erstellt, welche auf der linken Seite den vollständigen Graphen L und auf der rechten Seite den leeren Graphen $\varnothing$ enthält. [24]

3.4 Erzeugen von Variationen mithilfe der abgeleiteten Regeln

3.4.1 Ableiten eines neuen Winkelgraphen

Haben wir erfolgreich eine Graphgrammatik abgeleitet, können wir diese nun verwenden, um daraus neue Winkelgraphen zu erzeugen. Dazu wird zunächst eine Starter-Regel ausgewählt, um eine Grundlage für die weiteren Operationen zu schaffen. Es entsteht ein vollständiger Graph

G. Anschließend werden nach und nach weitere zufällige Produktionsregeln aus der Grammatik ausgewählt und auf den aktuellen Winkelgraphen angewandt. Dabei ist es egal, in welche Richtung die Regel verwendet wird. Wir können diese sowohl konstruktiv als auch destruktiv benutzen. Um eine Regel anzuwenden, muss der auszuschneidene Graph T als Teilgraph in G enthalten sein. Dies ist der Fall, wenn sich ein Homomorphismus $m : T \rightarrow G$ finden lässt. Das Finden eines Homomorphismus zwischen beliebigen Graphen gilt als NP-schweres Problem und ist somit äußerst kompliziert. [8] Für das Finden von Homomorphismen in planaren Graphen gibt es allerdings effiziente Algorithmen, die in linearer Zeit zu einem Ergebnis kommen. [11] Da wir ausschließlich mit planaren Graphen arbeiten, stellt dies also kein Problem für uns dar. Diesen Prozess können wir beliebig oft wiederholen und jederzeit abbrechen. Solang zumindest eine Starter-Regel angewandt wurde, erhalten wir in jedem Fall einen vollständigen Winkelgraphen und somit ein valides Ergebnis für diesen Teilschritt.

3.4.2 Festsetzen der Knotenpositionen

Jetzt gilt es nur noch den eben erzeugten Winkelgraphen in eine Outputstruktur mit festen Knotenpositionen umzuwandeln und gleichzeitig sicherzustellen, dass diese ohne jegliche Überschneidungen der Kanten dargestellt werden kann. Dazu verwenden wir einen ähnlichen Ansatz wie Bokeloh et al. [6] und stellen den Graphen als lineares Gleichungssystem dar, für welches wir anschließend eine Lösung finden. Die einzelnen Gleichungen dieses Systems lassen sich jeweils aus den Kanten des Winkelgraphen ableiten. Dazu betrachten wir die Positionen des Startknotens v_0 und des Endknotens v_1, sowie den Tangentenwinkel θ. Dieser kann wie folgt als zweidimensionaler Richtungsvektor dargestellt werden: $u = [cos(\theta), sin(\theta)]$. Es handelt sich hierbei um einen Einheitsvektor, der mit einer Kantenlänge s multipliziert auf die Position von v_0 addiert werden kann, um v_1 zu erhalten. Die gesamte Gleichung zum Beschreiben einer Kante lautet dann: $v_0 + su = v_1$ bzw. $v_0 + su - v_1 = 0$. [24] All diese Kantengleichungen können in eine Matrix-Gleichung der Form $Ax = 0$ gebracht werden, wobei x alle Unbekannten (Knotenpositionen und Kantenlängen) und A alle Koeffizienten dieser Unbekannten enthält. 0 stellt den Nullvektor des entsprechenden Vektorraums dar. Für jede Kantengleichung werden zwei neue Zeilen in die Matrix-Gleichung eingefügt, da wir mit zweidimensionalen Vektoren arbeiten. Eine Zeile beschreibt die x-Koordinaten, die andere Zeile die y-Koordinaten der Vektoren. Das Gleichungssystem, das durch dieses Vorgehen entsteht, ist stets unterbestimmt, d.h. es gibt mehr Unbekannte als Gleichungen im System. Die Konsequenz daraus ist, dass sich keine eindeutige Lösung finden lässt. Stattdessen können wir nur einen Raum an möglichen Lösungen finden, aus welchem wir dann stichprobenartig einzelne Lösungen auswählen können. Der Raum an mög-

lichen Lösungen lässt sich durch das Berechnen des Kerns der Matrix A finden. Der Kern einer Matrix M ist die Menge aller Vektoren x, für die gilt: $Mx = 0$.[1] Um nun den Kern von A zu finden, bringen wir A zunächst in die reduzierte Zeilenstufenform. Die reduzierte Zeilenstufenform einer Matrix ist stets eindeutig und erfüllt die folgenden Eigenschaften [25]:

- alle Zeilen, die ledglich aus Nullen bestehen, befinden sich ganz unten
- jede Zeile beginnt mit einer Folge von Nullen (ggf. mit Länge 0), gefolgt vom sogenannten *Pivot-Element*
- das Pivot-Element einer Zeile befindet sich stets rechts von den Pivot-Elementen aller darüberliegender Zeilen
- das Pivot-Element ist stets 1
- jede Spalte mit einem Pivot-Element enthält außer dem Pivot-Element nur Nullen

Zum Umwandeln einer Matrix in diese Form verwenden wir den Gauß-Jordan-Algorithmus. Dieser ist bereits in vielen anderen Quellen, wie z.B. [25], detailliert behandelt worden und soll von uns nicht näher betrachtet werden. Wichtig ist das Ergebnis, welches wir durch dessen Anwendung erhalten. Aus diesem können wir nun den Kern der Matrix berechnen. Dies wird nachfolgend an einem Beispiel demonstriert:

$$A = \begin{bmatrix} 1 & 2 & -1 & 1 & -4 \\ 2 & 3 & -1 & 1 & -11 \\ -2 & 0 & -3 & 1 & 22 \end{bmatrix} \longrightarrow rref(A) = \begin{bmatrix} 1 & 0 & 0 & -2 & -8 \\ 0 & 1 & 0 & 2 & 1 \\ 0 & 0 & 1 & 1 & -2 \end{bmatrix}$$

Die reduzierte Zeilenstufenform wird hier abgekürzt durch “rref” (reduced row echelon form). Der Kern von A kann nun berechnet werden, indem die folgende Gleichung gelöst wird:

$$\begin{bmatrix} 1 & 0 & 0 & -2 & -8 \\ 0 & 1 & 0 & 2 & 1 \\ 0 & 0 & 1 & 1 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

[1] https://www.studysmarter.de/studium/mathematik-studium/lineare-algebra/kern-einer-matrix/ [Letzter Zugriff am 01.07.2024]

Durch die Umwandlung in die reduzierte Zeilenstufenform kann die Lösung aus dieser Darstellung mehr oder weniger abgelesen werden. Jede Spalte ohne ein Pivot-Element repräsentiert eine freie Variable, was aufgrund der Unterbestimmung des Gleichungssystems zustande kommt. Wählen wir als freie Variablen $x_4 = t$ und $x_5 = s$, so ergibt sich für die weiteren Variablen: $x_1 = 8s + 2t,\ x_2 = -s - 2t,\ x_3 = 2s - t$. Der Kern von A lautet dann:

$$Kern(A) = \begin{bmatrix} 8s + 2t \\ -s - 2t \\ 2s - t \\ t \\ s \end{bmatrix} = \begin{bmatrix} 2 \\ -2 \\ -1 \\ 1 \\ 0 \end{bmatrix} t + \begin{bmatrix} 8 \\ -1 \\ 2 \\ 0 \\ 1 \end{bmatrix} s$$

Nun können wir beliebige Werte für s und t einsetzen, um eine konkrete Lösung für das Gleichungssystem zu erhalten. Leider bedeutet das Finden einer solchen Lösung nicht unbedingt, dass wir nun auch eine valide Darstellung für die Outputstruktur gefunden haben. Die gefundene Lösung könnte eventuell negative Werte für die Kantenlängen enthalten, welche nicht dargestellt werden können. Um dies zu umgehen, müssen wir die vorgeschlagenen Lösungen also stets auf ungültige Werte überprüfen und ggf. eine neue Lösung generieren. Neben dem Überprüfen auf negative Kantenlängen könnten wir hier außerdem weitere durch den Endnutzer angegebene Kriterien überprüfen und so z.B. alle Kantenlängen auf einen bestimmten Wertebereich beschränken. [24]

4 Implementierung

Im folgenden Kapitel betrachten wir die praktische Implementierung der zuvor vorgestellten Konzepte. Dazu werden zunächst einige spezifische Anforderungen an die zu entwickelnde Software aufgelistet. Anschließend wird darauf eingegangen, wie diese Anforderungen dann umgesetzt werden, indem die verwendeten Technologien und Entwurfsmuster vorgestellt werden.

4.1 Anforderungen an die Software

4.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben, *was* umgesetzt werden soll, also welche konkreten Funktionen von der Software bereitgestellt werden sollen. Diese lauten wie folgt:

1. es sollen beliebige polygonale 2D-Strukturen als Input eingelesen werden können
2. es sollen einige Beispielstrukturen als Input zur Verfügung gestellt werden, zwischen denen der Nutzer auswählen kann
3. es sollen automatisch zum Input lokal ähnliche Strukturen generiert werden können:
    a) aus einer eingelesenen Inputstruktur sollen automatisch Regeln für eine Graphgrammatik abgeleitet werden können
    b) aus einer gegebenen Graphgrammatik sollen verschiedene Graphen abgeleitet werden können
    c) aus einem solchen Graphen soll dann eine planare Outputstruktur mit fester Geometrie (also festen Knotenpositionen) erzeugt werden können

4. es soll eine grafische Benutzeroberfläche geben, in welcher der Nutzer Parameter für die Generierung einstellen, sowie zwischen den verschiedenen Inputstrukturen auswählen können soll
5. sowohl die Inputstrukturen, die daraus erzeugt Grammatik und die generierten Variationen sollen visualisiert werden können

4.1.2 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen beschreiben, *wie* die oben aufgelisteten Funktionen umgesetzt werden sollen, also welche Qualitätskriterien dabei eingehalten werden sollen. Diese lauten wie folgt:

1. die Anwendung soll auf Windows ausführbar sein
2. die Nutzeroberfläche soll einfach und übersichtlich gehalten werden
3. die implementierten Algorithmen sollen sich deterministisch verhalten und alle Ergebnisse sollen reproduzierbar sein
4. die Software soll modular aufgebaut sein, wobei die Module selbst eine hohe Kohäsion vorweisen und untereinander schwach gekoppelt sein sollen
5. alle wichtigen Komponenten sollen durch Tests abgedeckt sein
6. der Code soll verständlich sein und alle nicht-trivialen Bestandteile des Codes sollen mit Kommentaren versehen werden

4.2 Verwendete Technologien und Bibliotheken

Der gesamte Code ist in Java 16 geschrieben. Die grafische Benutzeroberfläche wird mithilfe des Java GUI-Toolkits Swing umgesetzt, welches Bestandteil der Java-Runtime ist und viele Bibliotheken zum Erstellen von simplen Nutzeroberflächen bereitstellt. Zum Rendern der erzeugten Strukturen wird die `Graphics` Komponente des Abstract Window Toolkits (AWT) verwendet, welches ebenfalls ein Teil der Java-Runtime ist. Die entwickelte Software wird hierbei als Teil eines größeren PCG-Projekts von Prof. Dr. Philipp Jenke umgesetzt, in welchem sich viele weitere Studenten-Projekte, sowie Implementierungen von weiteren Algorithmen und Datenstrukturen im Bereich der prozeduralen Generierung befinden. Die grundsätzlichen von uns verwendeten

Datenstrukturen und Algorithmen werden jedoch allesamt selbst implementiert und wir bedienen uns lediglich einiger Helfer-Funktionen, z.B. für das Einlesen von Dateien oder dem Loggen des Programmablaufs.

4.3 Architektur

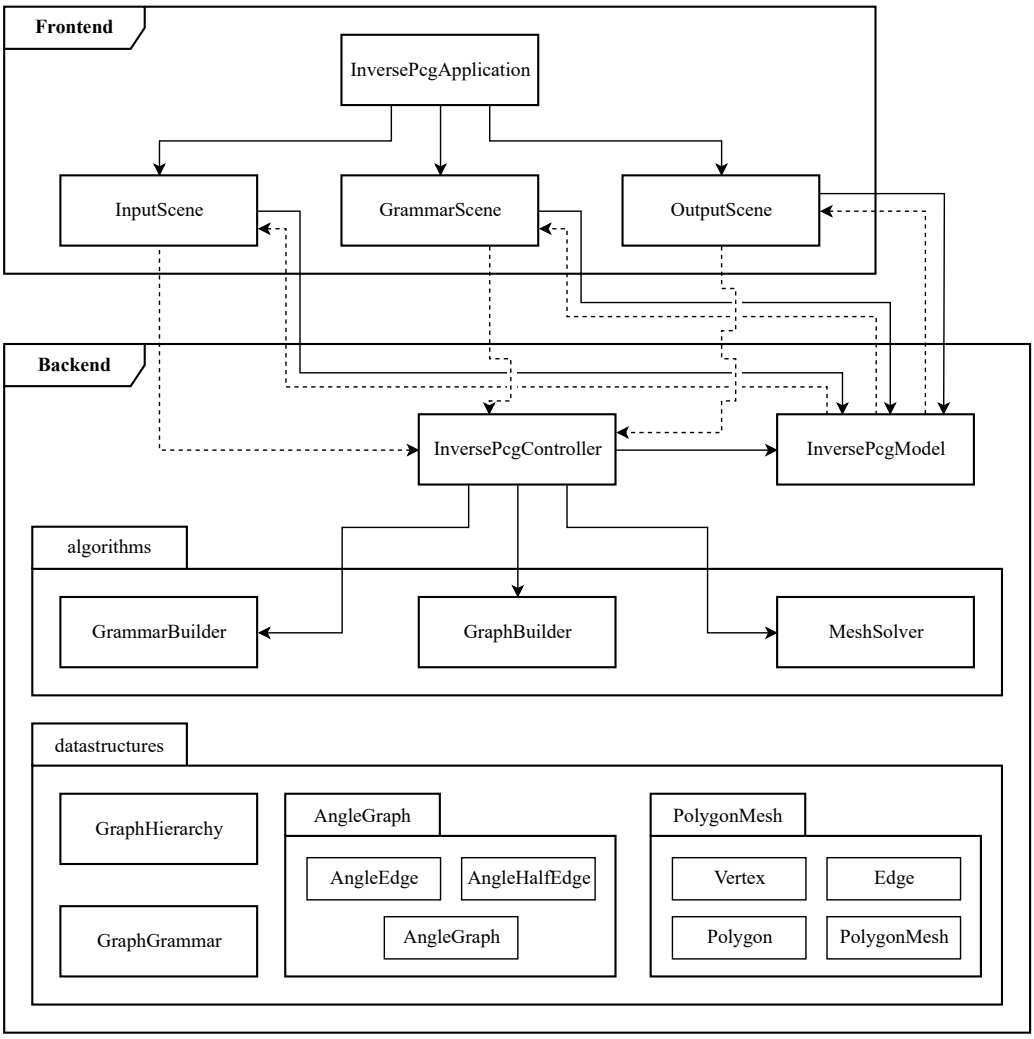

Abbildung 4.1: Architektur der entwickelten Software.

Um ein solches Softwareprojekt übersichtlich zu halten und in eine logische Struktur zu bringen, ist das Verwenden bestimmter Entwurfsmuster nötig. Wie in Abbildung 4.1 zu sehen ist, bedienen wir uns hierfür dem bewährten Muster *Model-View-Controller (MVC)*.

Das MVC Entwurfsmuster beschreibt drei separate logische Einheiten bzw. Komponenten. Das *Datenmodell (model)* verwaltet alle für die Anwendung wichtigen Daten und steht somit im Mittelpunkt. Das Datenmodell ist von den anderen Komponenten unabhängig und speichert die Daten einheitlich ab. Die anderen Komponenten können beliebig ausgetauscht werden, ohne dass das Datenmodell auf diese angepasst werden muss. Die *Ansicht (view)* ist für die Darstellung der Daten des Datenmodells zuständig und realisiert die Interaktion mit dem Endnutzer durch Bereitstellen einer Benutzeroberfläche. Die entsprechenden Daten werden von dieser Komponente nur dargestellt, nicht aber anderweitig verarbeitet. Für die Verarbeitung der Daten ist die *Steuerung (controller)* verantwortlich. Die Ansicht ist von der Steuerung unabhängig, da diese alle darzustellenden Daten ausschließlich aus dem Datenmodell bezieht. Die Steuerung wird von der Ansicht über Anfragen des Endnutzers informiert, verarbeitet diese und aktualisiert anschließend das Datenmodell. Die Ansicht wird daraufhin vom Datenmodell über die Änderung der Daten informiert und aktualisiert die Darstellung. [2]

In unserem Fall werden diese Komponenten wie folgt umgesetzt: Die Ansicht wird durch die Frontend-Komponenten realisiert, das Datenmodell wird in der Klasse `InversePcgModel` umgesetzt und die Steuerung setzt sich zusammen aus den Klassen `InversePcgController`, `GrammarBuilder`, `GraphBuilder`, und `MeshSolver`. Im Datenmodell sind neben den zu visualisierenden Datenstrukturen ebenfalls Parameter gespeichert, welche von der Steuerung zum Verarbeiten der Daten benutzt werden. Diese Parameter können in der Benutzeroberfläche angepasst werden, woraufhin die Ansicht die Parameterwerte im Datenmodell direkt anpasst. Die Steuerung bietet Schnittstellen an, welche von der Ansicht genutzt werden, um die Befehle des Benutzers auszuführen. Nach der Ausführung dieser Befehle nimmt die Steuerung Änderungen am Datenmodell vor, was von der Ansicht bemerkt wird, da die dort enthaltenen Szenen das Beobachter-Muster (Observer-Pattern) nutzen, um auf Änderungen im Datenmodell zu lauschen. Wird das Datenmodell geändert, so werden alle Beobachter darüber informiert. Anschließend wird die Darstellung der visualisierten Datenstrukturen in der Ansicht angepasst.

Dies sollte einen groben Überblick über den Aufbau der entwickelten Software geschafft haben. In den folgenden Kapiteln werden nun weitere Details zur Implementierung der einzelnen Komponenten vorgestellt.

4.4 Datenstrukturen

Schauen wir uns zunächst die implementierten Datenstrukturen etwas genauer an. Diese stellen unseren Input und Output dar, werden von den implementierten Algorithmen verarbeitet und durch die Benutzeroberfläche visualisiert. Grundsätzlich werden dabei vier verschiedene Datenstrukturen implementiert, die nun betrachtet werden.

4.4.1 PolygonMesh

Die `PolygonMesh` Klasse repräsentiert die in Unterabschnitt 3.2.1 vorgestellte polygonale Struktur, die der Algorithmus als Input bekommt und als Output erzeugt. Jedes `PolygonMesh` Objekt verwaltet eine Liste von Knoten (Klasse `Vertex`), eine Liste von Kanten (Klasse `Edge`) und eine Liste von Polygonen (Klasse `Polygon`). Diese werden alle auf der obersten Ebene (also in der `PolygonMesh` Klasse) verwaltet, um das Erzeugen von Duplikaten zu verhindern. Da sich mehrere Polygone die gleichen Kanten, und mehrere Kanten die gleichen Knoten teilen können, kann es sonst vorkommen, dass diese mehrfach in der Datenstruktur gespeichert werden. Durch die Verwaltung aller Instanzen der anderen Klassen in `PolygonMesh` selbst wird dies verhindert.

Ein `Vertex` besitzt lediglich nur ein einziges Feld: eine zweidimensionale Knotenposition und kann dadurch vollständig beschrieben werden. Eine `Edge` wird durch zwei verschiedene Knoten definiert. Für diese wird jeweils nur ein Index gespeichert, über welchen das eigentliche `Vertex` Objekt in der Knotenliste in `PolygonMesh` referenziert werden kann. Die Felder in `Polygon` werden ähnlich umgesetzt: auch hier werden wieder nur Indexe für die untergeordneten Objekte gespeichert. Jedes `Polygon` besitzt dabei eine Liste von Kanten- als auch Knoten-Indexen. Außerdem enthält jedes `Polygon` ein Feld für eine Farbe, welche später genutzt wird, um dieses in der grafischen Darstellung einzufärben.

Neben der Verwaltung der eigentlichen Datenstruktur stellt die `PolygonMesh` Klasse außerdem Funktionalität zum Serialisieren und Deserialisieren der Objekte bereit. Ein `PolygonMesh` Objekt kann in eine `.mesh` Datei umgewandelt werden und umgekehrt kann ein neues `PolygonMesh` Objekt durch das Einlesen einer `.mesh` Datei initialisiert werden. So können wir neue `PolygonMesh` Objekte zur Laufzeit erzeugen. In einer `.mesh` Datei werden alle Knotenpositionen, die Knoten-Indexe der einzelnen Polygone, sowie die Polygonfarben als Plain-Text abgespeichert. Dies ermöglicht das einfache Erstellen von neuen Polygonstrukturen, die dann dem Nutzer als Input für den Algorithmus bereitgestellt werden können.

4.4.2 AngleGraph

Die `AngleGraph` Datenstruktur implementiert den in Unterabschnitt 3.2.3 vorgestellten Winkelgraphen. Zusätzlich zur `AngleGraph` Klasse selbst, implementieren wir hier die Klassen `AngleEdge` und `AngleHalfEdge`, die die Kanten und Halbkanten repräsentieren, als auch eine innere Klasse `GraphBoundary`, die den GBS des jeweiligen Graphen darstellt. Eine eigene Klasse für die Knoten eines Winkelgraphen gibt es nicht, da diese in einem Winkelgraphen keine Eigenschaften besitzen. Stattdessen wird in `AngleGraph` nur die Gesamtanzahl n aller vorhandenen Knoten gespeichert, während die Kanten und Halbkanten statt Referenzen auf ein Knoten-Objekt lediglich Knoten-IDs von 0 bis $n - 1$ zugeordnet bekommen. Eine `AngleEdge` enthält zwei solcher Knoten-IDs, während die `AngleHalfEdge` nur an einem Knoten anliegt. Beide Kantentypen enthalten außerdem einen Tangentenwinkel. Die `GraphBoundary` verwaltet eine Liste an Indexen der enhaltenen Halbkanten, sowie Referenzen zu den sich daraus ergebenden positiven und negativen Drehungen, welche durch festgelegte Zahlenwerte repräsentiert werden.

Außerdem wird auch hier einiges an Funktionalität bereitgestellt. Der GBS kann automatisch aus den vorhandenen Halbkanten abgeleitet werden. Es gibt einige Konstruktoren für verschiedene Anwendungsfälle, darunter einen, der ein `PolygonMesh` in einen Winkelgraphen umwandeln kann. Außerdem wird ein tiefgehender Objektvergleich implementiert, welcher die zugrundeliegende Struktur der einzelnen Winkelgraphen miteinander vergleicht. Dies erleichtert das Finden von Duplikaten beim Erstellen der Graph-Hierarchie immens. Für die `GraphBoundary` wird Funktionalität bereitgestellt, die diesen in eine auf den Kantenwinkeln basierende Darstellung umwandeln kann, welche zwischen allen Winkelgraphen einheitlich verglichen werden kann und somit das Ableiten von Regeln für die Graphgrammatik ermöglicht.

4.4.3 GraphHierarchy

Die Graph-Hierarchie ist eine vergleichsweise simple Datenstruktur und ist vollständig in der Klasse `GraphHierarchy` implementiert. Diese enthält eine Liste für die verschiedenen Generationen, welche sich jeweils aus einer Reihe an `AngleGraph` Objekten zusammensetzen. Außerdem existiert ein Feld für die aktuelle Größe der Hierarchie, welche die aktuelle Anzahl an darin enthaltenen Winkelgraphen darstellt.

4.4.4 GraphGrammar

Die finale Datenstruktur implementiert die Graphgrammatik. Diese ist ebenfalls in einer einzigen Klasse `GraphGrammar` implementiert. Hier werden zwei Listen verwaltet: eine für die Starter-Regeln und eine für alle anderen Regeln. Starter-Regeln bestehen ausschließlich aus einem `AngleGraph`, da die andere Seite einer solchen Regel den leeren Graphen enthält, welcher nicht explizit repräsentiert werden muss. Alle anderen Regeln werden durch ein Paar von Winkelgraphen gebildet, welche jeweils die linke und die rechte Seite der Regel repräsentieren.

4.5 Datenmodell

Betrachten wir nun das Datenmodell. Dieses wird in der Klasse `InversePcgModel` realisiert. Dieses enthält Felder für die polygonale Inputstruktur, die daraus erzeugte Grammatik, sowie für die letztendlich erzeugte polygonale Outputstruktur. Außerdem werden hier alle für die Steuerung relevanten Parameter verwaltet, welche zunächst im Konstruktor mit vorgebenen Standardwerten initialisiert werden und anschließend vom Benutzer beliebig angepasst werden können. Die Liste an vorhandenen Parametern lautet wie folgt:

- `seed`: der Seed für die von den Algorithmen verwendeten Zufallsgeneratoren
- `maxGeneration`: die Maximalanzahl an Generationen in der Graph-Hierarchie beim Erstellen der Graphgrammatik
- `iterations`: die Anzahl an Iterationen beim Ableiten von neuen Winkelgraphen aus der Graphgrammatik
- `maxTries`: die maximale Anzahl an Versuchen für das Erzeugen einer validen Zeichnung für einen erzeugten Graphen
- `minRandVal`: der kleinstmögliche Zufallswert für die freien Variablen beim Erzeugen einer Graph-Zeichnung
- `maxRandVal`: der größtmögliche Zufallswert für die freien Variablen beim Erzeugen einer Graph-Zeichnung
- `waitTime`: die Wartezeit zwischen den einzelnen Iterationen beim Erzeugen des Outputs

Zum Umsetzen des Beobachter-Musters erweitert `InversePcgModel` die Klasse `Observable`, welche erneut von Prof. Dr. Philipp Jenke bereitgestellt wird. Diese erlaubt das Hinzufügen von `Observer` Objekten, die dann durch die ebenfalls gegebene Methode `notifyAllObservers` über Änderungen am Datenmodell informiert werden können.

4.6 Steuerung

Zum Verarbeiten der Daten im Datenmodell wird in der Klasse `InversePcgController` die Steuerungskomponente realisiert. Diese verwaltet die implementierten Algorithmen und bietet der Ansicht verschiedene Schnittstellen für die einzelnen Teilschritte des Verfahrens. Alle Funktionen, bei denen die Generierung von Zufallszahlen relevant ist, erhalten ihre Zufallswerte von einem in dieser Komponente verwalteten Zufallsgenerator, dessen generierten Ergebnisse durch den `seed` Parameter im Datenmodell gesteuert werden. Dadurch verhalten sich alle Algorithmen voll und ganz deterministisch.

Schauen wir uns nun die von der Steuerung verwalteten Algorithmen genauer an. Diese werden in drei Klassen unterteilt:

4.6.1 GrammarBuilder

Die `GrammarBuilder` Klasse enthält alle Algorithmen rund um das Ableiten der Graphgrammatik. Es wird ein `PolygonMesh` als Input entgegengenommen und eine `GraphGrammar` als Endergebnis erzeugt. Das Herzstück hierbei ist das inkrementelle Aufbauen einer Graph-Hierarchie. Um die erste Generation dieser Hierarchie zu bilden, gibt es Funktionalität, welche alle verschiedenartigen Kanten im ursprünglichen `PolygonMesh` Objekt in einen entsprechenden Winkelgraphen umwandelt. Die Tangentenwinkel der Kanten können dabei aus den Positionen der Start- und Endknoten berechnet werden. Die abgeleiteten Winkelgraphen in der ersten Generation der Hierarchie bestehen dann jeweils aus einer Halbkante a und einer weiteren Halbkante $\overline{a}$ mit dem zu a entgegengesetzten Kantenwinkel. Die zweite Generation kann ebenfalls direkt aus dem `PolygonMesh`-Input erzeugt werden. Dazu wird für jeden Knoten K in der Polygonstruktur ein neuer Winkelgraph G erzeugt. Für jede Kante, die mit K verbunden ist, wird anschließend eine neue Halbkante in G eingefügt. Ist dies für alle Knoten geschehen, so wird anschließend noch der GBS für alle erzeugten Winkelgraphen bestimmt. Somit haben wir nun auch alle Primitive für den entsprechenden Input gefunden. Anschließend werden alle weiteren Generationen der Graph-Hierarchie durch Anwenden von Klebeoperationen erzeugt,

welche ebenfalls in dieser Klasse implementiert werden. Das Erzeugen von neuen Generationen geschieht solange, bis die durch `maxGeneration` spezifierte Maximalanzahl an Generationen erreicht wurde. Jedes Mal, wenn ein neuer Graph erstellt wird, wird dieser mit den GBS aller vorher erzeugten Graphen abgeglichen um ggf. eine neue Regel ableiten zu können. Neben dem Abgleich der Graph Boundaries wird außerdem in jedem Schritt überprüft, ob ein vollständiger Graph generiert wurde, welcher dann zum Ableiten einer Starter-Regel verwendet wird. Konnte durch eine dieser beiden Überprüfungen eine Regel abgeleitet werden, so wird diese in die Grammatik eingefügt und anschließend der entsprechende Graph aus der Hierarchie entfernt. Bei erfolgreichem Durchlauf aller Algorithmen in dieser Klasse können wir ein `PolygonMesh` in eine nutzbare `GraphGrammar` umwandeln. Falls durch den Nutzer eine zu kleine Maximalanzahl an zu generierenden Graphen festgelegt wird, kann es jedoch sein, dass die Grammatik sehr wenige Regeln enthält. Dann kann es außerdem passieren, dass keine einzige Starter-Regel gefunden wird, wodurch in den Folgeschritten keine neuen Graphen daraus abgeleitet werden könnten. Um dieses Problem zu vermeiden und später zumindest noch einige Variationen generieren zu können, fügen wir der Grammatik direkt eine Starter-Regel hinzu, welche den leeren Graphen mit dem ursprünglichen Input-Graphen ersetzt.

4.6.2 GraphBuilder

In der `GraphBuilder` Klasse befindet sich alles an Funktionalität zum Ableiten von neuen Winkelgraphen aus einer gegebenen Graphgrammatik. Zunächst wird zufällig eine der vorhandenen Starter-Regeln ausgewählt, um eine Grundlage für alle weiteren Operationen zu schaffen. Wir erhalten einen vollständigen Graphen G. Anschließend wird für die im Datenmodell festgelegte Anzahl an Iterationen (`iterations`) immer wieder eine neue zufällige Regel ausgewählt und versucht, diese auf G anzuwenden. Dabei gibt es keine Garantien dafür, dass die gewählten Regeln überhaupt angewandt werden können und es kann passieren, dass sich G in einigen der Iterationen nicht verändert. Zum Anwenden einer Regel muss eine der beiden in der Regel enthaltenen Graphen als Subgraph in G vorhanden sein, d.h. wir müssen einen Homomorphismus finden, der von dem Graphen in der Regel auf G abbildet. Dafür gibt es effiziente Algorithmen (siehe [11]), jedoch wird von uns aufgrund von Zeitgründen lediglich ein Brute-Force Ansatz verwendet, der zufällige Abbildungen zwischen den Knoten der beiden Graphen generiert und anschließend für jede generierte Abbildung prüft, ob diese einen validen Homomorphismus darstellt. Dies stellt bei kleineren Graphen kein Problem dar, wird aber mit zunehmender Anzahl an Knoten sehr schnell äußerst zeitaufwändig und sollte in einer zukünftigen Version behoben werden.

4.6.3 MeshSolver

Abschließend betrachten wir noch die Klasse `MeshSolver`. Diese enthält Funktionalität zum Umwandeln eines Winkelgraphen in eine feste geometrische Darstellung in Form eines `PolygonMesh` Objektes. Im ersten Schritt wird hier durch die Kanten des Winkelgraphen iteriert und aus diesen die Matrix-Darstellung des Graphen abgeleitet. Diese Matrix wird dann per Gauß-Jordan-Algorithmus [25] in reduzierte Zeilenstufenform umgewandelt. Für jede Spalte in dieser Matrix, die kein Pivot-Element enthält, werden zufällige Werte für die entsprechenden freien Variablen generiert. Der Kern der Matrix kann direkt aus der reduzierten Zeilenstufenform abgeleitet werden, weshalb wir diesen nicht noch einmal explizit berechnen. Stattdessen setzen wir die bereits generierten Zufallszahlen direkt für die freien Variablen ein und berechnen daraus alle weiteren Knotenpositionen und Kantenlängen. Anschließend überprüfen wir diese Positionen auf eventuell auftretende negative Kantenlängen und generieren die Zufallszahlen ggf. neu. Dies wiederholen wir solange, bis entweder eine gültige Lösung gefunden werden konnte oder wir die vorgebenene Maximalanzahl an Versuchen (`maxTries`) erreicht haben. Anschließend werden die Knotenpositionen des generierten `PolygonMesh` Objektes normalisiert, sodass sich die gesamte Struktur bei der Darstellung auch vollständig im Viewport der Anwendung befindet.

4.7 Ansicht

Als letzte grundlegende Komponente der Anwendung betrachten wir die Ansicht, welche die Interaktion mit dem Benutzer realisiert und dafür eine Benutzeroberfläche bereitstellt. Diese Benutzeroberfläche wird mithilfe des Java GUI-Toolkits Swing implementiert. Dies geschieht in der Klasse `InversePcgApplication`, welche die gesamte Anwendung darstellt. Diese basiert auf der von Prof. Dr. Philipp Jenke bereitgestellten Klasse `CG2DApplication`, welche selbst die Swing-Komponente `JFrame` erweitert und Funktionalität zum einfachen Hinzufügen verschiedener Szenen bereitstellt. Wir implementieren drei verschiedene Szenen-Typen, die dieser Anwendung hinzugefügt werden. Die implementierten Szenen erweitern jeweils die Klasse `Scene2D`, welche ebenfalls von Prof. Dr. Philipp Jenke bereitgestellt wird. In dieser wird Funktionalität zum Zeichnen von Linien, Polygonen, etc. implementiert.

4.7.1 InputScene

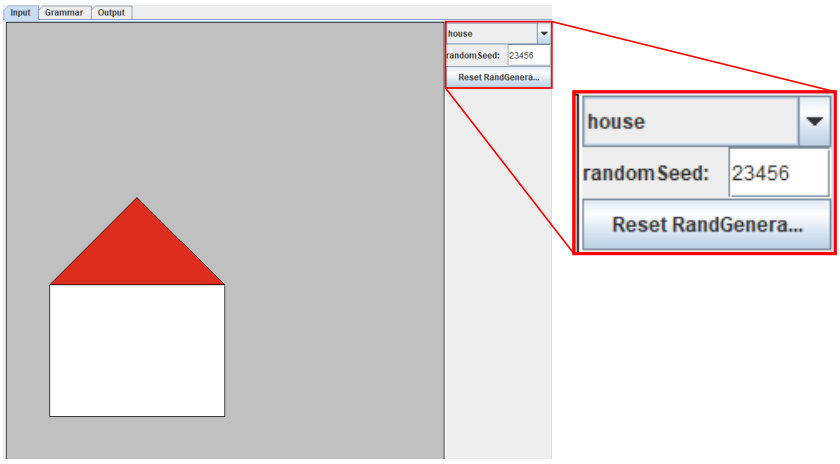

Abbildung 4.2: Die Input-Szene der Benutzeroberfläche.

Die `InputScene` wird zum Rendern der verschiedenen Inputstrukturen genutzt. Der Nutzer kann zwischen einer Reihe an bereitgestellten Strukturen auswählen, welcher dann in dieser Szene visualisiert werden. Diese Strukturen werden in Form von `.mesh`-Dateien gespeichert, zwischen welchen der Nutzer in einer `JComboBox` auswählen kann. Um aus den Dateien die eigentliche Datenstruktur zu erstellen, wird der ausgewählte Dateiname an die Steuerungskomponente weitergegeben. Sobald diese das entsprechende `PolygonMesh` erstellt hat, wird dieses im Datenmodell gespeichert. Die `InputScene` wird folglich über diese Änderung informiert und kann nun die ausgewählte Inputstruktur visualisieren. Außerdem kann in dieser Szene der `seed` für den Zufallsgenerator festgelegt werden.

4.7.2 GrammarScene

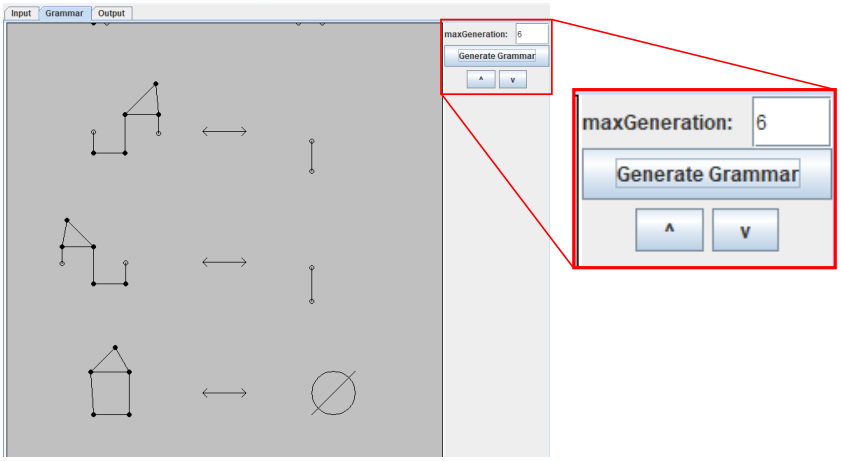

Abbildung 4.3: Die Grammatik-Szene der Benutzeroberfläche.

Die `GrammarScene` wird genutzt, um die aus dem Input erzeugte Graphgrammatik darzustellen. Um die Grammatik zu erhalten, wird zunächst erneut die Steuerungskomponente dazu beauftragt, diese aus dem momentan in der `InputScene` ausgewählten Input abzuleiten. Anschließend kann die Graphgrammatik aus dem Datenmodell entnommen und visuell dargestellt werden. Zum Steuern der Ableitung dieser Grammatik kann der Endnutzer hier den Parameter `maxGeneration` anpassen. Da die in den Produktionsregeln enthaltenen Winkelgraphen neben den Kantenwinkeln keine festen Geometrie-Eigenschaften besitzen, können diese nur angenähert dargestellt werden. Die einzelnen Kanten werden mit zufällig festgelegten Kantenlängen dargestellt und es wird versucht, die Winkel der Kanten weitesgehend zu respektieren. Soll jedoch eine Kante gezeichnet werden, die zwei bereits gezeichnete Knoten verbindet, so kann der Kantenwinkel nicht respektiert werden und die gezeichnete Kante verläuft in dem Winkel, der durch die beiden zu verbindenen Knoten vorgegeben wird. Trotzdem sind die entstehenden Darstellungen nützlich, um einen Einblick in die Funktionsweise hinter der Graphgrammatik zu bekommen.

4.7.3 OutputScene

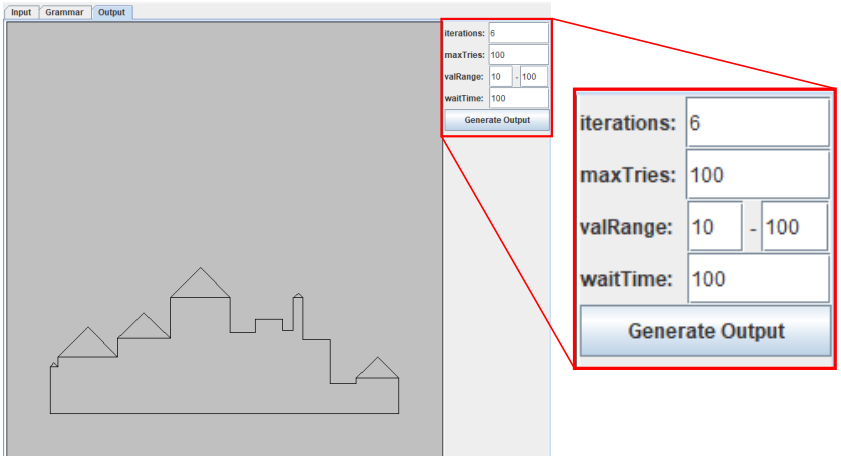

Abbildung 4.4: Die Output-Szene der Benutzeroberfläche.

In der `OutputScene` werden die final erzeugten Ergebnisse visualisiert. Dazu stehen dem Benutzer hier die Einstellungsmöglichkeiten für alle noch fehlenden Parameter (`iterations`, `maxTries`, `minRandVal`, `maxRandVal` und `waitTime`) zur Verfügung. `minRandVal` und `maxRandVal` werden dabei in der Darstellung zu `valRange` zusammengefasst. Mithilfe dieser Angaben kann die Steuerungskomponente einen neuen Output erzeugen, der zunächst wieder in das Datenmodell eingetragen wird, bevor dieser dann in der `OutputScene` visualisiert werden kann.

5 Auswertung

Werten wir die vorgestellte Implementierung nun aus. Dazu überprüfen wir zunächst die anfangs vorgestellten Anforderungen an die gesamte Software und diskutieren anschließend, wie erfolgreich dabei das eigentliche Verfahren umgesetzt wurde. Dazu werden einige Beispieldurchläufe mit verschiedenen Parameter-Werten betrachtet.

5.1 Überprüfen der Anforderungen

5.1.1 Funktionale Anforderungen

Funktionale Anforderung 1

Es sollen beliebige polygonale 2D-Strukturen als Input eingelesen werden können. Diese Anforderung wird durch die entwickelte `PolygonMesh` Datenstruktur realisiert. In dieser können beliebige Polygone in eine größere zusammenhängende Struktur gebracht werden. Diese Datenstruktur kann von der `GrammarBuilder` Klasse als Input entgegengenommen werden, was die Grundlage für das Durchführen aller weiteren Teilschritte des Verfahrens darstellt.

Funktionale Anforderung 2

Es sollen einige Beispielstrukturen als Input zur Verfügung gestellt werden, zwischen denen der Nutzer auswählen kann. Dies wird durch die vorgestellte `InputScene` realisiert. Hier kann der Benutzer ein Dropdown-Menü öffnen und erhält eine Liste an möglichen Beispielstrukturen, die alle vom `InversePcgController` in ein `PolygonMesh` Objekt umgewandelt werden können. Beispielstrukturen können ohne viel Aufwand und ohne Anpassung des Codes entfernt, hinzugefügt oder ausgetauscht werden, da diese in separaten `.mesh`-Dateien gespeichert werden und dynamisch zur Laufzeit eingelesen werden können.

Funktionale Anforderung 3

Es sollen automatisch zum Input lokal ähnliche Strukturen generiert werden können. Die Theorie hinter dem implementierten Verfahren beruht vollständig auf den in Kapitel 3 vorgestellten Konzepten und erzeugt den Output nach dem dort beschriebenen Vorgehen. Ist das Verfahren erfolgreich durchgelaufen, so lässt sich jeder Teil der dadurch erzeugten Outputstruktur im Input wiederfinden und die erzeugte Strukture ist somit lokal ähnlich zum Input. In vielen Fällen kann das Verfahren aufgrund von Defiziten in der `MeshSolver` Komponente jedoch nicht erfolgreich durchlaufen, wodurch diese Anforderung nicht immer vollständig erfüllt werden kann.

a) Aus einer eingelesenen Inputstruktur sollen automatisch Regeln für eine Graphgrammatik abgeleitet werden können. Diese Anforderung wird durch die Klasse `GrammarBuilder` realisiert, welche ein gegebenes `PolygonMesh` Objekt in eine `GraphGrammar` umwandeln kann.

b) Aus einer gegebenen Graphgrammatik sollen verschiedene Graphen abgeleitet werden können. Hierfür ist die `GraphBuilder` Klasse zuständig. Der dort entwickelte Algorithmus nimmt ein `GraphGrammar` Objekt entgegen und erzeugt daraus ein neues `AngleGraph` Objekt.

c) Aus einem solchen Graphen soll dann eine planare Outputstruktur mit fester Geometrie (also festen Knotenpositionen) erzeugt werden können. Dies wird vom `MeshSolver` übernommen, welcher ein `AngleGraph` Objekt in ein `PolygonMesh` Objekt umwandeln kann. Hier gibt es allerdings das Problem, dass in vielen Fällen keine valide Lösung für die Outputstruktur gefunden werden kann. Bei der Verarbeitung etwas größerer Winkelgraphen entstehen beim Lösen des entstandenen Gleichungssystems so viele freie Variablen, dass es eine ziemlich große Wahrscheinlichkeit gibt, dass zumindest eine dieser Variablen zum Erzeugen von ungültigen Werten für eine der anderen unbekannten Variablen führt. Dies versuchen wir zu umgehen, indem dieser Vorgang bis zu `maxTries` Mal wiederholt wird, in der Hoffnung, dass die jeweils nächste zufällige Lösung gültig sein wird. Da jedes Mal jedoch alle Variablen erneut generiert werden, ist die Fehlerwahrscheinlichkeit dabei so hoch, dass in den meisten Fällen auch nach etlichen Versuchen keine valide Lösung generiert werden kann. In diesem Fall visualisieren wir trotzdem die zuletzt erzeugte Struktur, jedoch wird diese nicht vollständig zum Input lokal ähnlich sein.

Funktionale Anforderung 4

Es soll eine grafische Benutzeroberfläche geben, in welcher der Nutzer Parameter für die Generierung einstellen, sowie zwischen den verschiedenen Inputstrukturen auswählen können soll.

Dies wird durch die Klasse `InversePcgApplication` realisiert. In den drei darin implementierten Szenen kann der Nutzer alle wichtigen Einstellungen vornehmen. In der `InputScene` kann der Seed für den Zufallsgenerator festgelegt und eine Inputstruktur ausgewählt werden. In der `GrammarScene` kann die Anzahl an Generation in der Graph-Hierarchie festgelegt werden. In der `OutputScene` können alle weiteren in Abschnitt 4.5 vorgestellten Parameter eingestellt werden.

Funktionale Anforderung 5

Sowohl die Inputstrukturen, die daraus erzeugt Grammatik und die generierten Variationen sollen visualisiert werden können. Die Visualisierung der hier genannten Datenstrukturen wird ebenfalls durch die `InversePcgApplication` übernommen. In der `InputScene` wird die ausgewählte Inputstruktur dargestellt, die `GrammarScene` übernimmt das Darstellen der daraus erzeugten Grammatik, und die `OutputScene` wird zum Visualisieren der verschiedenen erzeugten Variationen benutzt.

5.1.2 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderung 1

Die Anwendung soll auf Windows ausführbar sein. Der gesamte Code wurde auf einem Windows 11 PC und einem Laptop mit Windows 10 implementiert und getestet. Bei der Ausführung der Anwendung auf beiden Geräten gibt es keine Probleme. Die Anforderung wurde also erfüllt.

Nichtfunktionale Anforderung 2

Die Nutzeroberfläche soll einfach und übersichtlich gehalten werden. Die implementierte Benutzeroberfläche wurde sehr simpel gestaltet und stellt nur die wichtigsten Funktionen bereit. Es gibt lediglich drei verschiedene Szenen, welche jeweils nur aus einem Viewport und einer kurzen Reihe an Einstellungsmöglichkeiten, sowie einem Knopf zum Senden eines Befehls an die Steuerungskomponente bestehen. Alles wird in einem Fenster dargestellt, es gibt keine Popups und mit der Ausnahme des Dropdown-Menüs zum Auswählen der Inputstruktur auch keine aufklappbaren Komponenten mit versteckter Komplexität. Da die Anwendung vollständig mit Swing und AWT umgesetzt wurde, ist das Design der verwendeten GUI-Komponenten selbst ebenfalls sehr simpel gehalten.

Nichtfunktionale Anforderung 3

Die implementierten Algorithmen sollen sich deterministisch verhalten und alle Ergebnisse sollen reproduzierbar sein. Auch diese Anforderung wird erfüllt. Alle Algorithmen, bei denen das Erzeugen von Zufallszahlen eine Rolle spielt, erhalten die Zufallszahlen von ein und demselben Zufallszahlengenerator, der von der Steuerungskomponente verwaltet wird. Dieser wird stets mit einem vom Benutzer festgelegten Seed initialisiert und erzeugt bei gleichem Seed stets die gleichen zufälligen Zahlen. Werden vom Nutzer nach Start der Anwendung die exakt gleichen Aktionen immer und immer wiederholt, so sind die erzeugten Ergebnisse ebenfalls immer und immer wieder identisch.

Nichtfunktionale Anforderung 4

Die Software soll modular aufgebaut sein, wobei die Module selbst eine hohe Kohäsion vorweisen und untereinander schwach gekoppelt sein sollen. Wie in Abschnitt 4.3 ausführlich erklärt wurde, besteht die entwickelte Software aus vielen verschiedenen in sich logisch gekapselten Komponenten. Aufgrund des gewählten MVC-Entwurfsmusters sind die drei Hauptkomponenten Ansicht, Datenmodell und Steuerung lose untereinander gekoppelt und können größtenteils unabhängig voneinander existieren. [2] Es werden einheitliche Schnittstellen geboten oder die Informationen werden über das Beobachter-Muster zwischen den Komponenten ausgetauscht. In beiden Fällen wäre ein Austauschen der einzelnen Komponenten ledglich mit sehr geringem Aufwand verbunden. Die Kohäsion innerhalb der einzelnen Module ist hoch, da diese jeweils einen klar definierten Aufgabenbereich haben und die gesamte darin umgesetzte Funktionalität nur zum Umsetzen der entsprechenden Aufgabe des Moduls genutzt wird. Funktionalität, die sich von mehreren Komponenten geteilt wird, wurde in eine separate Klasse `Utils` ausgelagert. Die Komponenten mit der niedrigsten Kohäsion sind die Klassen `PolygonMesh` und `AngleGraph`, da diese neben dem Definieren der entsprechenden Datenstruktur auch komplexere Funktionalität für die Transformation dieser Datenstruktur enthalten. Diese Funktionalität hätte jeweils eventuell auch in eine separate Klasse ausgelagert werden können.

Nichtfunktionale Anforderung 5

Alle wichtigen Komponenten sollen durch Tests abgedeckt sein. Diese Anforderung wurde größtenteils umgesetzt. Es gibt separate Testklassen für die drei implementierten Algorithmus-Klassen `GrammarBuilder`, `GraphBuilder` und `MeshSolver`. In diesen wird jeweils der gesamte

Ablauf des entsprechenden implementierten Teilschritts getestet, indem die erhaltenen Endergebnisse überprüft werden. Außerdem befinden sich dort Tests für viele, aber nicht alle, Teilfunktionen, die dabei genutzt werden. Auch für die Datenstrukturen gibt es Tests, allerdings nur für die Klassen `AngleGraph` und `PolygonMesh`, da diese Funktionalität für komplexere Transformationen der entsprechenden Datenstrukturen enthalten.

Nichtfunktionale Anforderung 6

Der Code soll verständlich sein und alle nicht-trivialen Bestandteile des Codes sollen mit Kommentaren versehen werden. Jede von uns selbst implementierte und nicht von anderen Klassen geerbte Methode wurde mit Javadoc-Kommentaren versehen, die die genaue Funktionsweise, die involvierten Paramater und den Rückgabewert der Methode erklären. Komplexere Methoden wurden außerdem intern mit weiteren Kommentaren versehen, wenn die einzelnen Teilschritte als nicht-trivial erachtet wurden. Die Benennung der verschiedenen Klassen, Methoden und Variablen wurde so gewählt, dass diese so präzise wie möglich den entsprechenden Zweck beschreiben. Generell wurde der Code auf Basis der in [1] vorgestellten Konventionen erstellt. Damit gilt diese Anforderung als erfüllt.

5.2 Beispieldurchläufe

Nachfolgend werden einige der erhaltenen Ergebnisse präsentiert, welche mithilfe von verschiedenen Parameter-Konfigurationen erzeugt wurden. Zunächst zeigen wir einige erfolgreiche Ergebnisse. Es werden jeweils die verwendete Inputstruktur, die daraus erzeugte Grammatik, sowie die letztendliche Outputstruktur dargestellt. Die Grammatik kann in den meisten Fällen aufgrund der großen Anzahl an Regeln nicht vollständig dargestellt werden und es wird jeweils nur ein Ausschnitt an Regeln gezeigt. Außerdem werden die verwendeten Parameter-Konfigurationen für jedes Beispiel gezeigt.

5.2.1 Erfolgreiche Durchläufe

Für den in Abbildung 5.1 dargestellten Durchlauf verwendete Parameter: `seed = 23456`, `maxGeneration = 6`, `iterations = 5`, `maxTries = 100`, `minRandVal = 10`, `maxRandVal = 100`. Die Ergebnisse entsprechen den Erwartungen.

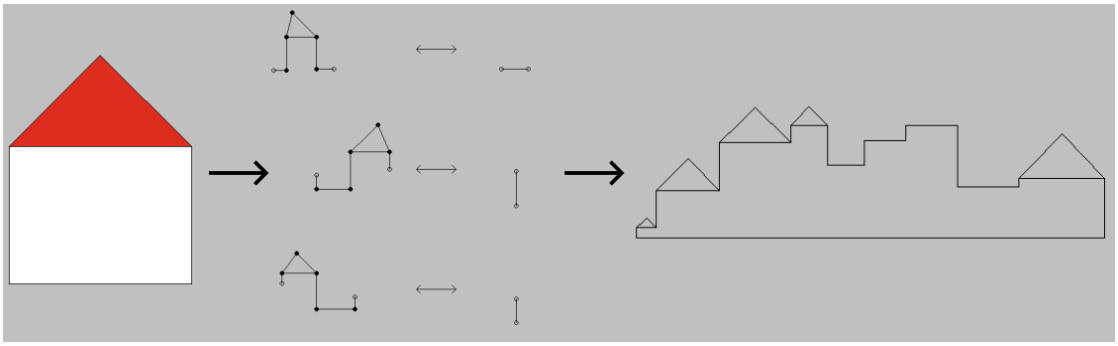

Abbildung 5.1: Erfolgreicher Durchlauf mit Input-Datei `house.mesh`.

Für den in Abbildung 5.2 dargestellten Durchlauf verwendete Parameter: `seed = 77`, `maxGeneration = 3`, `iterations = 7`, `maxTries = 100`, `minRandVal = 10`, `maxRandVal = 100`. Die Ergebnisse entsprechen den Erwartungen.

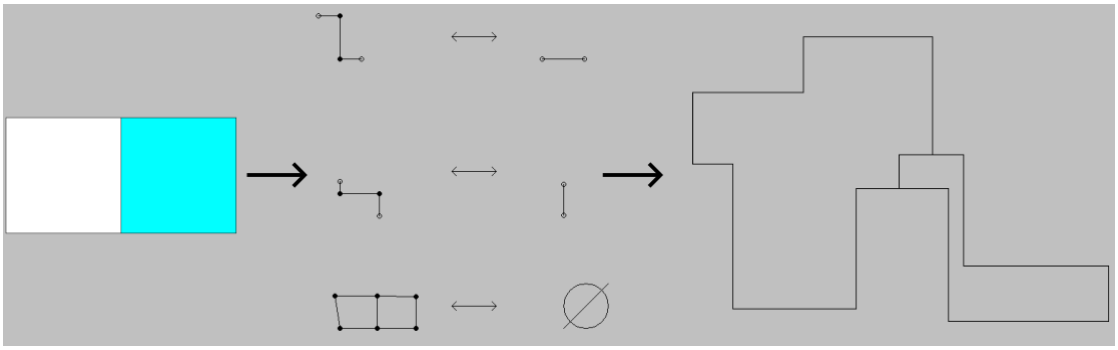

Abbildung 5.2: Erfolgreicher Durchlauf mit Input-Datei `square_double.mesh`.

Für den in Abbildung 5.3 dargestellten Durchlauf verwendete Parameter: `seed = 5555`, `maxGeneration = 3`, `iterations = 5`, `maxTries = 100`, `minRandVal = 10`, `maxRandVal = 10`. Die Ergebnisse entsprechen den Erwartungen. Durch das Limitieren der Zufallszahlen auf nur exakt einen Wert enthält der Output sehr viele Kanten mit der gleichen Kantenlänge und nimmt somit eine regelmäßige Form an. Es wirkt, als wäre der Output auf Basis eines Gitters modelliert worden.

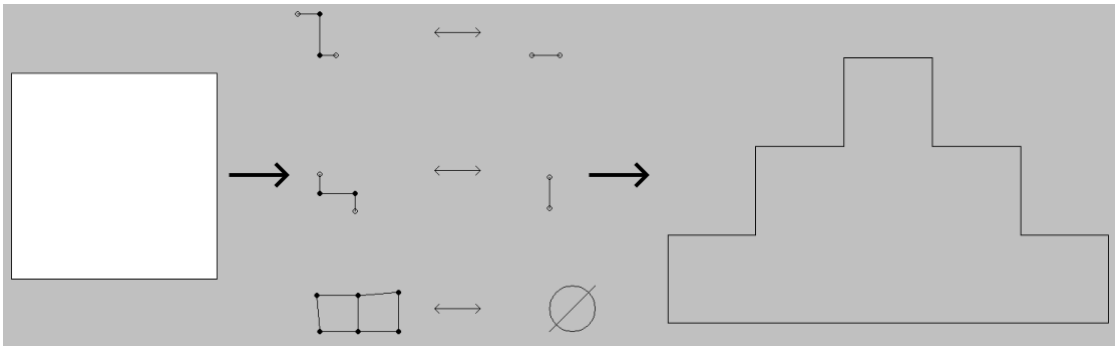

Abbildung 5.3: Erfolgreicher Durchlauf mit Input-Datei `square.mesh`.

Für den in Abbildung 5.4 dargestellten Durchlauf verwendete Parameter: `seed = 123`, `maxGeneration = 3`, `iterations = 10`, `maxTries = 100`, `minRandVal = 10`, `maxRandVal = 70`. Die Ergebnisse entsprechen den Erwartungen.

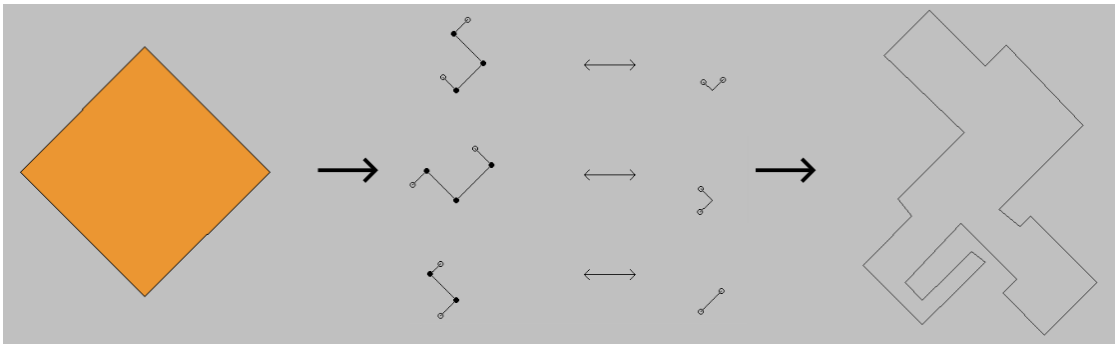

Abbildung 5.4: Erfolgreicher Durchlauf mit Input-Datei `rhombus.mesh`.

5.2.2 Fehlerhafte Durchläufe

Für den in Abbildung 5.5 dargestellten Durchlauf verwendete Parameter: `seed = 337`, `maxGeneration = 7`, `iterations = 7`, `maxTries = 100`, `minRandVal = 10`, `maxRandVal = 100`. Die Ergebnisse entsprechen nicht den Erwartungen. Alle dargestellten Kantenwinkel lassen sich zwar im Input wiederfinden und es wirkt so, als wäre die erzeugte Outputstruktur zumindest lokal ähnlich zum Input. Das Ergebnis ist jedoch trotzdem fehlerhaft, da sich einige der Kanten überschneiden.

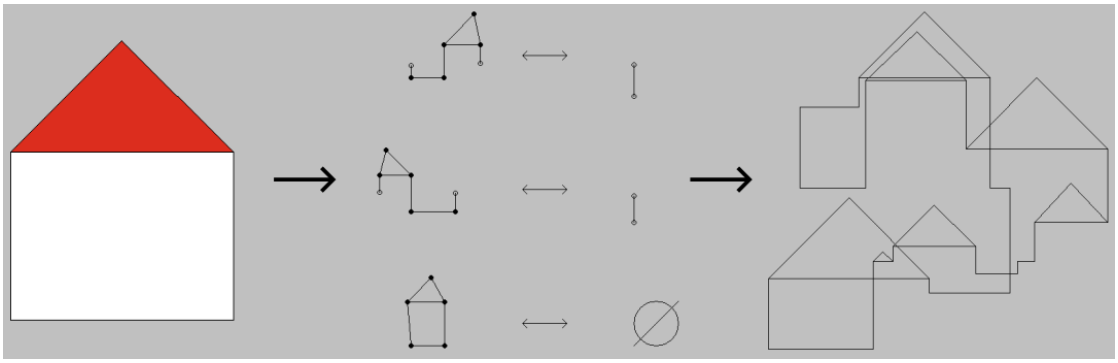

Abbildung 5.5: Fehlerhafter Durchlauf mit Input-Datei `house.mesh`.

Für den in Abbildung 5.6 dargestellten Durchlauf verwendete Parameter: `seed = 445`, `maxGeneration = 2`, `iterations = 7`, `maxTries = 10`, `minRandVal = 10`, `maxRandVal = 100`. Diesmal ist der dargestellte Output planar, jedoch liegt hier keine lokale Ähnlichkeit vor. Die Kantenwinkel stimmen zwar alle mit den Kantenwinkeln im Input überein, jedoch unterscheidet sich die Topologie der beiden Strukturen an einigen Stellen. Einige der Knoten in der Outputstruktur (an den "Spitzen") lassen sich nicht im Input wiederfinden.

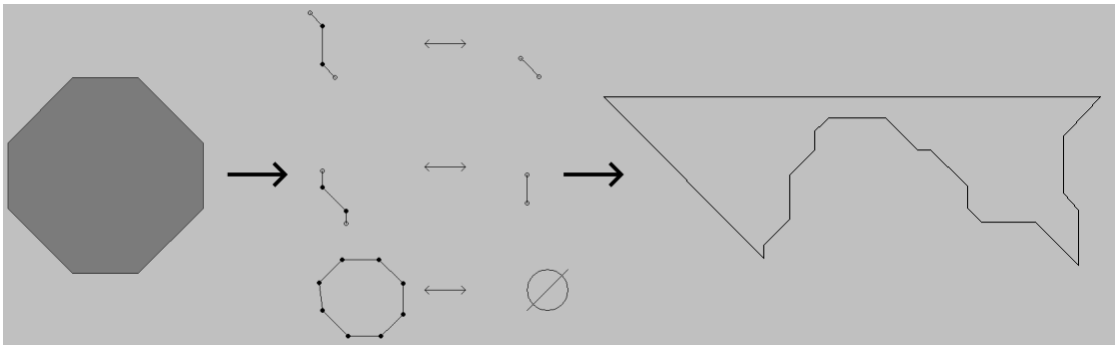

Abbildung 5.6: Fehlerhafter Durchlauf mit Input-Datei `octagon.mesh`.

Für den in Abbildung 5.7 dargestellten Durchlauf verwendete Parameter: `seed = 23456`, `maxGeneration = 30`, `iterations = 3`, `maxTries = 100`, `minRandVal = 10`, `maxRandVal = 100`. Der Output entspricht den geforderten Eigenschaften, ist sowohl zum Input lokal ähnlich als auch planar und somit nicht fehlerhaft. Trotzdem ist das Ergebnis unbrauchbar, da sich der Output kaum vom Input unterscheidet. Der Output besitzt zwar andere Kantenlängen, ist strukturell allerdings exakt gleich aufgebaut. Trotz der hohen Anzahl an zugelassenen Generationen konnte das Verfahren lediglich nur eine einzige Starter-Regel für die

Grammatik ableiten (beide dargestellten Starter-Regeln sind identisch). Aus dieser Grammatik kann keine hohe Vielfalt an Ergebnissen erzeugt werden.

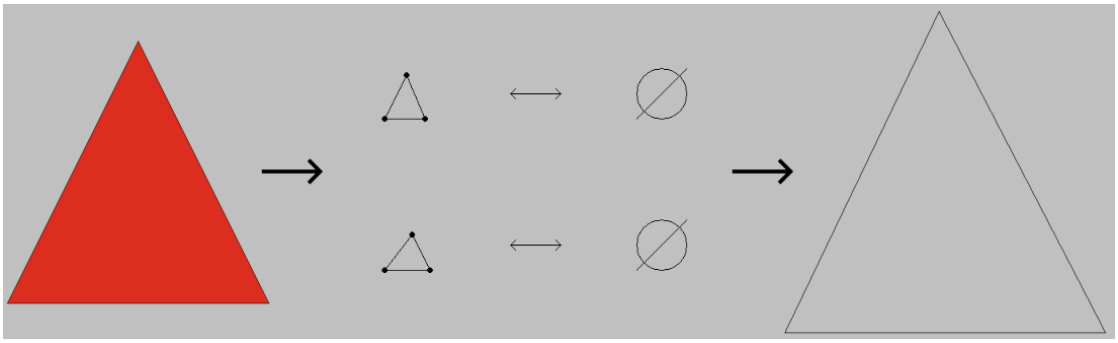

Abbildung 5.7: Ungenügender Durchlauf mit Input-Datei `triangle.mesh`.

5.3 Auswertung der Ergebnisse

Allgemein sind die Ergebnisse durchaus zufriedenstellend. Die erzeugten Strukturen erfüllen zwar in vielen Fällen nicht die Anforderungen an die lokale Ähnlichkeit und können auch oft nicht ohne Überschneidungen einiger der Kanten dargestellt werden, jedoch wird durch die von uns implementierte Software klar demonstriert, dass dieses Verfahren grundsätzlich funktioniert. Auch wenn es dabei noch viele Probleme gibt, die behoben werden müssen um dieses Verfahren wirklich in der Praxis einsetzen zu können, erfüllt die entwickelte Anwendung voll und ganz ihren Zweck als Prototyp.

6 Fazit

Abschließend soll diese Arbeit kurz reflektiert werden. Dazu fassen wir die gewonnenen Erkenntnisse zusammen und geben anschließend einen Ausblick auf mögliche Verbesserungen und Erweiterungsmöglichkeiten des vorgestellten Ansatzes und seiner Umsetzung.

6.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde die inverse prozedurale Generierung von Strukturen für virtuelle Welten untersucht, dies spezifisch für den zweidimensionalen Raum. Es wurden einige bereits entwickelte Ansätze im Bereich der "normalen" prozeduralen Generierung vorgestellt und das vorhandene Kernproblem solcher Verfahren aufgezeigt. Es wurde klargestellt, wie hier durch das inverse Vorgehen Abhilfe geschaffen werden kann, woraufhin einige der bereits existierenden inversen Verfahren erläutert wurden. Hier wurden erneut die vorhandenen Probleme aufgezeigt und anschließend ein neues Verfahren präsentiert, welches versucht, auch diese Probleme weitesgehend aus dem Weg zu räumen. Dieses Verfahren wurde detailliert vorgestellt und es wurde Schritt für Schritt erläutert, wie wir es schaffen, Muster in einer vorgegebenen Struktur zu erkennen, diese in Form einer Grammatik abzubilden und abschließend daraus beliebig viele Variationen abzuleiten, die dann für verschiedenste Zwecke genutzt werden können. Anhand einer prototypischen Implementierung haben wir dann gezeigt, dass dieses Verfahren auch in der Praxis funktioniert, auch wenn darin noch einige Probleme vorhanden sind.

6.2 Verbesserungsansätze und Erweiterungsmöglichkeiten

Das implementierte Verfahren sollte ausreichend demonstriert haben, was mithilfe des vorgestellten Ansatzes möglich ist. Dennoch gibt es hier natürlich einige Verbesserungsansätze und Möglichkeiten zur Erweiterung der betrachteten Konzepte, welche wir gern abschließend erwähnen möchten.

6.2.1 Eingrenzen des Hierarchie-Wachstums

In der momentanen Umsetzung ist es in den meisten Fällen nicht möglich, die Hierarchie wirklich zu leeren und diese wächst immer weiter bis in die Unendlichkeit. Es ist uns nur möglich mit diesem Vorgehen zu einem Ende zu kommen, da wir das Erzeugen der Hierarchie frühzeitig abbrechen, indem wir diese nur bis zu einer bestimmten Größe erweitern. Dadurch können wir zwar eine Vielzahl an Regeln finden, nicht aber kann garantiert werden, dass wir mit der erzeugten Graphgrammatik dann alle zum Input lokal Ähnlichen Polygonstrukturen erzeugen können. An dieser Stelle müsste das Verfahren um weitere Überprüfungen der erstellten Graphen erweitert werden, welche es uns erlauben, diese frühzeitig auch ohne das Finden von Regeln aus der Hierarchie zu entfernen. Einige solcher Ansätze werden im Paper von Merrell [24] diskutiert, jedoch kann selbst durch diese nicht immer garantiert werden, dass die Hierarchie wirklich geleert werden kann. [24]

6.2.2 Inkrementelles Festsetzen der Knotenpositionen

Ein weiteres Problem mit dem hier vorgestellten Verfahren liegt darin, wie wir die aus der Graphgrammatik abgeleiteten Graphen in eine feste geometrische Repräsentation bringen. Wie in Kapitel 5 festgestellt wurde, läuft dies in unserer Implementierung in vielen Fällen schief und es wird ein Output mit ungültigen Kantenlängen erzeugt. Auch bei vielen wiederholten Versuchen kann dieses Problem meist nicht umgangen werden und letztendlich wird ein Output erzeugt, in welchem sich einige der Kanten überschneiden und teilweise einen falschen Kantenwinkel besitzen. Ein Lösungsansatz hierfür wäre es, beim Festsetzen der Knotenpositionen inkrementell vorzugehen. Aktuell erzeugen wir nur die Winkelgraphen selbst nach einem solchen Vorgehen, indem wir in jeder Iterationen nur eine neue Regel auf den bereits erzeugten Graphen anwenden. Jedes Mal, wenn dann jedoch die Knotenpositionen für diesen Winkelgraphen bestimmt werden sollen, fangen wir damit komplett von Neuem an und bestimmen alle Knotenpositionen auf einmal. Wenn dabei dann etwas schiefläuft, generieren wir erneut alles von vorn. Würden wir stattdessen auch beim Bestimmen der Knotenpositionen inkrementell vorgehen und in jedem Schritt nur die Positionen der in der aktuellen Iteration neu hinzugefügten Knoten berechnen, so würde das Verfahren deutlich seltener Probleme beim Erzeugen des Outputs haben. Auch dies wird im Paper von Merrell diskutiert. [24]

6.2.3 Erweiterung in die dritte Dimension

Die letzte der zu erwähnenden Verbesserungen ist die Erweiterung dieses Verfahrens in den dreidimensionalen Raum. Diese Arbeit hat sich lediglich mit dem zweidimensionalen Raum befasst, da dies die Komplexität des Ganzen etwas verringert. Die grundlegenden vorgestellten Konzepte können allerdings auch im 3D angewandt werden, sind dann nur etwas komplexer in der Umsetzung. Hierdurch würde das Verfahren für viele weitere Anwendungsfälle in Frage kommen, die mit der aktuellen Umsetzung nicht abgedeckt werden können. Auch hierzu gibt es weitere Informationen im Paper von Merrell. [24]

Literaturverzeichnis

[1] *Java Code Conventions*. 1997. – https://www.oracle.com/technetwork/java/codeconventions-150003.pdf [Letzter Zugriff am 09.07.2024]

[2] *Model-View-Controller Pattern*. S. 353–402. In: *Learn Objective-C for Java Developers*. Berkeley, CA : Apress, 2009. – URL https://doi.org/10.1007/978-1-4302-2370-2_20. – ISBN 978-1-4302-2370-2

[3] ADAMS, David u. a.: Automatic generation of dungeons for computer games. In: *Bachelor thesis, University of Sheffield, UK*. (2002)

[4] BLATZ, Michael ; KORN, Oliver: *A Very Short History of Dynamic and Procedural Content Generation*. S. 1–13. In: KORN, Oliver (Hrsg.) ; LEE, Newton (Hrsg.): *Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation*. Cham : Springer International Publishing, 2017. – URL https://doi.org/10.1007/978-3-319-53088-8_1. – ISBN 978-3-319-53088-8

[5] BOKELOH, Martin ; WAND, Michael ; SEIDEL, Hans-Peter: A connection between partial symmetry and inverse procedural modeling. In: *ACM SIGGRAPH 2010 Papers*. New York, NY, USA : Association for Computing Machinery, 2010 (SIGGRAPH '10). – URL https://doi.org/10.1145/1833349.1778841. – ISBN 9781450302104

[6] BOKELOH, Martin ; WAND, Michael ; SEIDEL, Hans-Peter ; KOLTUN, Vladlen: An algebraic model for parameterized shape editing. In: *ACM Trans. Graph.* 31 (2012), jul, Nr. 4. – URL https://doi.org/10.1145/2185520.2185574. – ISSN 0730-0301

[7] CARLI, Daniel Michelon D. ; BEVILACQUA, Fernando ; TADEU POZZER, Cesar ; D'ORNELLAS, Marcos C.: A Survey of Procedural Content Generation Techniques Suitable to Game Development. In: *2011 Brazilian Symposium on Games and Digital Entertainment*, 2011, S. 26–35

[8] COOK, Stephen A.: The complexity of theorem-proving procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. New York, NY, USA : Association for Computing Machinery, 1971 (STOC '71), S. 151–158. – URL https://doi.org/10.1145/800157.805047. – ISBN 9781450374644

[9] EHRIG, H. ; PFENDER, M. ; SCHNEIDER, H. J.: Graph-grammars: An algebraic approach. In: *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, 1973, S. 167–180

[10] ENGELFRIET, J. ; ROZENBERG, G.: *NODE REPLACEMENT GRAPH GRAMMARS*. S. 1–94. In: *Handbook of Graph Grammars and Computing by Graph Transformation*, URL https://www.worldscientific.com/doi/abs/10.1142/9789812384720_0001

[11] EPPSTEIN, David: *Subgraph Isomorphism in Planar Graphs and Related Problems*. 1999

[12] FREIKNECHT, Jonas: Procedural content generation for games. (2021). – URL https://madoc.bib.uni-mannheim.de/59000

[13] GUMIN, Maxim: *Wave Function Collapse Algorithm*. September 2016. – URL https://github.com/mxgmn/WaveFunctionCollapse

[14] HENDRIKX, Mark ; MEIJER, Sebastiaan ; VAN DER VELDEN, Joeri ; IOSUP, Alexandru: Procedural content generation for games: A survey. In: *ACM Trans. Multimedia Comput. Commun. Appl.* 9 (2013), feb, Nr. 1. – URL https://doi.org/10.1145/2422956.2422957. – ISSN 1551-6857

[15] KOCH, Helge v.: On a Continuous Curve Without Tangent Constructable from Elementary Geometry. In: *Classics on fractals* (1904)

[16] KÖNIG, Barbara ; NOLTE, Dennis ; PADBERG, Julia ; RENSINK, Arend: *A Tutorial on Graph Transformation*. S. 83–104. In: HECKEL, Reiko (Hrsg.) ; TAENTZER, Gabriele (Hrsg.): *Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig*. Cham : Springer International Publishing, 2018. – URL https://doi.org/10.1007/978-3-319-75396-6_5. – ISBN 978-3-319-75396-6

[17] LAGAE, A. ; LEFEBVRE, S. ; COOK, R. ; DEROSE, T. ; DRETTAKIS, G. ; EBERT, D.S. ; LEWIS, J.P. ; PERLIN, K. ; ZWICKER, M.: A Survey of Procedural Noise Functions. In: *Computer Graphics Forum* 29 (2010), Nr. 8, S. 2579–2600

[18] LINDEN, Roland van der ; LOPES, Ricardo ; BIDARRA, Rafael: Procedural Generation of Dungeons. In: *IEEE Transactions on Computational Intelligence and AI in Games* 6 (2014), Nr. 1, S. 78–89

[19] LINDENMAYER, Aristid: Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. In: *Journal of theoretical biology* 18 (1968), Nr. 3, S. 280–299

[20] MANDELBROT, Benoit B.: *Fractals and the Geometry of Nature*. Bd. 1. WH freeman New York, 1982

[21] MCBRIDE, Martin: *L system Koch curve*. 2024. – https://graphicmaths.com/fractals/l-systems/l-systems/ [Letzter Zugriff am 27.06.2024]

[22] MERRELL, P: *Comparing model synthesis and wave function collapse*. 2021. – https://paulmerrell.org/wp-content/uploads/2021/07/comparison.pdf [Letzter Zugriff am 29.06.2024]

[23] MERRELL, Paul: Example-based model synthesis. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. New York, NY, USA : Association for Computing Machinery, 2007 (I3D '07), S. 105–112. – URL https://doi.org/10.1145/1230100.1230119. – ISBN 9781595936288

[24] MERRELL, Paul: Example-Based Procedural Modeling Using Graph Grammars. In: *ACM Trans. Graph.* 42 (2023), jul, Nr. 4. – URL https://doi.org/10.1145/3592119. – ISSN 0730-0301

[25] MEYER, Carl D.: *Matrix analysis and applied linear algebra*. SIAM, 2023

[26] PAPERT, Seymour: Computers for children. In: *Mindstorms: Children, computers, and powerful ideas* (1980), S. 3–18

[27] PERLIN, Ken: An image synthesizer. In: *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : Association for Computing Machinery, 1985 (SIGGRAPH '85), S. 287–296. – URL https://doi.org/10.1145/325334.325247. – ISBN 0897911660

[28] PERLIN, Ken: Improving noise. In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : Association for Computing Machinery, 2002 (SIGGRAPH '02), S. 681–682. – URL https://doi.org/10.1145/566570.566636. – ISBN 1581135211

[29] PRUSINKIEWICZ, Przemyslaw ; LINDENMAYER, Aristid: *Graphical modeling using L-systems*. S. 1–50. In: *The Algorithmic Beauty of Plants*. New York, NY : Springer New York, 1990. – URL https://doi.org/10.1007/978-1-4613-8476-2_1. – ISBN 978-1-4613-8476-2

[30] RAMANTO, Adhika S. ; MAULIDEVI, Nur U.: Markov Chain Based Procedural Music Generator with User Chosen Mood Compatibility. In: *International Journal of Asia Digital Art and Design Association* 21 (2017), Nr. 1, S. 19–24

[31] RODEN, Timothy ; PARBERRY, Ian: From Artistry to Automation: A Structured Methodology for Procedural Content Creation. In: RAUTERBERG, Matthias (Hrsg.): *Entertainment Computing – ICEC 2004*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, S. 151–156. – ISBN 978-3-540-28643-1

[32] SABIDUSSI, Gert: Graph derivatives. In: *Mathematische Zeitschrift* 76 (1961), S. 385–401. – URL https://doi.org/10.1007/BF01210984

[33] SIERPINSKI, Warclaw: Sur une courbe dont tout point est un point de ramification. In: *CR Acad. Sci.* 160 (1915), S. 302–305

[34] SMELIK, R.M. ; TUTENEL, T. ; DE KRAKER, K.J. ; BIDARRA, R.: A declarative approach to procedural modeling of virtual worlds. In: *Computers & Graphics* 35 (2011), Nr. 2, S. 352–363. – URL https://www.sciencedirect.com/science/article/pii/S0097849310001809. – Virtual Reality in Brazil Visual Computing in Biology and Medicine Semantic 3D media and content Cultural Heritage. – ISSN 0097-8493

[35] SPUFFORD, Francis: *Backroom boys: The secret return of the British boffin*. Faber & Faber, 2010

[36] TOGELIUS, Julian ; KASTBJERG, Emil ; SCHEDL, David ; YANNAKAKIS, Georgios N.: What is procedural content generation? Mario on the borderline. In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. New York, NY, USA : Association for Computing Machinery, 2011 (PCGames '11). – URL https://doi.org/10.1145/2000919.2000922. – ISBN 9781450308724

[37] ZUCKER, Matt: *The Perlin noise math FAQ*. 2001. – https://mzucker.github.io/html/perlin-noise-math-faq.html [Letzter Zugriff am 27.06.2024]

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.

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	Verwendung
ChatGPT	Künstliche Intelligenz verwendet als Hilfe bei der Literaturrecherche
Inkscape	Vektorgrafik-Software verwendet zur Erstellung von Abbildungen
draw.io	UML-Modellierungs-Software verwendet für den Architekturentwurf

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