

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

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

TODO SCHLÜSSELWÖRTER

Kurzzusammenfassung

TODO ZUSAMMENFASSUNG

Benjamin Schröder

Title of Thesis

Example-based inverse procedural generation for two-dimensional scenes

Keywords

TODO KEYWORDS

Abstract

TODO ABSTRACT

Inhaltsverzeichnis

Abbildungsverzeichnis	vi
Tabellenverzeichnis	vii
Abkürzungen	viii
1 Einleitung	1
1.1 Motivation	1
1.2 Problemstellung	2
1.3 Ziele und Vorgehen	2
2 Grundlagen	3
2.1 Prozedurale Generierung	3
2.2 Geschichte von PCG	4
3 Theorie	5
3.1 Überblick	5
3.2 Grundlagen	7
3.2.1 Input	7
3.2.2 Lokale Ähnlichkeit	7
3.2.3 Der Winkelgraph	8
3.2.4 Planarität und der Graph Boundary String	9
3.2.5 Teil-Operation	11
3.2.6 Klebe-Operation	12
3.3 Ablauf	13
3.3.1 Anpassen des Inputs	13
3.3.2 Finden der Primitive	13
3.3.3 Aufbauen der Graph-Hierarchie	14
3.3.4 Ableiten der Graph-Grammatik	15

4	Implementation	18
5	Auswertung	19
6	Fazit	20
	Literaturverzeichnis	21
A	Anhang	23
A.1	Verwendete Hilfsmittel	23
	Selbstständigkeitserklärung	24

Abbildungsverzeichnis

3.1	Überblick zum Ablauf des Verfahrens.	6
3.2	r-Ähnlichkeit.	8
3.3	Der Zusammenhang zwischen Drehwinkel und Planarität.	10
3.4	Positive und negative Drehungen.	11
3.5	Branch und Loop Gluing.	13
3.6	Double Pushout Produktionsregel.	16

Tabellenverzeichnis

A.1	Verwendete Hilfsmittel und Werkzeuge	23
-----	--	----

Abkürzungen

DPO Double Pushout.

GBS Graph Boundary String.

PCG Prozedurale Content Generierung.

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. Selbst in größeren Teams mit einer Vielzahl von Designern, nimmt die Erstellung von realistischen Welten einige Monate in Anspruch. [5] Hier kann an vielen 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 bilden eine Grundstruktur für ein komplexeres Design, bei dem die Entwickler dann nur noch kleinere Details per Hand abändern oder hinzufügen müssen.

Andererseits existieren auch viele Videospiele, wie z.B. Minecraft¹ oder Terraria², 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. Somit macht jeder Spieler eine andere Erfahrung und kann das Spiel außerdem gewissermaßen unbegrenzt oft durchspielen, ohne dass es repetitiv wirkt. So etwas wäre ohne Automatisierung gar nicht erst umsetzbar.

¹<https://www.minecraft.net/>

²<https://terraria.org/>

1.2 Problemstellung

Es gibt viele bekannte Verfahren, welche solche Ergebnisse unter der Verwendung von u.a. zellulären Automaten, generativen Grammatiken oder Constraint-basierten Graphen erzielen können. [7] Größtenteils beruhen diese jedoch auf der Anwendung von manuell erstellten Regeln, so z.B. eine Menge an gegebenen Produktionsregeln bei der Nutzung von Grammatiken. Das Erstellen solcher Regeln ist mit viel Arbeit und Trial-and-Error verbunden und kann ohne ein ausgeprägtes Verständnis des angewandten Verfahrens sehr schwierig werden. Dadurch kommt es für viele Designer letztendlich doch nicht in Frage. Hier setzt diese Arbeit an und untersucht die automatische Erstellung solcher Regeln.

1.3 Ziele und Vorgehen

Spezifisch soll versucht werden, Muster in Beispielstrukturen zu erkennen. Aus diesen Mustern sollen dann Regeln zum Zusammensetzen von Strukturen mit ähnlichen Eigenschaften abgeleitet werden.

Hier gibt es bereits verschiedene Verfahren, die einen solchen Ansatz verfolgen. Diese sind u.a. der Gitter-basierte Wave Function Collapse Algorithmus von Maxim Gumin³, die nach Symmetrien suchende inverse prozedurale Modellierung von Bokeloh et al. [1], oder das Polygon-basierte Verfahren von Paul Merrell. [8]

Diese Verfahren werden analysiert und anschließend das vielversprechendste davon praktisch umgesetzt. Das Endergebnis der Arbeit soll dann sein, dass das ausgewählte Konzept ausführlich und verständlich erläutert, und nach eigener Interpretation konkret implementiert wird. Im Rahmen dieser Arbeit soll dies lediglich für den zweidimensionalen Raum geschehen, könnte jedoch im Anschluss auch auf die dritte Dimension ausgeweitet werden.

Ebenfalls soll eine grafische Benutzeroberfläche bereitgestellt werden, über welche der Endnutzer Inputstrukturen auswählen, sowie Parameter zur Beeinflussung des Algorithmus anpassen kann.

³<https://github.com/mxgmn/WaveFunctionCollapse/>

2 Grundlagen

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. [2] Auch Musik kann durch solche Verfahren generiert werden. [9]

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. [12] 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 irgendwie anhand von einer begrenzten Menge an Nutzer-definierten Parametern erzeugt werden können. [11] 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. [10] Togelius et al. [12] 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. [12] 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 weitestgehend zusammen und sollte für unsere Zwecke ausreichen.

2.2 Geschichte von PCG

3 Theorie

Im Folgenden werden die theoretischen Konzepte hinter dem praktischen Teil der Arbeit betrachtet. Das implementierte Verfahren wird Schritt für Schritt vorgestellt und im Detail erläutert. Die vorgestellten Konzepte beruhen auf den Erkenntnissen von Paul Merrell in seiner Arbeit aus dem Jahr 2023 [8].

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

Beim Aufbau der Hierarchie werden die neu entstehenden Graphen auf bestimmte Eigenschaften überprüft, die es uns erlauben, daraus Regeln für eine Graph-Grammatik 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

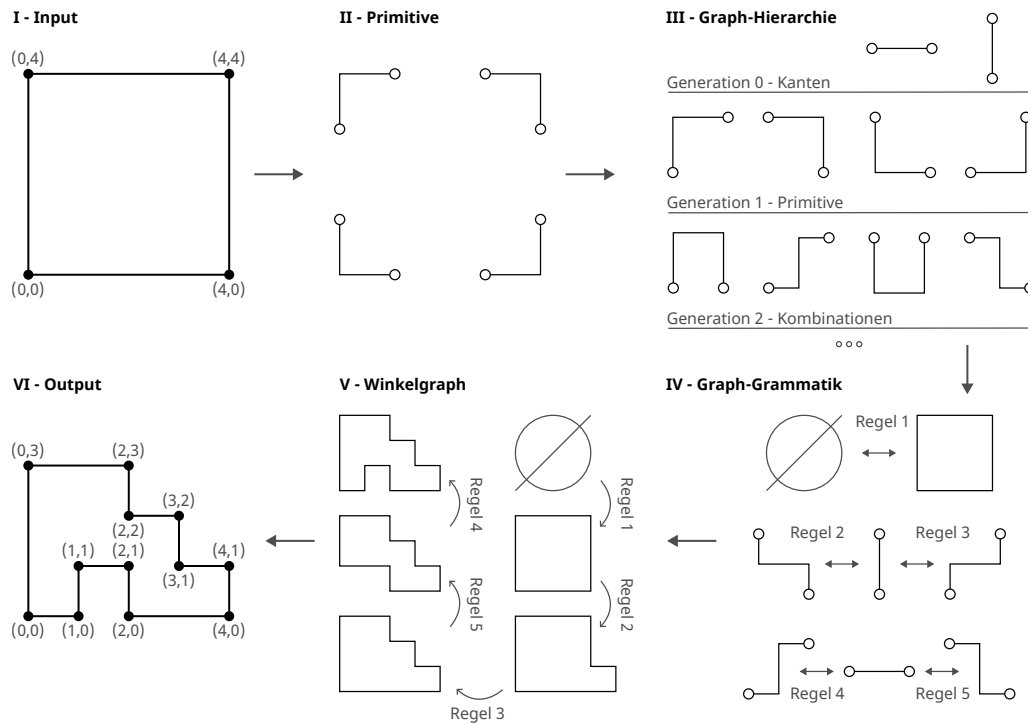


Abbildung 3.1: Überblick zum Ablauf des Verfahrens.

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.

Sobald nun eine Menge von Regeln für die Graph-Grammatik 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).

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, welche wiederum als Sammlung von Punkten und Kanten beschrieben werden können. So sind z.B. Kreise oder andere Strukturen mit Rundungen kein valider Input und könnten lediglich durch komplexe Polygone angenähert werden. Die einzelnen Polygone können außerdem mit Farben versehen werden, um verschiedene Arten von abgegrenzte Bereichen im Input zu markieren.

3.2.2 Lokale Ähnlichkeit

Ziel des Algorithmus ist es, Variationen des Inputs zu erzeugen. Dabei soll der Output eine gewisse Ähnlichkeit zum Input beibehalten. Global vorzugehen und die vollständigen Input- und Output-Strukturen miteinander zu vergleichen führt hierbei allerdings zu keinem vernünftigen Ergebnis. Der Output muss sich zumindest teilweise vom Input unterscheiden, ansonsten ist das Ergebnis nicht zu gebrauchen. Um Vergleiche auf einer kleineren Ebene vornehmen zu können, stellen wir hier das Konzept der *lokalen Ähnlichkeit* vor.

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 Konzept, das zum Verständnis beitragen kann, ist das der *r-Ähnlichkeit*, welche im Paper von Bokeloh et al. [1] 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

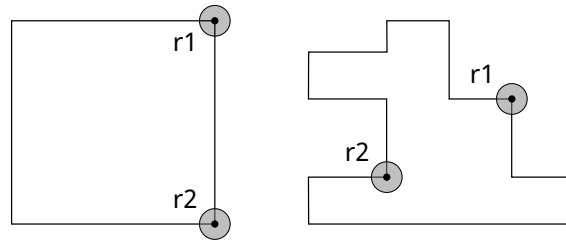


Abbildung 3.2: r-Ähnlichkeit.

uns die restliche Nachbarschaft egal ist. So können die betrachteten Strukturen beliebig skaliert werden und trotzdem ihre lokale Ähnlichkeit zueinander bewahren, solange alle Kantenwinkel dabei beibehalten werden.

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.

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.

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.

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.

Zum Vorbeugen dieses Problems definieren wir hier das Konzept der *Graph Boundary* und die dazugehörige Notation in Form vom *Graph Boundary String (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

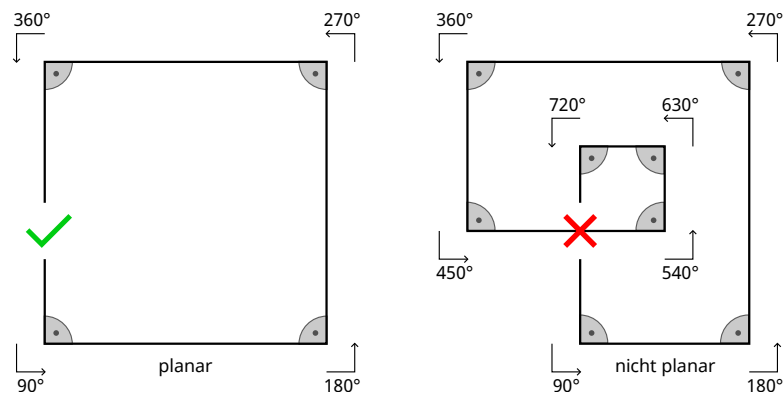


Abbildung 3.3: Der Zusammenhang zwischen Drehwinkel und Planarität.

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 $abc\wedge = bc \wedge a = c \wedge ab = \wedge abc$.

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 $400^\circ - 360^\circ = 40^\circ$ betragen. Wir wissen also nie, ob wir uns aktuell um den Winkel θ , $\theta + 360^\circ$, $\theta + 720^\circ$ oder noch mehr gedreht haben. Dieses Problem lässt sich durch Einführung des Konzepts der *positiven und negativen Drehungen* umgehen.

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 Kanten und wechseln dann auf eine negativ verlaufende Kante, so werden wir dabei in einigen Fällen diesen Schwellwert überschreiten. Falls dies geschieht,

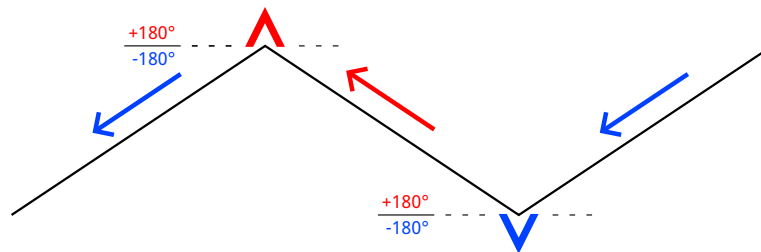


Abbildung 3.4: Positive und negative Drehungen.

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.

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.

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 .

3.2.5 Teil-Operation

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

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.

3.2.6 Klebe-Operation

Zwei entgegengesetzte Halbkanten k und \bar{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. Grundsätzlich gibt es zwei verschiedene Arten von Klebe-Operationen: Loop Gluing und Branch Gluing.

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 “ $a\bar{a}$ ” oder “ $\bar{a} \vee a\wedge$ ” 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:

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

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 \bar{a} und Graph H die Kante a , so kann ein Branch Gluing durchgeführt werden. Hierbei gibt es wieder zwei Optionen:

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

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

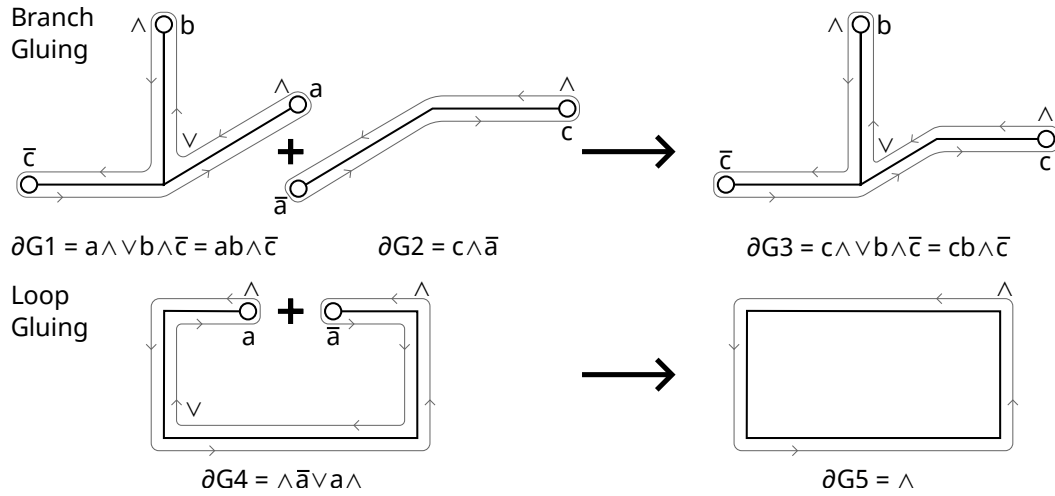


Abbildung 3.5: Branch und Loop Gluing.

3.3 Ablauf

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.

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.

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 die nachfolgenden Generationen automatisch generiert werden. Dazu werden alle Winkelgraphen der zuletzt generierten Generation enumeriert und alle durchführbaren Klebe-Operationen auf diese angewandt. Gibt es in einem der enumerierten Graphen einen noch offenen Kreis, der aber durch eine einfach Loop Gluing Operation geschlossen werden kann, so wird diese angewandt. 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.

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.

3.3.4 Ableiten der Graph-Grammatik

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 Graph-Grammatik 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 Graph-Grammatik 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.

Graph-Grammatiken

Bevor wir die Erzeugung dieser Datenstruktur genauer betrachten, soll erst einmal der Begriff der Graph-Grammatik klar definiert werden. Eine Graph-Grammatik 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. [4]

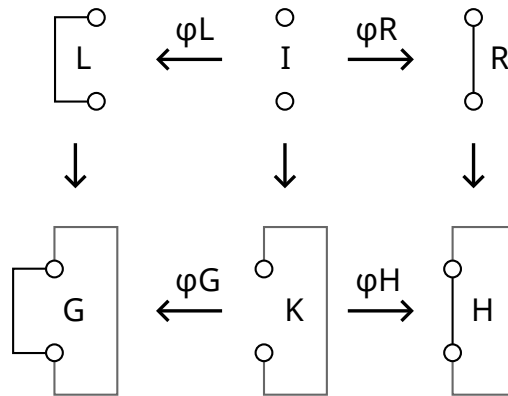


Abbildung 3.6: Double Pushout Produktionsregel.

Das Konzept der Graph-Grammatiken existiert bereits seit den frühen 70er Jahren und wurde mit dem Paper von Ehrig et al. [3] zum ersten Mal formal definiert. Seitdem haben sich sehr viele verschiedene Herangehensweisen in diesem Kontext etabliert, was zu viel Verwirrung führen kann. [6] Wir beschränken uns hier auf den ursprünglich präsentierten algebraischen bzw. Gluing-Ansatz von Ehrig et al. [3]. 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 Graph-Grammatiken genutzt werden kann. [8]

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 Graphmorphismen φ_L und φ_R untereinander abbilden. Ist L als Teilgraph in einem anderen Graphen G enthalten, 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. [3]

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 angewendet, 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* bezeichnen.

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.

4 Implementation

5 Auswertung

6 Fazit

Literaturverzeichnis

- [1] 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
- [2] 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
- [3] 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
- [4] 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
- [5] FREIKNECHT, Jonas: Procedural content generation for games. (2021). – URL <https://madoc.bib.uni-mannheim.de/59000>
- [6] 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
- [7] 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

- [8] 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
- [9] 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
- [10] 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
- [11] 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
- [12] 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

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
L ^A T _E X	Textsatz- und Layout-Werkzeug verwendet zur Erstellung dieses Dokuments

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