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

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

Tabellenverzeichnis

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 zumindest einige Monate in Anspruch. [9] 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. [9]

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. 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. [12] Ein Großteil dieser Verfahren erfordert menschliches Eingreifen in einige der Teilschritte. Es gibt allerdings Szenarien, in denen dies problematisch wird. 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. [4] 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 Produktionsregeln durch einen Menschen vorgegeben werden (bspw. [1]), bevor die automatische Generierung überhaupt beginnen kann. 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 wird der Einsatz eines solchen Verfahrens für viele Designer letztendlich nicht in Frage kommen. 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, um damit seine kreative Vision abzubilden. 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 klassischeren 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.

Es gibt bereits verschiedene Verfahren, die einen solchen Ansatz verfolgen. Diese sind u.a. der Gitter-basierte Wave Function Collapse Algorithmus von Maxim Gumin[3], die nach Symmetrien suchende inverse prozedurale Modellierung von Bokeloh et al. [2], oder das Polygon-basierte Verfahren von Paul Merrell. [14]

Im Rahmen dieser Arbeit werden jene Verfahren grob analysiert und anschließend das vielversprechendste davon praktisch umgesetzt. Das Endergebnis der Arbeit soll dann sein, dass die Funktionsweise eines ausgewählten Konzepts ausführlich und verständlich dargestellt, und nach eigener Interpretation konkret implementiert wird. Wir begrenzen uns dabei auf die Generierung von Strukturen im zweidimensionalen Raum. Die genauen Anforderungen an das darzustellende Konzept sowie an die umgesetzte Software werden in den dazugehörigen Kapiteln näher erläutert.

[3] https://github.com/mxgmn/WaveFunctionCollapse/

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. [4] Auch Musik kann durch solche Verfahren generiert werden. [15]

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. [21] 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. [19] 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. [16] Togelius et al. [21] 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. [21] 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 ersten und simpelsten Ansätze für PCG beruht auf der Generation von Pseudozufallszahlen (“pseudo random number generation (PRNG)”). [10] Das ab 1982 entwickelte und 1984 veröffentlichte Weltall-Erkundungsspiel “Elite” benutzt einen solchen Ansatz, um Eigenschaften der dort zu erkundenen Planeten automatisch zu generieren. [20] 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

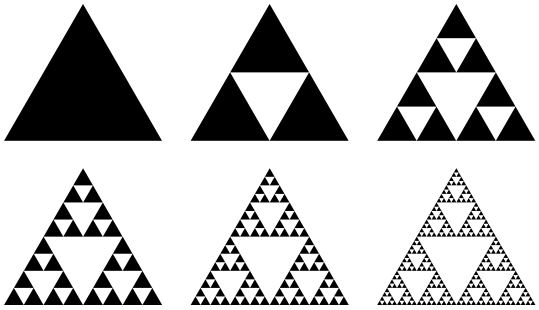

Abbildung 2.1: Konstruktion eines Sierpinski Dreiecks.

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. [20]

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. [13]

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 ei-

ne verkleinerte Kopie des gesamten Fraktals ist. [13] 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. Die ersten paar Schritte dieses Konstruktionsprozesses sind in Abbildung 2.1 zu erkennen. [18] 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. [13]

2.2.3 L-Systeme

2.2.4 Perlin Noise

2.2.5 Zelluläre Automaten

2.3 Inverse Verfahren

2.3.1 Model Synthesis

2.3.2 Nutzen von partieller Symmetrie

2.3.3 Wave Function Collapse

2.3.4 Inverses Ableiten einer Graph-Grammatik

3 Konzept

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 [14].

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 keine Halbkanten mehr besitzen. Aus diesen lässt sich eine sogennante 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.

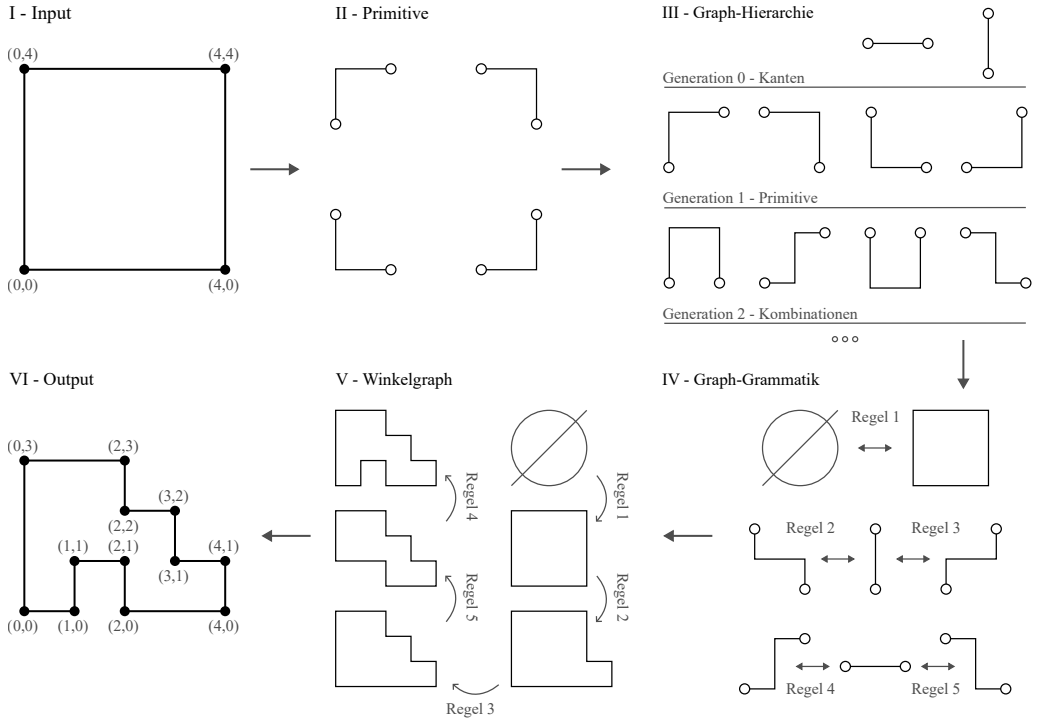

Abbildung 3.1: Überblick zum Ablauf des Verfahrens.

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.

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 Output-Strukturen sollen dabei *lokal ähnlich* zum gegebenen Input sein. 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 Konzept, das zum Verständnis beitragen kann, ist das der *r-Ähnlichkeit*, welche im Paper von Bokeloh et al. [2] 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

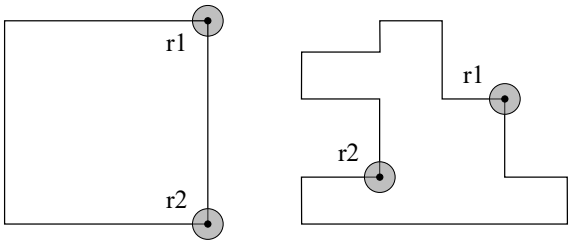

Abbildung 3.2: r-Ähnlichkeit.

Nachbarschaft egal ist. So können die betrachteten Strukturen beliebig skaliert werden und trotzdem ihre lokale Ähnlichkeit zueinander bewahren, solang 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

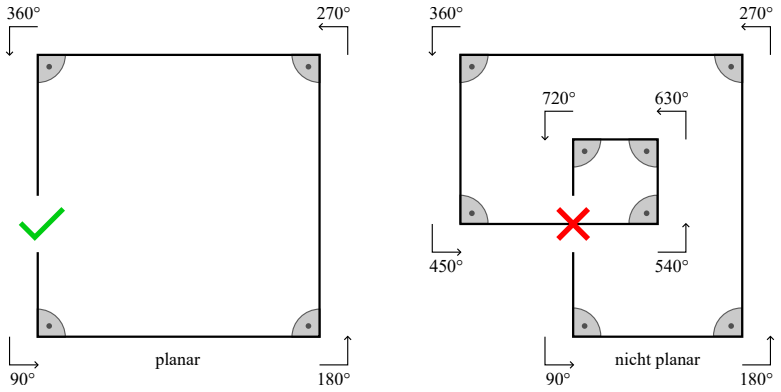

Abbildung 3.3: Der Zusammenhang zwischen Drehwinkel und Planarität.

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

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.

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, 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

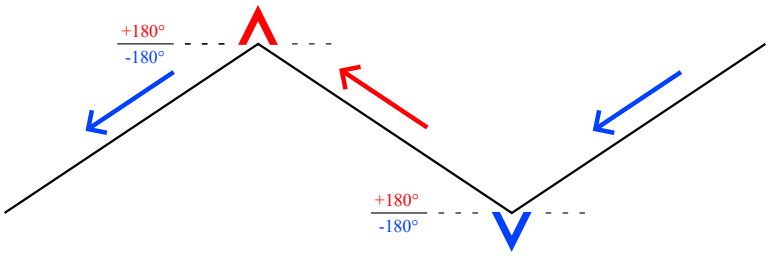

Abbildung 3.4: Positive und negative Drehungen.

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 $\overline{k}$ 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.

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.

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.

3.3 Finden der Graph-Grammatik

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

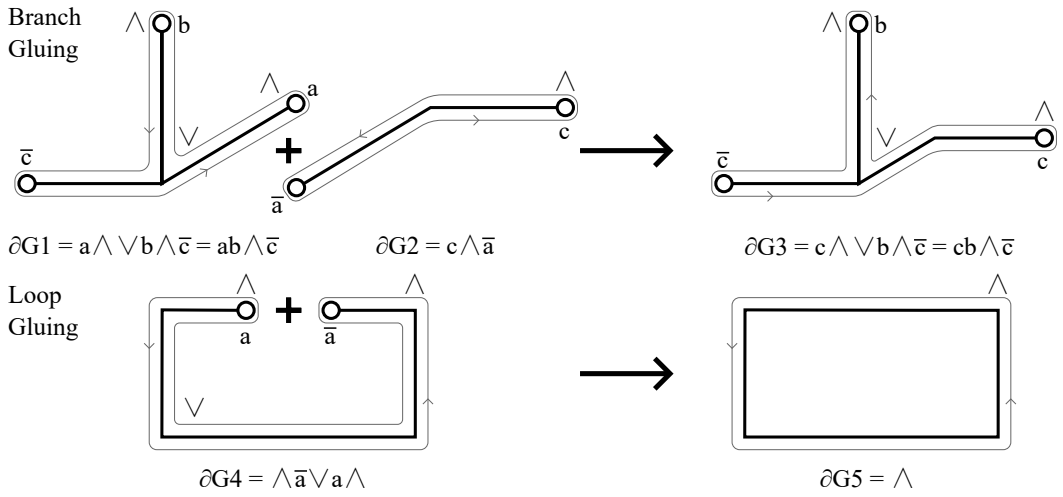

Abbildung 3.5: Branch und Loop Gluing.

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 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.

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. [7]

Das Konzept der Graph-Grammatiken existiert bereits seit den frühen 70er Jahren und wurde mit dem Paper von Ehrig et al. [6] zum ersten Mal formal definiert. Seitdem haben sich sehr viele verschiedene Herangehensweisen in diesem Kontext etabliert, was zu viel Verwirrung führen kann. [11] Wir beschränken uns hier auf den ursprünglich präsentierten algebraischen bzw. Gluing-Ansatz von Ehrig et al. [6]. 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. [14]

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.

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: $(x, y) \in E(A) \to (h(x), h(y)) \in E(B)$. [17]

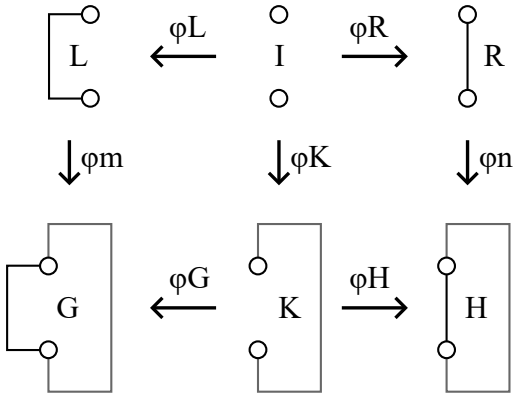

Abbildung 3.6: Double Pushout Produktionsregel.

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 $\varphi n : R \to H$. [6]

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.

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.

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 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.

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.

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.

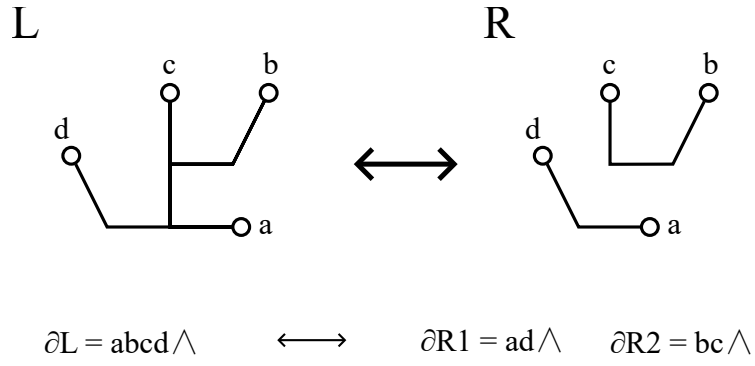

Abbildung 3.7: Ersetzen eines Graphen L mit mehreren Graphen $\{R_1, R_2\}$.

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.

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

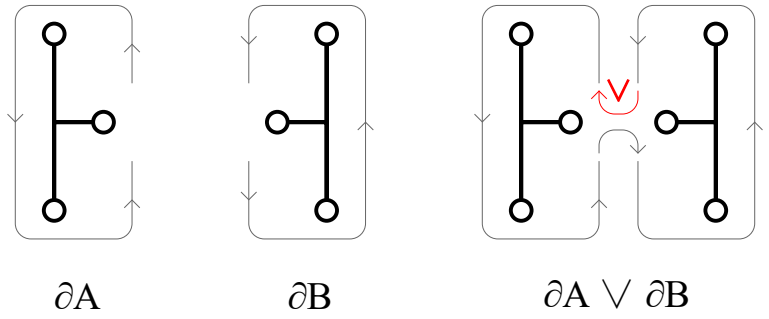

Abbildung 3.8: Kombinieren des GBS zweier Graphen mit zusätzlicher negativer Drehung.

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.

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.

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.

Eingrenzen des Hierarchie-Wachstums

Im Optimalfall lassen sich in den ersten paar Generationen der Hierarchie direkt so viele Produktionsregeln ableiten, dass die Hierarchie dadurch vollständig geleert werden kann. Meist wird dies allerdings nicht eintreffen und die Anzahl der Graphen wächst unkontrollierbar in die Höhe. Nachfolgend werden einige Optimierungen vorgestellt, um dies etwas weiter einzugrenzen. Leider gibt es selbst dann nach wie vor viele Input-Strukturen, bei denen sich das Hierarchie-Wachstum nicht ausreichend einschränken lässt. Das Verfahren wird hierdurch dennoch flexibler als auch effizienter.

Eine dieser Optimierungen beläuft sich darauf, ... TODO

3.4 Erzeugen von Variationen mithilfe der abgeleiteten Regeln

3.4.1 Ableiten eines neuen Winkelgraphen

Haben wir erfolgreich eine Graph-Grammatik abgeleitet, können wir diese nun verwenden, um daraus neue Winkelgraphen abzuleiten. 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 $m : T \to G$ finden lässt. Das Finden eines Homomorphismus zwischen beliebigen Graphen gilt als NP-schweres Problem und ist somit äußerst kompliziert. [5] Für das Finden von Homomorphismen in planaren Graphen gibt es allerdings effiziente Algorithmen, die in linearer Zeit zu einem Ergebnis kommen. [8] 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 Output-Struktur 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. [3] und stellen den Graphen als lineares Gleichungssystem dar, für welches wir anschließend eine Lösung finden. Die einzelnen Gleichungen dieses Systems lassen sich aus den Kanten des Winkelgraphen ableiten. Die Kanten lassen sich beschreiben durch die Position des Startknotens v_0 und des Endknotens v_1. Um zu garantieren, dass dabei der Tangentenwinkel der Kante mit dem entsprechenden Winkel θ im Kantenlabel übereinstimmt, fügen wir ebenfalls einen Richtungsvektor u in die Gleichung mit ein. Dieser kann wie folgt berechnet werden: $u = [cos(\theta), sin(\theta)]$. Hierbei handelt es sich 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 lautet dann: $v_0 + su = v_1$ bzw. $v_0 + su - v_1 = 0$. All diese Kantengleichungen können in eine Matrix-Gleichung der Form $Ax = 0$ gebracht werden, wobei x alle Knotenpositionen und Kantenlängen und A alle Koeffizienten dieser Unbekannten enthält. Durch das Berechnen des Kerns der Matrix A lassen sich anschließend mögliche Lösungen für das Gleichungssystem ableiten. Das erstellte Gleichungssystem wird hierbei stets unterbestimmt sein, da es mehr Unbekannte als Gleichungen gibt. Daraus ergibt sich wiederum, dass es immer entweder keine oder unendlich viele Lösungen geben wird. Wird eine Lösung gefunden, so enthält diese einige freie Variablen. Somit finden wir statt einer konkreten Lösung immer nur einen Raum an möglichen Lösungen. Diesen Raum nennen wir Basis des Kerns der Matrix. Durch Festlegen von beliebigen Werten für die freien Variablen können wir daraus nun konkrete Lösungen für das Gleichungssystem entnehmen.

Leider bedeutet das Finden einer Lösung für das Gleichungssystem nicht unbedingt, dass wir nun auch eine valide Darstellung für die Output-Struktur 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.

4 Implementierung

4.1 Anforderungen an die Software

4.2 Architektur

4.3 Verwendete Technologien und Bibliotheken

4.4 Datenstrukturen

4.5 Algorithmen

5 Auswertung

5.1 Überprüfen der Anforderungen

5.2 Erreichen des Forschungsziels

5.3 Probleme und Erweiterungsmöglichkeiten

6 Fazit

Literaturverzeichnis

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

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

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

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

[10] 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

[11] 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

[12] 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

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

[14] 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

[15] 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

[16] 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

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

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

[19] 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

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

[21] 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
LaTeX	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