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

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. [2] 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. 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. [4] 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 Gumin[3], die nach Symmetrien suchende inverse prozedurale Modellierung von Bokeloh et al. [1], oder das Polygon-basierte Verfahren von Paul Merrell. [6]

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

In diesem Kapitel werden einige grundlegende Konzepte behandelt, welche zum Verständnis dieser Arbeit beitragen. Zunächst wird erklärt, was genau unter dem Begriff der prozeduralen Generierung zu verstehen ist, woraufhin einige der fundamentalen Verfahren vorgestellt werden.

2.1 Prozedurale Generierung

Prozedurale Generierung, oft auch Prozedurale Content Generierung (PCG), beschreibt eine Menge von Verfahren zum algorithmischen Erstellen von Inhalten ("Content"). Dabei handelt es sich meist um Verfahren, die automatisch Texturen oder verschiedene Gebilde im Kontext von Videospielen erzeugen können, so z.B. Landschaften, Flüsse, Straßennetze, Städte oder Höhlenstrukturen. Auch Musik kann durch solche Verfahren generiert werden, was für diese Arbeit allerdings weniger relevant ist. [9]

TODO: - Eingehen auf Zufälligkeit & deterministisches Verhalten (Reproduzierbarkeit durch Seed) -> Quelle 9 - Eingehen auf Unterscheidung zwischen assisted/non-assisted -> Quelle 14

- Auf Begriffe Prozedural, Content, und Generierung einzeln eingehen ?

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 veschiedenen 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. 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. Mit diesem Problem haben sich Togelius et al. [9]

bereits ausführlich befasst, weshalb dies hier nicht weiter thematisiert werden soll. Für diese Arbeit soll die oben genannte Definition ausreichen.

2.2 Verwendung von PCG

Da die Entwicklung von Videospielen aufgrund der großen Anzahl an benötigten Inhalten sehr schnell sehr aufwändig werden kann, findet PCG vor allem in dieser Industrie einen großen Nutzen. Gerade das Erstellen von immersiven Welten erfordert eine Vielzahl von verschiedensten detaillierten Modellen und kann manuell nur mit sehr großem Arbeitsaufwand umgesetzt werden. Das Automatisieren der Generierung von Inhalten kann den Entwicklerstudios hier eine bedeutende Menge an Zeit und Kosten sparen, die dann an anderen Stellen eingesetzt werden können. In vielen Fällen kann sogar Speicherplatz gespart werden, indem die Generierung der Inhalte zur Laufzeit stattfindet.

PCG hat bereits in vielen bekannten Videospielen Verwendung gefunden. Schon im Jahr 1980 wurde

TODO: Aufzählen von Spielen mit PCG Algorithmen

2.3 Perlin Noise

Ein fundamentales Konzept im Bereich von PCG ist das Verwenden von Rauschfunktionen, oder auch *Noise*. Der wohl bekannteste Vertreter dieses Konzepts ist das 1985 von Ken Perlin entwickelte [7] und 2002 verbesserte [8] *Perlin Noise*, welches seit dessen Veröffentlichung nicht mehr aus der Welt der Computergrafik wegzudenken ist. Mithilfe von Perlin Noise können eine Reihe von Zufallswerten erzeugt werden. Hierbei sind sich nah beieinander liegende Werte stets sehr ähnlich und es gibt keine starken Ausschläge, weshalb der entstehende Verlauf sehr organisch wirkt. Aufgrund von dieser Eigenschaft eignet sich Perlin Noise perfekt zum Erzeugen von natürlichen Strukturen wie z.B. Hügellandschaften oder Inselgruppen. Generell ist der erzeugte Kurvenverlauf vielseitig einsetzbar und findet somit in vielen verschiedenen Anwendungen einen Nutzen, darunter auch im Bereich der Animation. [3]

TODO: Einfügen von Vergleich zwischen Random Noise und Perlin Noise

Neben der vielseitigen Einsetzbarkeit von Perlin Noise gibt es außerdem den Vorteil, dass dieses Verfahren sehr günstig sowohl in Bezug auf die Berechnungszeit als auch in Bezug auf die Speicherverwendung ist. Einzelne Punkte im Verlauf lassen sich unabhängig voneinander berechnen, wodurch sich der Berechnungsprozess wunderbar parallelisieren lässt. Dies wird mit der immer weiter voranschreitenden Entwicklung von Grafikkarten und Prozessoren auch zu einem immer größeren Vorteil. [3]

Perlin Noise kann für eine beliebige Anzahl an Dimensionen berechnet werden. Dazu wird der n-dimensionale Raum in eine reguläre gitterartige Struktur aufgespalten. Die Punkte auf dem Gitter sind dabei all jene, die an ausschließlich ganzzahligen Koordinaten liegen. Im zweidimensionalen Raum wäre dies also die Menge an Punkten $\{(x, y) \parallel x, y \in \mathbb{N}\}$. Alle anderen Punkte im Raum befinden sich dann jeweils innerhalb einer der von den Gitterpunkten aufgespannten Zellen.

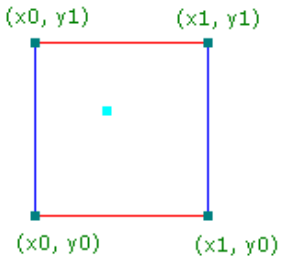

Abbildung 2.1: Beispiel einer Gitterzelle im 2D

Jeder der Gitterpunkte bekommt außerdem einen pseudozufälligen Gradienten (Richtungsvektor der Länge 1) zugeordnet.

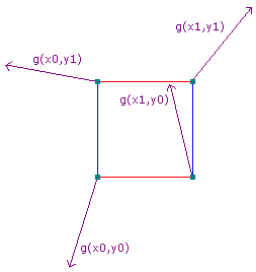

Abbildung 2.2: Pseudozufällige Gradienten für die Eckpunkte einer Zelle im 2D

Soll jetzt der Noise-Wert für einen Punkt im Raum berechnet werden, werden zunächst die Eckpunkte der betroffenen Zelle und deren zugeordnete Gradienten ermittelt. Außerdem werden die Vektoren berechnet, die von den Eckpunkten in Richtung des aktuellen Punktes zeigen.

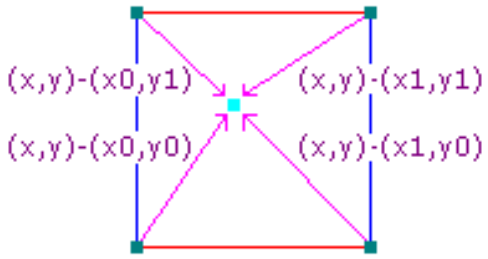

Abbildung 2.3: Vektoren von den Eckpunkten zu einem Punkt im Inneren einer Zelle im 2D

Anschließend wird dann für jeden Eckpunkt das Skalarprodukt aus dem dortigen Gradienten und dem Vektor in Richtung des Punktes gebildet. Der Mittelwert all dieser Skalarprodukte ergibt dann den finalen Noise-Wert. [7][1]

TODO: Erklärung mathematisch beschreiben, Abbildungen erneuern/in eine Abbildung zusammenfassen

2.4 L-Systeme

Ein weiteres bekanntes Konzept ist das der L-Systeme.

2.5 Fraktale

[5]

example-based model synthesis -> (weitere Merrell Verfahren) -> wave function collapse -> example-based procedural modeling using graph grammars

[1] Erklärung in Anlehnung an https://mzucker.github.io/html/perlin-noise-math-faq.html

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 [6].

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. Diese Inputstruktur wird anschließend umgewandelt in einen Graphen, in welchem die Punkte des Polygons als Knoten und die Verbindung zwischen den Punkten als Kanten dargestellt werden. Die Darstellung als Graph ist nützlich, da in dieser 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 komplizierter werdenden Graphen führt. Beim Aufbau der Hierarchie werden die neu entstehenden Graphen auf bestimmte Eigenschaften überprüft, die es uns erlauben, daraus Regeln für ein Graphersetzungssystem abzuleiten. Das einfachste Beispiel hierfür sind vollständige Graphen, also Graphen, die nur noch aus in sich geschlossenen Kreisen bestehen und keine Halbkanten mehr besitzen. Aus diesen lässt sich eine sogenannte Startregel ableiten, welche den leeren Graphen mit dem gefundenen

vollständigen Graphen ersetzt. Das Finden von weiteren Regeln ist deutlich komplizierter und wird später im Detail erläutert.

Sobald man nun eine Menge von Regeln für das Graphersetzungssystem gefunden hat, kann man diese verwenden um verschiedenste zum Inputgraphen ähnliche Graphen abzuleiten, indem zufällig verschiedene Regeln nach und nach angewendet werden. Für einen solchen Graphen müssen dann noch konkrete Knotenpositionen und Kantenlängen bestimmt werden, sodass dieser wieder als Struktur aus Polygonen dargestellt werden kann. Hier findet das Verfahren schließlich auch sein Ende.

3.2 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 Punkten und Kanten beschrieben werden kann. So sind z.B. Kreise oder andere Strukturen mit Rundungen kein valider Input und können wenn dann nur durch komplexe Polygone angenähert werden.

Zur Verarbeitung des Inputs wird dieser in einen Graphen 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. Außerdem können die einzelnen Polygone mit Farben versehen werden, um verschiedene abgegrenzte Bereiche zu markieren. Die Kanten im Graphen werden mit einem entsprechenden Label versehen, welches neben den Start- und Endknoten ebenfalls Informationen zum Tangentenwinkel, sowie zu den Farben der links und rechts anliegenden Polygone enthält. Ein Kantenlabel besitzt die Form $\tilde{a} = (l, r, \theta)$, wobei $\tilde{a}$ die Bezeichnung der Kante, l und r die Farben der anliegenden Polygone, und θ der Tangentenwinkel der Kante sind.

3.3 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 zwingend 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.

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

[3] Lagae, A. ; Lefebvre, S. ; Cook, R. ; DeRose, T. ; Drettakis, G. ; Ebert, D.S. ; Lewis, J.P. ; Perlin, K. ; Zwicker, M.: A Survey of Procedural Noise Functions. In: *Computer Graphics Forum* 29 (2010), Nr. 8, S. 2579–2600

[4] 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

[5] Mandelbrot, Benoit B. ; Frame, Michael: Fractals. In: *Encyclopedia of physical science and technology* 5 (1987), S. 579–593

[6] 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

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

[8] Perlin, Ken: Improving noise. In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : Association

for Computing Machinery, 2002 (SIGGRAPH '02), S. 681–682. – URL https://doi.org/10.1145/566570.566636. – ISBN 1581135211

[9] 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