HAW HAMBURG

BACHELOR THESIS
Van Jonas Scharenberg

Prozedurale Generierung von Landkarten mittels des Wave Function Collapse Algorithmus

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

Van Jonas Scharenberg

Prozedurale Generierung von Landkarten mittels des Wave Function Collapse Algorithmus

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. Birgit Wendholt

Eingereicht am: 12. Juli 2022

Van Jonas Scharenberg

Thema der Arbeit

Prozedurale Generierung von Landkarten mittels des Wave Function Collapse Algorithmus

Stichworte

Prozedurale Generierung, Computergrafik, Wave Function Collapse, Textursynthese, Landkartengenerierung

Kurzzusammenfassung

Diese Arbeit behandelt den Wave Function Collapse Algorithmus von Maxim Gumin und dessen Einordnung in das Themenfeld der Prozeduralen Generierung von digitalen Inhalten. Das Ziel ist die Entwicklung eines Landkartengenerators, mit dem fiktive Landkarten auf Knopfdruck von Nutzerinnen und Nutzern erstellt werden können. Dafür werden etablierte Techniken der Prozeduralen Generierung dargelegt. Das logische Konzept und die Evaluation des Prototypen verdeutlichen den Erfolg mit geeigneten Inputbildern sowie das Erweiterungspotential dieses Ansatzes.

Van Jonas Scharenberg

Title of Thesis

Procedural generation of maps with Wave Function Collapse Algorithm

Keywords

Procedural Content Generation, Computer Graphics, Wave Function Collapse, Texture Synthesis, Map Generation

Abstract

In this thesis, the Wave Function Collapse Algorithm by Maxim Gumin will be addressed and classified into the field of Procedural Content Generation. The goal is the development of a map generator, which will enable users to generate fictional maps with one click. For this purpose, established techniques of Procedural Conten Generation will be

discussed. The logical concept and evaluation of the prototype illustrate the success of this approach as well as its potential of expandability.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Mit der schnell fortschreitenden Evolution der Computerspiele und dem Einzug der Digitalisierung in den Alltag besteht immer mehr Bedarf an komplexen digitalen Inhalten. Das wird auch bedingt durch die immer größeren Rechenkapazitäten von Hardware und leistungsfähigerer Software und führt zu immer mehr Zeit, die Entwickler für die Erstellung von digitalen Assets aufwenden müssen[24]. Besonders der hohe Detailgrad von grafischen Assets sowie immer komplexere Strukturen von Videospielen sorgen dafür, dass gerade in den vergangenen Jahren Techniken zur Generierung solcher Assets immer mehr an Relevanz erlangt haben.

Ein Lösungsansatz für dieses Problem ist die Prozedurale Content Generierung (im Folgenden als PCG abgekürzt). Mithilfe von PCG ist es möglich, Videospielinhalte prozedural generieren zu lassen, anstatt sie per Hand erstellen zu müssen. Nicht nur Texturen lassen sich so erstellen, ganze Landkarten, Level oder Dungeons können so generiert werden. So bietet PCG das Potential, nicht nur viel Zeit und Aufwand bei der Produktion digitaler Inhalte zu sparen, sondern auch die Größe von Applikationen drastisch zu reduzieren.

Ein Algorithmus innerhalb des PCG, der seit seiner Entwicklung im Jahr 2016 von Maxim Gumin großen Anklang in der Spieleentwicklung gefunden hat, ist der Wave Function Collapse[11] (im Folgenden auch WFC). Der WFC gehört zum Themenbereich der Textursynthese und wurde ursprünglich für zweidimensionale Texturen konzipiert. So ist der Einsatz des Algorithmus möglich, um grafische Assets zur Laufzeit zu generieren. Große Texturen, wie Landkarten, müssen also nicht gespeichert werden, sondern können ganz nach aktuellem Bedarf erstellt werden.

1.2 Ziele

Als Teil dieser Arbeit soll ein Prototyp für einen Landkartengenerator enwickelt werden. Dies soll mithilfe des Wave Function Collapse Algorithmus realisiert werden. Für diesen Zweck muss der Wave Function Collapse so modifiziert werden, dass sich ein Nutzer mit den gewünschten Inputbildern eine Fantasy-Landkarte auf Knopfdruck prozedural generieren lassen kann.

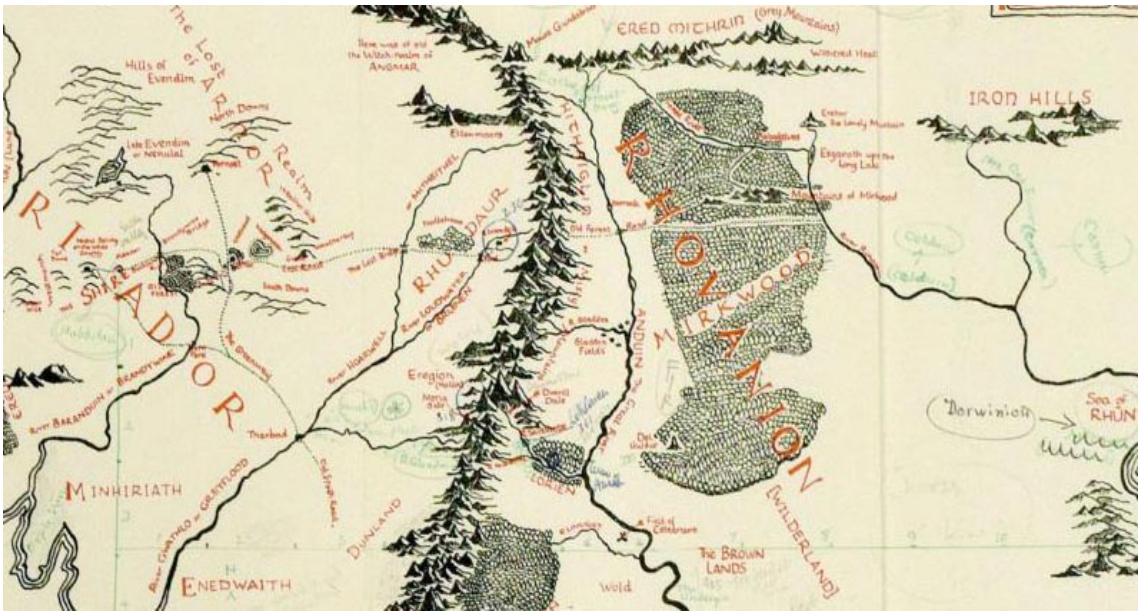


Abbildung 1.1: Fantasy-Karte von Mittelerde aus *Der Herr der Ringe*[6]

Die auf Abb. 1.1 gezeigte Fantasy-Landkarte stammt aus dem Kontext der *Herr der Ringe*-Reihe und dient als Inspiration für diese Arbeit. Es sollen also mithilfe des WFC Algorithmus Landschaften generiert werden, die voneinander abgegrenzt werden können. Dem Nutzer soll es dabei möglich sein, Landkarten verschiedener Größe zu generieren und die Häufigkeit verschiedener Elemente zu beeinflussen, um so Karten mit verschiedenen Eigenschaften zu erstellen bzw. Prioritäten nach Bedarf zu setzen.

1.3 Struktur der Arbeit

Diese Arbeit setzt sich aus folgenden Bestandteilen zusammen:

1. Einleitung
2. Grundlagen und Methoden der Prozeduralen Content Generierung
    - Eine Auswahl an wichtigen Techniken und Methoden innerhalb der PCG wird vorgestellt und näher beleuchtet. Außerdem wird der WFC in das Feld der PCG eingeordnet.
3. Konzept
    - Das logische Konzept des in dieser Arbeit entwickelten Prototypen wird präsentiert.
4. Umsetzung
    - Details zur Implementierung sowie des Software Engineering-Prozesses werden vorgestellt.
5. Evaluation
    - Der Prozess der Entwicklung sowie der entstandene Prototyp werden bezüglich der Ziele und Anforderungen evaluiert.
6. Schluss

2 Grundlagen und Methoden der Prozeduralen Content Generierung (PCG)

Das folgende Kapitel beinhaltet relevante theoretische Konzepte und Grundlagen, die die Basis für diese wissenschaftliche Arbeit darstellen. Somit werden neben dem Wave Function Collapse auch alternative Lösungsansätze für die in Kapitel 1 beschriebene Problemstellung angerissen. Die ausgewählten Themenbereiche werden grundlegend beschrieben und hinsichtlich des aktuellen Stands der Technik näher beleuchtet. Darüber hinaus werden wichtige Anwendungsbereiche für PCG inklusive wegbereitender Beispiele genannt.

2.1 Definition

Prozedurale Content Generierung, im Folgenden mit PCG abgekürzt, ist ein Ansatz zur automatischen Erstellung von Inhalten in der Informatik sowie weiteren Naturwissenschaften. R. Smelik et al.[25] definieren PCG als jegliche Form von automatisch generierten Assets, die auf einem limitierten Set von user-definierten Inputparametern basieren[25]. Ein kleines Set an Inputparametern wird folglich mithilfe bestimmter Algorithmen, auch *amplification algorithms* genannt, in ein großes Set an Outputdaten transformiert[23]. Gerade im Bereich der Videospielproduktion wird PCG oft verwendet. So definieren Togelius et al. PCG als den Prozess algorithmischer Kreierung von Videospielinhalten mit limitiertem oder indirektem User-Input[28].

In diesem Kontext ist es außerdem notwendig, auch das Wort Content zu definieren. Shaker et al.[24] definieren Content als (fast) alles, was in einem Videospiel enthalten ist: Level, Landkarten, Regeln, Texturen, Story, Gegenstände, Quests, Musik, Waffen,

Fahrzeuge, Charaktere etc..[24] Etwas allgemeiner halten es Freiknecht et al. mit ihrer Definition von Content als digitales Asset für Videospiele, Simulationen oder Filme[10].

Zusammenfassend ist PCG also

> *ein Ansatz zur automatischen Erstellung von digitalen Inhalten für Spiele, Simulationen oder Filme, der auf vordefinierten Algorithmen und Patterns basiert und dabei einen minimalen Userinput benötigt[9].*

2.2 Klassifizierung von PCG

Das große Feld der PCG umfasst nicht nur zahlreiche Techniken und Methoden, mit denen Content prozedural generiert werden kann, ganz allgemein lassen sich nach Shaker et al. außerdem mehrere essentielle Ansätze unterscheiden[24]:

- Online vs. Offline: Während bei der Online PCG Content zur Laufzeit erstellt wird, bezieht sich Offline PCG auf die Erstellung von Content während der Entwicklung eines Videospiels.
- Notwendig vs. Optional: Notwendiger generierter Content ist existenziell für die Nutzung der Applikation bzw des Spiels. Optionaler Content kann nach Belieben ausgetauscht oder verworfen werden.
- Grad und Dimension der Kontrolle: Es ist möglich, die Ergebnisse von PCG auf verschiedene Arten zu beeinflussen und so zu kontrollieren. Die Verwendung eines Seeds erlaubt beispielsweise, ein prozedural erstelltes Asset genau so wieder zu erstellen.
- Generisch vs. Adaptiv: Der generische Ansatz ist dem Namen nach ein übergreifender Ansatz, der den User und seine Nutzung des Contents nicht berücksichtigt. Beim adaptiven Ansatz wird die Contentgenerierung auf den Nutzer angepasst.
- Stochastisch vs. Deterministisch: Das exakte Ergebnis von stochastischem PCG kann nicht vorhergesagt werden, während deterministischer PCG mit den gleichen Inputparametern immer zum selben Ergebnis kommt.
- Constructive vs. Generate-and-test: Der konstruktive Ansatz erstellt den Content ganzheitlich, während beim Generate-and-test Ansatz einzelne Teile aufeinander aufbauend generiert und getestet werden.

- Automatic generation vs. Mixed authorship: Bei der automatischen Generierung wird nur limitierter Input erlaubt bzw nur das Verändern von Parametern durch die Entwickler. Mixed authorship dagegen erlaubt den Input von Usern oder anderen Designern.

2.3 Anwendungsbereiche von PCG

Prozedurale Content Generierung findet nicht nur in der Informatik bzw. der Computergrafik Anwendung, auch in weiteren wissenschaftlichen Disziplinen wie bspw. der Biologie oder Kulturwissenschaft wird PCG verwendet. PCG zeichnet sich hier durch eine enorm variable Einsetzbarkeit aus und kann somit an viele konkrete Problemstellungen angepasst werden. So lässt sich mithilfe von PCG eine Simulation bspw. von Populationen genau so erstellen, wie ein Modell in der urbanen Stadtplanung.

Abbildung 2.1: Beispiel eines prozedural generierten Planeten in No Man's Sky[2]

Neben der Anwendung im wissenschaftlichen Feld bzw. in der Forschung, wird auch im Entertainment-Sektor immer größerer Wert auf die Verwendung von PCG gelegt. Spieleklassiker wie Minecraft und Far Cry setzen PCG-basierte Methoden für die Generierung der Spielwelten und Dungeons ein, aber auch Gegnerdesign, die Stärke und Spawnhäufigkeit von Gegnern, die Vegetation in bestimmten Biomen und viele weitere Parameter

lassen sich mithilfe von PCG-Algorithmen und -tools generieren[14]. In dem im Jahr 2016 erschienenen Action-Adventure-/ und Survival-Spiel No Man's Sky werden neben Items, Raumschiffen und Waffen sogar ganze Planeten inklusive deren Biome prozedural generiert, siehe Abbildung 2.1. Im Kontext der Historie von PCG ist außerdem das Videospiel *Rogue* zu nennen[8]. Das um das Jahr 1980 entwickelte Spiel gilt eines der einflussreichsten, PCG verwendenden Spiele, da nicht nur Dungeons, sondern ganze Spielmechaniken mithilfe von PCG entwickelt wurden, was in einem ganz neuen Genre, heutzutage bekannt als *Rogue-Like*, resultierte[29]. Aufgrund der Vielfalt der zu verwendenden Items, Waffen sowie Gegnertypen war Rogue zu dem Zeitpunkt innovativ und unterschied sich bezüglich des Ansatz von allen anderen Videospielen[8].

Ein weiteres Einsatzgebiet ist die Filmindustrie. Disney Research, ein Netzwerk von Forschungslaboren zur Unterstützung der The Walt Disney Company, hat bereits mehrere Publikationen zum Einsatz von PCG in der Filmproduktion herausgegeben, z.B. zur virtuellen Bearbeitung von Terrain.[3]

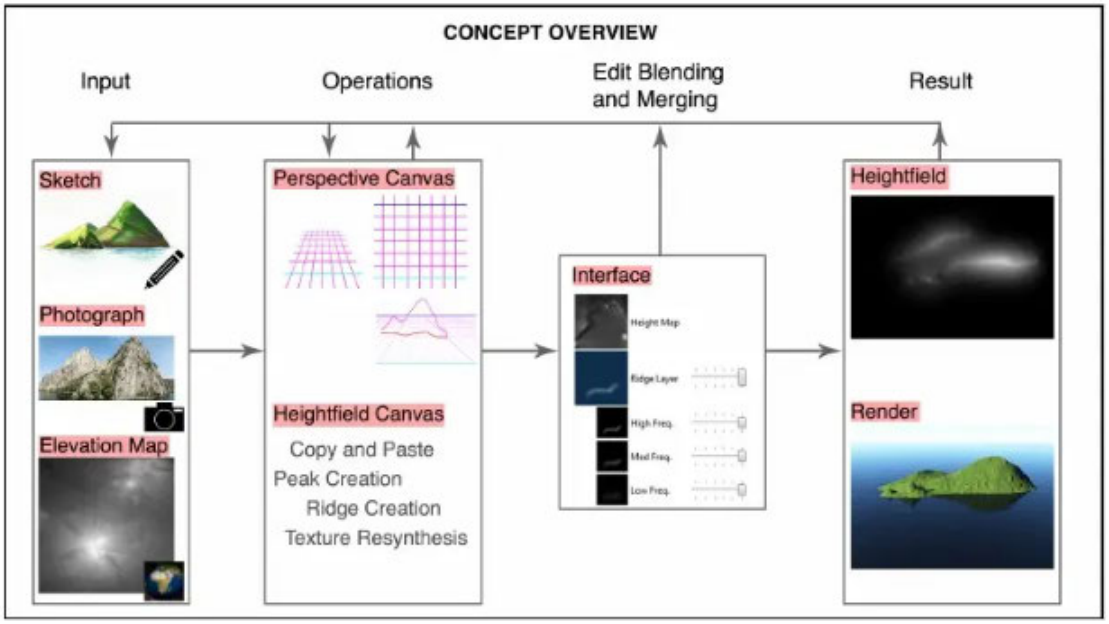


Abbildung 2.2: Workflow für die virtuelle Bearbeitung von Terrain[3]

In Abb. 2.2 wird verdeutlicht, wie der Workflow bei der Bearbeitung von Terrain aussieht. So können nicht nur Fotos als Input verwendet werden, sondern auch Zeichnungen oder Höhenkarten.

2.4 Generelle Techniken und Methoden

2.4.1 L-Systeme

L-Systeme, benannt nach dem theoretischen Biologen Aristid Lindenmayer, wurden 1968 von Lindenmayer als formale Sprache entwickelt, die als Grundlage einer axiomatischen Theorie biologischer Entwicklung diente. Auch Lindenmayer selbst veröffentlichte im Jahr 1990 gemeinsam mit Przemyslaw Prusinkiewicz Ansätze zur Verwendung von L-Systemen zur Modellierung von grafischen Objekten, speziell Pflanzen[19]. In der Computergrafik wurden die Vorteile dieser formalen Sprache schnell erkannt, sodass diese dank der authentischen Ergebnisse große Anwendung findet.

Konzeptionell sind L-Systeme Ersetzungssysteme, bei denen simple Symbole oder Formen anhand von vordefinierten Regeln durch komplexere Strukturen ersetzt werden. Im Gegensatz zu Chomsky-Grammatiken, bei denen das Ersetzungsverfahren sequentiell abläuft, werden die Regeln bei L-Systemen parallel auf alle Symbole angewendet. Ein klassisches Beispiel für L-Systeme ist die Snowflake Curve, die in Abb 2.3 zu sehen ist. Die Basis bilden die Formen *initiator* und *generator*. In jedem Durchlauf wird jede

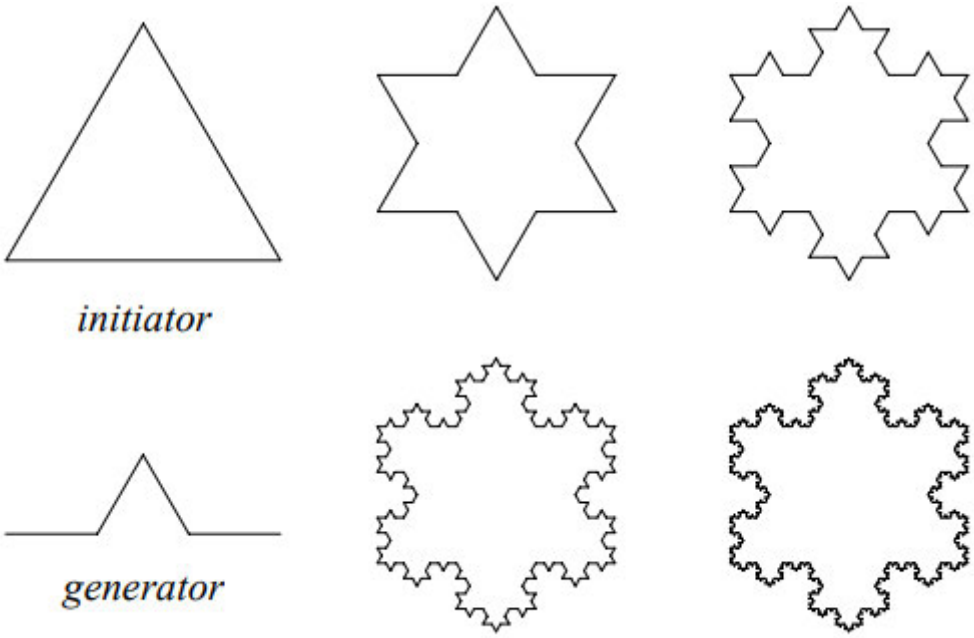


Abbildung 2.3: Snowflake Beispiel[19]

durchgehende Linie mit einer Kopie der generator-Form ersetzt, sodass bereits nach vier Generationen aus der initiator-Form eine detaillierte Schneeflocke zu erkennen ist.

Da es verschiedene Varianten an L-Systemen gibt, unter anderem kontextsensitive, deterministische sowie nicht-deterministische wie bspw. stochastische und randomisierte L-Systeme, die jedoch den Rahmen dieser Arbeit sprengen würden, wird im Folgenden ausschließlich auf die grundlegende Variante, das kontextfreie 0L-System sowie die graphische Interpretation von L-Systemen nach Lindenmayer und Prusinkiewicz eingegangen[19].

Das kontextfreie 0L-System G lässt sich formal als 3-Tupel definieren:

$$G = \{\mathcal{V}, w, \mathcal{P}\}$$

- $\mathcal{V}$: Endliches Alphabet
- w: Ein nicht-leeres Startwort, auch Axiom genannt. Es gilt: $w \in \mathcal{V}^+$
- $\mathcal{P}$: Die (endliche) Menge an Produktions-/ bzw. Transformationsregeln mit $\mathcal{P} \subset \mathcal{V} \times \mathcal{V}^*$

Für die Computergrafik werden die L-Systeme bedeutsam, wenn die Symbole des Alphabets für grafische Formen und Operationen stehen. Lindenmayer und Prusinkiewicz beschreiben den Ansatz der *turtle interpretation of strings*. Die *turtle interpretation of strings* geht von einem Zustand der Schildkröte aus, welcher als 3-Tupel x, y, α definiert ist, wobei x und y die Position der Schildkröte im karthesischen Koordinatensystem repräsentieren und der Winkel α die aktuelle Ausrichtung bzw. den Blickwinkel der Schildkröte. Mithilfe folgender Symbole lassen sich einfach grafische Strukturen erstellen:

- F: Gehe einen Schritt der Länge d vorwärts und ziehe dabei eine Linie
- f: Gehe einen Schritt der Länge d vorwärts, ohne eine Linie zu ziehen
- +: Rotiere nach links um den Winkel δ. Der neue Zustand ist $(x, y, \alpha + \delta)$
- -: Rotiere nach rechts um den Winkel δ. Der neue Zustand ist $(x, y, \alpha - \delta)$

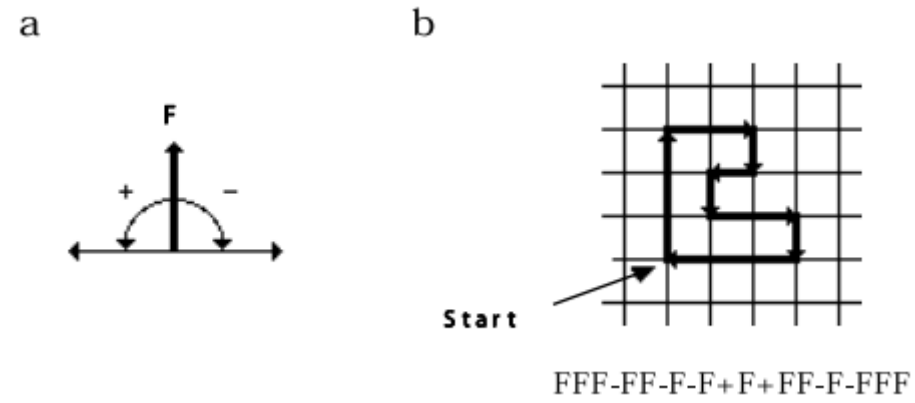
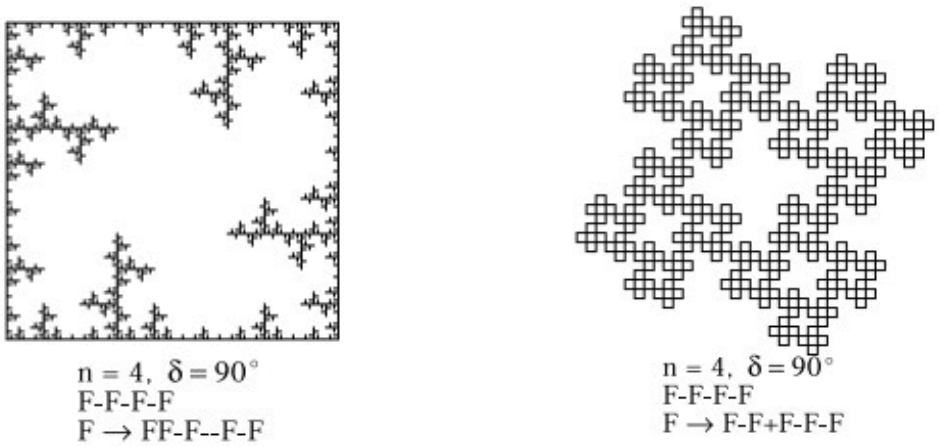


Abbildung 2.4: (a) Turtle Interpretation der Symbole F, + und -. (b) Interpretation einer Zeichenkette. Der Winkel δ ist in diesem Beispiel 90°[19]

Weitere Varianten der L-Systeme verwenden zusätzliche Parameter, Symbole sowie einen Stack, also einen Zwischenspeicher für Zustände, wodurch sich wesentlich komplexere Strukturen generieren lassen, siehe Abb. 2.5.



(a) Beispiel 1 (b) Beispiel 2

Abbildung 2.5: Einfache Beispiele mit veränderten Parametern (insbesondere der Regel) für die Verwendung von L-Systemen sowie der Turtle Interpretation [19]

2.4.2 Shape Grammars

Shape Grammars wurden erstmals in einer Arbeit von George Stiny und James Gips von 1972 erwähnt und beschrieben[26]. Stiny konkretisiert Shape Grammars als regelbasiertes System zur Generierung von zwei-/ oder dreidimensionalen Grafiken.

Die Basis stellen dabei die Regeln dar, mithilfe derer Formen durch andere Formen ersetzt werden. Die Regeln bestimmen, in welcher Größe sowie mit welchem Aussehen die Form gezeichnet wird. Konzeptionell orientiert sich Stiny an den in Kapitel 2.2.1 beschriebenen L-Systeme, weshalb Shape Grammars als eine erweiterte Form der L-Systeme angesehen werden können. Eine Shape Grammar setzt sich aus folgenden Regeln zusammen. Zu-

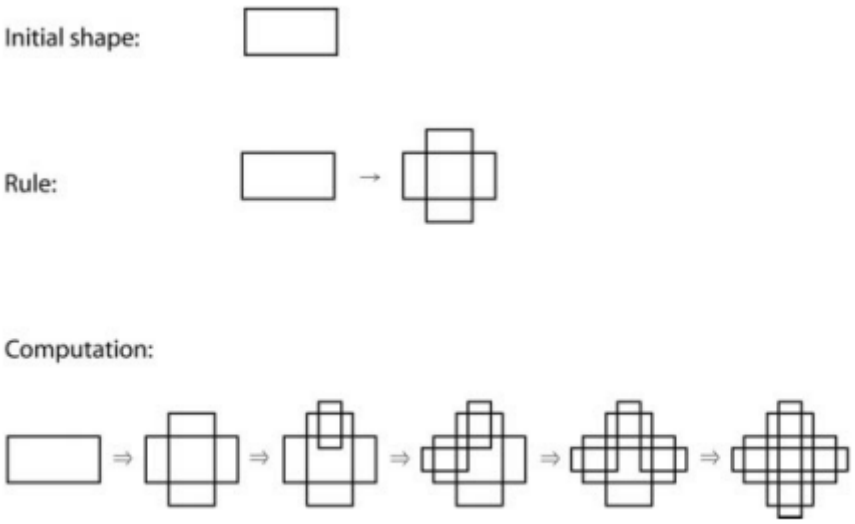


Abbildung 2.6: Shape Grammar - Beispiel für die iterative Anwendung von Regeln auf eine initiale Form[27]

nächst werden Terminalsymbole definiert. Diese werden durch Formen verkörpert, die nicht in weitere Formen transformiert werden, sondern ein Ende der Transformationskette darstellen. Anschließend werden die Terminalsymbole durch eine Menge an Regeln erweitert. Diese beschreiben eindeutig, welche Form sich aus einer Form ableiten lässt.

Formal lässt sich eine Shape Grammar (SG) laut Stiny als Quadrupel definieren:

$$SG = \{\mathcal{V}_T, \mathcal{V}_M\, \mathcal{R}, I\}$$

- $\mathcal{V}_T$: Eine endliche Menge an Formen.
- $\mathcal{V}_{T*}$: Eine endliche Menge an Formen, die aus der Kombination von Elementen aus $\mathcal{V}_T$, inklusive verschiedener Ausrichtung sowie Größe, entstehen können.
- $\mathcal{V}_M$: Eine endliche Menge an Formen, die in andere Formen transformiert werden können. Es gilt: $\mathcal{V}_{T*} \cap \mathcal{V}_M = \emptyset$.
- $\mathcal{R}$: Eine endliche Menge an geordneten Tupeln (u, v), welche Regeln für die Transformation von Formen darstellen. Sie werden folgendermaßen notiert $u \rightarrow v$, wobei u mit v ersetzt wird.
- I : Der initiale Zustand, bestehend aus Elementen aus $\mathcal{V}_M$ und $\mathcal{V}_{T*}$.

Sind alle Formen, Regeln, Terminierungssymbole sowie der Initialzustand definiert, kann die Generierung gestartet werden. Ausgehend vom initialen Zustand I werden so lange Regeln aus $\mathcal{R}$ rekursiv angewendet, bis die Ergebnisform ausschließlich aus Terminalsymbolen aus $\mathcal{V}_{T*}$ besteht und somit keine Regeln mehr angewendet werden können[26].

2.5 Textursynthese

2.5.1 Definition

Ursprünglich wurde der Wave Function Collapse im Feld der Textursynthese erfunden und angewendet. Allgemein geht es bei der Textursynthese um einen Prozess, bei dem ein Outputbild generiert wird, das eine Textur aufweist, die große Ähnlichkeit mit einem kleineren Inputbild hat[15]. Einige Experten unterscheiden außerdem einen beispiel-basierten Ansatz mit einer ausschließlich algorithmus-basierten Textursynthese ohne Inputbild.

Darüber hinaus wird bei der Textursynthese zwischen *parametric methods* und *non-parametric methods* unterschieden. Die parametrisierten Methoden zeichnen sich durch die Berechnung von Daten aus, die das Inputbild repräsentieren. Auf Grundlage dieser Daten wird anschließend ein Outputbild generiert, das durch die selben Daten repräsentiert wird wie das Original. Bei den nicht-parametrisierte Methoden werden lokale

benachbarte Muster aus dem Inputbild extrahiert und im Outputbild gemäß den Nachbarschaften neu angeordnet. Beim in Kapitel 2.6 vorgestellten Wave Function Collapse handelt es sich um ein nicht-parametrisiertes Verfahren.

2.5.2 Beispiele

Ein klassisches Beispiel für eine nicht-parametrisierte Anwendung von Textursynthese wird in dem Paper *Texture Synthesis by Non-parametric Sampling* von Efros und Leung vorgestellt[7]. Mit dem Ziel, so viel lokale Struktur wie möglich zu bewahren, wird das Outputbild ausgehend von einem initialen Seed um Pixel für Pixel erweitert. Auf Basis von als Markov Random Field (MRF) modellierten Texturen werden die Nachbarschaften jedes Pixels erfasst und Häufigkeiten errechnet.

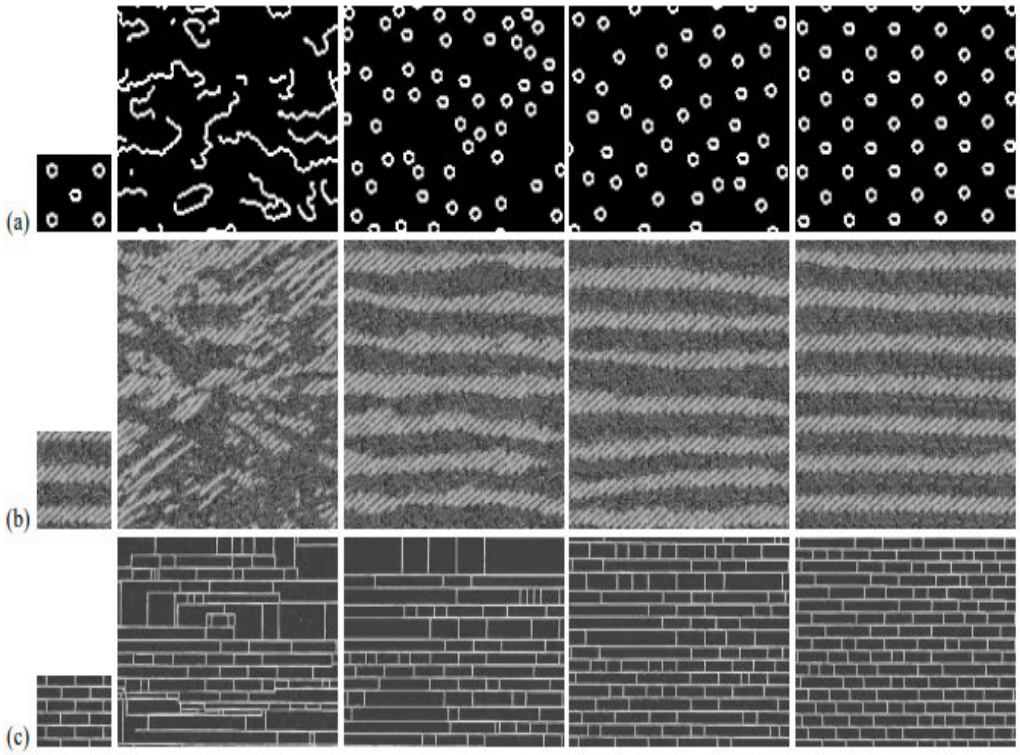


Abbildung 2.7: Mit gegebenem Inputbild (links) wurden mithilfe des Algorithmus vier Outputbilder mit den *neighborhood window*-Größen 5, 11, 15 und 23 (von links nach rechts) erzeugt[7]

Wie in Abb 2.7 zu sehen ist, erstellt der Algorithmus aus einem Inputbild Outputbilder mit sehr ähnlichen Texturen. Ein wichtiger, anpassbarer Faktor ist dabei das *neighborhood window*, welches sich direkt auf die Ähnlichkeit der Texturen von Input- und Outputbild auswirkt. Mit einem kleinen Fenster von 5 ist grobe Struktur der Textur zwar wiederzuerkennen, die Pixel im Outputbild werden jedoch relativ zufällig angeordnet. Je größer das Fenster ist, desto weniger spielt Zufall eine Rolle bis der Output fast eine Vervielfältigung des Inputs darstellt (Fenstergröße 23). Die Fenstergröße wirkt sich also direkt auf den Grad an Zufall in den sich ergebenden Texturen aus.

2.6 Wave Function Collapse

2.6.1 Allgemeines

Der Wave Function Collapse ist ein Algorithmus, der von Maxim Gumin entwickelt und veröffentlicht wurde. Das Ziel des Algorithmus ist die Generierung von Bildern auf Basis von kleineren Beispielbildern. Damit ist der Algorithmus dem Feld der Textursynthese sowie dem Feld der PCG zuzuordnen. Der WFC basiert auf einer *greedy Search* und kann als Methode des *Constraint Solvings* klassifiziert werden[15]. Essentiell dabei ist die lokale Ähnlichkeit, die Input- und Outputbilder bei korrekter Funktion des Algorithmus aufweisen. Als lokale Ähnlichkeit definiert Maxim Gumin, dass das Outputbild ausschließlich aus N x N Mustern besteht, die mindestens ein Mal im Inputbild vorhanden sind[11].

Die grundlegende Idee für den Wave Function Collapse stammt aus der Quantenmechanik, in der ein physikalisches System vollständig durch einen quantenmechanischen Zustand beschrieben werden kann. Dieser lässt sich in einer Wellenfunktion beschreiben und kann als Wahrscheinlichkeitsverteilung aller momentan möglichen Zustände verstanden werden. Während des Wave Function Collapse wird mithilfe von Messungen die Anzahl der möglichen Zustände letztlich auf einen einzelnen Zustand reduziert.

Mit dieser Inspiration definierte Maxim Gumin im Jahr 2016 einen Algorithmus, der diese Idee auf Farbwerte der Pixel im Outputbild anwendet. Bei der Generierung des Outputbildes werden nacheinander einzelne Pixel kollabiert, woraufhin alle daraus hervorgehenden Konsequenzen für mögliche Zustände benachbarter Pixel propagiert werden. Dies geschieht so lange, bis jedem Pixel ein konkreter Wert zugeordnet werden kann[11].

Mittlerweile gibt es bereits mehrere wissenschaftliche Arbeiten, in denen der Wave Function Collapse erweitert wurde. Beispielsweise beschreiben Kim et al.[18] die Modifikation des Wave Function Collapse auf Basis einer graph-basierten Datenstruktur, die die Anwendung des WFC für dreidimensionale Inhalte ermöglicht.

2.6.2 Funktionsweise des Algorithmus

Grundlegend können der klassischen Version des Algorithmus vier elementare Aufgaben zugeordnet werden:

1. Lokale Muster aus dem Inputbild extrahieren
2. Diese Muster mit einem Index versehen, um Nachbarschaftsbedingungen (im Folgenden Adjacency Contraints) zu überprüfen
3. Inkrementell auf Basis der Adjacency Constraints ein Outputbild generieren
4. Das Outputbild im Format des Inputbildes ausgeben[15]

Diese vier Aufgaben lassen sich aus der allgemeinen Beschreibung des Algorithmus ableiten, die Maxim Gumin auf seiner Website definiert.

Diese in Abb. 2.8 dargestellte generische Beschreibung des WFC-Algorithmus veranschaulicht die Arbeitsweise des Algorithmus, ohne jedoch explizit auf die Wahl der Adjacency Constraints einzugehen. Die Methode zur Erkennung von Adjacency Constraints spielt jedoch eine wichtige Rolle, wodurch zwei Varianten des Algorithmus unterschieden werden können, das Simple Tiled Model sowie das Overlapping Model, auf welche in den folgenden Kapiteln näher eingegangen wird.

Generell kann die Schleife (4.) in Abb. 2.8 als essentieller Part des Algorithmus betrachtet werden. Jede Iteration besteht aus zwei Teilschritten: Der Beobachtung (4i.) sowie der Propagierung (4ii.)

Bei der Beobachtung wird zunächst das Wave-Element mit dem kleinsten Entropiewert ungleich Null gewählt. Der Entropiewert ist hier von wesentlicher Bedeutung, da er die Gewichtung der möglichen Muster gemäß ihrer Frequenz an dieser Position, bzw. an diesem Wave-Element indiziert. Er gibt also Aufschluss darüber, wie viele Muster an diesem Wave-Element noch möglich sind: Je niedriger der Entropiewert, desto weniger Optionen sind vorhanden. Nachdem ein Element gewählt ist, kann dieses nun *kollabiert*

1. Lies das Input-Bild und zähle die N x N Patterns.
    i. (optional) Füge die rotierten und reflektierten Patterns hinzu.
2. Erstelle ein Array mit den Dimensionen des Outputs (auch als *Wave* bezeichnet). Jedes Element des Arrays repräsentiert den Zustand einer N x N Region im Output-Bild. Der Zustand beschreibt mithilfe eines Boolean-Koeffizienten für jedes mögliche Muster, ob das Muster an dieser Position der Wave verboten oder noch erlaubt ist. *False* besagt, dass das Muster an dieser Position verboten ist, *true* besagt, dass es noch erlaubt ist.
3. Initialisiere die Wave in vollständig unbeobachtetem Zustand (z.B. alle Boolean-Koeffizienten sind *true*.
4. Wiederhole die folgenden Schritte:
    i. Beobachtung:
        a. Finde das Wave-Element mit dem minimalen Entropy-Wert, der nicht Null ist. Kann kein Element gefunden werden, beende die Schleife (4.) und gehe über zu Schritt (5.).
        b. Kollabiere das gefundene Element in einen konkreten Zustand unter Berücksichtigung dessen Koeffizienten und der Häufigkeitsverteilung der Muster.
    ii. Propagierung: Propagiere die Auswirkungen der gewonnenen Informationen an benachbarte Elemente.
5. Nun sollten alle Wave-Elemente entweder in einem vollständig beobachteten Zustand (alle Koeffizienten außer einem sind *false*) oder in einem Zustand der Kontradiktion sein (alle Koeffizienten sind *false*). Im ersten Fall wird das Ergebnis ausgegeben. Im zweiten Fall wird der Algorithmus beendet, ohne ein Ergebnis auszugeben.

Abbildung 2.8: Beschreibung des WFC-Algorithmus nach Maxim Gumin[11]

werden (Schritt 4ib), es wird also eines der möglichen Muster ausgewählt, wodurch das Element nun in einen konkreten Zustand übergeht.

Die Propagierung erfolgt mithilfe der neu gewonnenen Informationen aus dem Beobachtungsschritt. Für jeden Nachbarn des gewählten Elements wird nun berechnet, welche Muster durch die Kollabierung nun nicht mehr möglich sind. Wenn sich der Zustand des Nachbarn aufgrund der Kollabierung verändert, werden wiederum seine benachbarten

Wave-Elemente betrachtet, sodass es potentiell zu Zustandsänderungen in der gesamte Wave kommen kann.

2.6.3 Simple Tiled Model

Die grundlegende Form des Algorithmus zeichnet sich durch ein Set an vorab definierten Tiles, bzw. eine Tilemap, und die zugehörigen Adjacency Constraints aus. Die Adjacency Constraints werden, anders als beim Overlapping Model, vom Anwender definiert. Somit ist bereits vor dem Start des Algorithmus festgelegt, welche Tiles in welcher Ausrichtung nebeneinander platziert werden können[11] .

Die Constraints werden in einer XML- oder JSON-Datei gespeichert und können zur Laufzeit ausgelesen und verarbeitet werden. Jedes Tile im Output-Bild ist also nur von seinen vier unmittelbaren Nachbarn beschränkt[12].

2.6.4 Overlapping Model

Charakteristisch für das Overlapping Model ist die automatische Erkennung der Muster sowie der Adjacency Constraints. Außerdem kann die Liste der erkannten Muster optional um die rotierten und reflektierten Versionen der Muster ergänzt werden.

Der Prozess der Mustererkennung läuft folgendermaßen ab: Ein N x N großes Quadrat, wobei N die Höhe bzw. Breite der Muster repräsentiert, durchläuft Pixel für Pixel alle möglichen Positionen des Inputbildes. Der jeweils aktuelle Inhalt des Quadrats wird (je nach Anwendungsfall mitsamt Rotation und Reflektion) in einer Liste der möglichen Muster als neues Muster gespeichert. Der Speicherprozess geschieht jedoch nur, wenn das Muster noch nicht in der Liste vorhanden ist. Ansonsten wird das Gewicht des Musters (Parameter *weight*) in der Liste inkrementiert.

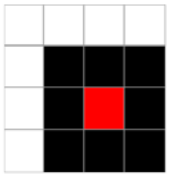


Abbildung 2.9: *Red Maze* Inputbild (4 x 4 Pixel)[15]

Dieses Vorgehen wird in dem Paper *WaveFunctionCollapse is Constraint Solving in the Wild* von Isaac Karth und Adam M. Smith verständlich veranschaulicht[15]. Abbildung 2.9 zeigt ein 4 x 4 Pixel großes Inputbild, aus dem sich mit dem vorgestellten Verfahren mit einer Mustergröße von 2 x 2 Pixel folgende Muster ergeben:

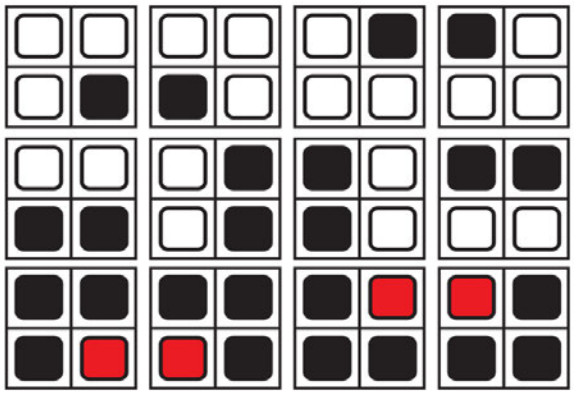


Abbildung 2.10: Alle Muster (mit Rotation und Reflektion), die sich aus dem *Red Maze* Inputbild ergeben[15]

Wenn die Erkennung der Muster abgeschloßen ist, werden die Adjacency Constraints erstellt. Dafür wird für alle in der Liste enthaltenen Muster überprüft, mit welchen anderen Mustern eine Nachbarschaft besteht. Die Nachbarschaften sind richtungssensitiv und bestehen, wenn alle sich überlappenden Pixel die gleichen Farbwerte haben[15].

2.6.5 Probleme des WFC und Vergleich zu Paul Merrell's Model Synthesis

Bei Betrachtung des WFC-Algorithmus ist es notwendig, auch auf die Model Synthesis von Paul Merrell[21] einzugehen. Die Model Synthesis ist ein dem WFC sehr ähnlicher Algorithmus, der bereits im Jahr 2007 von Paul Merrell entwickelt wurde[21]. Laut Merrell lässt sich von zwei Versionen des gleichen Algorithmus sprechen. Gerade an den Ergebnissen der Algorithmen bei kleinen Outputs, siehe Abb. 2.11, lassen sich kaum Unterschiede erkennen.

Bei genauerer Beleuchtung der Algorithmen lassen sich jedoch wichtige Unterschiede feststellen. Einerseits unterscheiden sich die Algorithmen in der Reihenfolge, in der Zellen ausgewählt werden. Die Model Synthesis wählt die nächste Zelle gemäß der Scanline aus,

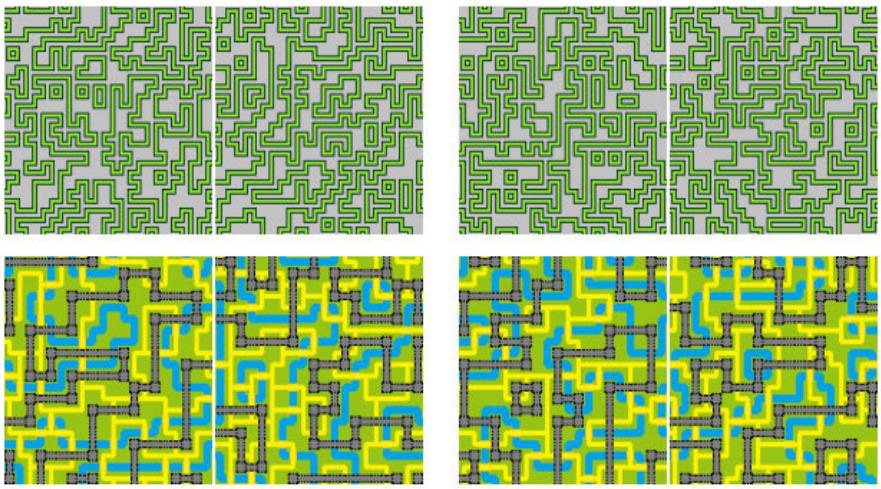


Abbildung 2.11: Die mit Model Synthesis erstellten Bilder (links) sind den mit WFC erstellten Bildern (rechts) sehr ähnlich[21]

während der WFC wie beschrieben die Zelle mit dem niedrigsten Entropiewert selektiert. Dies kann sich gerade bei sehr großen Texturen negativ auf die Laufzeit auswirken und die Fehlerrate erhöhen. Andererseits wird das Modell beim WFC im Ganzen modifiziert, während es bei der Model Synthesis in kleinere Parts unterteilt und letztendlich wieder zusammengesetzt wird.[20] Dies hat zur Folge, dass es beim WFC-Algorithmus, der ursprünglich für 2D-Texturen entwickelt wurde, bei großen Inputtexturen, bspw. 3D-Texturen, und bei sehr großen Outputgrößen zu sehr langen Laufzeiten kommen kann. Für kleine Texturen, wie in dieser Arbeit verwendet, ist der WFC-Algorithmus jedoch ideal, da 2D-Texturen verwendet werden und der Output aufgrund des Anwendungsfalls nicht zu groß wird. Wie in Abb.2.12 zu sehen ist, liefert der WFC Algorithmus bei kleineren Texturen zuverlässig Ergebnisse in kürzester Zeit.

	Model Syn (s)	WFC (s)
Parse Input	3.545	0.386
Synthesis	32.161	7.699
Generate Output	0.081	0.104
Total	35.787	8.186

Abbildung 2.12: Vergleich der Laufzeiten von Model Synthesis und WFC für die Bearbeitung von 54 kleinen Texturen[21]

3 Konzept

Das folgende Kapitel umfasst die funktionalen Anforderungen an den im Rahmen dieser Arbeit entwickelten Prototypen. Außerdem wird das logische Konzept beschrieben, ohne näher auf technische Eigenschaften oder konkrete Implementierungen einzugehen. Es soll folglich eine abstrakte Übersicht über eine mögliche Implementierung gegeben werden.

Die Basis des Konzepts bildet das in Kapitel 2.6.3 beschriebene Simple Tiled Model, welches für den spezifischen Use Case angepasst wird.

3.1 Funktionale Anforderungen

3.1.1 Funktionalität

Der Prototyp soll nach den Grundsätzen des WFC-Algorithmus verschiedene Inputbilder nach vorgegebenen Constraints zu einer Landkarte zusammensetzen. Dem User soll es so möglich sein, auf Knopfdruck eine zufällige Landkarte zu generieren.

3.1.2 Performance

Je nach Input- und Outputgröße und der verwendeten WFC-Variante variieren die Ergebnisse des WFC-Algorithmus bezüglich der Performance stark. Der Prototyp soll performanceoptimiert sein und schnellstmöglich Ergebnisse liefern.

3.1.3 Parametrisierbarkeit

Es soll möglich sein, den Prototyp über verschiedene Parameter anzupassen, sodass das Ergebnis den Wünschen des Users entsprechend variabel sein kann. Dafür soll beispielsweise die Größe des Outputbildes angepasst werden können.

3.1.4 Input- und Outputformat

Der Prototyp soll Bilder in Form von PNG-Dateien als Input akzeptieren. Die Inputbilder haben dabei alle die gleiche Höhe und Breite, die der Variable *tileSize* der Application-Klasse entsprechen müssen. Als Output soll ebenfalls ein Bild als PNG-Datei entstehen, auf dem der Input eindeutig wiedergefunden werden kann. Die Größe des Outputbildes lässt sich in den Parametern des Algorithmus einstellen und beträgt ein Vielfaches der Größe der Inputbilder.

3.1.5 Landkarte

Die generierte Landkarte sollte über folgende Merkmale verfügen:

- Die Landkarte sollte über verschiedene Landschaftsmerkmale verfügen (z.B. Bäume, Berge)
- Bestimmte Landschaftsmerkmale sollten sich erkennbar zu Gruppen zusammenformen (z.B. Berge fügen sich zu einem Gebirge zusammen)
- Da keine reale Region abgebildet werden, sondern eine Fantasy-Karte generiert werden soll, sollen vereinzelt prägnante Merkmale generiert werden (bspw. Kirchen oder Burgen)
- Die Landmasse sollte durch Wasser begrenzt sein

3.2 Modifikationen am WFC

3.2.1 Vorbereiten des Inputs

In der Originalimplementierung verwendet Maxim Gumin für das Simple Tiled Model XML-Dateien für die Speicherung und Definition der benötigten Tiles.

In Abbildung 3.1 ist zu erkennen, dass Gumin nicht nur die in dem Simple Tiled Model verwendeten Tiles bzw Inputbilder definiert, außerdem versieht er sie mit einem Gewicht (weight) sowie der gewünschten Symmetrie (symmetry). Die Symmetrie spielt später beim Hinterlegen der Bilder in der Musterliste eine wichtige Rolle, da so bestimmt wird, welche Rotationen zusätzlich hinterlegt werden müssen. Die Gewichte geben Auskunft

über die Häufigkeit, mit der dieses Teil im Outputbild vorkommen soll. Darüber hinaus sind in dieser Datei die Nachbarschaften eindeutig definiert.

```
<set size="14">
        <tiles>
                <tile name="bridge" symmetry="I" weight="1.0"/>
                <tile name="component" symmetry="X" weight="20.0"/>
                <tile name="connection" symmetry="T" weight="10.0"/>
                <tile name="corner" symmetry="L" weight="10.0"/>
                <tile name="substrate" symmetry="X" weight="2.0"/>
                <tile name="t" symmetry="T" weight="0.1"/>
                <tile name="track" symmetry="I" weight="2.0"/>
                <tile name="transition" symmetry="T" weight="0.4"/>
                <tile name="turn" symmetry="L" weight="1.0"/>
                <tile name="viad" symmetry="I" weight="0.1"/>
                <tile name="vias" symmetry="T" weight="0.3"/>
                <tile name="wire" symmetry="I" weight="0.5"/>
                <tile name="skew" symmetry="L" weight="2.0"/>
                <tile name="dskew" symmetry="\" weight="2.0"/>
        </tiles>
        <neighbors>
                <neighbor left="bridge" right="bridge"/>
                <neighbor left="bridge 1" right="bridge 1"/>
                <neighbor left="bridge 1" right="connection 1"/>
                <neighbor left="bridge 1" right="t 2"/>
                <neighbor left="bridge 1" right="t 3"/>
```

Abbildung 3.1: Auszug aus der data.xml Datei aus Gumin's *Circuit*-Beispiel[11]

Für diesen Prototyp wird anstatt der XML-Datei eine JSON-Datei verwendet, da so die Erstellung und Bearbeitung dieser Datei für die Verwendung erleichtert und übersichtlicher gemacht wird.

3.2.2 Hinterlegen der Inputbilder

Die Tiles bzw. Inputbilder werden gemeinsam mit der zugehörigen JSON-Datei in einem Ordner hinterlegt, dessen Pfad dem SimpleTiledModel in der Application-Klasse als String übergeben wird. So können alle zugehörigen Dateien in einer Vorverarbeitungsphase vor der Ausführung des WFCs eingelesen und zwischengespeichert werden.

3.2.3 Manuelles Erstellen der Constraint-Datei

Das Simple Tiled Model benötigt im Vorfeld eine Constraint-Datei, die bereits in den vorangegangenen Kapiteln beschrieben wurde. Diese trägt den Namen *constraints.json* und wird vom Entwickler manuell erstellt. In dieser Datei werden neben den Nachbarschafts-Constraints auch die zu verwendenden Offsets definiert.

Wie in Abb. 3.2 zu erkennen ist, lassen sich in der Constraint-Datei folgende Parameter definieren:

- offsets: Für diesen Use Case werden die vier Offsets „top“, „right“, „bottom“ und „left“ verwendet sowie die dazugehörigen Offsets im Koordinatensystem definiert.
- adjacencyRules: Hier werden alle Inputbilder hinterlegt. Für jedes Inputbild wird die zugehörige Datei („file“), die Symmetrie („symmetry“), das Gewicht für die gewünschte Häufigkeit („weight“) sowie die erlaubten Nachbarschaften bzw. Inputbilder in Richtung der jeweiligen Offsets bestimmt.

```
{
  "offsets": {
    "top": [ 0, -1],
    "right": [ 1, 0],
    "bottom": [ 0, 1],
    "left": [ -1, 0]
  },
  "adjacencyRules": {
    "border": {
      "file": "border.png",
      "symmetry": "X",
      "weight": 1,
      "top": [ "textureWater"],
      "right": [ "border"],
      "bottom": [ "treePines","rocksMountain"],
      "left": [ "border"]
    },
```

Abbildung 3.2: Auszug aus einer beispielhaften constraints.json-Datei

3.2.4 Alternatives Finden der Nachbarschafts-Constraints

Alternativ zur Verwendung einer manuell zu erstellenden Constraint-Datei wäre es auch möglich, die Nachbarschafts-Constraints algorithmisch zu erfassen. Dafür müssten die Muster an den Rändern um Pixel für Pixel hinsichtlich der Farbwerte abgeglichen werden, was wiederum die Laufzeit je nach Größe und Anzahl an Inputbildern enorm erhöhen würde. In einer optimierten Version könnten lediglich drei Punkte (z.B. (0,0), (length/2, 0) und (length,0)) miteinander hinsichtlich der RGB-Werte verglichen und somit die Laufzeit reduziert werden.

Problematisch bei diesem Ansatz ist jedoch die Erstellung der Inputbilder. Die Bilder müssten perfekt miteinander abschließen, damit sie als Nachbarn erkannt werden können. Die Erstellung passender Inputbilder ist im Rahmen dieser Arbeit nicht möglich.

3.2.5 Parameter

Um den Ablauf und somit das Ergebnis des WFC-Algorithmus zu beeinflussen und wie gewünscht zu einem Ergebnis kommen zu können, ist es notwendig folgende Parameter zu übergeben:

- IModel: Die verwendete Variante des WFC-Algorithmus. Für den Landkartengenerator wird das SimpleTiledModel übergeben. Alternativ wäre es mit angepassten Inputbildern auch möglich, das in Kapitel 3.2.4 beschriebene AutoSimpleTiledModel zu verwenden.
- IWaveDrawer: Der WaveDrawer verarbeitet das Ergebnis des WFC-Algorithmus und erstellt die Output-Datei.
- OutputSize: Ein Integer-Array mit zwei Elementen. Diese definieren, wie groß das generierte Bild sein soll. Der erste Wert definiert die Anzahl an Tiles in der Horizontalen, während der zweite Wert die Anzahl an Tiles in der Vertikalen beschreibt.

3.2.6 Variablen

Die zentrale Klasse des Prototyps *WaveFunctionCollapse* benötigt folgende Variablen, um für den Algorithmus essentielle Daten zwischenzuspeichern und zur Laufzeit darauf zuzugreifen:

- Map<Coordinates, Cell<T» := Eine Hashmap, die die Wave repräsentiert. In der Map werden die Koordinaten mit den zugehörigen Zellen gespeichert.
- List<String> := Eine Liste, in der alle verwendeten Directions als Strings enthalten sind.
- List<Tile<T» := Eine Liste mit allen eingelesenen Tiles.
- IWaveDrawer<T>:= Eine Instanz für das spätere Zeichnen des Ergebnisses.

3.3 Durchlaufen des Wave Function Collapse

Wie bereits in Kapitel 2.6.2 beschrieben, kann der Wave Function Collapse in vier Schritte unterteilt werden:

- 1. Muster aus einem Inputbild extrahieren
- 2. Diese Muster mit einem Index versehen, sodass Adjacency Constraints definiert werden können
- 3. Inkrementell auf Basis der Adjacency Constraints ein Outputbild generieren
- 4. Rendern und Ausgeben des Outputbildes

Für diese Applikation können diese Schritte jedoch leicht modifiziert werden. In Schritt 1 werden anstelle der Extraktion der Muster aus dem Inputbild mehrere Inputbilder verwendet. Da für den Prototypen das Simple Tiled Model verwendet wird, müssen in Schritt 2 auch keine Muster mit Indizes versehen und automatisch Adjacency Constraints definiert werden. Stattdessen wird jedes Inputbild in die Constraints-Datei eingetragen und die erlaubten Nachbarschaften werden wie in Kapitel 3.2.3 beschrieben definiert.

Für diesen Prototyp bildet Schritt 3 das Herzstück und wird im Folgenden näher erläutert.

3.3.1 Initialisierung der Wave

Bei Beginn jeder Generierung sowie nach jeder Kontradiktion ist das Setzen der Wave auf den Initialzustand notwendig. Dafür wird jede Zelle der Wave mit einer Liste aller Tiles aus dem Simple Tiled Model initialisiert.

Algorithm 1 Initialisierung der Wave

```
procedure INITIALIZEWAVE(List<Tile> tiles)
    for all Cell C ∈ wave do
        C ← tiles
    end for
end procedure
```

3.3.2 Generelle Lösungsroutine

Dieser Algorithmus bildet die Basis des Prototyps und ist dafür zuständig, das Outputbild gemäß der Adjacency Constraints mit konkreten Tiles zu belegen. Das Kernelement ist hierbei eine Schleife, die so lange wiederholt wird, bis alle Zellen nur noch genau einen State haben, also kollabiert sind und der Algorithmus folglich zu einem Ergebnis gekommen ist. Wie auch in Maxim Gumin's Originalimplementierung wird in diesem Prototypen kein Backtracking verwendet. Kommt es während des Durchlaufens der Lösungsroutine zu einem Widerspruch, wird die Wave wieder in den Initialzustand versetzt und der Algorithmus wird neugestartet.

Algorithm 2 General Process

```
procedure GENERATE
    initializeWave()
    totalContradictions ← 0
    while ∃ Cell ∈ wave with more than 1 state do
        findLowestEntropyCell()
        if Contradiction then
            totalContradictions++
            initializeWave()
            findLowestEntropyCell()
        end if
        replaceStates(Cell lowestEntropyCell)
        Stack < Cell > cellsToUpdate ← lowestEntropyCell
        while !cellsToUpdate.IsEmpty() do
            propagate(Stack<Cell> cellsToUpdate)
        end while
    end while
    drawWave()
end procedure
```

Zunächst wird die im Konstruktor kreierte Wave mittels der *initializeWave()*-Methode in den Initialzustand versetzt. Wie in Kapitel 3.3.1 erwähnt, wird jede Zelle des Outputgrids mit einer Liste aller möglichen Tiles bzw. aller Inputbilder versehen. Die lokale Variable *totalContradictions* wird zu Beginn mit 0 initialisiert und bei jedem Auftreten einer Kontradiktion inkrementiert.

In der Schleife wird zunächst der Beobachtungsschritt bzw. die Observierung in Form der Methoden *findLowestEntropyCell()* sowie *replaceStates(Cell lowestEntropyCell)* durchge-

führt. Wie der Name bereits andeutet, ist die Aufgabe der Methode *findLowestEntropyCell()*, die Zelle zurückzugeben, die aktuell den niedrigsten Entropiewert ungleich Null aufweist.

Im zweiten Teil des Beobachtungsschritts wird nun die gefundene Zelle kollabiert. In der *replaceStates*-Methode wird ein konkreter State zufällig, aber gewichtet durch die in der constraints.json-Datei vergebene Häufigkeitsverteilung, aus allen möglichen States ausgewählt. Anschließend werden alle States dieser Zelle entfernt, sodass nur noch der ausgewählte State übrig bleibt.

Der Propagierungsschritt wird im folgenden Kapitel 3.3.3 näher erläutert.

3.3.3 Propagierung

Im Anschluss an den Kollaps der ausgewählten Zelle folgt die Propagierung in Form der Methode *propagate(Stack<Cell> cellsToUpdate)*, die so wiederholt aufgerufen wird, bis keine Elemente mehr im Stack vorhanden sind.

Algorithm 3 Propagation

```
procedure PROPAGATE(Stack<Cell> cellsToUpdate)
    C ← top element of cellsToUpdate
    for all dir ∈ directions do
        N ← neighbor of C in direction dir
        N_S ← all possible states for N
        O ← Set of options left for C in direction dir
        N_S ← N_S ∩ O
        if N_S has changed then
            cellsToUpdate ← N
        end if
    end for
end procedure
```

In der propagate-Methode werden ausgehend von der aktuellen, kollabierten Zelle die Nachbarzellen in allen vier Richtungen herausgefunden. Anschließend werden die States der Nachbarn entsprechend der Nachbarschafts-Constraints aktualisiert. Bei der Anpassung der States der Nachbarn ist es essentiell zu prüfen, ob sich die States geändert haben. Haben sich die States eines Nachbarn geändert, wird der Nachbar auch zum Stack hinzugefügt, sodass alle notwendigen Aktualisierungen in weiteren Propagierungszyklen durchgeführt werden können.

3.3.4 Zeichnen der Wave

Nachdem der Algorithmus erfolgreich durchgelaufen ist, wird das Ergebnis der Klasse WaveDrawer übergeben und die Methode *drawWave(Map<Coordinates,Cell> wave)* wird mit der resultierenden Wave als Parameter aufgerufen.

Diese Methode hat die Aufgabe, alle Koordinaten des Outputbildes zu durchlaufen und die Tiles, die als konkrete States in den Zellen gespeichert sind, an die jeweilige Position in eine neue PNG-Datei zu schreiben. Diese PNG-Datei stellt das Ergebnis dar.

4 Umsetzung

4.1 Nichtfunktionale Anforderungen

1. **Separation of Concerns**: Die Applikation sollte gemäß dem Prinzip *Divide and Conquer*in Bausteine unterteilt sein, die jeweils für genau eine Aufgabe verantwortlich sind. Die Verantwortlichkeiten sollten klar erkennbar und abgrenzbar sein.
2. **Minimierung von Abhängigkeiten**: Abhängigkeiten sollten auf ihre Notwendigkeit geprüft werden und wenn möglich vermieden werden (Lose Kopplung). Besonders zyklische Abhängigkeiten sollten vermieden werden.
3. **Verständlichkeit und Übersichtlichkeit**: Auch Entwickler, die nicht an der Entwicklung des Projekts beteiligt waren, sollen schnell die Struktur und die Zuständigkeiten der einzelnen Komponenten durchdringen können. Dies wird durch folgende Aspekte erzielt:
   - JavaDoc-Kommentare
   - Selbsterklärende Komponenten-, Klassen-, Methoden- und Variablennamen
   - Übersicht durch Auslagern von Hilfsmethoden
   - Zerteilen von größeren Methoden in kleine Methoden
   - Besondere Erklärung von wichtigen Parts im Code
4. **Erweiterbarkeit**: Nachträglich soll das Projekt einfach um Funktionalität ergänzt werden können.
5. **Redundanzfreiheit**: Wenn derselbe Code in mehreren Stellen des Projekts verwendet wird, sollte dieser in eine eigene Methode ausgelagert werden. Dies sorgt für bessere Wartbarkeit und bildet die Basis für Wiederverwendung.

6. **Klare Schnittstellendefinitionen**: Um keine Details der Implementierung offenzulegen, sollten Komponenten über klare Schnittstellen verfügen. So können Abhängigkeiten minimiert und darüber hinaus für Austauschbarkeit von Teilen der Implementierung gesorgt werden.

4.2 Vorgehensmodell

Als Vorgehensmodell wurde für dieses Projekt ein agiler, inkrementeller Ansatz gewählt. Als Vorbild dienten bekannte agile Verfahren, wie Scrum. Es wurden Sprints mit einer Dauer von zwei Wochen gewählt, um das Projekt strukturiert mit regelmäßigen Planungs- und Evaluierungsphasen bearbeiten zu können.

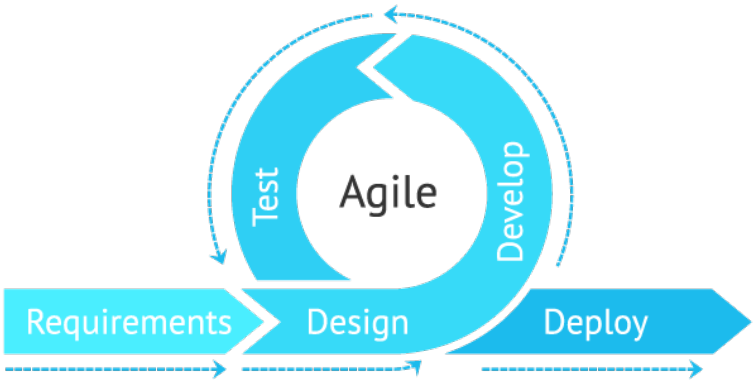


Abbildung 4.1: Agiler Ansatz für die Umsetzung dieses Projekts[1]

Grob kann das Projekt in folgende Arbeitspakete unterteilt werden:

- Einarbeitung in die Original-Implementierung von Maxim Gumin[11]
- Grundlegende 2D-Implementierung des WFC-Algorithmus mit Orientierung an der Arbeit von Ben Dzaebel[5]
- Entwicklung des Inputs in erforderlichem Format (Bilder sowie constraint-Dateien)
- Erweiterung/Modifikation des WFC-Algorithmus

4.3 Verwendete Tools und Technologien

Java Für dieses Projekt wurde Java als Programmiersprache verwendet. Die Wahl fiel auf Java, da es bereits mehrere Implementierungen des WFC-Algorithmus mit Java als Referenz gibt und Java als Programmiersprache für diesen Anwendungsfall gut geeignet ist. Außerdem spielte die Präferenz des Autors eine große Rolle bei der Wahl der Programmiersprache. Für die Applikation wurde die JDK 14[1] verwendet.

Git Version Control Für eine bessere Versionskontrolle, bzw. um auch zu späteren Zeitpunkten wieder zu dem Stand früherer Implementierungen zurückkehren zu können, wurde Git verwendet.[2]

IntelliJ Idea Als Entwicklungsumgebung für diesen Prototyp wurde IntelliJ Idea[3] verwendet. Diese IDE wurde aufgrund von persönlichen Präferenzen des Autors gewählt.

Maven Maven[4] ist ein Tool, mit dem sich der Build-Prozess stark vereinfachen lässt. So müssen Abhängigkeiten nicht eigenständig vom Nutzer aufgelöst werden.

4.4 Verwendete Libraries

JSON-Simple Für das Auslesen der Constraints-Dateien wird die Bibliothek JSON-Simple[5] verwendet.

[1] Siehe: https://www.oracle.com/de/java/technologies/javase/jdk14-archive-downloads.html (Access: 21.06.2022
[2] Siehe: https://git-scm.com/ (Access: 21.06.2022)
[3] Siehe: https://www.jetbrains.com/de-de/idea/ (Access: 21.06.2022)
[4] Siehe: https://maven.apache.org/ (Access: 21.06.2022)
[5] Siehe: https://mvnrepository.com/artifact/com.googlecode.json-simple/json-simple/1.1.1 (Access: 21.06.2022)

4.5 Architektur des Systems

Die Applikation setzt sich aus den drei Komponenten **Model**, **WaveDrawer** und **WFC**, einem Utility-Package sowie einer zentralen Application-Klasse zusammen. Die Komponenten geben ihre Funktionalität mittels Interfaces nach außen. So ergibt sich folgendes Komponentendiagramm.

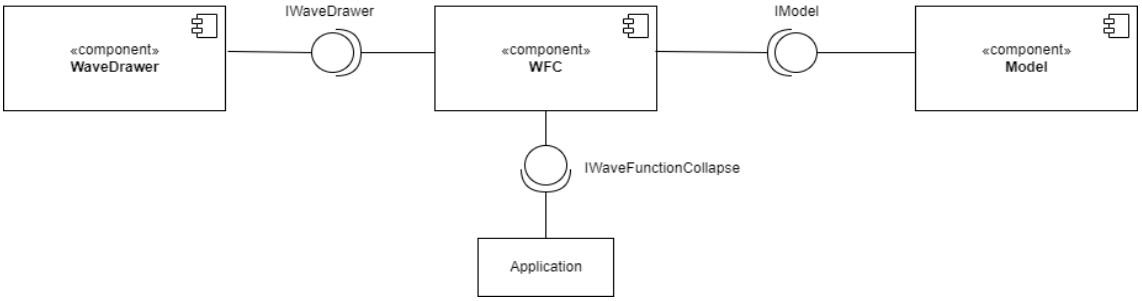


Abbildung 4.2: Top-Level Komponentendiagramm für diesen Prototyp (erstellt mit Draw.io; siehe https://app.diagrams.net/)

4.6 Referenzierte Projekte und verwendete Daten

Thematisch ähnliche Projekte Als Vorlage für diese Arbeit gilt hauptsächlich die Original-Implementierung des WFC-Algorithmus von Maxim Gumin[11]. Darüber hinaus wurden zwei weitere Projekte als Orientierung für die Implementierung zu Hilfe genommen. Einerseits das im Rahmen einer Bachelorarbeit erstellte Projekt *Erweiterung des Wave Function Collapse Algorithmus zur Synthese von 3D-Voxel-Modellen anhand von nutzergenerierten Beispielen* von Ben Dzaebel[5] und andererseits eine im Zuge des Master Grundprojekts entwickelte generische Implementierung des WFC von Christian Dorn[4], die im PCG Repository der HAW Hamburg zu finden ist[13]. Letztlich wurde eine C#-Implementierung von Joseph Parker[22] als weitere Orientierung genommen.

Während die Struktur der Kernschleife des WFC-Algorithmus von Dorn[4] zum Großteil verwendet werden konnte, mussten primär die Vor- und Nachverarbeitung auf diesen Anwendungsfall angepasst werden. Besonders die Teile aus Implementierung von Ben Dzaebel, die Inspiration für diese Arbeit boten, bedurften großen Bearbeitungsaufwand, da die Arbeit auf 3D-Texturen ausgelegt ist.

Inputbilder Die Bilder, die als Input verwendet werden, stammen hauptsächlich aus einem lizenzfreien Kartographie-Pack des Artists Kenney.nl[17]. Die enthaltenen PNG-Dateien passen nicht nur optisch gut für diesen Anwendungsfall, sie eignen sich auch dank des symmetrischen Formats von 128 x 128 Pixeln optimal für diese Anwendung. Weitere Inputbilder wurden manuell mithilfe des Tools GIMP[6] erstellt.

4.7 Implementierung

4.7.1 Komponente Model

Die Komponente Model sorgt für die Vorbereitung der Daten gemäß der WFC-Variante Simple Tiled Model. Der Input wird eingelesen und mithilfe des Interfaces IModel an den WFC-Algorithmus übergeben. Aufgrund der Nutzung des Interfaces ist das Projekt potentiell um weitere WFC-Varianten, wie das Overlapping Model erweiterbar.

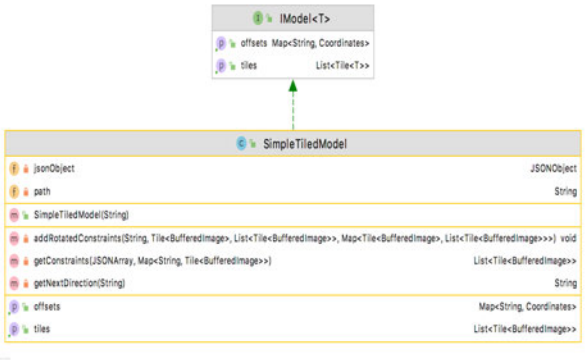


Abbildung 4.3: UML Diagramm der Klasse SimpleTiledModel sowie des Interfaces IModel

[6]Siehe: https://www.gimp.org/

4.7.2 Komponente WFC

Die Komponente WFC beinhaltet alle Funktionalitäten, die für die Durchführung des Wave Function Collapse notwendig sind. Die Basis bildet dabei die Klasse *WaveFunctionCollapse*, die die Anwendungslogik beinhaltet und den Start des Generierungsprozesses über das Interface *IWaveFunctionCollapse* nach außen gibt. Dies wird über die Methode generate() ermöglicht, die aus der zentralen Application-Klasse heraus aufgerufen wird.

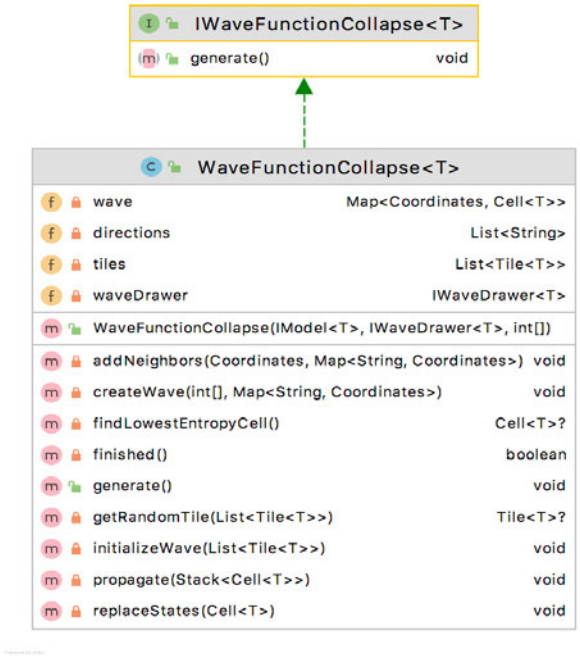


Abbildung 4.4: UML Diagramm der Klasse WaveFunctionCollapse sowie des zugehörigen Interfaces

4.7.3 Komponente WaveDrawer

Die Komponente WaveDrawer sorgt für die Nachbearbeitung der Daten nach dem erfolgreichen Durchlaufen des WFC-Algorithmus. Dafür wird aus der Komponente WFC

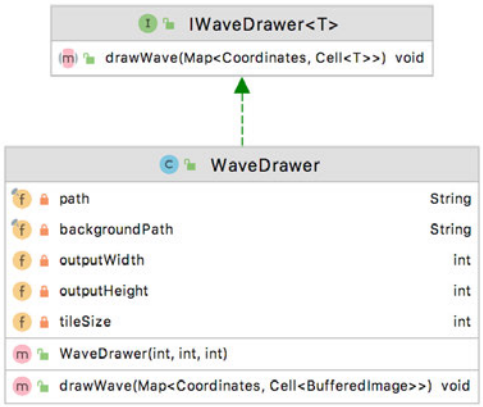


Abbildung 4.5: UML Diagramm der Klasse WaveDrawer sowie des zugehörigen Interfaces

heraus auf die Interface-Methode *drawWave(Map<Coordinates,Cell<T»)* zugegriffen, sodass aus den Daten der generierten Wave eine PNG-Datei erstellt werden kann.

4.7.4 Package Utility

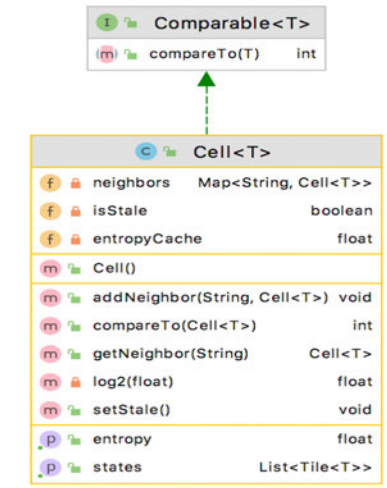
Das Package *Utility* umfasst verschiedene Klassen, die in mehreren Komponenten genutzt werden und essentiell für die Struktur der Applikation und die Verwendung des Algorithmus sind.

Wichtige Utility-Klassen

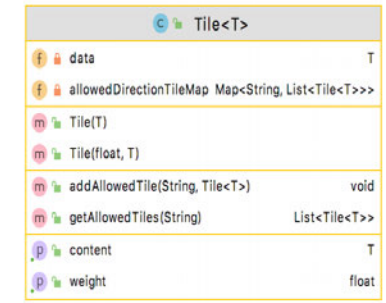
Cell Die Klasse Cell repräsentiert eine einzelne Zelle im WFC Algorithmus. Jede Zelle kennt die vier benachbarten Zellen und verfügt über eine Liste, in der alle möglichen States bzw. Tiles hinterlegt sind. Ist nur noch ein State in der Liste, gilt die Zelle als kollabiert.

Um Zellen miteinander hinsichtlich des Entropiewertes vergleichen zu können, implementiert diese Klasse das Interface *Comparable*.

Tile Die Klasse Tile repräsentiert ein einzelnes Tile im WFC Algorithmus. Neben den Daten, die in dem Tile gespeichert sind, verfügt jedes Tile auch über ein Gewicht sowie eine HashMap, in der alle erlaubten Tiles in den jeweiligen Richtungen gespeichert sind.



(a) Klasse *Cell*



(b) Klasse *Tile*

Abbildung 4.6: UML-Diagramme von wichtigen Utility-Klassen

4.7.5 Klasse Application

Die Klasse *Application* stellt die zentrale Klasse der Anwendung dar. Aus ihr heraus kann der Algorithmus gestartet werden. Außerdem lässt sich über die Parameter die Größe des Outputbildes verändern. Mithilfe des Interfaces IWaveFunctionCollapse wird aus der Klasse Application heraus eine Instanz des Algorithmus erstellt und mittels der generate-Methode der Generierungsprozess gestartet.

Die Applikation kann mit Werten für die verhältnismäßige Größe des Outputbildes gestartet werden, also zwei Integern für die Höhe und Breite. Die letztendliche Größe des Outputbildes lässt sich jedoch aus diesen Parameter multipliziert mit der Größe der Tiles errechnen. Dies stellt sicher, dass das Outputbild größer als die Inputbilder ist. Werden beim Start der Anwendung unpassende oder unzureichende Parameter übergeben, wird der Algorithmus mit Default-Werten gestartet.

5 Evaluation

Das folgende Kapitel dient der Evaluation und Diskussion des entwickelten Prototyps. Neben der Identifikation von Problemen und möglichen Lösungen und der Reflektion über die Einhaltung von Zielen und Anforderungen soll auch auf mögliche Anwendungsgebiete eingegangen werden.

5.1 Präsentation der Ergebnisse

Nachfolgend werden einige der generierten Landkarten präsentiert. Dabei ist zu beachten, dass die generierten Karten stark von den in der Constraint-Datei gewählten Parametern *weight*, *symmetry* sowie Nachbarschaftsregeln abhängt.

5.1.1 Generierte Landkarten

Die Abbildung 5.1 zeigt ausdrücklich den Erfolg des WFC-Algorithmus sowie der Variante SimpleTiledModel. Als Input für diese Karte dienten ausschließlich Bilder von Wegen (Gerade, Kurve, Sackgasse, 3-Wege-Kreuzung). Die Karte verdeutlicht das erfolgreiche Erfassen und Umwandeln der Constraint-Datei, der Rotationen und Gewichte.

Auf Abb. 5.2 ist die Generierung von Inseln zu sehen. Als Input dienten hierfür lediglich die Grenzen der Küsten sowie die Ecken der Inseln (und die Gebirge als Platzhalter für verschiedene Landschaften). Mit detaillierteren Inputbildern wäre es auch möglich, die rechteckige Form der Inseln zu vermeiden. Da die Inputbilder jedoch manuell erstellt wurden, war es in dieser Arbeit leider nicht möglich, vielfältigere Inputbilder zur Verfügung zu stellen.

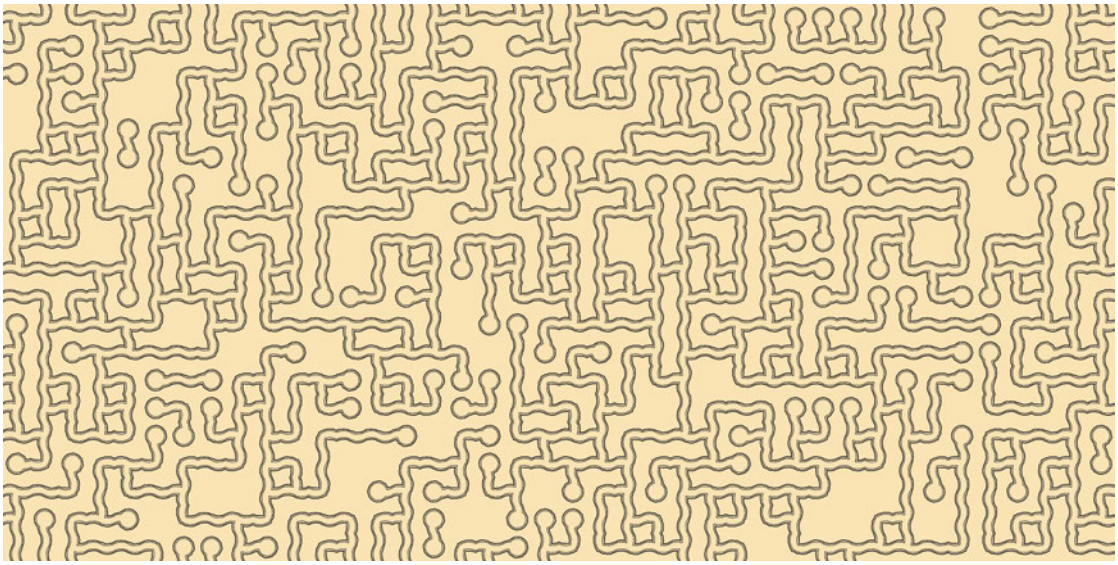


Abbildung 5.1: Generierte Karte mit ausschließlich Wegen

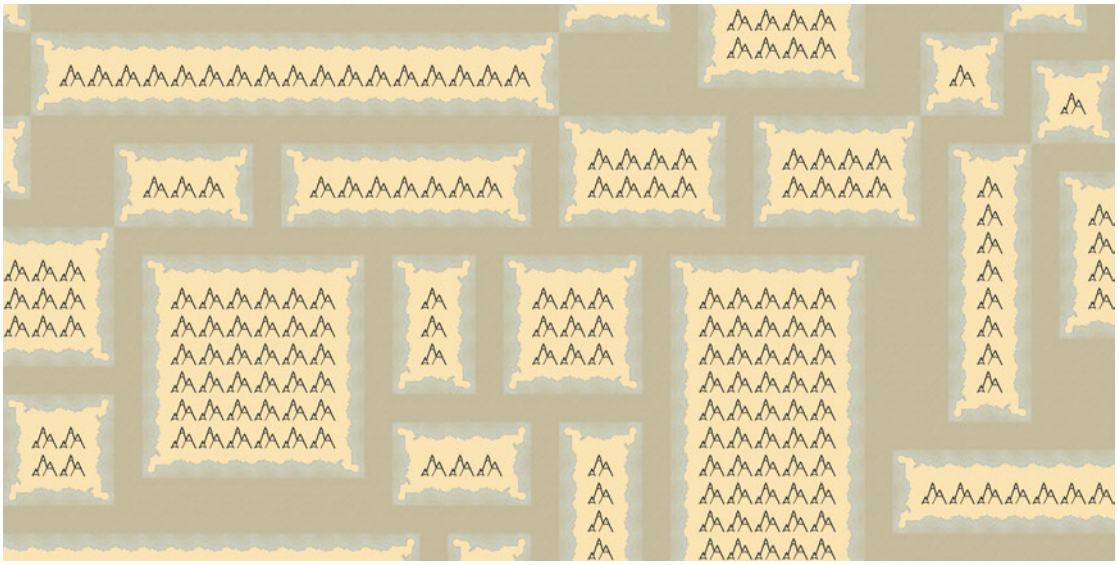


Abbildung 5.2: Generierte Karte mit mehreren kleinen Inseln

Abb. 5.3 verdeutlicht die Auswirkungen der Gewichte auf die Generierung. Während Seen oder Kirchen aufgrund der niedrigen Gewichte sehr vereinzelt auf der Karte vertreten sind, kommen Gebirge wesentlich häufiger vor. So lassen sich vom Nutzer mit der Änderung des Parameters *weight* einfach Prioritäten setzen und so sehr unterschiedliche Ergebnisse generieren. Auf Abb. 5.4 ist gut zu erkennen, wie sich die Nachbarschafts-Constraints auf die Generierung der Karte auswirken. In den Constraints ist beispiels-

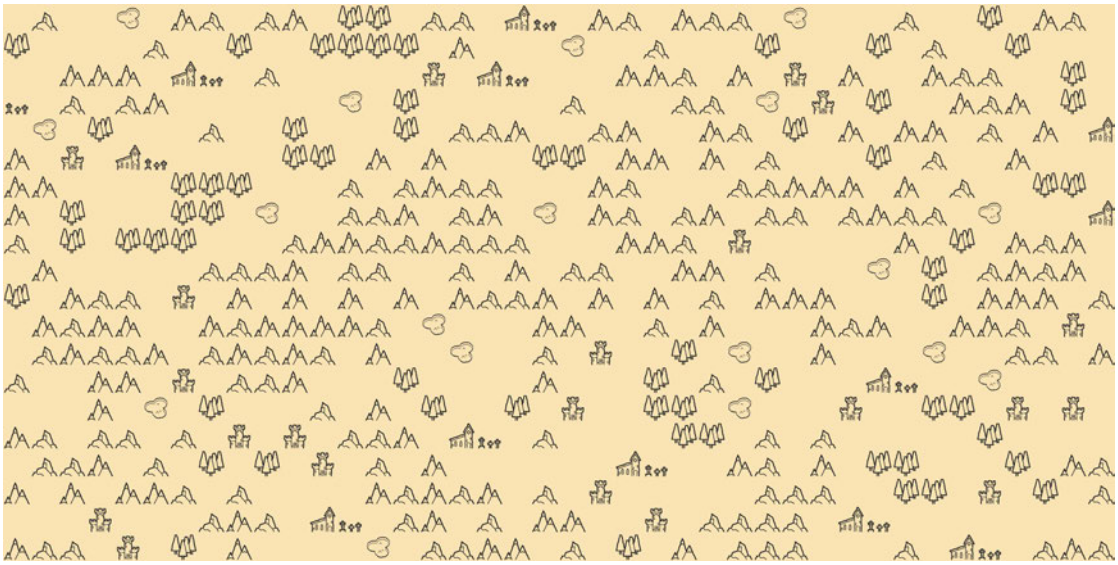


Abbildung 5.3: Generierte Karte mit vielen Gebirgen

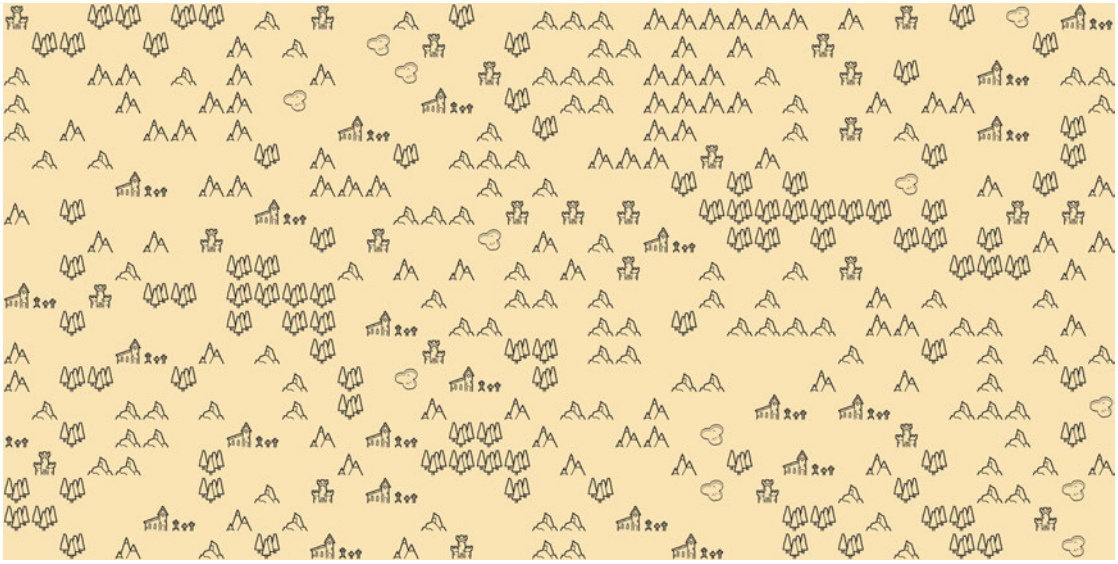


Abbildung 5.4: Generierte Karte mit verschiedenen Elementen

weise definiert, dass Grabsteine nur rechts von Kirchen vorkommen dürfen und dass Baumgruppen sowie Gebirge jeweils nebeneinander auftreten dürfen, von andersartigen Tiles aber durch ein leeres Tile, welches auch als Input definiert wird, getrennt sein müssen. Auch das leere Tile kann gewichtet werden, sodass die Karte beliebig dicht gefüllt werden kann.

Auf der Abbildung 5.5 ist schlussendlich eine sehr großflächige Karte zu sehen. Diese verfügt über die Maße 8960 x 5120 Pixel.

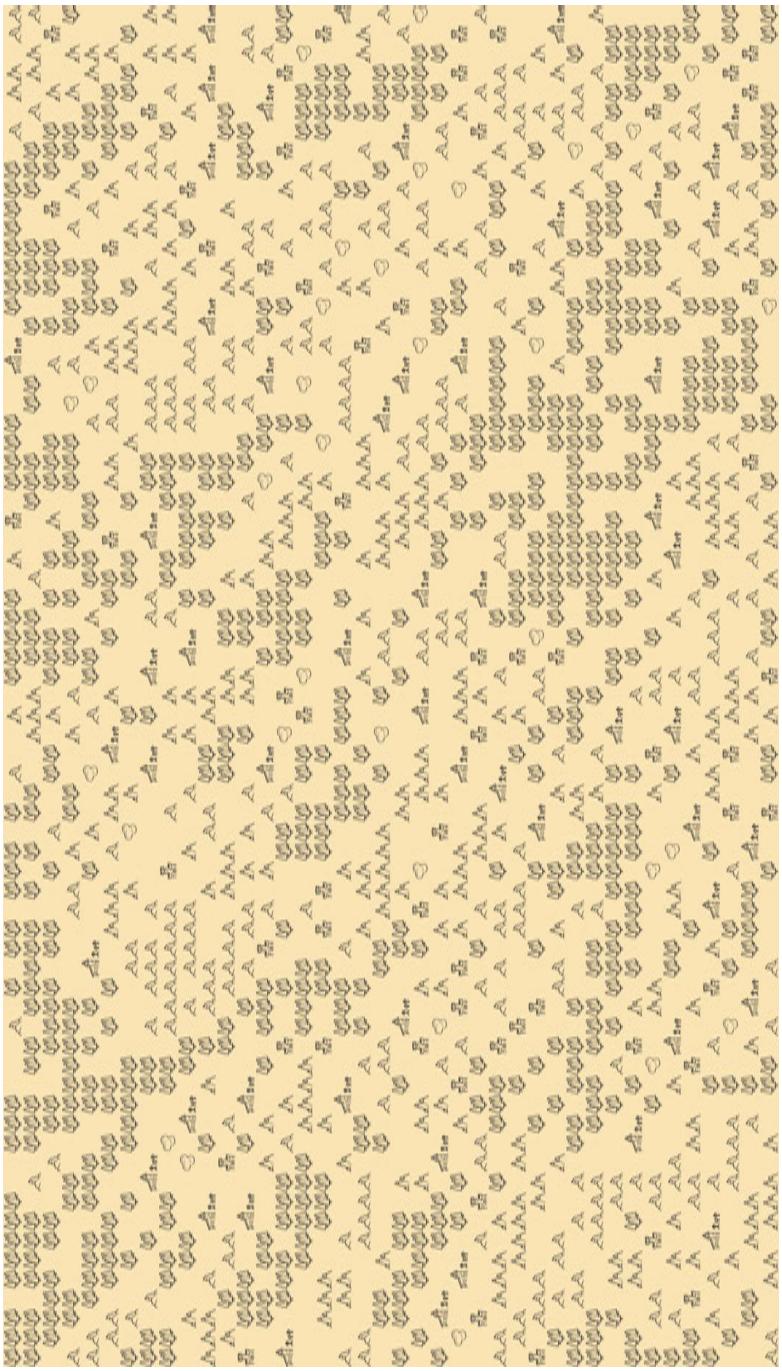


Abbildung 5.5: Sehr große generierte Karte mit verschiedenen Elementen

5.2 Evaluierung des Prototyps

5.2.1 Ziele und Anforderungen

Der im Rahmen dieser Arbeit entwickelte Prototyp erfüllt viele der vorab definierten Ziele und Anforderungen und liefert befriedigende Ergebnisse. Der Algorithmus liefert mit einer zu vernachlässigenden Fehlerrate Ergebnisse. Bei den gewählten Input- und Outputgrößen kommt es zwar gelegentlich zu Kontradiktionen, es wurden jedoch zuverlässig Ergebnisse in kurzer Zeit generiert. Die dabei generierte Landkarte ist natürlich stark von verwendeten Inputbildern und den definierten Gewichten und Nachbarschafts-Constraints in der Constraint-Datei abhängig. Wie bereits in Kapitel 2.6.5 angerissen, kann es bei enorm großen Outputgrößen und bei sehr komplexen Inputgrößen zu Performanceproblemen kommen. Dies ist jedoch bei der Arbeit mit dem WFC nicht zu vermeiden und für diesen Anwendungsfall zu vernachlässigen.

Die funktionalen Anforderungen betreffend liefert der Prototyp mit den meisten Parameterkombinationen eine sehr gute Performance. Die Parametrisierbarkeit wurde jedoch nur bedingt eingehalten. Es ist zwar möglich, verschiedene Parameter den Vorstellungen entsprechend anzupassen, hinsichtlich der Benutzerfreundlichkeit gibt es jedoch Optimierungsbedarf. Dies ist bedingt durch das manuelle Erstellen der Constraint-Datei sowie die nicht vorhandene UI.

Die generierten Landkarten betreffend wurde die meisten Anforderungen erfüllt. Der Detailgrad der Karten würde sich mit detaillierteren und vielfältigeren Inputbilder erhöhen lassen. Das Ziel der Begrenzung der Landmasse durch Wasser wurde dagegen nur teilweise erfüllt. Es wurden zwar Karten erstellt, auf denen Wasser mehrere Inseln umgibt, siehe Abbildung 5.2, ein zufriedenstellendes Ergebnis ist jedoch aufgrund fehlender Varianz der Inputbilder nicht zu generieren.

Auf die Einhaltung der nichtfunktionalen Anforderungen wurde großer Wert gelegt. Die Komponenten sind klar voneinander abgegrenzt und übersichtlich hinterlegt. Darüber hinaus ist der Prototyp durch die klare Schnittstellendefinition leicht erweiterbar.

5.2.2 Probleme und Erweiterungspotential

Anforderungen an den Input

Ein essentielles Problem bei der Nutzung des Prototyps sind die speziellen Anforderungen an den Input. Die Bilder, die als Input verwendet werden, müssen ein einheitliches, symmetrisches Format aufweisen. Mit dem gewählten Input beträgt das benötigte Format 128 x 128 Pixel. Wäre dieses Projekt in Zusammenarbeit mit einem professionellen UI-Artist entstanden, wären die Ergebnisse wesentlich überzeugender gewesen. Die Inputbilder hätten an den Bedarf angepasst werden können, wodurch eine viel dynamischere Karte möglich gewesen wäre. Auch das relativ große Inputformat hätte verkleinert werden können, um mehr Dynamik zu ermöglichen.

Manuelles Erstellen der Constraints-Datei

Ein weiteres Problem ist das manuelle Erstellen der Constraint-Datei. Da in der JSON-Datei für jedes Tile für jede Richtung genau definiert werden muss, welche anderen Tiles dort platziert werden dürfen (Nachbarschafts-Constraints), wird die Datei schnell sehr groß und unübersichtlich. Bei fehlerhaften Eingaben kann es passieren, dass der Algorithmus zu keinem Ergebnis kommt, da immer wieder Kontradiktionen auftreten. Eine mögliche Lösung für dieses Problem wäre die Entwicklung eines Editors, mit dem die Tiles für die Nachbarschafts-Constraints nur ausgewählt werden und automatisch in die JSON-Datei geschrieben werden.

User Interface

Um die Handhabung mit der Applikation für den Nutzer enorm zu erleichtern, wäre eine UI sinnvoll. Dies war im Rahmen dieser Arbeit aus zeitlichen Gründen nicht möglich. Mit einer UI ließe sich die Anwendung auch um zusätzliche Features erweitern, die dem Nutzer persönlichen Gestaltungsspielraum bieten würden. So könnte der Nutzer bestimmte Landschaften, bspw. Wald oder Gebirge, auf der Karte bereits vor Start des Algorithmus setzen, die dem WFC als Startpunkt für die Generierung dienen könnten. So könnte die Karte nach den persönlichen Wünschen des Nutzers generiert werden. Auch eine Auswahl bestimmter Tiles, die für die Karte verwendet werden sollen, wäre so denkbar.

5.3 Potentielle Anwendungsfelder

Wie bereits in Kapitel 2.3 erwähnt, findet PCG gerade in Videospielen Anwendung, um große Landschaften und ähnliche Videospielinhalte flexibel zu generieren und so den Aufwand für die Kreierung so niedrig wie möglich zu halten. Dieser Prototyp könnte, ggf. nach spezifischen Erweiterungen, die Landkarte für bspw. Open World-Spiele generieren. Besonders diese große Spiele stehen vor der Herausforderung, immer neue Inhalte zu erstellen, damit sich die Inhalte nicht zu oft wiederholen und damit eintönig werden[16]. So könnte das Gameplay stark verbessert werden. Aber auch in 2D gehaltene Spiele, die eine Mini-Map oder Übersichtskarte benötigen, könnten den Generator für ihre Zwecke verwenden. Dafür wäre es wahrscheinlich notwendig, dass ein UI-Artist entsprechende Input-Tiles entwirft. Je nach Bedarf könnte der Algorithmus entweder direkt ins Spiel integriert werden, sodass zur Laufzeit, also *online*, neue, zufällige Karten generiert werden, oder *offline* von den Entwicklern verwendet werden, sodass nur in der Entwicklung Karten generiert werden und dem Nutzer immer die gleiche Karte zur Verfügung steht.

Aber nicht nur in Videospielen könnte der Prototyp Verwendung finden. Auch manche Filme könnten von einer Fantasy-Landkarte profitieren. Mit den richtigen Inputbildern könnten generierte Karten, wie beispielsweise in der Filmreihe *Der Herr der Ringe* zum Einsatz kommen. Die aus diesem Kontext stammende Fantasy-Karte von *Mittelerde* diente diesem Prototypen auch als Inspiration, siehe Abb. 1.1. So könnte der Generator auch für Fanfiction in diesem Kontext genutzt werden.

6 Schluss

In dieser Arbeit wurde gezeigt, dass es möglich ist, mithilfe des Wave Function Collapse Algorithmus Landkarten zu erstellen. Die Qualität der Landkarten ist jedoch stark abhängig von folgenden Parametern:

- Größe und Vielfalt der Inputbilder
- Größe des zu generierenden Outputbildes
- Gewichtung der Inputbilder (Parameter weight)
- Definition der Nachbarschafts-Constraints

Allgemein kann gesagt werden, dass der Prototyp gute Ergebnisse liefert. Diese ließen sich jedoch mit vielfältigeren und qualitativeren Inputbildern stark verbessern.

Ein Manko des Prototyps ist die Benutzerfreundlichkeit bzw. Usability. Ein User Interface würde viel zur Benutzerfreundlichkeit beitragen und damit die Qualität der Applikation deutlich erhöhen. Besonders die manuelle Erstellung der Constraint-Datei ist mit großem Aufwand für den Nutzer verbunden und ließe sich, wie in Kapitel 5.2.2 erwähnt, mit einem Editor vereinfachen. Mit der Entwicklung eines Editors könnte sich in einer aufbauenden Arbeit befasst werden.

Abschließend kann gesagt werden, dass die Implementierung des WFC zur Erstellung von Landkarten erfolgreich umgesetzt wurde, es jedoch auch an einigen Stellen Erweiterungs- bzw. Verbesserungspotential gibt. Nach der Ergänzung um weitere Features könnte der Prototyp für die Generierung von Fantasy-Karten in Videospielen sowie Filmen oder im Kontext weiterer Kulturprojekte eingesetzt werden.

Literaturverzeichnis

[1] : *Vorgehensmodell Softwareentwicklung - Agile Entwicklung.* – URL https://wiki.mint-system.ch/vorgehensmodell-softwareentwicklung.html#agile-entwicklung

[2] : *No Man's Sky - Website.* 2016. – URL https://www.nomanssky.com/press/

[3] Bradbury, Gwyneth A. ; Choi, Il ; Amati, Cristina ; Mitchell, Kenny ; Weyrich, Tim: Frequency-based controls for terrain editing. In: *Proceedings of the 11th European Conference on Visual Media Production, 2014.* New York, USA, 2014, S. 1–10. – URL http://doi.acm.org/10.1145/2668904.2668944

[4] Dorn, Christian: *Generische Implementierung des Wave Function Collapse Algorithmus.* 2022

[5] Dzaebel, Ben: *Erweiterung des Wave Function Collapse Algorithmus zur Synthese von 3D-Voxel-Modellen anhand von nutzergenerierten Beispielen.* 2020. – URL https://users.informatik.haw-hamburg.de/~abo781/abschlussarbeiten/ba_dzaebel.pdf. – https://git.haw-hamburg.de/acg303/wfc

[6] Eckrich, Tobias M.: *Präzise Karte von Mittelerde aus Herr der Ringe.* – URL https://www.tolkiengesellschaft.de/17109/die-wohl-praeziseste-karte-mittelerdes/

[7] Efros, Alexei A. ; Leung, Thomas K.: Texture Synthesis by Non-parametric Sampling. In: *Proceedings of IEEE International Conference on Computer Vision.* Berkeley, California, USA, 1999. – URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.14&rep=rep1&type=pdf

[8] Freiberg, Sven: *Procedural Generation of Content in Video Games.* 2016. – URL https://users.informatik.haw-hamburg.de/~abo781/abschlussarbeiten/ba_freiberg.pdf

[9] Freiknecht, Jonas: *Procedural Content Generation for Games*. 2020

[10] Freiknecht, Jonas ; Effelsberg, Wolgang: A Survey on the Procedural Generation of Virtual Worlds. (2017). – URL https://www.mdpi.com/2414-4088/1/4/27/pdf

[11] Gumin, Maxim: *Maxim Gumin - Official Github page*. 2016. – URL https://github.com/mxgmn/WaveFunctionCollapse

[12] Heaton, Robert: *The Wave Function Collapse Algorithm explained very clearly*. 2018. – URL https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/

[13] Jenke, Philipp: *PCG repository des VR AR Labs der HAW Hamburg*. – URL https://git.haw-hamburg.de/vr_ar_lab/pcg

[14] Johansson, Gustav: *PCG in Game Bits and its Effect on Player Behaviour*. 2020

[15] Karth, Isaac ; Smith, Adam M.: WaveFunctionCollapse is Constraint Solving in the Wild. In: *Proceedings of FDG '17*. Hyannis, MA, USA, 2017. – URL https://doi.org/10.1145/3102071.3110566

[16] Kegel, Barbara D. ; Haahr, Mads: Procedural Puzzle Generation: A Survey. In: *IEEE Transactions on Games - Vol. 12*, IEEE, 2020, S. 21–40. – URL https://ieeexplore.ieee.org/document/8718565

[17] Kenney.nl: *Cartography pack by Kenney NL*. – URL https://opengameart.org/content/cartography-pack

[18] Kim, Hwanhee ; Lee, Seongtaek ; Lee, Hyundong ; Hahn, Taesung ; Kang, Shinjin: *Automatic Generation of Game Content using Graph-based Wave Function Collapse Algorithm*. 2019. – URL https://ieee-cog.org/2019/papers/paper_187.pdf

[19] Lindenmayer, Aristid ; Prusinkiewicz, Przemyslaw: Graphical modeling using Lsystems. (1990)

[20] Merrel, Paul C.: *Model Synthesis*. 2009. – URL http://www.cs.unc.edu/xcms/wpfiles/dissertations/merrell.pdf

[21] Merrell, Paul: *Comparing Model Synthesis and Wave Function Collapse*. 2021. – URL https://paulmerrell.org/wp-content/uploads/2021/07/comparison.pdf

[22] Parker, Joseph: *WFC Implementation for Unity.* – URL https://github.com/selfsame/unity-wave-function-collapse

[23] Roden, Timothy ; Parberry, Ian: *From artistry to automation: A structured methodology for procedural content creation.* Entertainment Computing–ICEC 2004, 2004. – 301–304 S

[24] Shaker, Noor ; Togelius, Julian ; Nelson, Mark J.: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016. – ISBN 978-3-319-42714-0

[25] Smelik, Ruben M. ; Tutenel, Tim ; Kraker, Klaas J. de ; Bidarra, Rafael: *A declarative approach to procedural modeling of virtual worlds.* Computers and Graphics, 2011

[26] Stiny, George ; Gips, James: Shape Grammars and the Generative Specification of Painting and Sculpture. In: *Information Processing 71.* Amsterdam, Niederlande : C V Freiman, 1972

[27] Stiny, George ; Knight, Terry: Classical and non-classical computation. (2001). – URL https://www.andrew.cmu.edu/user/ramesh/teaching/course/48-747/subFrames/readings/Knight&Stiny-2001-ARQ5-4_355-372.ClassicalAndNonClassicalComputation.pdf

[28] Togelius, Julian ; Kastbjerg, Emil ; Schedl, David ; Yannakakis, Georgios N.: What is procedural content generation? Mario on the borderline. In: *Proceedings of the 2nd Workshop on Procedural Content Generation in Games.* Copenhagen, Denmark, 2011. – URL http://julian.togelius.com/Togelius2011What.pdf

[29] Wichman, Glenn R.: *A Brief History of „Rogue“.* 1997. – URL http://web.archive.org/web/20071217204920/http://www.wichman.org/roguehistory.html

Erklärung zur selbstständigen 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