HAW HAMBURG

BACHELOR THESIS
Lasse Jahn Abrams

Erweiterung des Wave Function Collapse Algorithmus durch eine umgebungsabhängige dynamische Gewichtung

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

Lasse Jahn Abrams

Erweiterung des Wave Function Collapse Algorithmus durch eine umgebungsabhängige dynamische Gewichtung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Thomas Lehmann

Eingereicht am: 14. September 2023

Lasse Jahn Abrams

Thema der Arbeit

Erweiterung des Wave Function Collapse Algorithmus durch eine umgebungsabhängige dynamische Gewichtung

Stichworte

Wave Function Collapse, Prozedurale Content Generierung, Modellsynthese, Textursynthese, Constraint Solving

Kurzzusammenfassung

Der Wave Function Collapse Algorithmus dient dazu, aus einer Menge von Mustern und Regeln prozedural komplexe Bilder zu generieren. Obwohl er ursprünglich für die Textursynthese entwickelt wurde, findet er inzwischen Anwendung in der prozeduralen Content Generierung für die Spieleentwicklung. Der erzeugte Output weist die Eigenschaft der lokalen Ähnlichkeit auf. Diese Eigenschaft kann jedoch bei größeren Ausgabebildern zu homogenen Ergebnissen führen, was besonders in der prozeduralen Content Generierung unerwünscht ist. Die Zielsetzung dieser Bachelorarbeit besteht darin, den Algorithmus durch eine umgebungsabhängige dynamische Gewichtung zu erweitern. Diese Erweiterung berücksichtigt den Kontext der umliegenden Muster, um die Wahrscheinlichkeiten für die Auswahl zukünftiger Muster zu beeinflussen. Durch diese Methode weist der generierte Output eine erhöhte Vielfalt und Variation auf, sodass dieser für die Anwendung in der Spieleentwicklung besser geeignet ist.

Lasse Jahn Abrams

Title of Thesis

Extending Wave Function Collapse Algorithm with an environment-dependent dynamic Weighted Choice

Keywords

Wave Function Collapse, Procedural Content Generation, Model Synthesis, Texture Synthesis, Constraint Solving

Abstract

The Wave Function Collapse algorithm is used to generate procedurally complex images from a set of patterns and rules. Although it was originally developed for Texture Synthesis, it is being utilized in Procedural Content Generation for game development. The generated output has a trait of Local Similarity. This trait can lead to homogeneous results for larger output images, which is especially undesirable in Procedural Content Generation. The objective of this bachelor thesis is to extend the algorithm by adding an environment-dependent dynamic weighting. This extension takes into account the context of the surrounding patterns to influence the probabilities for selecting future patterns. Through this method, the generated output exhibits increased diversity and variation, making it more suitable for use in game development.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Von Videospielen wird verlangt, dass diese immer größer und detaillierter werden. Um dies zu erreichen benötigt ein Spiel Content. Die manuelle Erstellung von Content ist ein aufwändiger Prozess und somit mit großem Zeitaufwand und Kosten verbunden [9]. Daher wird immer häufiger Prozedurale Content Generierung (PCG) genutzt, um Content automatisch zu generieren. Hierbei werden zahlreiche verschiedene Algorithmen eingesetzt.

Ein Algorithmus der im Bereich der Prozeduralen Content Generierung angewendet wird, ist der Wave Function Collapse (WFC) Algorithmus. Der 2016 von Max Gumin veröffentlichte Algorithmus [8] ist ein beispielbasierter Algorithmus zur Bildgenerierung. Aus einer kleinen Input-Bitmap entnimmt der Algorithmus Muster und setzt diese anhand von Adjacency Constraints zu größeren lokal ähnlichen Output-Bitmaps zusammen [11]. Obwohl der Algorithmus anfänglich zur Textursynthese eingeführt wurde, hat dieser Anwendung im Game-Design gefunden.

1.2 Problemstellung und Ziel

Die Eigenschaft der lokalen Ähnlichkeit führt bei der Synthese von größerem Output zu gleichförmigen Ergebnissen. Dies geschieht, da Muster aufgrund der Adjacency Constraints nur durch angrenzende Muster beeinflusst werden. Zudem ist die Frequenz der Muster, wegen der globalen statischen Gewichte, über den gesamten Output ähnlich. Im ursprünglichen Anwendungsfall der Textursynthese ist dies eine gewünschte Eigenschaft. Das resultiert jedoch bei Anwendung zur Prozeduralen Content Generierung im Game-Design, aufgrund der Homogenität, zu uninteressanten Ergebnissen. Das ist dargestellt in Abbildung 1.1 (1).

Das Ziel der Arbeit ist die Erweiterung des WFC Algorithmus durch ein Konzept, welches das Problem der Gleichförmigkeit löst. Dazu wird ein Prototyp erstellt, der den modifizierten Algorithmus implementiert. Es werden Tilemap generiert, anhand dessen das neue Konzept evaluiert und verglichen werden kann.

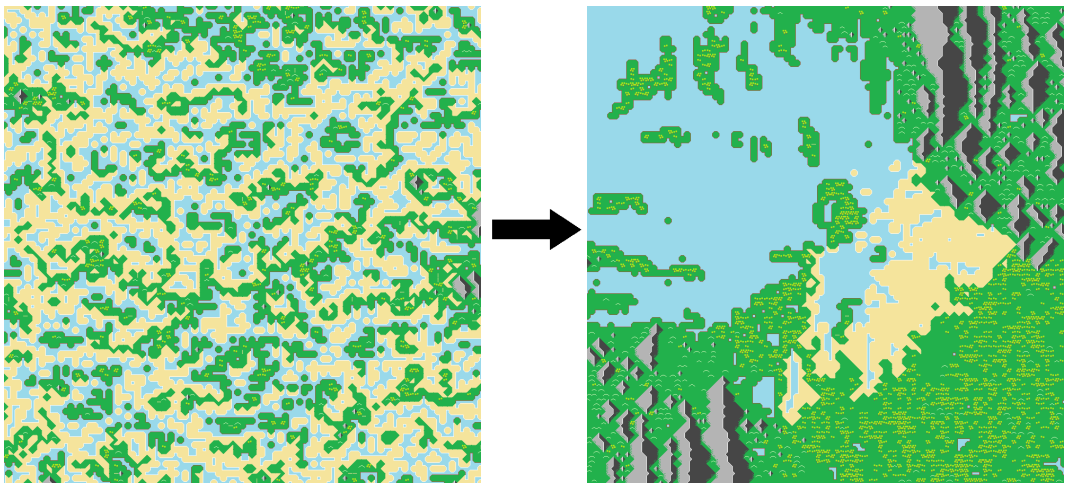

Abbildung 1.1: (1) stellt ein Ergebnis des unveränderten WFC Algorithmus dar. Die Anordnung der Tiles auf der Tilemap scheint willkürlich und homogen. (2) hingegen zeigt das Resultat des durch die umgebungsabhängige dynamische Gewichtung erweiterten WFC Algorithmus. Obwohl beide Tilemaps mit dem gleichen Tileset generiert wurden, ist (2) variabler, mit klar erkennbaren Strukturen.

1.3 Problemlösung

Im WFC wird beim Kollabieren einer Zelle die Auswahl eines Musters durch einen begrenzten Kontext beeinflusst. Der Kontext ist durch direkt benachbarte Muster mittels Adjacency Constraints und einer globale statische Gewichtung zur Auswahl der Muster gegeben. So ergibt sich die Eigenschaft der lokalen Ähnlichkeit im Output. Um das Problem der Gleichförmigkeit zu lösen, wird die lokale Ähnlichkeit abgeschwächt, indem der kollabierenden Zelle mehr Kontext zur Verfügung gestellt wird. Der erweiterte Kontext ergibt sich aus dem Beobachtungsraum. Dort werden allen bereits gesetzten Muster in einem Radius um die Zelle erfasst. Außerdem wird der Output in verschiedene Regionen eingeteilt. Aus dem erweiterten Kontext wird während der Laufzeit eine umgebungsab-

hängige dynamische Gewichtung berechnet, welche die globalen statischen Gewichtung zur Auswahl eines Musters ersetzt. Die Berechnung erfolgt auf Basis von Werten, die einem Regelsatz in Form einer formalen Grammatik entnommen werden. Abhängig von den umliegenden Mustern und der Region der Zelle werden individuelle Gewichte zur Auswahl eines Musters berechnet. Durch den umgebungsabhängigen Kontext ergeben sich interessantere Tilemaps, wie in Abbildung 1.1 (2) dargestellt.

1.4 Aufbau der Arbeit

Die Arbeit ist wie folgt aufgebaut. Das Kapitel **Stand der Technik** bietet einen umfassenden Überblick über die theoretischen Grundlagen. Es werden verschiedene Methoden der Prozeduralen Content Generierung erläutert. Dies umfasst die formalen Grammatiken, die Textursynthese und die Modellsynthese. Der Schwerpunkt liegt auf dem Wave Function Collapse.

Danach folgt das Kapitel **Konzept**. Die zugrunde liegende Problemstellung wird erläutert und ein Lösungskonzept dafür vorgestellt. Es werden zudem Anforderungen und Spezifikationen des entwickelten Prototyps, welcher das Lösungskonzept implementiert, ausgeführt.

In der **Umsetzung** wird das Vorgehensmodell erklärt und die Architektur und das Softwaredesign des Prototyps beschrieben.

Das Kapitel **Evaluation** zeigt die Ergebnisse auf. Es werden die Stärken, Schwächen und mögliche Verbesserungsmöglichkeiten des Ansatzes diskutiert. Zudem wird die Laufzeit des Algorithmus und der Benutzeroberfläche diskutiert.

Zum **Schluss** erfolgt eine Zusammenfassung der Arbeit und einen Ausblick auf potenzielle zukünftige Forschungsmöglichkeiten.

2 Stand der Technik

Dieses Kapitel beinhaltet die zugrunde liegenden theoretischen Grundlagen. Betrachtet werden Konzepte der Prozedurale Content Generierung (PCG), mit Fokus auf den in dieser Arbeit angewendeten Wave Function Collapse Algorithmus.

2.1 Prozedurale Content Generierung

Prozedurale Content Generierung (PCG) ist die automatische Erstellung von Content mittels Algorithmen durch begrenzten oder indirekten menschlichen Input [33]. Je nach zu erzeugendem Content gibt es zahlreiche verschiedene Methoden zur prozeduralen Generierung.

2.2 Formale Grammatiken

Eine Grammatik ist definiert durch ein 4-Tupel $G = (N, \Sigma, P, S)$.

1. N ist eine endliche Menge an nichtterminalen Symbolen
2. Σ ist eine endliche Menge an terminalen Symbolen
3. P ist eine endliche Menge an Regeln (Produktionen)

   $((\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*)$
4. S ist das Startsymbol

In jeder Iteration wird ein nichtterminales Symbol durch neue Symbole, beschrieben durch eine Produktionsregel, ersetzt. Durch das sequenzielle Ersetzen der Symbole durch Produktionen, werden als Sätze bezeichnete Ketten aus terminalen Symbolen erzeugt. Obwohl formale Grammatiken zunächst zur Modellierung natürlicher Sprache eingeführt wurden, haben diese in der PCG weit verbreitete Anwendung gefunden [31].

Kontextfreie Grammatik

Eine Art der formalen Grammatiken sind die kontextfreien Grammatiken. Die Produktionsregeln sind definiert durch $A \rightarrow \alpha$. A ist ein einzelnes nichtterminales Symbol und α ein String beliebiger terminaler und nichtterminaler Symbole. In der Informatik haben kontextfreie Grammatiken zahlreiche Anwendungen. Im Folgenden werden zwei genannt. Sie werden zur Repräsentation und Generierung von kontextfreien Daten genutzt, einschließlich der Modellierung von JSON-Datenstrukturen [24]. Sie können rekursiv Zeichenketten erzeugen, die zur Beschreibung des Ablaufes eines Videospiellevels genutzt werden [7].

L-Systeme

L-Systeme wurden 1968 von dem Biologen Aristid Lindenmayer eingeführt zur Modellierung des organischen Wachstums von Pflanzen. L-Systeme sind eine Art formale Grammatik. Die Anwendung der Produktionsregeln geschieht nicht mehr sequenziell. Pro Iterationsschritt werden alle Symbole des Satzes parallel ersetzt [31].

Durch das Interpretieren der Symbole eines Satzes als Instruktionen, werden typischerweise Fraktale oder Pflanzen modelliert.

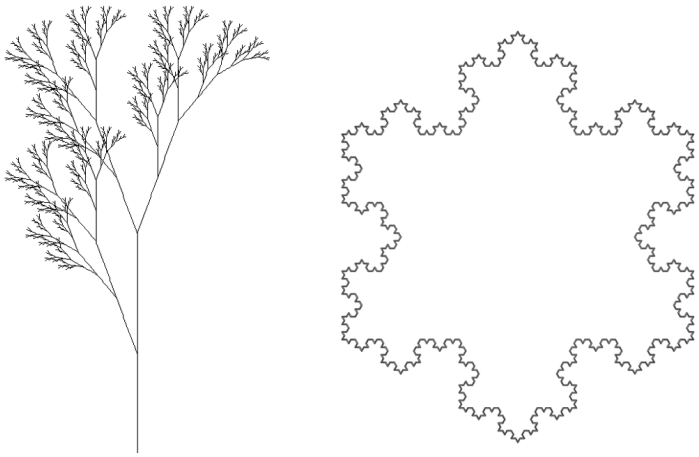

Abbildung 2.1: Baum und Fraktal generiert mittels L-System Grammatik.[1]

Shape-Grammar

Shape-Grammatiken sind formale Grammatiken, die zur Beschreibung von zweidimensionalen und dreidimensionalen Formen dienen. Statt Symbolen der formalen Grammatik werden Figuren verwendet. Beginnend mit der Startfigur wird diese mittels der Produktionsregeln durch eine neue Figur rekursiv ersetzt. Die neue Figur besteht immer aus der vorhandenen Figur und den Figuren der linken und/oder rechten Seite [32]. Ein häufiger Anwendungsfall ist die prozedurale Generierung von Gebäuden [22].

2.3 Textursynthese

Textursynthese ist die automatische Generierung neuer Texturen aus einem Input-Sample. Die generierten Texturen sind von beliebiger Größe und äquivalent zum Input. Die Textursynthese wird in zwei Kategorien eingeteilt: die parametrischen Methoden und die nichtparametrischen Methoden.

Die parametrische Methode läuft wie in Abbildung 2.2 dargestellt ab. Zunächst werden aus dem Input-Sample eine Menge von Charakteristiken, auch Statistiken genannt, entnommen. Diese definieren die zugrunde liegende Struktur der Textur. Die Statistiken werden auf ein Sample von additiven weißen gaußschen Rauschen angewendet, sodass der

[1] Bildquelle: http://paulbourke.net/fractals/lsys/ (Letzter Zugriff: 27.07.2023)

synthetisierte Output die gleichen statistischen Eigenschaften wie der Input erhält. Es gibt zahlreiche Algorithmen zur Extraktion der Statistiken und wie diese auf den Output angewendet werden. Insbesondere die Textursynthese unter Verwendung von Convolutional Neural Networks fällt in diese Kategorie [27].

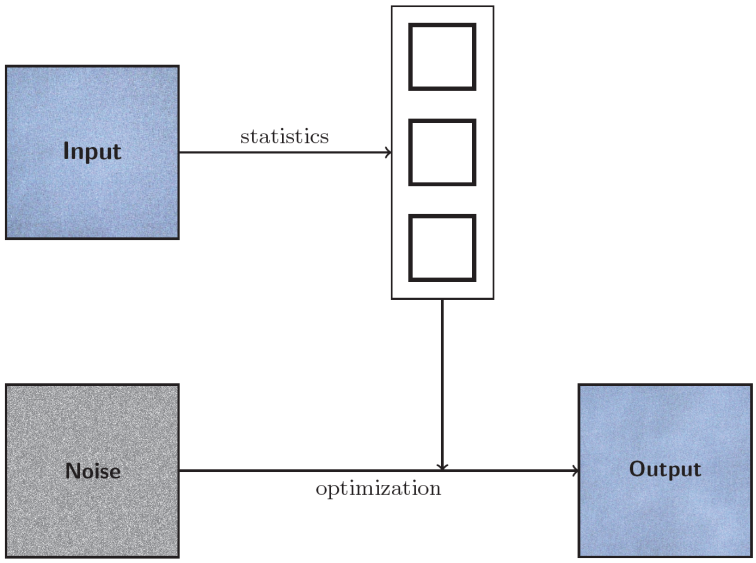

Abbildung 2.2: Ablauf der parametrischen Textursynthese [27].

Im Gegensatz zu den parametrischen Methoden wird der Input in den nichtparametrischen Methoden nicht mit Statistiken charakterisiert. Neue Texturen werden durch Neuordnung von Teilen des Inputs synthetisiert [27].

Eine der bedeutendsten Methoden der nichtparametrischen Textursynthese ist das *Non-parametric Sampling* von Elfros und Leung [3]. Ein Pixel p wird als Ausgangspunkt ausgesucht, von dort aus wird eine neue Textur Pixel um Pixel nach außen hin synthetisiert. Beim Synthetisieren eines Pixels wird die $N \times N$ große lokale Nachbarschaft betrachtet und bereits synthetisierte Pixel in dem Fester als Kontext erfasst. Daraufhin werden in der Input-Textur Pixel mit einem ähnlichen Kontext in der lokalen Nachbarschaft gesucht. Der zu synthetisierende Pixel nimmt den Farbwert eines Pixels an, der einen ähnlichen Kontext besitzt [3].

Basierend auf der pixelbasierten Synthese wurden zahlreiche Methoden entwickelt, die Texturen aus Patches zusammensetzen. Patches sind dem Input entnommene Bildaus-

schnitte. Eine dieser Methoden ist Elfros und Freeman's Image Quilting. Anstatt schrittweise Pixel zu synthetisieren, werden schrittweise Patches nebeneinander überlappend angeordnet. Zur Auswahl, welcher Teil eines Patches in der Textur genutzt wird, wird ein Schnitt zwischen den überlappenden Blöcken, innerhalb der überlappenden Fläche gemacht. Der Schnitt geschieht entlang der Minimal Error Boundary, der Weg an dem die minimale Abweichung zwischen den Pixeln beider Patches vorhanden ist. So wird ein nahtloser Übergang sichergestellt [4].

2.4 Modellsynthese

In seiner Dissertation *Model Synthesis* [19] aus dem Jahr 2009 hat Paul Merrell die gleichnamige Modellsynthese vorgestellt. Die Modellsynthese baut auf den Ideen der nichtparametrischen Textursynthese auf und passt diese an die Anforderungen der 3D-Modellierung an. Eine Synthese im zweidimensionalen Raum ist weiterhin möglich.

Die erste von zwei Methoden ist die *Diskrete Modellsynthese*. Dem Input-Modell werden diskrete Modellstücke, sogenannte *model pieces* entnommen. Ist das Input-Modell zweidimensional wird dieses durch ein Gitter in quadratische *model pieces* eingeteilt. Ein dreidimensionaler Input wird in gleich große Würfel als *model pieces* unterteilt. Identische *model pieces* werden logisch als eines behandelt. Jedes unterschiedliche *model piece* besitzt ein korrespondierendes *label*.

Aus dem Input-Modell werden für jedes *model piece* die Adjacency Constraints abgeleitet. Adjacency Constraints beschreiben, wie die *model pieces* relativ zueinander platziert werden dürfen.

Die *model pieces* dienen als Grundbausteine zur Generierung eines beliebig großen Output-Modells. Das Output-Modell soll zwei Bedingungen erfüllen. Es soll dem Input-Modell ähnlich sein und es soll konsistent sein. Ein Modell gilt als konsistent, wenn alle Adjacency Constraints erfüllt sind, und es somit keine Konflikte in der Anordnung gibt [18].

Im Folgenden wird anhand Abbildung 2.4 erklärt, wie die diskrete Modellsynthese im 2-dimensionalen Raum abläuft.

1. Aus dem Input-Modell (Example Model) werden *model pieces* und korrespondierende *labels* erstellt. Zudem werden die Adjacency Constraints abgeleitet

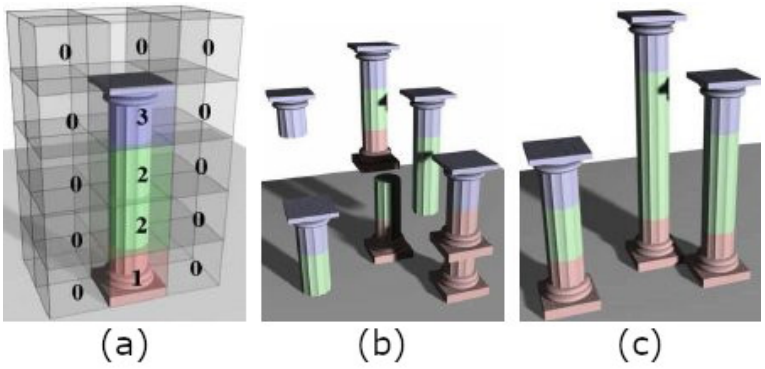

Abbildung 2.3: Eine Säule ist in *model pieces* aufgeteilt (a). Diese sind zufällig angeordnet (b). Sie sind entsprechend der Adjacency Constraints angeordnet (c) [18].

2. In dem unvollständigen Output-Modell (M) wird die Domäne jeder Zelle mit dem *label* jedes *model piece* gefüllt.

3. Dieser Schritt geschieht in einer Schleife, bis in der Domäne jeder Zelle genau ein *label* ist:

   a) Wähle eine Zelle für den Output. Entferne aus der Domäne der Zelle alle *labels* außer einem *label*. ($C_0(M)$)

   b) Die Propagierung der Constraints geschieht durch einen Constraint-Satisfaction-Ansatz, den Arc Consitency Algorithmus #3 [16]. Der Algorithmus propagiert rekursiv über alle Zellen. Es werden alle *labels* aus der Domäne einer Zelle entfernt, welche die Adjacency Constraints der möglichen *labels* der benachbarten Zellen nicht erfüllen ($C_1(M)$, $C_2(M)$). Letztendlich bleiben nur *labels* übrig, welche die Regeln erfüllen ($C(M)$).

Die zweite Methode ist die *Kontinuierliche Modellsynthese*. Sie wird angewendet, falls die *model pieces* des Inputs nicht in ein Gitter passen.

Als Input wird ein geschlossenes polyedrisches Objekt gegeben. Basierend auf den Ebenen des Objektes werden parallele Ebenen erstellt. Aus deren Schnittpunkten ergibt sich eine Menge an möglichen Kanten und Vertices, die äquivalent zu Zellen der *kontinuierlichen Modellsynthese* fungieren. So wie die Domäne einer Zelle zu Beginn alle *model piece* enthält, besteht die Domäne der Kanten und Vertices aus allen im Input-Objekt vorkommenden Anordnungen. Adjacency Constraints sind durch Nachbarschaftsverhältnisse von

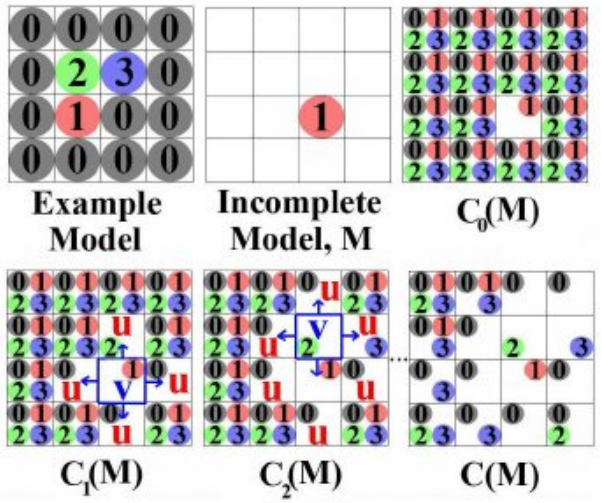

Abbildung 2.4: Ein Iterationsschritt der diskreten Modellsynthese [18].

Kanten und Vertices im Input abzuleiten. Die Generierung läuft auch äquivalent zur *kontinuierliche Model Synthese* ab. In einer Schleife wird jeweils eine Kante oder Vertices kollabiert und Adjacency Constraints propagiert. Am Ende ergibt sich ein vollständig kollabiertes Output-Modell, welches dem Input-Modell ähnlich ist [21].

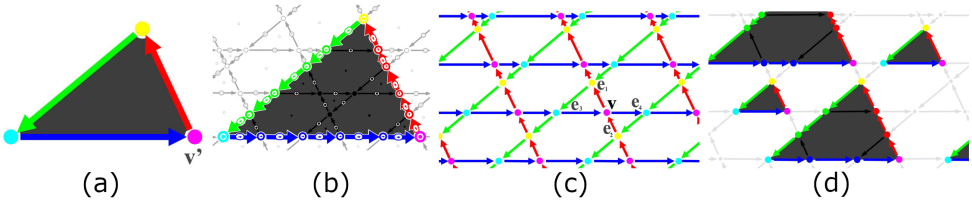

Abbildung 2.5: Kontinuierliche Modellsynthese in 2D. Input Modell besteht aus Kanten und Vertices (a und b). Aus den Parallelen Linien ergibt sich eine Menge an möglichen Kanten und Vertices (c). Aus dieser Menge wird der Output generiert [21].

2.5 Wave Function Collapse

Der Wave Function Collapse (WFC) Algorithmus wurde 2016 von Maxim Gumin veröffentlicht [8] und hat seitdem viel Aufmerksamkeit auf sich gezogen. In verschiedenen Indie-Spielen hat der Algorithmus seitdem Anwendung gefunden (s. 2.8).

Der WFC beruht auf Paul Merrels diskreter Modellsynthese. Wie bei der Modellsynthese werden aus einem Input *model pieces* und die zugehörigen Adjacency Constraints entnommen. *model pieces* werden im WFC Muster genannt. Daraus wird ein Output generiert, der dem Input ähnlich ist.

Gumin beschreibt dies als lokale Ähnlichkeit. Das bedeutet, dass im Output nur solche Muster vorhanden sein werden, die auch im Input präsent sind und die Häufigkeit der Muster im Output soll ähnlich der im Input sein.

Es gibt jedoch einige wesentliche Unterschiede. Das Overlapping-Model des WFC (s. 2.5.1) arbeitet mit überlappenden Mustern aus $N \times N$ Zellen anstelle von *model pieces* aus einzelnen Zellen. Außerdem wird zur Auswahl der Zelle, die kollabiert wird, die Heuristik der niedrigsten Entropie genutzt. Zudem wurde ein Tile-Symmetrie System zur Verkleinerung des Inputs, bzw. Vergrößerung des Tilesets eingeführt [8].

Üblicherweise wird der WFC im zweidimensionalen angewendet, sodass die Muster quadratischen Tiles entsprechen, die zu Tilemaps zusammengeführt werden.

Der WFC Algorithmus kann in vier zentrale Aufgaben aufgeteilt werden. Der erste Schritt ist das Entnehmen der Muster aus dem Input. Als Nächstes werden Adjacency Constraints für die entnommenen Muster erstellt. Der dritte und vierte Schritt geschehen in einer Schleife und stellen die eigentliche Generierung des Outputs dar. Es wird eine Zelle kollabiert und Veränderungen in den Zellen durch Constraints auf die benachbarten Zellen propagiert [11].

Algorithm 1 Wave Function Collapse

```
Musterentnahme()                          ▷ Vorbereitung 2.5.1
Constrainterstellung()                    ▷ Vorbereitung 2.5.1
while nicht abgeschlossen do
    Observierung()                        ▷ Observierung 2.5.2
    Propagierung()                        ▷ Propagierung 2.5.3
end while
```

Diese Aufgaben und Variationen werden in den folgenden Sektionen, basierend auf der Beschreibung des WFC Algorithmus durch Karth und Smith [12], erläutert.

2.5.1 Vorbereitung

Musterentnahme

Gegeben ist ein Sample-Input. Aus dem Bild werden Muster als Grundbausteine zur Generierung des Outputs entnommen. Unterschieden wird hierbei zwischen zwei Modellen.

Overlapping Model Mittels eines *sliding window*, welches Zelle um Zelle über den Input fährt, werden alle $N \times N$ großen überlappende Muster dem Input entnommen. Duplikate werden gezählt und entfernt. Die entnommenen Muster werden gegebenenfalls durch deren Spiegelungen und Rotationen ergänzt.

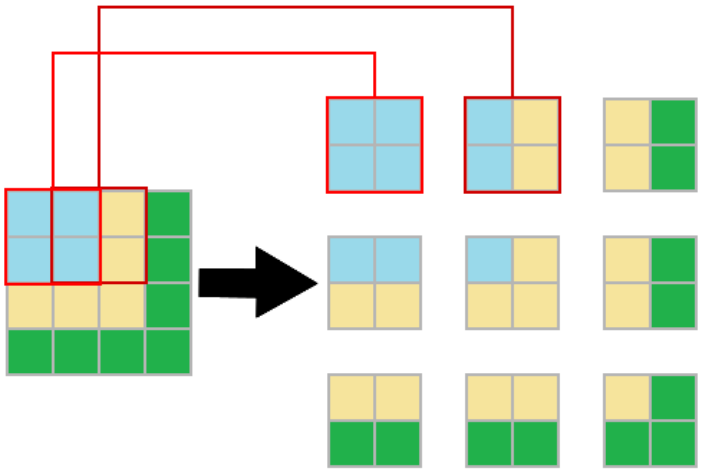

Abbildung 2.6: Extraktion der Muster aus einem Input mit N=2 beim Overlapping Model.

Simple Tiled Model Dieses Modell gleicht der diskreten Modellsynthese. Der Input wird in ein Gitter eingeteilt und alle Muster in den Zellen werden erfasst. Duplikate werden gezählt und entfernt. Auch hier besteht die Möglichkeit, die erfassten Muster durch deren Symmetrien und Rotationen zu erweitern.

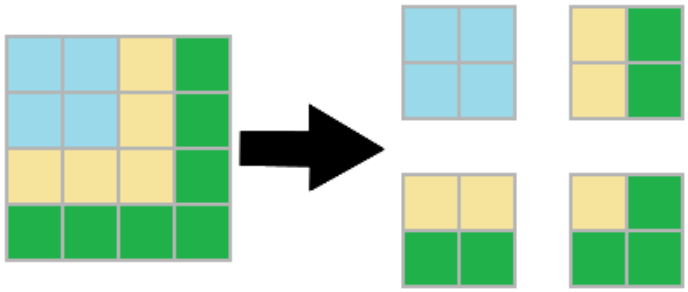

Abbildung 2.7: Extraktion der Muster aus einem Input mit N=2 beim Simple Tiled Model.

Constrainterstellung

Aus der Menge der erfassten Muster wird eine Index-Datenstruktur erstellt. In der Datenstruktur werden die Adjacency Constraints für jedes Muster beschrieben. Auch hier wird zwischen den zwei Modellen unterschieden.

Overlapping Model In diesem Modell wird der Input nicht zur Erstellung der Regeln genutzt. Eine Anordnung der Tiles ist valide, wenn die Überschneidung dieser bei einem Offset von x, y übereinstimmen.

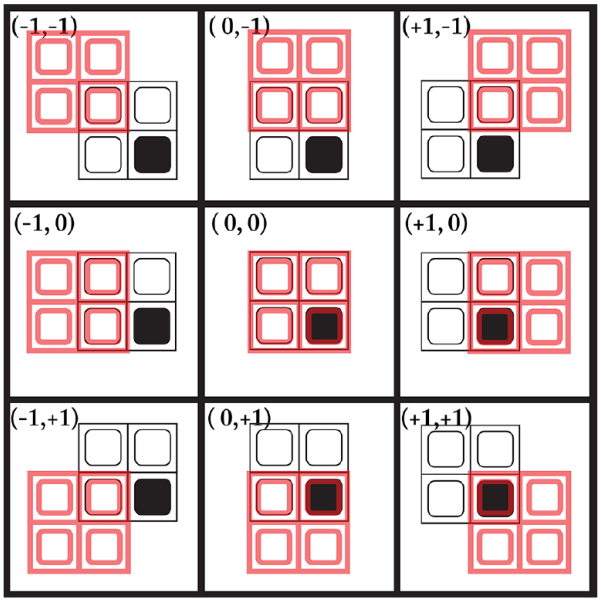

Abbildung 2.8: Es gibt neun Anordnungen, wie sich zwei 2×2 Muster überlappen können [11].

Simple Tiled Model Eine Anordnung ist valide, wenn die gleiche Nachbarschaftsbeziehung zweier Muster im Input einmal vorkam. Dies gleicht der Adjacency Constraints der diskreten Modellsynthese.

Ohne Input-Image

Obwohl der WFC von der Textursynthese inspiriert ist, kann Output mit dem Simple Tiled Model ohne Input-Sample generiert werden. Der Schritt der Musterentnahme aus dem Input-Sample wird ersetzt durch die manuelle Angabe von Mustern. Im Pattern Building Process werden die Adjacency Constraints nicht auf Basis des Inputs erstellt, sondern werden entweder manuell konfiguriert oder auf Basis von angegebenen Kanten berechnet (s. 2.6) [23].

2.5.2 Observierung

Zunächst wird die Position mit der geringsten Entropie gesucht. Im Sinne des WFC bedeutet Entropie die Anzahl der möglichen Muster, die in der Domäne einer Zelle sind.

Wenn keine Zelle bisher kollabiert wurde, entspricht die Entropie jeder Zelle der Anzahl aller Muster.

Wurde die Zelle mit dem minimalen Entropiewert gefunden, wird diese kollabiert. Eines der Muster in der Domäne der Zelle wird durch eine gewichtete zufällige Auswahl bestimmt. Die Gewichtung ergibt sich durch die Frequenz des Musters im Input-Sample. Falls mehrere Zellen den gleichen minimalen Entropiewert aufweisen, wird eine zufällige Zelle mit minimaler Entropie ausgewählt und kollabiert.

2.5.3 Propagierung

Der Propagierungsschritt des WFC gleicht dem der Modellsynthese und implementiert auch den Arc Consistency Algorithmus #3. Ist eine Zelle kollabiert, so wird diese markiert. Von der Markierung aus werden die Adjacency Constraints in die benachbarten Zellen propagiert. Entsprechend der Adjacency Constraints werden Muster aus der Liste der möglichen Muster entfernt. Wenn ein Muster aus einer Zelle entfernt wird, wird diese Zelle markiert. Von dort aus wird mit der aktualisierten Menge an Mustern weiter in benachbarte Zellen propagiert. So wird die Menge der möglichen Muster jeder Zelle aktualisiert.

2.5.4 Ausnahmebehandlung

Im WFC kann es geschehen, dass Zellen nach der Propagierung eine Entropie mit dem Wert 0 haben. Das bedeutet, dass es kein Muster für diese Zelle gibt, welches alle Constraints erfüllt. Die Zelle kann somit nicht kollabiert werden. Wenn das geschieht, wird der WFC neu gestartet. Das kann zu Problemen führen, weil bei einer großen Anzahl an Constraints der Algorithmus immer öfter in Konflikte laufen wird. Das würde bei der Einführung weiterer Constraints (s. 2.7.1) oder bei einer stark durch Adjacency Constraints eingeschränkten Tilesets geschehen.

Eine Möglichkeit damit umzugehen ist das Backtracking. Tritt ein Fehler auf, wird in einen früheren Zustand zurückgegangen und ein anderer Pfad gewählt. Hatten mehrere Zellen beim Kollabieren die geringste Entropie, kann eine andere Zelle kollabiert werden. Zudem kann beim Kollabieren einer Zelle ein anderes Muster ausgewählt werden.

In der diskreten Modellsynthese hat Paul Merrell das Problem durch das *Modifying in Parts* gelöst. Die Methode ist auch auf den WFC anwendbar. Er hat festgestellt, dass die Erfolgsrate bei der Generierung mit vielen Constraints immer schlechter wird, je größer der Output ist. Beim *Modifying in Parts* wird nur einen $N \times N$ großes Fenster des Outputs bearbeitet. Dieser kann schnell und mit hoher Erfolgsrate gelöst werden. Das Fenster wird schrittweise über den gesamten Output verschoben und die Zellen innerhalb des Fensters werden kollabiert [19].

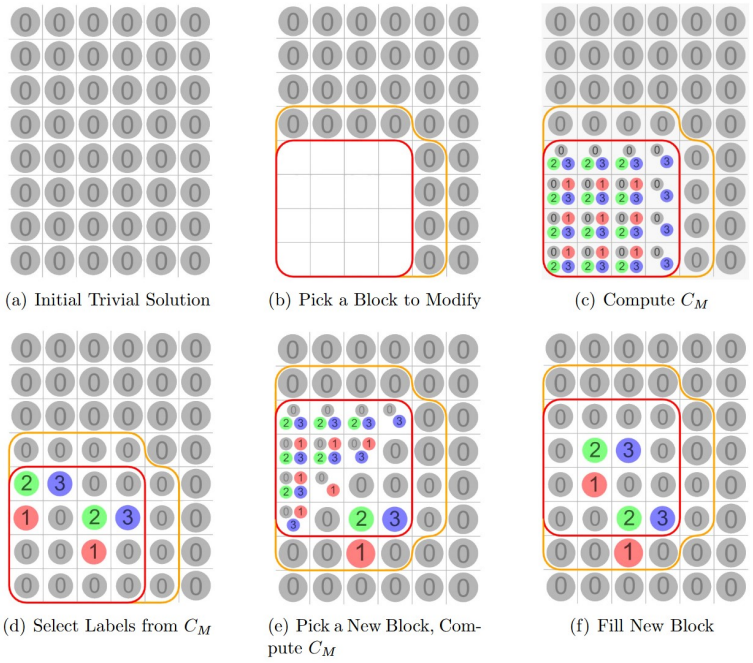

Abbildung 2.9: Modifying in Parts nach Merrell [19].

2.6 Automatische Tile Erstellung

Um mehr Variation und Auswahlmöglichkeiten der Tiles zu bekommen, lassen sich diese durch deren Transformationen erweitern. Die Erstellung der Constraints im Overlapping Model bleibt unverändert. Im Simple Tiled Model müssen Adjacency Constraints generiert werden. Diese können nicht vollständig aus dem Input abgeleitet werden, da die durch Transformationen erstellten Tiles nicht im Input vorkommen.

2.6.1 Symmetrie

In Abbildung 2.10 sind die acht möglichen Transformationen eines Tiles bei fixierter Orientierung angegeben, und mit einem Index markiert.

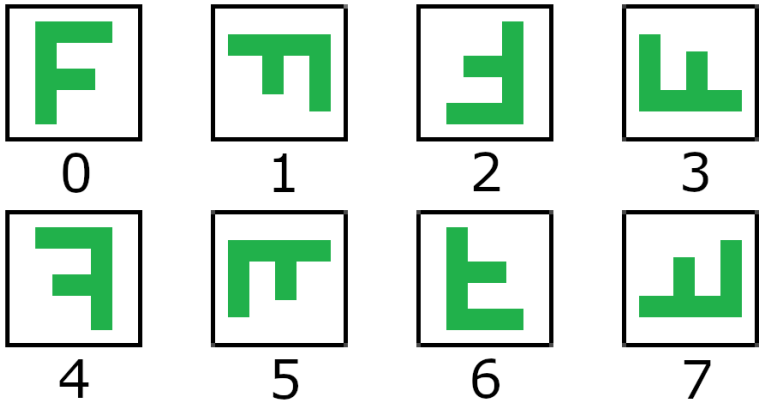

Abbildung 2.10: Das Tile ist um 0°, 90°, 180° und 270° rotiert (0-3). Das Spiegelbild ist ebenfalls um 0°, 90°, 180° und 270° rotiert (4-7).[2]

Abhängig von der Symmetrie eines Tiles, muss dieses unterschiedlich oft transformiert werden, um alle Anordnungen zu erstellen. In Tabelle 2.1 sind die benötigten Transformationen für alle Symmetriearten angegeben [1].

Symmetrie	Abkürzung	Benötigte Transformationen
Keine	F	0-7
Vertikale oder Horizontale Achse	I	0-3
Vertikale und Horizontale Achse	T	0-1
Diagonale Achse	L	0-3
Beide diagonale Achsen	/	0-1
Alle Achsen	X	0

Tabelle 2.1: Alle möglichen Tile-Symmetrien

2.6.2 Matching

Um Adjacency Constraints automatisch zu konfigurieren, benötigt jede Kante ein Kantenlabel. Passen die Labels der Kanten zweier Tiles zueinander, dürfen diese nebenein-

[2] Abbildung [1] nachempfunden

ander platziert werden [23]. Für jede Kante eines Tiles werden alle Tiles, bei denen entsprechenden Kantenlabels übereinstimmen, als Adjacency Constraints erfasst.

2.7 Erweiterungen des WFC

Aufgrund seiner Popularität wurden zahlreiche Erweiterungen des WFC entwickelt. Erweiterungen ermöglichen es den Output des Algorithmus an die entsprechenden Use-Cases anzupassen. So kann der Benutzer mehr Kontrolle über die Generierung erhalten oder die Laufzeit verbessern. Im Folgenden werden einige der Erweiterungen genannt.

2.7.1 Constraints

Zusätzlich zu den lokalen Adjacency Constraints können nichtlokale Constraints eingeführt werden. In der Observations-Propagierungs-Schleife werden nichtlokalen Constraints zu beliebigen Zelle propagiert [28]. Cheng et al. erweiterten den WFC Algorithmus um nichtlokale Constraints.

Globale Constraints begrenzen die maximale Anzahl eines bestimmten Musters und/oder geben eine minimale Anzahl vor, die ein Muster im Output vorkommen muss.

Distance Contrainst etablieren eine minimale und/oder maximale Distanz, die von einem Muster zu einem anderen Muster bestehen muss [1].

In Tessera werden **Path Contraints** eingeführt, die mittels Pfaddiagrammem die Konnektivität zwischen Tiles sicherstellen [23].

2.7.2 Lokationsheuristik

Zur Auswahl der zu kollabierenden Zelle hat Gumin die Heuristik der geringsten Entropie gewählt. Ihm ist aufgefallen, dass Menschen oftmals selber dieser Heuristik folgen. [8] Ein Beispiel dafür ist das Zahlenrätsel Sudoku. Bei Sudoku sucht der Spieler zunächst immer die Felder mit den wenigsten möglichen Zahlen.

Unterschiedliche Heuristiken haben einerseits verschiedene Fehlerraten. Heuristiken, die Zellen einer wachsenden Region zusammenhängender Muster kollabieren, haben eine ähnliche niedrige Fehlerrate. Das sind die Heuristiken *Lowest-Entropie*, *Spiral* und *Lexical*.

Heuristiken, die über der gesamten Tilemap hinweg Zellen kollabieren, wie *Anti-Entropie* und *Hilbert*, haben im Vergleich eine schlechtere Fehlerrate. Andererseits können verschiedene Heuristiken vor allem durch richtungsabhängige Details einen ästhetischen Einfluss auf das Ergebnis haben [12].

2.7.3 Musterheuristik

Im WFC wird beim Kollabieren einer Zelle ein valides Muster zufällig ausgewählt. Die zufällige Auswahl ist gewichtet nach der Häufigkeit die ein Muster im Input vorkommt. Die Gewichtung kann auch manuell angegeben werden, um den Benutzer mehr Einfluss auf den Output zu geben [15]. Die gewichtete Heuristik nennt sich *weighted-choice*. Karth und Smith stellen zusätzlich die Heuristiken *rarest*, *common* und *lexical* vor [12].

2.7.4 Räumliche Einteilung

Der Output kann räumlich eingeteilt werden. Langendam und Bidarra teilen den Output mittels Heatmap ein, um räumlich variierende Gewichte zur Auswahl der Muster zu nutzen [15].

Alternativ kann der Output in Chunks eingeteilt werden, das wird genutzt um Run-Time Generierung zu ermöglichen. Bis auf das Propagieren der Contraints auf benachbarte Chunks, können diese individuell gelöst werden. Dementsprechend ist es möglich Gewichte zur Tileauswahl in den Chunks individuell zu setzen [30]. Auch die Lokationsheuristiken, Musterheuristiken und nichtlokale Constraints können individuell angepasst werden.

2.8 Anwendungen des WFC

Der WFC Algorithmus kann an verschiedene Anforderungen angepasst werden und bietet somit eine breite Masse an Anwendungsmöglichkeiten.

Eingeführt wurde der Algorithmus zur Erzeugung zweidimensionaler Bitmaps und Tilemaps um, zweidimensionale Texturen zu erzeugen. So wurde zum Beispiel Game-Content im Spiel Rodina [5] texturiert. Caves of Quid generiert aus Tiles zweidimensionale Tilemaps als Spielewelt [6].

Erweiterungen des WFC ermöglichen die Generierung von dreidimensionalem Content. Mit dem Overlapping Model werden Modelle aus Voxeln generiert [2]. Stålberg generiert im Spiel Bad North kleine Inseln aus einem Tileset an dreidimensionalen Modulen [26].

Abbildung 2.11: Mit dem WFC Algorithmus generierte Insel aus dem Spiel Bad North[3].

2.9 Modellierung mit Graph-Grammatiken

Die Modellsynthese und der WFC sind Algorithmen, welche sich an der nichtparametrischen Textursynthese orientieren. Die diskrete Modellsynthese und der WFC basieren auf ein Gitter. Muster müssen auf das Gitter passen, wodurch die Variabilität der erzeugbaren Formen begrenzt wird.

Der Fokus weiterer Forschung liegt vor allem darin sich von der Gitterdarstellung zu lösen. Die kontinuierliche Modellsythese ist eine Methode ohne Gitter auszukommen. Ein neuer Ansatz ist die beispielbasierte prozedurale Modellierung mit Graph-Grammatiken [20]. Das Ziel dieser Methode ist weiterhin dasselbe: aus einem Input-Sample soll ein ähnlicher Output erzeugt werden. Jedoch wird der Input und der Output werden als zweidimensionale oder dreidimensionale Formen durch Graphen dargestellt.

In Abbildung 2.12 ist der Ablauf der beispielbasierte prozedurale Modellierung mit Graph-Grammatiken in den drei folgenden Schritten dargestellt.

Im ersten Schritt werden Muster, sogenannte Primitives, aus dem Input entnommen. Primitives sind die individuellen Graphen eines Inputs, bei dem jede Kante halbiert

[3]Bildquelle: https://store.steampowered.com/app/688420/Bad_North_Jotunn_Edition/ (Letzter Zugriff: 04.07.2023)

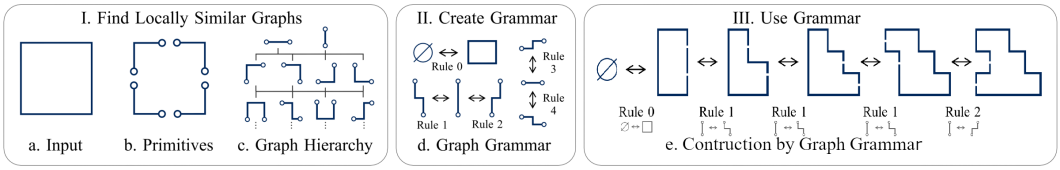

Abbildung 2.12: Ablauf der beispielbasierte prozedurale Modellierung mit Graph-Grammatiken [20].

wurde. Aus den Primitives wird eine Graph-Hierarchie erstellt, welche jede mögliche Kombination von n Primitives beinhaltet.

Im zweiten Schritt werden aus der Hierarchie Produktionsregeln einer Grammatik abgeleitet. Diese Produktionsregeln können alle Graphen der Hierarchie produzieren.

Im dritten Schritt wird die erstellte Grammatik angewendet, um neue Formen zu erstellen. Beginnend mit einem leeren Graphen werden iterativ Produktionsregeln zufällig angewandt. Die Generation kann jederzeit beendet werden und es ergibt sich je nach Iterationstiefe ein beliebig komplexer Graph. Zur Darstellung muss die Position der Vertices bestimmt werden. Es wird eine zufällige Lösung berechnet, bei der sich die Kanten nicht schneiden.

Am Ende ergibt sich ein zum Input lokal Ähnlicher Output. Er enthält also nur Primitives die auch im Input vorkommen [20].

3 Konzept

Das Ziel der Arbeit ist die Synthese von Tilemaps aus einem Tileset, die als Game-Space genutzt werden können. Generierte Tilemaps sollen interessant sein und das Problem der Gleichförmigkeit, welches sich bei der Synthese großer Tilemaps mittels des WFC-Algorithmus herausbildet, lösen. Im folgenden Kapitel wird die Problemstellung dargelegt und ein Konzept zur Lösung des Problems vorgestellt. Zudem werden Anforderungen und Spezifikationen des in dieser Arbeit entwickelten Prototyps erläutert.

3.1 Problem

Eine Eigenschaft des unmodifizierten WFC ist die lokale Ähnlichkeit, die Gumin wie folgt definiert hat:

- Der Output beinhaltet nur $N \times N$ Muster, die auch im Input vorhanden sind
- Die Frequenz eines Musters im Output, soll der im Input ähnlich sein

Wird der WFC so konfiguriert, dass er ohne Input arbeitet, dann ist diese Definition der lokalen Ähnlichkeit nicht mehr zutreffend. Eine passende Neudefinition lautet wie folgt:

- Der Output beinhaltet nur Muster, die vorgegeben wurden
- Die Frequenz eines Musters im Output, soll den vorgegebenen Gewichten entsprechen

Beide Definitionen sind durch eine weitere Eigenschaft zu erweitern:

- Im Output werden Muster durch benachbarte (lokale) Muster mittels Adjacency Constraints eingeschränkt.

In der Textursynthese ist die lokale Ähnlichkeit eine gewünschte Eigenschaft, da sie es ermöglicht, ähnliche lokale Strukturen über die gesamte Textur hinweg zu synthetisieren. Aber insbesondere bei der Anwendung des WFC im Level-Design führt dies zu uninteressanten Ergebnissen [23]. Auf lokaler Ebene liefert der WFC zufriedenstellende, einzigartige Ergebnisse, da sich in dieser Größe kaum untereinander ähnliche Strukturen herausbilden können. Auf globaler Ebene, somit bei der Synthese größerer Tilemaps, ist zu erkennen, dass die lokale Ähnlichkeit zu homogenen Ergebnissen führt.

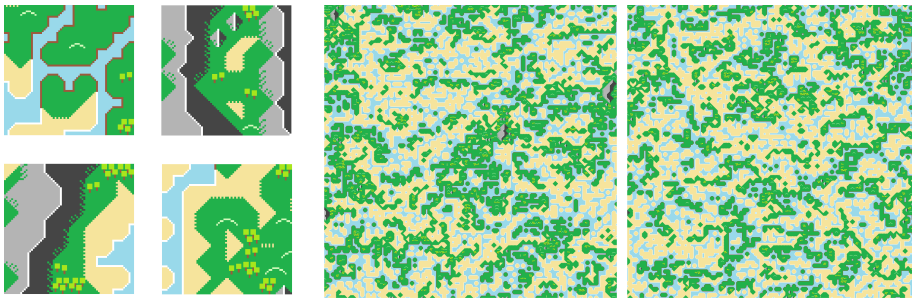

Abbildung 3.1: Die generierten kleinen Tilemaps sind einzigartig (a). Im Gegensatz dazu sind die großen Tilemaps homogen (b).

Die Nutzung der großen gleichförmigen Tilemaps in Abbildung 3.1 (b) als Spielewelt, würde zu einem uninteressanten Spielerlebnis führen.

3.2 Erweiterung des WFC

3.2.1 Problemlösung

Damit interessantere Tilemaps generiert werden, ist es nötig größere zusammenhängende Strukturen zu erzeugen und mehr Variation innerhalb der Tilemaps zu schaffen. Damit dies möglich ist, muss der Einfluss der lokalen Ähnlichkeit verringert werden. Eine Möglichkeit ist die Einführung von nichtlokalen Constraints, wie in Unterabschnitt 2.7.1 beschrieben. Durch diese wird die Auswahl der Muster zusätzlich durch Zellen, die nicht zur direkten Nachbarschaft gehören, beeinflusst und die lokale Ähnlichkeit abgeschwächt. Die Einführung weiterer Constraints hat jedoch zwei Nachteile. Erstens müssen, um starken Einfluss auf die Eigenschaften der Tilemap nehmen zu können, viele Constraints eingeführt werden. Der Mehraufwand der Erstellung der Constraints steigt bei großen Tilesets

exponentiell an. Daher werden weitere Constraints in Beispielanwendungen nur für einige wenige spezielle Tiles genutzt. Beispielsweise Game-Items wie Schlüssel und Kisten [1]. Zweitens bedeutet die Einführung weiterer Constraints, dass es häufig zu Widersprüchen kommt. Vor allem, wenn zahlreiche weitere Contraints eingeführt werden, muss der Algorithmus oftmals neu starten oder im Falle der Implementierung von Backtracking sich oftmals zurücksetzen. Dies würde zu einer drastischen Verschlechterung der Laufzeit führen. Im schlimmsten Fall kann es sein, dass aufgrund der vielen Constraints keine valide Lösung gefunden werden kann.

3.2.2 Weighted Choice

Die zweite Möglichkeit Einfluss auf die Auswahl der Tiles zu nehmen und somit auch auf die lokale Ähnlichkeit, ist die Musterheuristik der Weighted-Choice (s. 2.7.3). Im originalem WFC wird eine initiale statische Gewichtung für die gesamte Tilemap definiert. Die Erweiterung des WFC durch eine räumliche Einteilung durch Chunks (s. 2.7.4), ermöglicht es die Gewichte der Tiles individuell in unterschiedlichen Abschnitten der Tilemap anzupassen. Für jeden Chunk können unterschiedliche Gewichtung angegeben werden. Hier wird das Konzept weitergeführt. Die Größe der Blöcke, welche die Tilemap unterteilen, wird verringert bis zur kleinsten Ebene: einer Zelle. Für jede Zelle wird eine individuelle Gewichtung der Tiles angegeben. Es wird eine Methodik benötigt, mit der man die Gewichte für jede Zelle anpassen kann. Diese muss nicht statisch vor der Laufzeit definiert werden, sondern kann dynamisch zur Laufzeit erstellt werden. In den folgenden Abschnitten wird ausgeführt, wie diese Gewichte dynamisch erstellt werden.

3.2.3 Beobachtungsraum

Es ist es naheliegend, dass die Gewichtung durch Tiles der anderen Zellen der Tilemap beeinflusst wird. Das Ziel ist lokale Ähnlichkeit abzuschwächen, zudem sollen zusammenhängende Strukturen generiert werden. Der dafür benötigte Kontext, soll nicht nur aus den benachbarten Zellen bestehen. Trotzdem soll ein lokaler Kontext weiterhin bestehen bleiben. Um diese zu erreichen, wird ein Beobachtungsraum als Grundlage zur dynamischen Berechnung der Gewichte genutzt. Der Beobachtungsraum beinhaltet nur die Zellen im Umfeld der kollabierenden Zelle sind.

Nachdem eine Zelle zum Kollabieren ausgewählt wurde, werden alle Zellen innerhalb des Radius r um diese erfasst. Um eine bessere Kontrolle zur späteren Berechnung der dynamischen Gewichtung zu erhalten, werden die Zellen in n Ringe aufgeteilt, entsprechend der Entfernung zur Zelle in der Mitte. Jeder Ring hat eine Breite, die zusammenaddiert den Radius des Beobachtungsraumes ergeben. Entfernungen zur mittleren Zelle werden mittels euklidischen Abstandes[1] berechnet.

6,4	5,7	5,0	4,5	4,1	4,0	4,1	4,5	5,0	5,7	6,4
5,8	5,0	4,2	3,6	3,2	3,0	3,2	3,6	4,2	5,0	5,8
5,4	4,5	3,6	2,8	2,2	2,0	2,2	2,8	3,6	4,5	5,4
5,1	4,1	3,2	2,2	1,4	1,0	1,4	2,2	3,2	4,1	5,1
5,0	4,0	3,0	2,0	1,0	0	1,0	2,0	3,0	4,0	5,0
5,1	4,1	3,2	2,2	1,4	1,0	1,4	2,2	3,2	4,1	5,1
5,4	4,5	3,6	2,8	2,2	2,0	2,2	2,8	3,6	4,5	5,4
5,8	5,0	4,2	3,6	3,2	3,0	3,2	3,6	4,2	5,0	5,8
6,4	5,7	5,0	4,5	4,1	4,0	4,1	4,5	5,0	5,7	6,4

Abbildung 3.2: Zelle A wird kollabiert. Der Beobachtungsraum dieser Zelle ist in zwei Ringe mit einer Breite von 2 eingeteilt. Der euklidische Abstand zur mittleren Zelle ist angegeben. Alle Zellen, die einen Abstand ≤ 2 (Breite des grünen Ringes) besitzen, werden dem grünen Ring zugeschrieben. Alle Zellen, die einen Abstand ≤ 4 (Breite des grünen und blauen Ringes addiert) und nicht zum grünen Ring gehören, sind Teil des blauen Ringes.

Die Neuberechnung der Gewichtung basiert auf Basis der bereits kollabierten Zellen im Beobachtungsraum. Zur Berechnung werden vom Benutzer erstellte Regelsätze genutzt, die im folgenden Unterabschnitt 3.2.4 ausgeführt werden.

3.2.4 Regelsätze

Gegeben ist eine Menge an bereits kollabierten Tiles T, die durch den Beobachtungsraum der zu kollabierenden Zelle a erfasst wurden. Eingeführt wird eine Menge an Tile-Kategorien K, um ähnliche Tiles zusammen zu kategorisieren. Jedes Tile $t \in T$ hat eine

[1] http://de.wikipedia.org/w/index.php?title=Euklidischer%20Abstand&oldid=212120720 (Letzter Zugriff: 08.07.2023)

Tile-Kategorie $k \in K$. Jedes Tile ist außerdem einem Ring $r \in R$ zugeteilt. Mit diesen Informationen werden für alle möglichen Tiles $u \in U$ der Zelle a dynamische Gewichte erstellt.

Zur Berechnung wird eine Konfigurationsdatei als Satz an Regeln benötigt, ein Beispiel A ist im Anhang zu finden. In der Konfigurationsdatei wird beschrieben, mit welchem Wert die Gewichtung eines Tiles $u \in U$ einer Tile-Kategorie k angepasst wird. Das geschieht entsprechend der Tile-Kategorie k und Ring r eines Tiles t im Beobachtungsraum. Je größer der Wert, desto wahrscheinlicher, dass ein Tile u der Tile-Kategorie k gewählt wird.

Algorithm 2 Berechnung der dynamischen Gewichtung

```
for each Tile u ∈ U do
    initialize weight 0
    for each Tile t ∈ T im Beobachtungsraum do
        value = getValueFromRegelsatz(Ring r, Tiletype k)
        value / Number of Tiles in Ring r                    ▷ (a)
        weight += value
    end for
    weight = weight * staticWeight                           ▷ (b)
end for
```

(a) Ringe können eine unterschiedliche Anzahl an Zellen haben. Damit die dynamische Gewichtung nicht zu stark durch den Ring mit den meisten Zellen beeinflusst wird, wird der Wert durch die Anzahl der Zellen im Ring normiert. Jeder Ring hat somit den gleichen Einflusss auf die Gewichtung. (b) Tiles mit der gleichen Tile-Kategorie haben die gleiche Gewichtung. Um die Frequenz innerhalb Tiles der gleichen Tile-Kategorie anpassen zu können, wird die Gewichtung mit einem statischen Gewicht multipliziert.

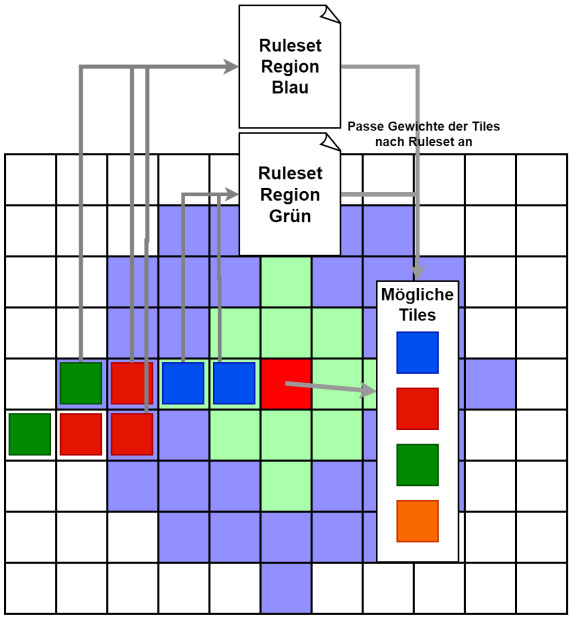

Abbildung 3.3: Eine Zelle (rot) wird kollabiert, diese kann in eine Menge an Tiles kollabiert werden. Die Gewichte dieser Tiles werden durch kollabierte Zellen im Beobachtungsraum entsprechend der Regelsätze beeinflusst.

3.2.5 Gebiete

Einteilung der Spielwelt in Regionen ist ein wichtiger Bestandteil der prozeduralen Generierung. Regionen haben unterschiedliche Charakteristiken, aufgrund unterschiedlicher Parameter bei der Generierung. So wird eine größere Variation innerhalb einer Spielwelt erzeugt [17]. In dieser Arbeit wird die Tilemap ebenfalls in Regionen eingeteilt.

Die Einteilung der Tilemap geschieht durch ein Voronoi-Diagramm. Dies ist die Zerlegung des Raumes in Regionen, durch eine vorgegebene Menge an Punkten, als Zentren, innerhalb des Raumes. Alle Punkte werden der Region zugeordnet, deren Mittelpunkt sie am nächsten sind [25].

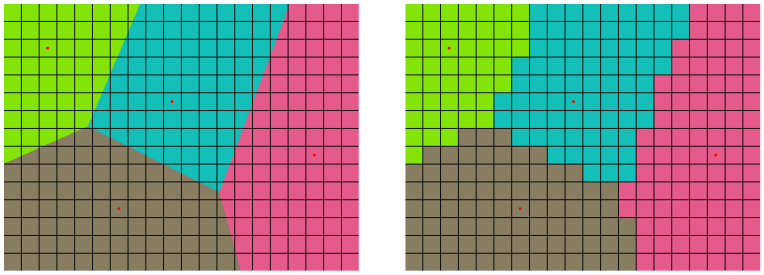

Abbildung 3.4: Einteilung Zellen einer Tilemap durch ein Voronoi Diagramm.

Der Nutzer definiert n Regionen. Für diese muss eine Zelle der Tilemap als Mittelpunkt angegeben werden. Für jede Zelle der Tilemap wird der euklidische Abstand zu allen Mittelpunkten berechnet. Die Zelle wird der Region zugeordnet, dessen Mittelpunkt den geringsten Abstand hat.

Damit Regionen unterschiedliche Charakteristiken aufweisen, müssen Parameter der Generierung in den verschiedenen Regionen verändert werden. Für jede Region muss der Nutzer ein Regelsatz definieren. Beim Kollabieren einer Zelle wird zur Berechnung der umgebungsabhängigen dynamischen Gewichtung der entsprechende Regelsatz einer Region genutzt. Die Regelsätze unterschiedlicher Regionen unterscheiden sich durch die Werte, die sich aus Tile-Kategorie und Ring ergeben. Übergänge von einer Region auf eine andere werden kein Problem darstellen, da die Einhaltung der Adjacency-Constraints weiterhin gegeben ist.

3.2.6 Zweiter Durchlauf

Beim Kollabieren einer Zelle kann es dazu kommen, dass im Beobachtungsraum keine/weniger Tiles vorhanden sind. Besonders am Anfang des Algorithmus wird der umgebungsabhängigen dynamischen Gewichtung wenig Kontext zur Berechnung geboten. Daher wird ein zweiter Durchlauf durchgeführt. Beim Kollabieren im ersten Durchgang wird das Verhältnis zwischen Anzahl der Zellen im Beobachtungsraum und der Anzahl der kollabierten Zellen erfasst. Nach Beendigung des ersten Durchlaufes des WFC werden alle Tiles mit einem Verhältnis, welches einen Grenzwert unterschreitet, zurückgesetzt. Mit der teilweise gelösten Tilemap wird der WFC neu gestartet. In diesem Durchlauf wird der Berechnung der umgebungsabhängigen dynamischen Gewichtung für die zurückgesetzten Tiles mehr Kontext geboten. Theoretisch sollte dies zu einer qualitativ besseren Auswahl des Tiles führen.

Inwieweit sich das auf die Qualität der Ergebnisse auswirkt, wird in der Evaluation diskutiert.

3.3 Anforderungen

3.3.1 Funktionale Anforderungen

Im Rahmen dieser Arbeit soll ein Software-Prototyp umgesetzt werden. Dieser soll zweidimensionale Tilemaps mittels WFC erstellen. Der Algorithmus soll die Erweiterung der umgebungsabhängigen dynamischen Gewichtung implementieren. Die vom Prototypen generierten Tilemaps, sollen dementsprechend interessant sein. Was dies bedeutet wird durch folgende Anforderungen formuliert.

1.1 Ein Faktor für eine interessante Tilelemap ist ein großes Tileset. Ein gegebenes Tileset soll erweitert werden, sodass sich durch die automatische Tile Erstellung die maximale Anzahl an individuellen Tiles ergibt. Damit die Tiles in der generierten Tilemap fehlerfrei angewendet werden, sollen Adjacency Constraints anhand von Kantenlabels automatisch erstellt werden.

1.2 Es sollen sich größere zusammenhängende Strukturen herausbilden. Tiles die nicht direkt durch Constraints im Zusammenhang stehen, sollen trotzdem voneinander abhängig sein. Welche Strukturen sich herausbilden wird vom Tileset und der Konfiguration der Generierung abhängig sein.

1.3 Um Diversität innerhalb der Tilemap zu erhalten, sollen sich Regionen herausbilden, die sich durch ihre Charakteristiken unterscheiden. Obwohl Regionen mit dem gleichen Tileset arbeiten, sollen sich diese Charakteristiken durch Strukturen, Frequenz der Tiles und typische Anordnungen der Tiles unterscheiden.

1.4 Übergänge zwischen Regionen sollen fließend sein.

1.5 Die Anordnung der Tiles sollen weiterhin auf lokaler Ebene, ein Bildausschnitt aus einer größeren Tilemap, die lokale Ähnlichkeit erfüllen.

Der erweiterte WFC Algorithmus soll durch ein User-Interface konfiguriert und ausgeführt werden können. Es gibt folgende Anforderungen, die in Unterabschnitt 3.4.3 weiter spezifiziert sind.

2.1 Die Benutzeroberfläche soll eine einfache Konfiguration der Tiles eines Tilesets ermöglichen. Es sollen Informationen, die automatische Tile und Adjacency Constraint Erstellung erlauben, angegeben werden.

2.2 Eine Benutzeroberfläche soll eine einfache Konfiguration des Regelsatzes für die umgebungsabhängige dynamische Gewichtung ermöglichen.

2.3 Eine Benutzeroberfläche soll das Ausführen des konfigurierten WFC Algorithmus möglich machen.

3.3.2 Nicht funktionale Anforderungen

Für die nicht funktionale Anforderungen wurde sich an den bekannten Qualitätsmerkmalen des internationaler Standard ISO/IEC 25010 orientiert [10]. Je nach Anwendungsfall haben einige Anforderungen eine höhere Priorität als andere. Da der Prototyp ein Proof of Concept ist, hat zum Beispiel die Sicherheit eine geringe Priorität. Die nicht funktionalen Anforderungen **Wartbarkeit** und **Benutzbarkeit** werden priorisiert.

Um die Wartbarkeit des Systems sicherzustellen, wird Code-Qualität und die Einhaltung von Coding-Prinzipien angestrebt. Darüber hinaus ist es erforderlich, den Code vollständig zu kommentieren. Zusätzlich soll der Prototyp in klar definierte Module mit abgegrenzten Verantwortlichkeiten aufgeteilt sein. Die Benutzbarkeit soll durch eine verständliches User-Interface garantiert werden.

Dieser Schwerpunkt auf Wartbarkeit und Benutzbarkeit soll dem Nutzer helfen, das im Prototypen präsentierte Konzept zu verstehen. Auch soll so sichergestellt werden, dass das System für zukünftige Entwicklungen geeignet ist.

3.4 Spezifikationen

3.4.1 WFC

Gemäß Abschnitt 2.5 werden geeignete Spezifikationen für den Algorithmus ausgewählt. Es gibt zwei Arten des WFC Algorithmus, das Overlapping Model und das Simple Tiled Model (s. 2.5.1). Der Prototyp soll das Simple Tiled Model des WFC implementieren, um die vorgestellte Erweiterung des WFC (s. 3.2) zu veranschaulichen. Das Konzept ist

auch auf das Overlapping Model anwendbar, jedoch würde die zusätzliche Komplexität dieser Methode von dem Konzept ablenken.

Der Prototyp soll die standardmäßige Lokationsheuristik der niedrigsten Entropie nutzen. Zusätzlich soll er durch weitere Lokationsheuristiken erweitert werden.

Der Algorithmus soll ohne Input-Sample (s. 2.5.1) arbeiten. Somit muss der Nutzer entsprechende Muster erstellen. Die Muster sollen zweidimensionale, quadratische Tiles sein, die zusammen ein Tileset ergeben. Aus diesem soll eine Tilemap variabler Größe als Output in Form eines Bildes generiert werden.

3.4.2 Tileset

Der Aufwand der manuellen Erstellung eines Tilesets steigt bei steigender Anzahl an Tiles exponentiell an. Jedes Tile und entsprechende Spiegelungen und Rotationen müsste manuell erstellt werden. Für jedes Tile müssten außerdem alle Adjacency Constraints für jede Kante manuell angegeben werden. Der Prototyp soll dem Nutzer ermöglichen mit minimalem Aufwand Tilesets zu erstellen. Eine automatische Erstellung des Tilesets (s. 2.6) vereinfacht diesen Prozess, deswegen soll die Software diese folgendermaßen implementieren.

Durch die Eigenschaften eines des Tiles ergeben sich die Transformationen des Tiles, durch die das Tileset erweitert werden soll. Durch die Symmetrie eines Tiles sind alle Transformationen nach Tabelle 2.1 bestimmbar. Jedoch ist es nicht immer sinnvoll alle Transformationen eines Tiles zu nutzen. Daher muss zudem für jedes Tile angegeben werden, ob dieses rotiert und gespiegelt werden darf.

Für alle Tiles des erweiterten Tilesets sollen die Adjacency Constraints definiert werden. Wie in Abbildung 3.5 beschrieben, wird für jede Kante eines Tiles ein Kantenlabel definiert. Da in diesem Prototyp die Tiles quadratisch sind, gibt es vier Kanten: *top*, *bottom*, *left* und *right*. Ist eine Kante asymmetrisch, muss diese segmentiert werden.

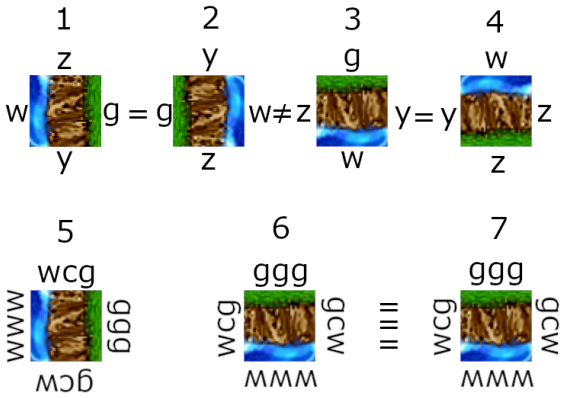

Abbildung 3.5: Beschreibung der Kanten von Tiles durch Labels

Im Folgendem wird das Zusammenpassen der Kantenlabels anhand von Abbildung 3.5 erläutert. Wenn das Tile ein Label pro Seite hat, können symmetrische Kanten (w und g) bei Übereinstimmung der Labels korrekt aneinander angeordnet werden (1 und 2). Anordnungen zwischen asymmetrischen Kanten (y und z) sind trotz Übereinstimmung der Label inkorrekt (3 und 4). Wenn asymmetrische Kanten vorhanden sind, muss das Label der Kante segmentiert werden. Zwei Kanten können nebeneinander angeordnet werden, wenn die gegenüberliegenden Teilsegmente des Labels übereinstimmen (6 und 7).

Im Prototypen ist ein Kantenlabel ein String. Segmente des Labels sind die individuellen Character des Strings. Es stimmen zwei Kanten überein, wenn der String der einen Kante mit dem umgekehrten String der anderen Kante übereinstimmt.

Wird ein Tile transformiert, müssen entsprechend die Labels der Kanten angepasst werden. Bei der Rotation eines Tiles werden die Labels entsprechend der Kante mitgenommen. Bei der Spiegelung an der vertikalen Achse eines Tiles werden die Labels an den Kanten *left* und *right* getauscht. Der String an den Kanten *top* und *bottom* wird umgekehrt.

3.4.3 User-Interface

Der WFC soll mittels Konfigurationsdateien angepasst werden. Über ein User-Interface wird ein Nutzer in der Lage sein ein Tileset zu erstellen, welches als JSON-Datei abgespeichert wird. Ein Beispiel einer solchen Konfigurationsdatei A ist im Anhang zu finden.

Der Nutzer erstellt die individuellen Tiles, die automatisch anhand deren Eigenschaften ergänzt werden. Die Informationen, die über die graphische Oberfläche angegeben werden:

- Arbeitsverzeichnis - Verzeichnis in dem Konfigurationsdateien, Bilder der Tiles und Ergebnisse gespeichert werden
- Eigenschaften eines Tiles
  - Bilddatei im Arbeitsverzeichnis
  - Kategorie (wird in 3.2.4 erläutert)
  - Symmetrie nach Tabelle 2.1
  - Ist Rotation erlaubt?
  - Ist Spiegelung erlaubt?
  - String als Label für alle Kanten

Durch die Einführung der umgebungsabhängigen dynamischen Gewichtung und der Regionen muss das User-Interface erweitert werden. Aus den zusätzlichen Informationen wird eine Konfigurationsdatei für die Regelsätze erstellt. Informationen, die über die graphische Oberfläche angegeben werden:

- Größe der Tilemap - Anzahl der Zellen in der horizontalen und vertikalen Richtung
- Region - Name und Koordinaten des Mittelpunktes
- Beobachtungsraum - Anzahl und Breite der Ringe
- Werte zur Berechnung der Gewichtung je nach Region, Ring und Tile-Kategorie

Ist die Konfiguration abgeschlossen, muss der Algorithmus über das User-Interface gestartet werden. Dazu wird Folgendes benötigt:

- Knopf zum Starten des Algorithmus
- Auswahl des WFC - Soll der unmodifizierte oder modifizierte Algorithmus verwendet werden?
- Lokationsheuristik - Eine Auswahl der implementierten Heuirstiken

4 Umsetzung

Im folgenden Kapitel werden Details der Implementierung ausgeführt. Dazu wird die Auswahl des Vorgehensmodells und der verwendeten Technologien zur Entwicklung des Software-Prototypen begründet. Des Weiteren wird die Umsetzung der im Kapitel 3 vorgestellten Konzepte beschrieben.

4.1 Vorgehensmodell

Zur Entwicklung des Software-Prototyps wurde das inkrementelle und iterative Vorgehensmodell aus der agilen Softwareentwicklung angewendet. In diesem Modell werden die Softwareanforderungen in mehrere Module aufgeteilt und inkrementell entwickelt [29]. Innerhalb eines Moduls wird ein iterativer Entwicklungsprozess verfolgt. Die Entwicklungsphasen der Anforderungsanalyse, Design, Implementierung und Evaluation werden iterativ durchlaufen [34]. Dieses Vorgehensmodell wurde aufgrund seiner Flexibilität ausgewählt. Durch die inkrementelle und iterative Entwicklung können im Laufe der Entwicklung Anforderungen auf Basis von Zwischenergebnissen, sowohl iterativ innerhalb eines Moduls, als auch modulübergreifend, angepasst werden.

Die in Kapitel 3 genannten Anforderungen und Spezifikationen wurden wie folgt aufgeteilt und inkrementell implementiert:

- Erzeugung des Tilesets durch Symmetrien und Kantenlabels auf Basis einer JSON-Konfigurationsdatei
- Erzeugung der Regelsätze auf Basis einer weiteren JSON-Konfigurationsdatei
- Modifizierung des WFC Algorithmus durch die umgebungsabhängige dynamische Gewichtung
- Einführung weiterer Lokationsheuristiken

- Einteilung der Tilemap in Regionen
- Modul zur Erzeugung der Konfigurationsdateien
- User-Interface

4.2 Verwendete Technologien

Die Umsetzung des Software-Prototyps geschieht auf Basis eines Procedural Content Generation Repositorys der HAW Hamburg, das von Prof. Dr. Phillip Jenke geleitet wird. In diesem Projekt werden verschiedene Algorithmen zur prozeduralen Content Generation zusammengetragen. Als Grundlage dieser Arbeit dient eine generische Implementation des WFC Algorithmus. Aufgrund des PCG-Repositorys ist Java als Programmiersprache gegeben. Umgesetzt wurde das Projekt mit Java SE 14[1]. Zur Build-Automatisierung und Abhängigkeitsverwaltung nutzt das Projekt Gradle[2].

Zur Entwicklung des Prototyps wurde die Entwicklungsumgebung IntelliJ Idea[3]genutzt. Die Versionsverwaltung erfolgte mit Git[4].

4.2.1 Bibliotheken

Neben den Java Standardbibliotheken wurden zwei weitere Bibliotheken angewendet. JSON.simple[5] ist eine Bibliothek, die grundlegende Funktionen zum Lesen und Schreiben von JSON-Dateien bereitstellt. Hier genutzt zur Verwaltung der Konfigurationsdateien des Tilesets und der Regelsätze. Zur interaktiven Konfiguration und Ausführung des modifizierten WFC Algorithmus wurde die Java Swing Bibliothek[6] genutzt. Sie bietet Funktionen zur Erstellung einer graphischen Benutzeroberfläche.

[1] https://www.oracle.com/de/java/technologies/javase/jdk14-archive-downloads.html (Letzter Zugriff: 14.07.2023)
[2] https://gradle.org/ (Letzter Zugriff: 14.07.2023)
[3] https://www.jetbrains.com/idea/ (Letzter Zugriff: 14.07.2023)
[4] https://git-scm.com/ (Letzter Zugriff: 14.07.2023)
[5] https://github.com/fangyidong/json-simple (Letzter Zugriff: 14.07.2023)
[6] https://docs.oracle.com/javase%2F7%2Fdocs%2Fapi%2F%2F/javax/swing/package-summary.html (Letzter Zugriff: 14.07.2023)

4.3 Architektur

Die Model-View-Controller-Architektur ist ein Architekturmuster, bei der eine Anwendung in drei eigenständige Komponenten unterteilt wird. Vorteil dieser Architektur ist die Entkopplung der Komponenten und damit einhergehend eine klare Trennung der Verantwortlichkeiten. Komponenten können unabhängig voneinander entwickelt und getestet werden. Das bestehende System kann einfach angepasst werden, durch das Hinzufügen oder Austauschen von Komponenten [14]. Im Zusammenspiel mit dem beschriebenen Vorgehensmodell (s. 4.1) bietet die MVC-Architektur eine hohe Flexibilität.

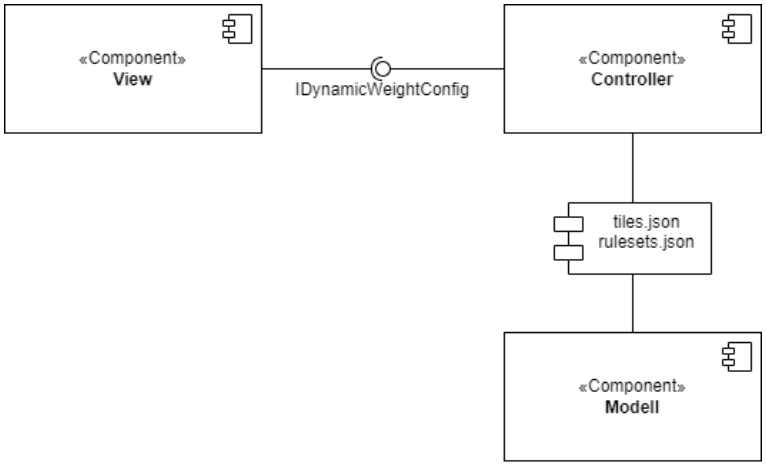

Abbildung 4.1: Komponentendiagramm

4.3.1 View

Die View Komponente implementiert das User-Interface. Hier werden Parameter zur Ausführung des WFC gesetzt. Die Eingabe der Parameter geschieht durch die bereitgestellten Funktionen der Java Swing Bibliothek.

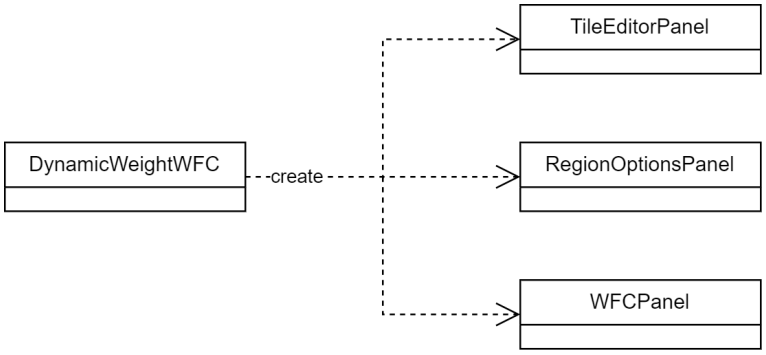

Abbildung 4.2: View Komponente - Klassendiagramm

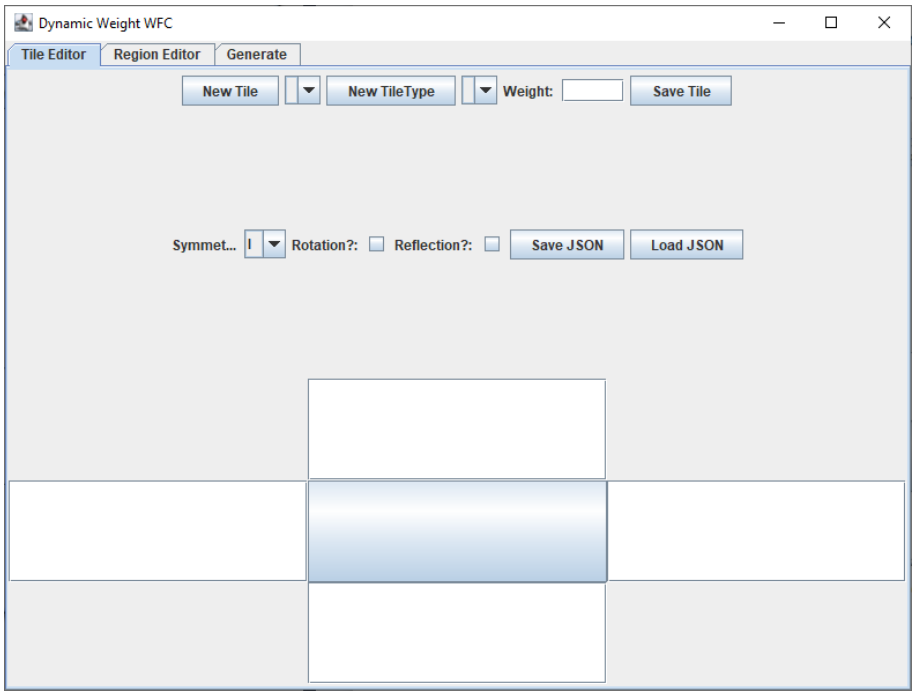

Abbildung 4.3: Im Tile Editor Panel wird das Tileset konfiguriert. Benötigte Funktionen wurden in Unterabschnitt 3.4.3 angegeben.

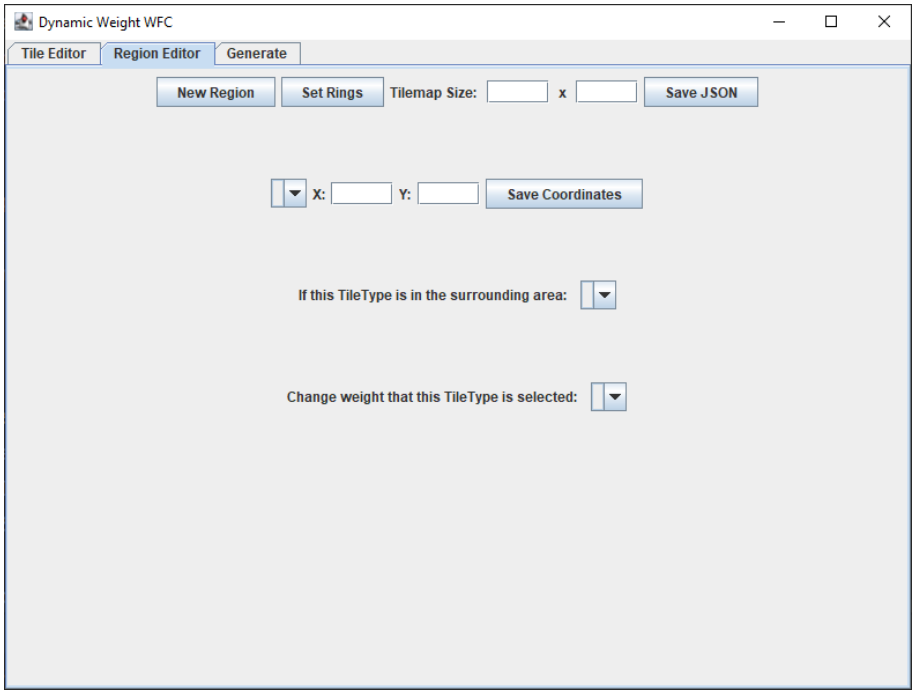

Abbildung 4.4: Im Region Editor Panel werden die Regeln zur umgebungsabhängigen dynamischen Gewichtung der einzelnen Regionen konfiguriert. Benötigte Funktionen wurden in Unterabschnitt 3.4.3 angegeben.

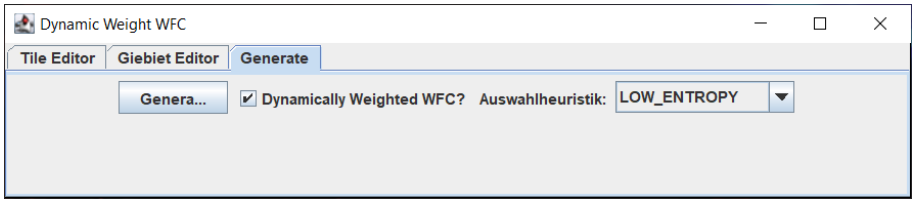

Abbildung 4.5: Im Generate Panel kann der Algorithmus gestartet werden. Der Nutzer gibt an, ob der modifizierte Algorithmus verwendet werden soll. Außerdem welche Lokationsheuristik genutzt wird.

4.3.2 Controller

Die Controller Komponente nimmt Eingaben aus der View Komponente entgegen und leitet diese normiert an das Modell weiter. Die Klasse **DynamicWeightConfig** dient als Datenstruktur zur Abbildung der Parameter mittels der Hilfsklassen **Offsets**, **Tile**, **TileTypes**, **Region**, **Symmetry** und **Edges**. Die Klasse **DynamicWeightConfig**

konvertiert die Parameter in das JSON-Format. Erstellt werden die vom Modell genutzten Konfigurationsdateien, *tiles.json* beinhaltet alle Informationen zum Tileset und *rulesets.json* alle Informationen zu den Regionen und zugehörigen Regelsätzen. Die Klasse **WFCStarter** übergibt der Modell Komponente die benötigten Parameter und startet den WFC Algorithmus.

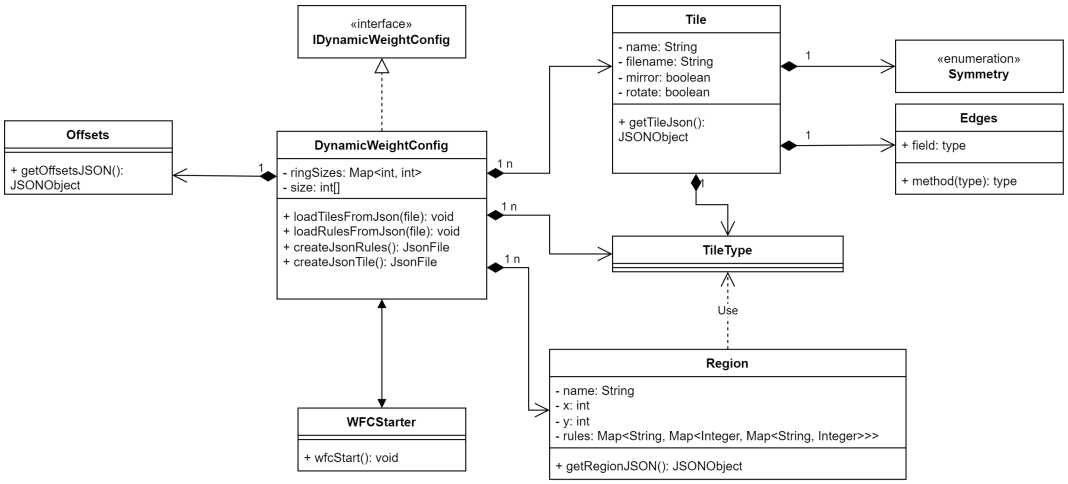

Abbildung 4.6: Controller Komponente - Klassendiagramm

4.3.3 Model

Die Modell Komponente beinhaltet sämtliche Applikationslogik des modifizierten WFC Algorithmus. Durch das PCG-Repository ist eine generische Implementierung des WFC gegeben.

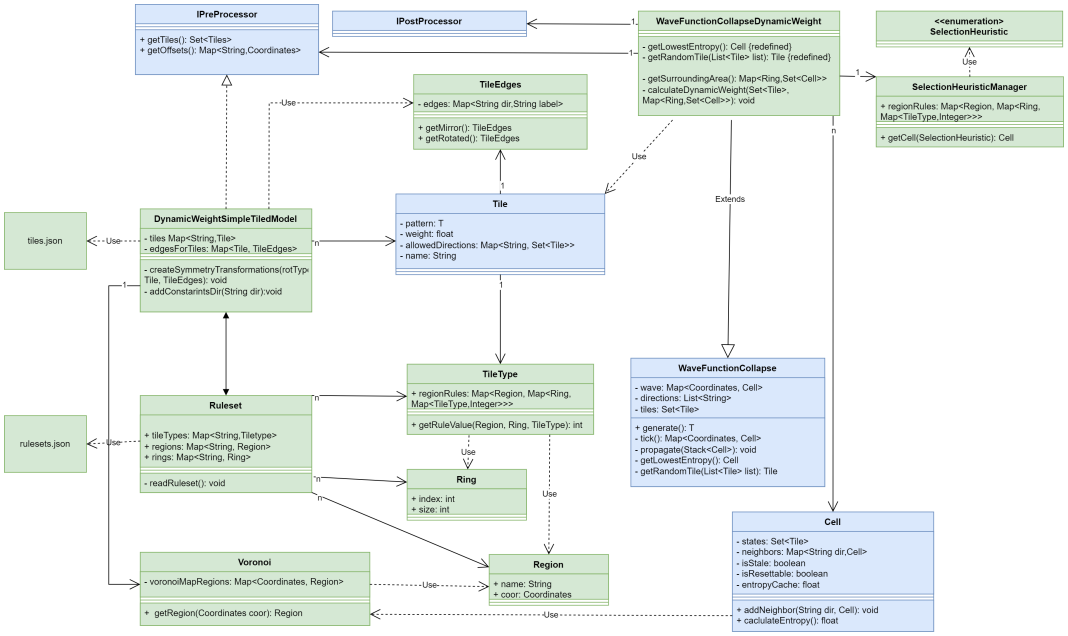

Abbildung 4.7: Modell Komponente - Klassendiagramm. Blau markiert sind Klassen, die durch die generische Implementierung gegeben sind. Grün markiert sind Klassen, die zur modifizierten Implementierung gehören.

Die Klasse **WaveFunctionCollapse** implementiert den WFC Algorithmus. Die Methode generate() startet den Algorithmus. Die Klasse **WaveFunctionCollapseDynamicWeigth** erbt von dieser, und implementiert das in Abschnitt 3.2 vorgestellte Konzept. Durch das Ersetzen der Methode getRandomTile() wurde die zufällige gewichtete Auswahl eines Tiles beim Kollabieren durch die dynamische Gewichtung ersetzt. Durch das Ersetzen der Methode getLowestEntropy() wurde die Lokationsheuristik angepasst. Die Methode getCell() des **SelectionHeuristicManager** gibt die zu kollabierende Zelle entsprechend der angegebenen Lokationsheuristik zurück.

Die Klasse **WaveFunctionCollapse** benötigt einen PreProcessor, der das Interface **IPreProcessor** implementiert. Das Interface stellt den Vorbereitungsschritt (s. 2.5.1) dar und fordert zwei Methoden ein. Mithilfe der Methode getOffsets(), werden die Richtungsvektoren der Nachbarzellen angegeben. In dem 2D Simple-Tiled-Model dieser Anwendung sind es vier Nachbarzellen. Die Methode getTiles() gibt die Liste aller Tiles, mit konfigurierten Adjacency Constraints zurück.

Die Klasse **DynamicWeightSimpleTiledModel** implementiert dieses Interface. Mittels der durch die Controller Komponente erstellte Konfigurationsdatei *tiles.json* werden

die Tiles erstellt. Wie in Unterabschnitt 3.4.2 beschrieben, wird das Tileset automatisch erweitert und Adjacency Constraints auf Basis der Kantenlabels erstellt. Kantenlabels werden durch die Klasse **TileEdges** repräsentiert.

Die Klasse **RulesetCreator** erstellt aus der Konfigurationsdatei *rulesets.json* die Regeln für die dynamische Gewichtung. Als Container für die Regeln wird die Klasse **TileTypes** genutzt.

4.3.4 Ablauf des Algorithmus

Der Ablauf des modifizierten WFC Algorithmus ist in Abbildung 4.8 dargestellt. Der grundlegende Ablauf aus Vorbereitung, Observierung und Propagierung des Algorithmus bleibt bestehen. Der Vorbereitungsschritt wird durch die Klasse **DynamicWeightSimpleTiledModel** beschrieben. Der Observierungschritt wird modifiziert. Nach der Auswahl des Tiles werden zwei Zwischenschritte eingefügt - das Erfassen des Beobachtungsraumes und darauf folgend die Berechnung der dynamischen Gewichtungen. Der Propagierungsschritt bleibt bestehen. Eingefügt wurde zusätzlich ein zweiter Durchlauf für Tiles mit zu wenig Kontext.

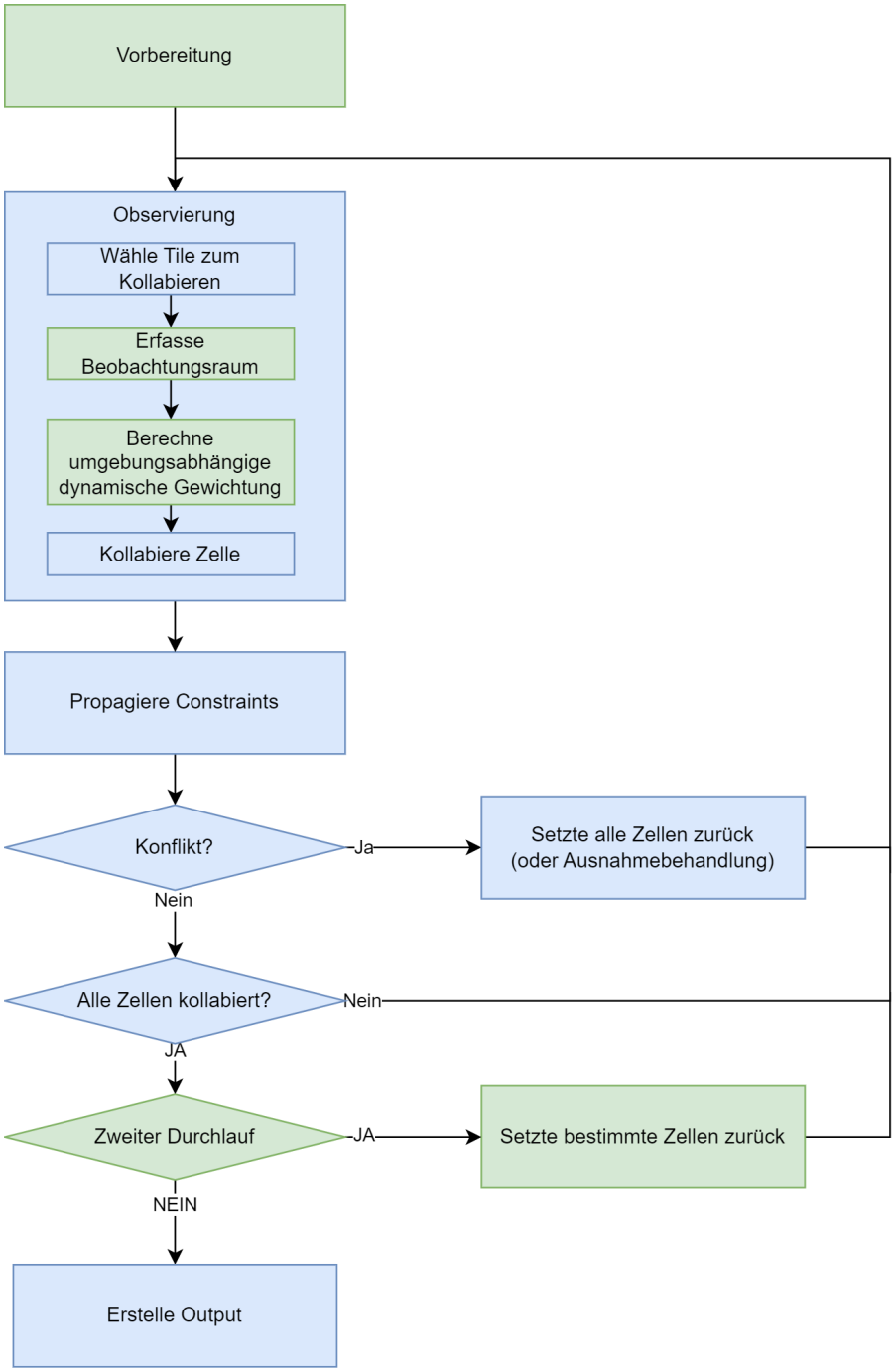

Abbildung 4.8: Ablaufdiagramm des modifizierten WFC Algorithmus. Neu eingeführte Schritte des Algorithmus sind in grün angegeben.

5 Evaluation

Die Evaluation von prozedural generierten Content jeglicher Art hängt stark von subjektiven Faktoren ab. Somit ist es schwierig allgemein gültige empirische Maßstäbe zu finden. Im Endeffekt werden Contentgeneratoren nach deren Fähigkeit, die Anforderungen des Nutzers zu erfüllen, bewertet [31]. Anforderungen sind grundsätzlich abhängig vom Anwendungsfall. Je nach Anwendungsfall können Ziele in der Generierung stark voneinander abweichen. Abhängig sind Anforderungen auch von der Art der Prozeduralen Content Generierung und den Designanforderungen des Nutzers. In Abschnitt 3.3 wurden allgemeine Anforderungen an die Ergebnisse des Prototyps formuliert. Im Folgenden wurden diese mit einem Tilesets aus dem originalen WFC Algorithmus [8] und anhand eines definierten Anwendungsfalls evaluiert.

5.1 Beispiel nach Gumin

Maxim Gumin hat im WaveFunctionCollapse Repository anhand mehrerer Tilesets die Ergebnisse des Algorithmus vorgestellt [8]. Zur Darstellung wird das Tileset *Knots* verwendet. Die Konfiguration des modifizierten Algorithmus ist wie folgt:

- 48×48 große Tilemap
- Drei Regionen: oben links, oben rechts, und unten mitte
- Beobachtungsraum aus zwei Ringen mit einer Breite von jeweils 2
- Eine individuelle Tile-Kategorie für jedes Tile
- Konfiguration der Tiles und Konfiguration einer Region sind im Anhang A.

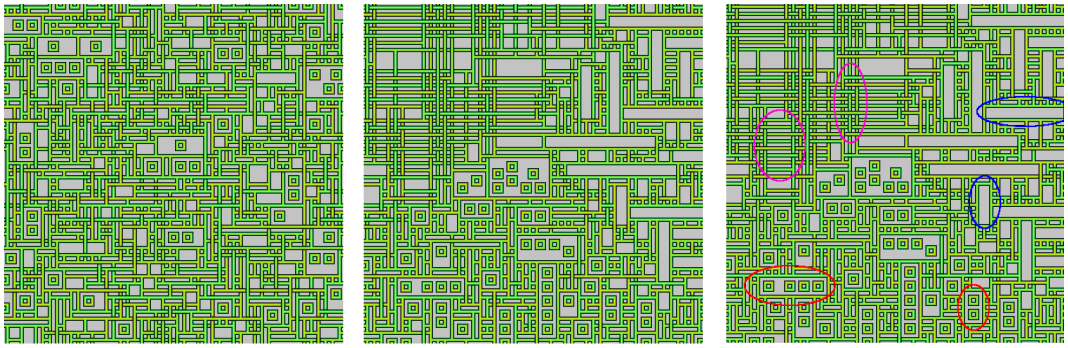

Abbildung 5.1: Ergebnisse des Tileset *Knots*. Standard WFC Algorithmus (a). Modifizierter WFC Algorithmus (b) und (c).

Die Abbildung 5.1 zeigt die Ergebnisse des unmodifizierten WFC Algorithmus in (a), und das Ergebnis des modifizierten Algorithmus in (b), welches die Anforderungen aus Abschnitt 3.3 erfüllen soll.

Das Tileset *Knots* besteht aus fünf Tiles. Durch die automatische Tileseterweiterung, werden daraus 13 Tiles. Alle möglichen Orientierungen der Tiles kommen in dem Output vor und erfüllen die Adjacency Constraints (s. Anforderung 1.1).

In dem unmodifizierten WFC Algorithmus (a) ist das Problem der Gleichförmigkeit (s. 3.1) zu erkennen. Im Gegensatz dazu sind in (b) drei Regionen, oben-links, rechts und unten, klar erkennbar (s. Anforderung 1.3). Diese weisen unterschiedliche typische Anordnungen von Tiles auf. Laut den Anforderungen sollen sich in den verschiedenen Regionen auch unterschiedliche größere Strukturen bilden. Trotz der geringen Menge an Tiles bilden sich diese Strukturen in (c) umkreist heraus (s. Anforderung 1.2). Wie in Anforderung 1.4 gefordert, sind die Übergänge zwischen den Regionen aufgrund der Einhaltung der Adjacency Constraints nahtlos.

5.2 Anwendungsbeispiel

Anforderungen sind grundsätzlich abhängig vom Anwendungsfall, daher wird ein konkretes Anwendungsbeispiel definiert. Anhand dieses wird der modifizierte WFC Algorithmus

evaluiert. Generiert wird eine Tilemap die Terrain darstellen soll, ähnlich wie im rundenbasierten Strategiespiel Civilization 1[1]. Dafür wurde ein Tileset aus 32 unterschiedlichen Tiles angefertigt. Daraus soll eine plausible, natürliche Spielwelt aus verschiedene Landschaftsarten, wie Berge, Wälder, Gewässer, etc. entstehen.

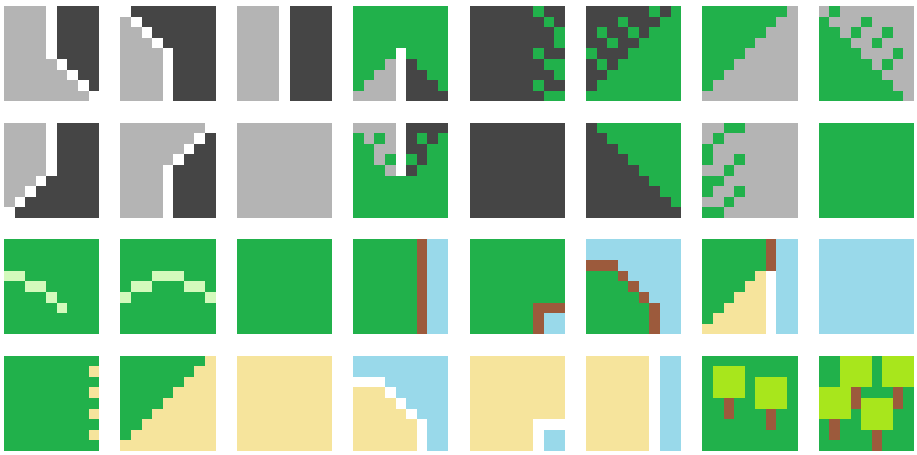

Abbildung 5.2: Alle Tiles des genutzten Tileset.

Generiert werden 64×64 große Tilemaps. Diese sind in vier Regionen eingeteilt, in jeder Ecke eine Region. Der Beobachtungsraum besteht aus drei Ringen und hat einen Radius von 7 Zellen. Die Breiten der Ringe von innen nach außen sind: 2 Zellen, 3 Zellen und 2 Zellen.

[1] https://civilization.com/de-DE/civilization-1/ (Letzter Zugriff: 27.07.2023)

5.2.1 Ergebnisse

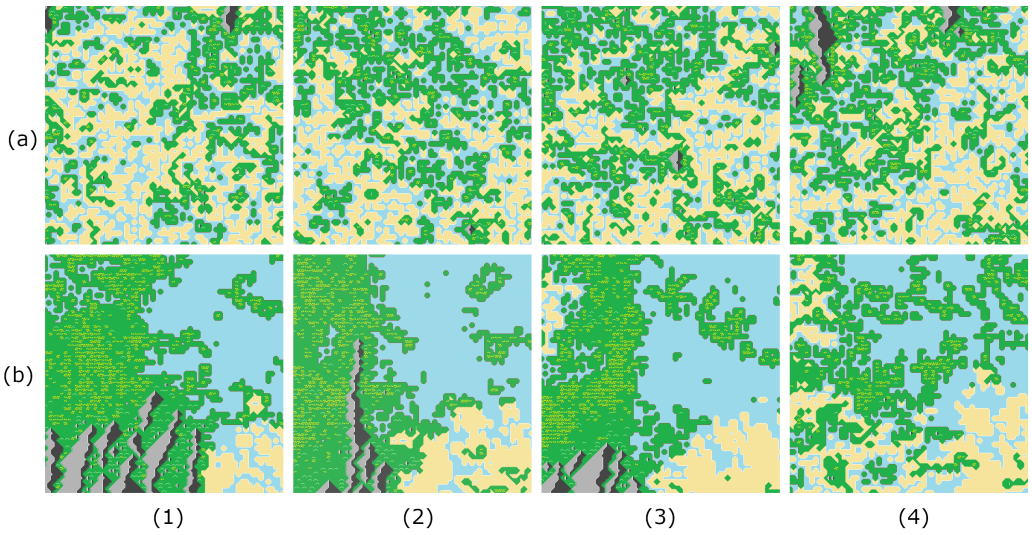

Abbildung 5.3: Tilemaps generiert mit dem originalen WFC Algorithmus (a). Tilemaps generiert mit dem modifizierten Algorithmus mit niedrigster Entropie als Lokationsheuristik (b).

Im Vergleich zu dem originalen WFC ist ein deutlicher Unterschied zu erkennen. Während in (a) das gesamte Ergebnis homogen ist, bilden sich in (b) wie konfiguriert die vier Regionen heraus (s. Anforderung 1.3). Übergänge zwischen den Regionen sind nahtlos (s. Anforderung 1.4).

Die Anforderung 1.2, dass sich Strukturen herausbilden, ist nur teilweise erfüllt. Im Endeffekt sind Strukturen Ansammlungen an Tiles mit bestimmten Tile-Kategorien. Der Beobachtungsraum wird zwar als Kontext erfasst, jedoch geschieht keine richtungsabhängige Anpassung der dynamischen Gewichte. Tiles derselben Kategorie und desselben Rings haben unabhängig von ihrer Position zum kollabierenden Tile den gleichen Einfluss auf die Gewichtung. Trotz dieser Begrenzung reicht es um natürliche Strukturen wie Inseln, Gebirgsketten, Täler und Ozeane zu erzeugen.

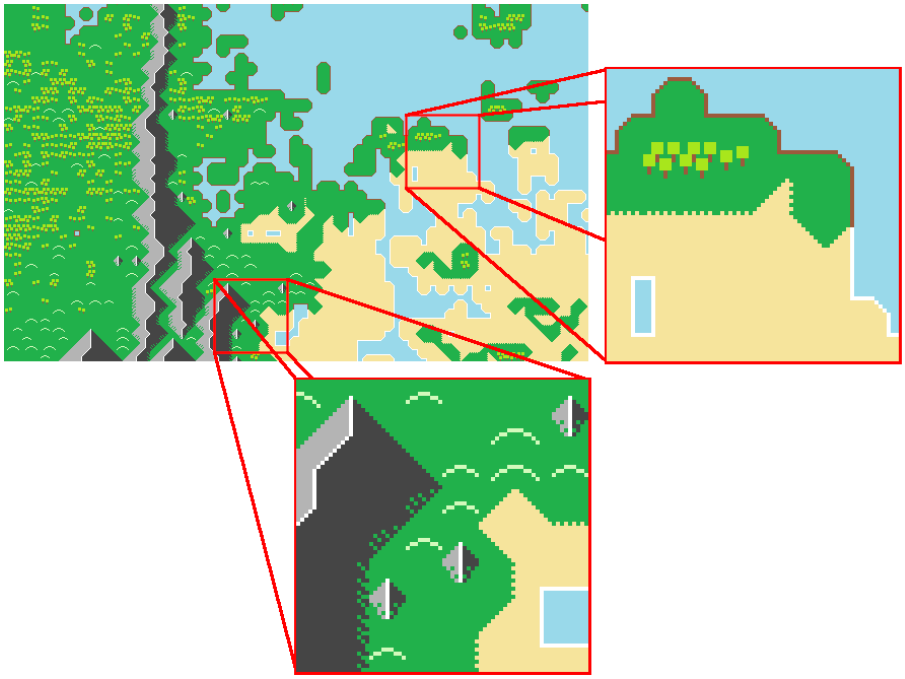

Abbildung 5.4: Zwei Ausschnitte aus einer großen Tilemap.

In Abbildung 5.4 sind Ausschnitte aus der Tilemap vergrößert dargestellt. Aus den Ergebnissen ist ersichtlich, dass die automatische Tileset Erweiterung funktioniert. Das Tileset wurde von 35 Tiles auf 60 Tiles erweitert. Adjacency Constraints wurden erzeugt und in den Ergebnissen kommen keine unerwarteten Nachbarschaftsverhältnisse vor (s. Anforderung 1.1).

Es ist zu erkennen, dass die lokale Ähnlichkeit, trotz Abschwächung durch die dynamische Gewichtung, weiterhin eingehalten wird. Auf der lokalen Ebene sind Ergebnisse weiterhin ähnlich zu dem des unmodifizierten Algorithmus (s. Anforderung 1.5).

5.2.2 Andere Lokationsheuristiken

Die Ufer der generierten Karten in Abbildung 5.3 sind teilweise stark verzweigt. Diese Verzweigungen lassen die Küsten unnatürlich wirken. Auf einige Tilemaps (b4) wirkt sich diese Erscheinung stark aus.

Ufer-Tiles sind äquivalent zu Pfad-Tiles, aus denen, wegen der Adjacency Constraints, Pfad-Graphen entstehen. Jeder Graph muss zusammenhängend sein und kann zyklisch

oder azyklisch sein [23]. Die zusammengesetzten Ufer-Tiles können sehr verzweigt sein, weil eine valide Anordnung von Tiles nur entsteht, wenn diese zusammenhängende Graphen bilden. Tiles werden jedoch zufällig zusammengesetzt, sodass häufig erst nach vielen Verzweigungen zufällig eine valide Anordnung gefunden wird. Die Zerklüftung wird verstärkt durch die Entropie Heuristik, wodurch zunächst die am stärksten eingeschränkten Zellen kollabiert werden, was häufig die Ufer-Tiles sind. So wird die Tilemap zunächst mit Tiles gefüllt, die kaum durch die dynamische Gewichtung beeinflusst werden (in diesem Tileset Ufer-Tiles), da die Auswahl der möglichen Tiles durch die Adjacency Constraints bereits eingeschränkt ist.

Es besteht die Möglichkeit, die Lokationsheuristik der niedrigsten Entropie durch andere Heuristiken (s. 2.7.2) zu ersetzten. So werden Zellen kollabiert, die mehr Tiles zur Auswahl haben. Die dynamische Gewichtung hat somit einen größeren Einfluss auf das Ergebnis.

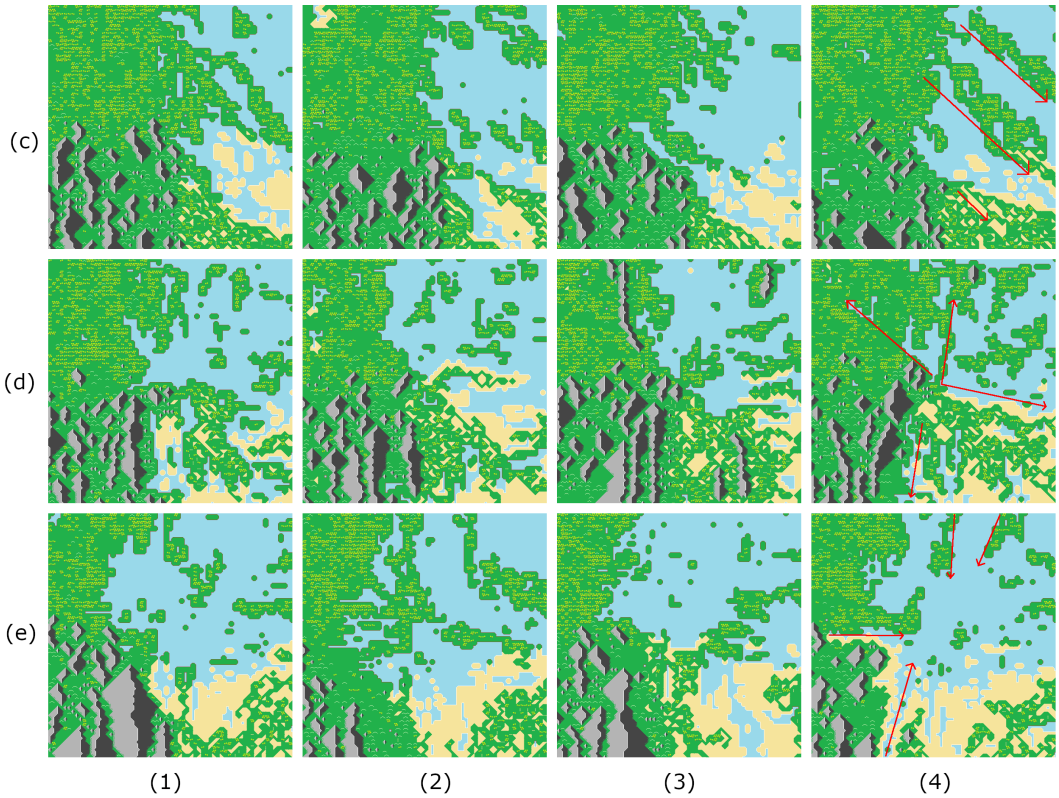

Abbildung 5.5: Ergebnisse anderer Heuristiken. Scanline(c), Innen nach Außen (d), Außen nach Innen (e).

In Abbildung 5.5 sind Ergebnisse anderer Heuristiken mit der gleichen Konfiguration dargestellt. Die Ergebnisse sind ähnlich zu denen der Entropie Heuristik. Es gibt jedoch Unterschiede. Die Verzweigungen an den Ufern sind weniger stark ausgeprägt, sodass die Ufer natürlicher wirken. Die Heuristiken weisen jedoch einen richtungsabhängigen Bias auf, dargestellt in (4).

Die Wahl der Heuristik hängt vom Tileset und den Anforderungen am Output ab. Einerseits führt die Heuristik der niedrigsten Entropie zu geringeren Fehlerraten. Außerdem gibt es keinen richtungsabhängigen Bias. Andererseits ist bei Tilesets mit sich gegenseitig stark einschränkenden Tiles, wie es bei Pfad-Tiles der Fall ist, der Einfluss der umgebungsabhängigen dynamischen Gewichtung auf die Auswahl der Tiles geringer.

Die vorgestellten Heuristiken ermöglichen der dynamischen Gewichtung einen größeren Einfluss auf den Output. Trotz des richtungsabhängigen Bias ist in diesem Beispiel die Verwendung der alternativen Heuristiken vorteilhaft.

5.2.3 Zweiter Durchlauf

Die Aufgabe des zweiten Durchlaufes ist es Zellen, die beim ersten Durchlauf einen zu geringen Kontext hatten, zurückzusetzen und mit mehr Kontext neu zu generieren.

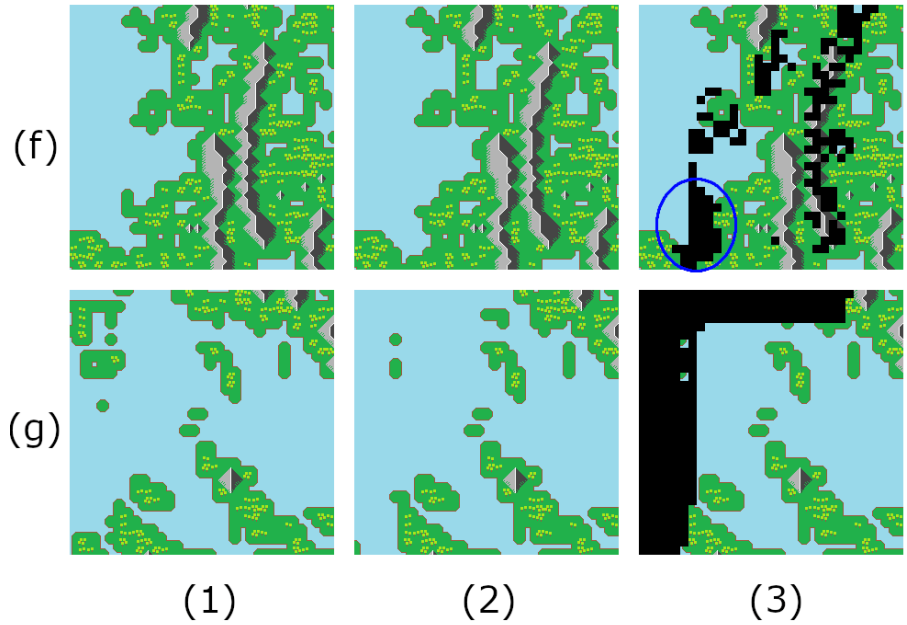

Abbildung 5.6: Ergebnis mit der niedrigsten Entropie Heuristik (f). Ergebnisse mit der Scanline Heuristik (g). Ergebnis des ersten Durchlaufes (1). Ergebnisse des zweiten Durchlaufes (2). Zurückgesetzte Tiles in schwarz (3).

Die Unterschiede zwischen dem ersten und zweiten Durchlauf sind bei der niedrigsten Entropie Heuristik gering. (f)

Aufgrund der Entropie Heuristik, werden zuerst Tiles gesetzt, die durch Nachbarzellen stark eingeschränkt sind (Ufer-Tiles, Gebirgs-Tiles). Das sind Tiles, die aufgrund des fehlenden Kontextes zurückgesetzt werden. Zellen, die stark durch Adjacency Constraints eingeschränkt waren, sind es auch nach dem Zurücksetzen. Damit Zellen nicht so stark eingeschränkt sind, müssen zudem die benachbarten Zellen, aufgrund eines geringen Kontextes, zurückgesetzt werden. Ein Block zurückgesetzter Zellen (blau umkreist) bietet genügend Freiraum, dass die Gewichte mit neuem Kontext berechnet werden können. Beim Vergleich von (f1) und (f2) ist zu sehen, dass die Küste leicht angepasst ist. Größtenteils sind die Zellen trotzdem soweit eingeschränkt, dass die dynamische Gewichtung kaum Einfluss hat.

Bei Verwendung anderer Heuristiken kann der zweite Durchlauf einen größeren Einfluss nehmen. Die Ergebnisse des mit der Scanline Heuristik zeigen auf, dass ein zusammenhängender Block an Zellen zurückgesetzt wird. Es wird genügend Freiraum geboten, um

einige Merkmale der Tilemap deutlich zu verändern. In (g2) wurde beispielsweise eine gesamte Insel durch Wasser ersetzt.

5.2.4 Laufzeiten

Jede Zelle muss mindestens einmal kollabiert werden, was eine Komplexität von $O(n)$ ergibt. Für jede dieser Zellen werden die Constraints auf alle anderen Zellen propagiert mit einer Komplexität von $O(n)$. Die Zeitkomplexität des WFC Alorithmus ist somit $O(n) \times O(n) = O(n^2)$.

Das Berechnen der dynamische Gewichtung, einschließlich dem Erfassen des Beobachtungsraumes, hat ebenfalls eine Komplexität von $O(n)$ und wird genauso oft wie die Propagierung ausgeführt. Der modifizierte Algorithmus hat somit eine Zeitkomplexität von $O(n) \times (O(n) + O(n)) = O(n^2)$.

Das Diagramm in Abbildung 5.7 bestätigt die Laufzeitkomplexität, durch ein quadratisches Wachstum der Laufzeiten. Der unmodifizierte WFC Algorithmus ist am schnellsten. Der modifizierte Algorithmus ist, aufgrund der zusätzlichen Komplexität durch des Beobachtungsraumes, langsamer. Die Laufzeit wird durch steigende Größe des Beobachtungsraumes schlechter. In Abbildung 5.8 ist zu sehen, dass der zweite Durchlauf die Laufzeit des Algorithmus leicht verschlechtert. Die Laufzeit des modifizierten Algorithmus hat trotzdem die gleiche Laufzeitkomplexität und sind somit hinsichtlich der besseren Ergebnisse akzeptabel.

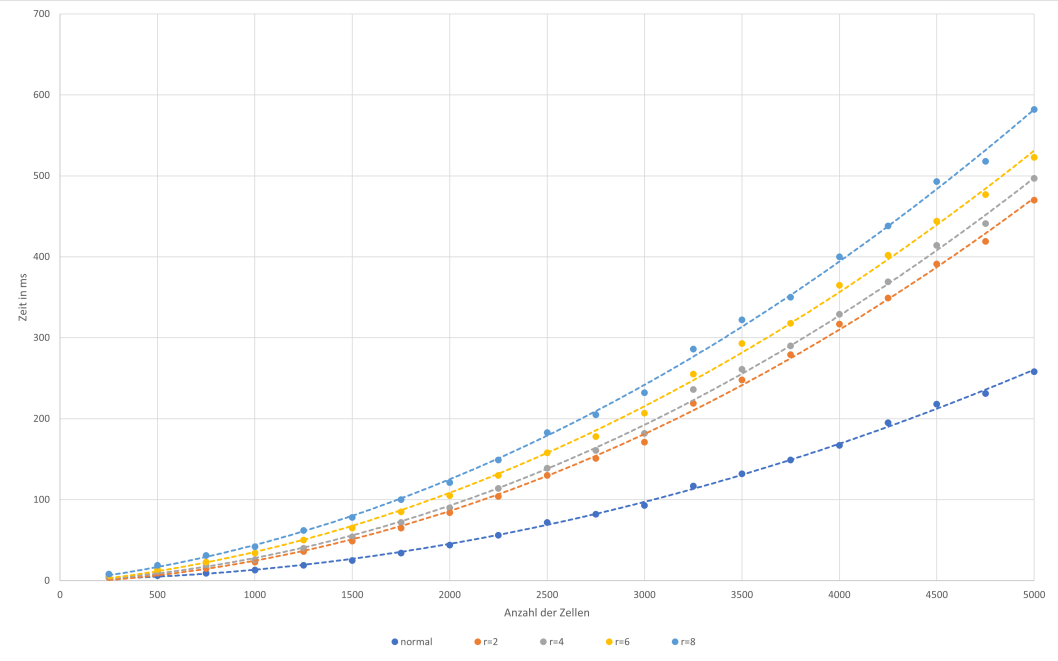

Abbildung 5.7: Laufzeiten des Algorithmus. Der Radius des Beobachtungsraumes ist durch r gegeben. Der zweite Durchlauf wird durchgeführt

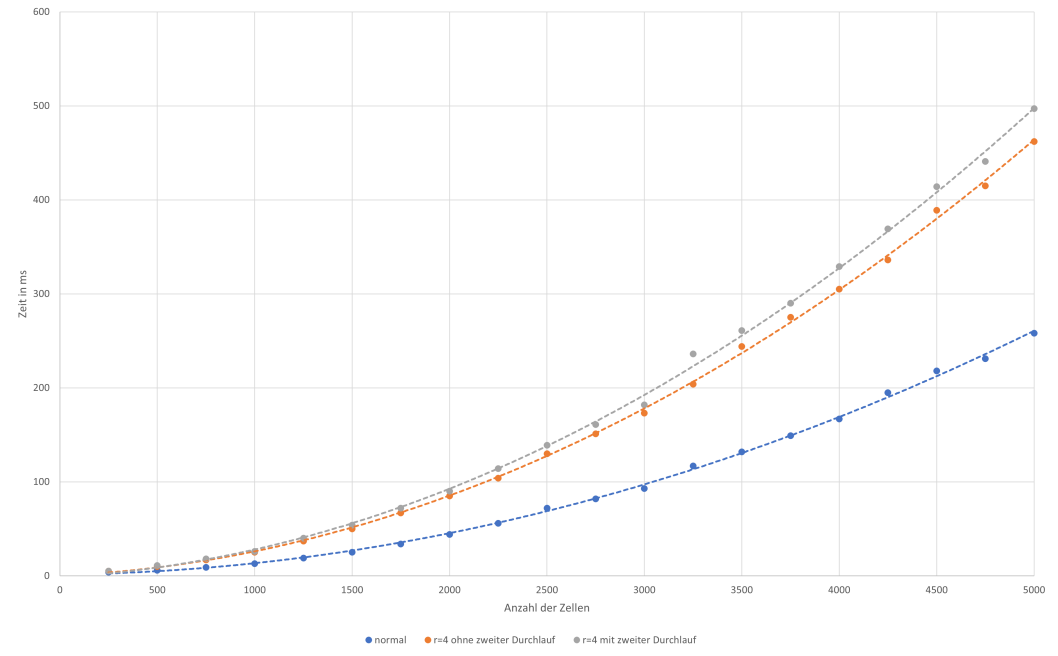

Abbildung 5.8: Vergleich der Laufzeit mit und ohne zweiten Durchlauf

5.3 Konfiguration

Der erste Schritt ist die Erstellung des Tilesets. Laut Anforderung 2.1 soll eine einfache Konfiguration möglich sein. Die Angabe der Attribute Tile-Kategorie, Symmetrie, Rotation und Reflexion ist eine simple Auswahl an vorgegebenen Attributen. Diese sind schnell und unkompliziert gesetzt.

Im Gegensatz dazu können bei der Setzung der Kantenlabels Fehler passieren. Die automatische Erstellung der Adjacency Constraints benötigt Kantenlabel mit der gleichen Anzahl an Segmenten. Somit muss zu Beginn feststehen, wie viele Segmente ein Kantenlabel besitzen soll. Falls ein Tile hinzugefügt wird, welches mehr Segmente zur Beschreibung der Kanten benötigt, müssen die Kantenlabel aller Tiles angepasst werden. Bei einem großen Tileset ist es möglich, dass der Benutzer aufgrund der Vielzahl der Kantenlabels die Übersicht verliert. Fehler sind jedoch im Output ersichtlich und können schnell korrigiert werden. Im Endeffekt erspart die Angabe der Labels dem Nutzer die manuelle Erstellung von Adjacency Constraints, was mit einem größeren Mehraufwand verbunden wäre.

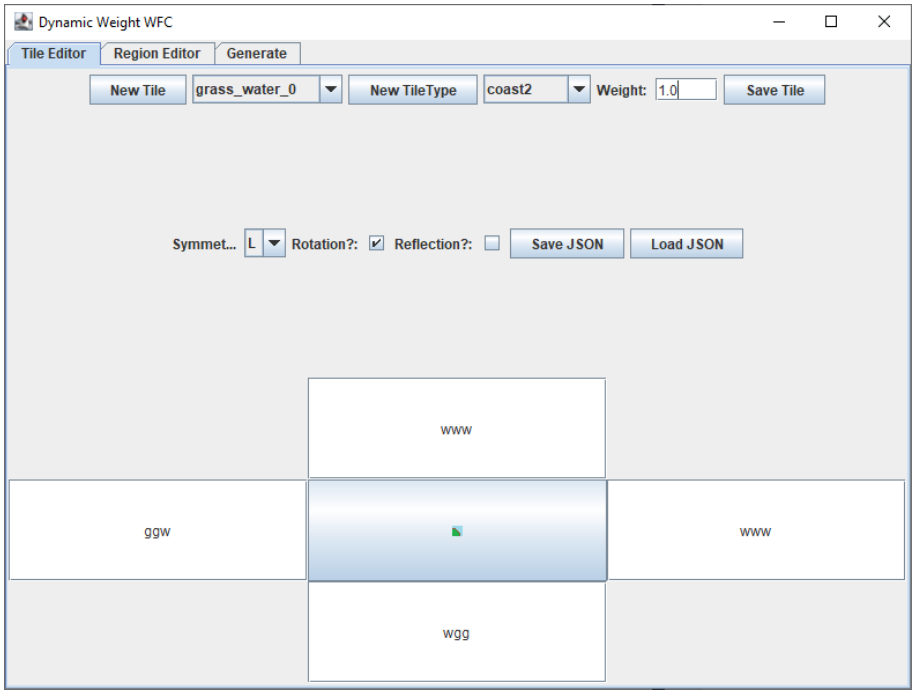

Abbildung 5.9: Konfiguration eines Tiles

Der zweite Schritt ist die Erstellung der Regelsätze. Entsprechend der Anforderung 2.2 soll eine einfache Konfiguration möglich sein. Das Erstellen der Region ist durch die Angabe von Name und Koordinaten des Mittelpunktes geschehen. Der Beobachtungsraum wird durch die Anzahl und Breite der Ringe angegeben.

Für jeden Ring legt der Nutzer fest, mit welchem Wert die Wahrscheinlichkeit, dass ein Tile der Tile-Kategorie A ausgewählt wird, wenn ein Tile der Tile-Kategorie B im Ring des Beobachtungsraumes vorhanden ist, angepasst wird. Problem dieses Ansatzes ist, dass es $T * T * R1 * R2$ (T = Anzahl der TileTypes, R1 = Anzahl der Ringe, R2 = Anzahl der Regionen) Eingabemöglichkeiten gibt. Bei einem großen Tileset mit vielen TileTypes wird die manuelle Eingabe sehr aufwändig, daher werden diese mit dem Wert 0 initialisiert. Der Nutzer kann so gezielt einzelne Werte anpassen.

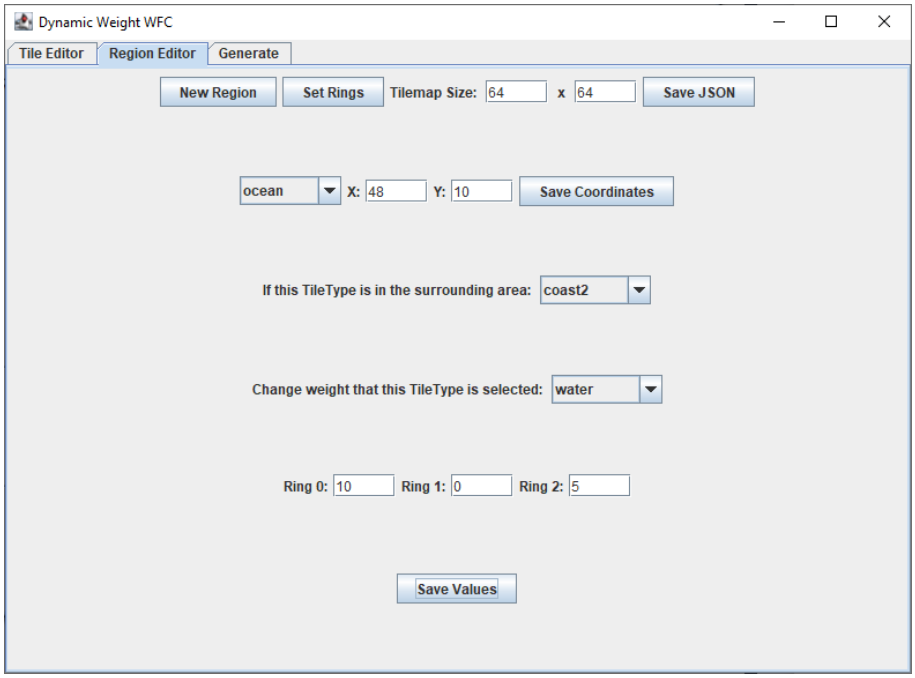

Abbildung 5.10: Konfiguration der Regelsätze

5.4 Nicht funktionale Anforderungen

Wie in Abschnitt Unterabschnitt 3.3.2 genannt, liegt der Fokus auf die nicht funktionalen Anforderungen der Benutzbarkeit und Wartbarkeit. Benutzbarkeit ist durch das User-Interface gegeben. Die Nutzung davon wurde in Abschnitt 5.3 diskutiert.

Die Wartbarkeit ist vor allem durch Modularität sichergestellt. Verantwortlichkeiten sind aufgrund der Model-View-Controller-Architektur klar getrennt. Innerhalb dieser Komponenten werden Anforderungen und Spezifikationen aufgeteilt und inkrementell umgesetzt, wodurch die Modularität weiter verbessert wird. Dies wird durch die Verwendung der objektorientierten Sprache Java unterstützt, wobei Javadoc-Kommentare dazu beitragen, den Code verständlich zu halten.

6 Schluss

6.1 Fazit

Im Rahmen dieser Arbeit wird ein Konzept zur Erweiterung von Maxim Gumin's Wave Function Collapse Algorithmus vorgestellt und ein Prototyp zur Anwendung des Konzeptes entwickelt.

Im WFC wird beim Kollabieren einer Zelle ein Tile zufällig ausgewählt, welches nach der Propagierung der Adjacency Constraints die Zelle passt. Diese zufällige Auswahl ist gewichtet. Im originalen Algorithmus werden Gewichte statisch vor der Laufzeit gesetzt. Ersetzt wird dies durch eine dynamisch zur Laufzeit berechnete Gewichtung. Diese hängt von zwei Faktoren ab. Zum einen von den bereits kollabierten Tiles im Beobachtungsraum um der Zelle. Zum anderen von der Region der Tiles, welche mittels Voronoi-Diagramm ermittelt wurden. Ein konfigurierbarer Regelsatz beschreibt, wie dieser Kontext zur Berechnung der Gewichte angewendet wird. Zusätzlich werden in einem zweiten Durchlauf des Algorithmus Tiles, die beim Kollabieren einen zu geringen Kontext hatten, zurückgesetzt und neu generiert.

Über eine graphische Benutzeroberfläche kann der Nutzer Tilesets erstellen. Tiles werden automatisch durch deren Transformationen erweitert. Adjacency Constraints werden für alle Tiles auf Basis der Kantenlabel automatisch generiert.

Der erweiterte WFC Algorithmus generiert Tilemaps, die im Gegensatz zum originalem Algorithmus mehr Variation bieten und besser als Game-Content geeignet sind. Die verbesserte Qualität ist jedoch mit einer schlechteren Laufzeit verbunden, die allerdings die gleiche Laufzeitkomplexität besitzt.

6.2 Ausblick

Das in dieser Arbeit vorgestellte Konzept bietet vielfältige Möglichkeiten für zukünftige Verbesserungen und Erweiterungen. WFC wurde bereits auf dreidimensionale Modelle übertragen [13]. Es ist naheliegend, das Konzept ebenfalls für die Generation mit dreidimensionalen Mustern anzupassen.

Es können verschiedene Heuristiken zur Erfassung des Beobachtungsraumes eingeführt werden. Einerseits kann die, durch den euklidischen Abstand gegebene, Kreisform ersetzt werden. Andererseits könnte die Einteilung des Beobachtungsraumes angepasst werden, sodass dieser nicht mehr in Ringe aufgeteilt wird.

Darüber hinaus wäre eine Anpassung/Verbesserung des Algorithmus zur Berechnung der dynamischen Gewichtung möglich. Falls im Beobachtungsraum spezielle Anordnung mehrerer Muster vorhanden sind, könnten diese einbezogen werden. Durch eine Anpassung des Algorithmus könnten die Regelsätze vereinfacht werden.

Das Konzept kann auch auf das Overlapping Modell des WFC Algorithmus übertragen werden. Da in diesem Verfahren Muster aus dem Input entnommen werden, wäre eine Herausforderung die Erstellung der benötigten Regelsätze. Gegebenenfalls können diese durch automatisch aus einem Input erstellt werden. Die verschiedenen Regionen können durch mehrere Input-Samples dargestellt werden, die jeweils eine Region repräsentieren.

Literaturverzeichnis

[1] Cheng, Darui ; Han, Honglei ; Fei, Guangzheng: *Automatic Generation of Game Levels Based on Controllable Wave Function Collapse Algorithm*. S. 37–50. In: *International Conference on Evolutionary Computation*, 2020. – ISBN 978-3-030-65735-2

[2] Dzaebel, Ben: *Erweiterung des Wave Function Collapse Algorithmus zur Synthese von 3D-Voxel-Modellen anhand von nutzergenerierten Beispielen*, Hochschule für Angewandte Wissenschaften Hamburg, Diplomarbeit, 2020

[3] Efros, Alexei ; Leung, Thomas: Texture Synthesis by Non-parametric Sampling. In: *Proceedings of the IEEE International Conference on Computer Vision* 2 (1999), 09

[4] Efros, Alexei A. ; Freeman, William T.: Image quilting for texture synthesis and transfer. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001)

[5] EllipticGames: *Rodina 1.3.4*. Steam. 2018. – URL https://steamcommunity.com/games/314230/announcements/detail/3369147113795750369. – [Online; abgerufen am 24-Juli-2023]

[6] FreeholdGames: *Caves of Qud*. Steam. 2015. – URL https://store.steampowered.com/app/333640/Caves_of_Qud/. – [Online; abgerufen am 24-Juli-2023]

[7] Gellel, Alexander ; Sweetser, Penny: A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels. In: *Proceedings of the 15th International Conference on the Foundations of Digital Games*. New York, NY, USA : Association for Computing Machinery, 2020 (FDG '20). – ISBN 9781450388078

[8] Gumin, Maxim: *Wave Function Collapse Algorithm*. 09 2016. – URL https://github.com/mxgmn/WaveFunctionCollapse. – [Online; abgerufen am 01-September-2023]

[9] Hendrikx, Mark ; Meijer, Sebastiaan ; Van Der Velden, Joeri ; Iosup, Alexandru: Procedural Content Generation for Games: A Survey. In: *ACM Trans. Multimedia Comput. Commun. Appl.* 9 (2013), feb, Nr. 1. – ISSN 1551-6857

[10] ISO/IEC: ISO/IEC 25010: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models / International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). 2011 (25010). – ISO/IEC Standard

[11] Karth, Isaac ; Smith, Adam M.: WaveFunctionCollapse is Constraint Solving in the Wild. In: *Proceedings of the 12th International Conference on the Foundations of Digital Games*. New York, NY, USA : Association for Computing Machinery, 2017 (FDG '17). – ISBN 9781450353199

[12] Karth, Isaac ; Smith, Adam M.: WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning. In: *IEEE Transactions on Games* 14 (2022), Nr. 3, S. 364–376

[13] Kleineberg, Marian: *Infinite procedurally generated city with the Wave Function Collapse algorithm — marian42.de*. 2019. – URL https://marian42.de/article/wfc/. – [Online; abgerufen am 02-August-2023]

[14] Kowarschick, Wolfgang ; Reenskaug, Trygve M. H. ; Stoiber, Dietmar ; Berkovitz, Joe: *Model-view-controller-paradigma*. – URL https://glossar.hs-augsburg.de/Model-View-Controller-Paradigma. – [Online; abgerufen am 16-Juni-2023]

[15] Langendam, Thijmen Stefanus L. ; Bidarra, Rafael: MiWFC - Designer Empowerment through Mixed-Initiative Wave Function Collapse. In: *Proceedings of the 17th International Conference on the Foundations of Digital Games*. New York, NY, USA : Association for Computing Machinery, 2022 (FDG '22). – ISBN 9781450397957

[16] Mackworth, Alan K.: Consistency in Networks of Relations. In: *Artif. Intell.* 8 (1977), feb, Nr. 1, S. 99–118. – ISSN 0004-3702

[17] Matthews, Elizabeth A. ; Malloy, Brian A.: Procedural generation of story-driven maps. In: *2011 16th International Conference on Computer Games (CGAMES)*, 2011, S. 107–112

[18] Merrell, Paul: Example-based model synthesis. In: *I3D '07: Symposium on Interactive 3D graphics and games*, 04 2007, S. 105–112

[19] Merrell, Paul: *Model Synthesis*, University of North Carolina at Chapel Hill, Dissertation, 2009

[20] Merrell, Paul: Example-Based Procedural Modeling Using Graph Grammars. In: *ACM Trans. Graph.* 42 (2023), jul, Nr. 4. – ISSN 0730-0301

[21] Merrell, Paul ; Manocha, Dinesh: Continuous Model Synthesis. In: *ACM Trans. Graph.* 27 (2008), 12, S. 158. ISBN 9781450318310

[22] Müller, Pascal ; Wonka, Peter ; Haegler, Simon ; Ulmer, Andreas ; Van Gool, Luc: Procedural Modeling of Buildings. In: *ACM Trans. Graph.* 25 (2006), 07, S. 614–623

[23] Newgas, Adam: Tessera: A Practical System for Extended WaveFunctionCollapse. In: *Proceedings of the 16th International Conference on the Foundations of Digital Games*. New York, NY, USA : Association for Computing Machinery, 2021 (FDG '21). – ISBN 9781450384223

[24] Nguyen, Phuc: *An introduction to parsing context-free language*. April 2020

[25] Okabe, Atsu ; Boots, Barry ; Sugihara, Kokichi ; Chiu, Sung: *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Bd. 43. 01 2000. – ISBN 9780471986355

[26] Oskar Stålberg, Richard M. ; Kvale, Martin: *Bad North*. Steam. 2018. – URL https://store.steampowered.com/app/688420/Bad_North_Jotunn_Edition/. – [Online; abgerufen am 24-Juli-2023]

[27] Raad Cisa, Lara ; Davy, Axel ; Desolneux, Agnès ; Morel, Jean-Michel: A survey of exemplar-based texture synthesis. In: *Annals of Mathematical Sciences and Applications* 3 (2017), 07

[28] Sandhu, Arunpreet ; Chen, Zeyuan ; McCoy, Joshua: Enhancing Wave Function Collapse with Design-Level Constraints. In: *Proceedings of the 14th International*

Conference on the Foundations of Digital Games. New York, NY, USA : Association for Computing Machinery, 2019 (FDG '19). – ISBN 9781450372176

[29] Santamaria, Oscar: *Incremental model: What it is and how to implement it.* Mar 2023. – URL https://www.plutora.com/blog/incremental-model-what-and-how-to-implement-it. – [Online; abgerufen am 14-Juni-2023]

[30] Scholz, Dominik: *Tile-Based Procedural Terrain Generation.* January 2019

[31] Shaker, Noor ; Togelius, Julian ; Nelson, Mark J.: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016

[32] Stiny, George ; Gips, James: 'Shape Grammars and the Generative Specification of Painting and Sculpture'. In: *IFIP Congress* Bd. 71, 01 1971, S. 1460–1465

[33] 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). – ISBN 9781450308724

[34] Wikipedia: *Iterative and incremental development — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Iterative%20and%20incremental%20development&oldid=1138733898. 2023. – [Online; abgerufen am 14-Juni-2023]

Glossar

Adjacency Constraint Regeln, die beschreiben wie benachbarte Elemente relativ zueinander angeordnet werden können.

Chunk Ein quadratisches Segment einer Tilemap aus $N \times N$ Zellen.

Entropie Maß für den Informationsgehalt einer Zelle einer Tilemap. Informationsgehalt entspricht der Anzahl der möglichen Muster für diese Zelle.

Kantenlabel Beschriftung, die spezifische Informationen über eine der Kanten eines Tiles darstellt.

model piece Bezeichnung der diskreten Modellstücke in der Modellsynthese. Dienen als Grundbausteine zur Synthese des größeren Outputs. Äquivalent zu einem Muster aus dem WFC.

Muster Diskrete Elemente im WFC, die als Grundbausteine zur Synthese des größeren Outputs dienen. Im zweidimensionalen sind das Tiles. Im dreidimensionalen können Muster 3D-Modelle sein. Äquivalent zu einem model piece aus der Modellsynthese.

Tile Quadratische Grafik, die zusammengesetzt mit weiteren Tiles ein größeres Gesamtbild ergeben.

Tilemap Ein zweidimensionales Gitter, in dem Tiles gleicher Größe zu einem Gesamtbild angeordnet werden.

Tileset Eine Sammlung individueller Tiles.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

PCG Prozedurale Content Generierung.

WFC Wave Function Collapse.

A Anhang

Listing A.1: tiles.json

```
{
  "tiles": {
    "corner": {
      "rotate": true,
      "mirror": false,
      "file": "corner.png",
      "top": "k", "right": "k", "bottom": "e", "left": "e",
      "symmetry": "L",
      "weight": 1.0,
      "tiletype": "corner"
    },
    "t": {
      "rotate": true,
      "mirror": false,
      "file": "t.png",
      "top": "e", "right": "p", "bottom": "p", "left": "p",
      "symmetry": "T",
      "weight": 1.0,
      "tiletype": "t"
    },
    "line": {
      "rotate": true,
      "mirror": false,
      "file": "line.png",
      "top": "e", "right": "p", "bottom": "e", "left": "p",
      "symmetry": "I",
      "weight": 1.0,
```

```
      "tiletype": "line"
    },
    "cross": {
      "rotate": true,
      "mirror": false,
      "file": "cross.png",
      "top": "p", "right": "p", "bottom": "p", "left": "p",
      "symmetry": "I",
      "weight": 1.0,
      "tiletype": "cross"
    },
    "empty": {
      "rotate": true,
      "mirror": false,
      "file": "empty.png",
      "top": "e", "right": "e", "bottom": "e", "left": "e",
      "symmetry": "X",
      "weight": 1.0,
      "tiletype": "emp"
    }
  },
  "offsets": {
    "top": [0,-1],
    "left": [-1,0],
    "bottom": [0,1],
    "right": [1,0]
  }
}
```

Listing A.2: rulesets.json

```
{
  "tilemap_size": {
    "x": 48, "y": 48
  },
  "regions": {
    "region_1": {
      "centerY": 0, "centerX": 0,
      "tiletypes": {
        "corner": {
          "rings": {
            "0": {
             "corner":-1, "t":3, "line":5, "cross":5, "emp":0
            },
            "1": {
              "corner":-1, "t":0, "line":0, "cross":5, "emp":0
            }
          }
        },
        "t": {
          "rings": {
            "0": {
              "corner":-1, "t":-1, "line":5,"cross":5, "emp":0
            },
            "1": {
              "corner":-1, "t":-1, "line":3, "cross":0, "emp":0
            }
          }
        },
        "line": {
          "rings": {
            "0": {
              "corner":-1, "t":-1, "line":2, "cross":1, "emp":3
            },
            "1": {
```

```
            "corner":-1, "t":-1, "line":3, "cross":5, "emp":0
          }
        }
      },
      "cross": {
        "rings": {
          "0": {
            "corner":-1, "t":-1, "line":3, "cross":1, "emp":2
          },
          "1": {
            "corner":-1, "t":-1, "line":2, "cross":3, "emp":2
          }
        }
      },
      "emp": {
        "rings": {
          "0": {
            "corner":-1, "t":10, "line":5, b"cross":0, "emp":0
          },
          "1": {
            "corner":-1, "t":5, "line":5, "cross":5, "emp":0
          }
        }
      }
    }
  },
  "ring_sizes": {
    "0": 3, "1": 3
  }
}
```

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