



Wave-Function-Collapse

Funktionsweise und Anwendungsfälle

Bachelorarbeit

Betreuer: Prof. Dr. Matthias Hopf

Studiengang: Bachelor Media Engineering

Fakultät: Elektrotechnik Feinwerktechnik Informationstechnik

Davoud Tavakol

Matrikelnummer: 3540912

Sommersemester 2023

Abgabedatum: 20. Juli 2023

Hinweis: Diese Erklärung ist in alle Exemplare der Abschlussarbeit fest einzubinden. (Keine Spiralbindung)

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Tavakol

Vorname: Davoud

Matrikel-Nr.: 3540912

Fakultät: EFI

Studiengang: Media Engineering

Semester: 8

Titel der Abschlussarbeit:

Wave-Function-Collapse - Funktionsweise und Anwendungsfälle

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Nürnberg 28.06.2023

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung der/des Studierenden zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,
 genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigefügt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Nürnberg 28.06.2023

Ort, Datum, Unterschrift Studierende/Studierender

Datenschutz: Die Antragstellung ist regelmäßig mit der Speicherung und Verarbeitung der von Ihnen mitgeteilten Daten durch die Technische Hochschule Nürnberg Georg Simon Ohm verbunden. Weitere Informationen zum Umgang der Technischen Hochschule Nürnberg mit Ihren personenbezogenen Daten sind unter nachfolgendem Link abrufbar: <https://www.th-nuernberg.de/datenschutz/>

Zusammenfassung

In dieser Bachelorarbeit wird auf die Funktionsweise des Wave-Function-Collapse (WFC) Algorithmus von Maxim Gumin eingegangen und wie dieser für Textursynthesen sowie zur Erstellung von prozedural generierten Content verwendet werden kann. Dazu wird zuerst auf gängige Textursynthese Algorithmen eingegangen und inwiefern sich WFC von diesen Unterscheidet. Später wird die Model Synthese von Paul Merrell herangezogen um Ähnlichkeiten zwischen den beiden Algorithmen aufzuzeigen und zu vergleichen. Es wird klar das beide Algorithmen sich in ihrer Methodik sogenannte Constraint-Satisfaction-Probleme (CSP) zu lösen als identisch erweisen. Danach werden die Unterschiede der jeweiligen Implementierungen der Algorithmen aufgezeigt, wie z.B. die Blockweise Generierung der Model Synthese und das Overlapping Model von Gumin. Zum Schluss wird das Overlapping Modell detailliert dargestellt und darauf eingegangen, weshalb diese Implementation zu Bekanntheit gelangt ist und welche Anwendungsfälle mit WFC generell möglich sind.

Abstract

This bachelor thesis describes the Wave-Function-Collapse (WFC) algorithm by Maxim Gumin and how it can be used for texture synthesis and the creation of procedurally generated content. Firstly, common texture synthesis algorithms and how WFC differs from them are discussed. Later, the model synthesis of Paul Merrell is used to show similarities between the two algorithms and compare them. It becomes clear that both algorithms are identical in their methodology to solve so-called constraint satisfaction problems (CSP). Afterwards, the different implementations of the respective algorithms are pointed out, such as the modifying in blocks generation of the model synthesis and the overlapping model of Gumin. Finally, the Overlapping Model is explained in detail and why this implementation has gained so much attention lately and which use cases are generally possible with WFC.

Inhaltsverzeichnis

| | |
|--|-----------|
| 1 Einleitung | 5 |
| 2 Textursynthesen im Vergleich | 6 |
| 2.1 Pixel basierende Textursynthese | 7 |
| 2.2 Pyramid basierte Textur Synthese | 9 |
| 2.3 Patch basierte Textursynthese | 11 |
| 3 Wave-Function-Collapse | 13 |
| 3.1 Constraint-Satisfaction-Problem (CSP) | 14 |
| 3.1.1 CSP-Beispiel an Sudoku | 16 |
| 3.2 WFC und die Model Synthese | 19 |
| 3.2.1 Unterschiede zwischen WFC und Model Synthese | 21 |
| 3.2.2 Simple Tile Model und Overlapping Model | 23 |
| 3.3 Beispiel Implementierung | 28 |
| 3.3.1 Weitere Implementierungen von WFC | 31 |
| 4 Fazit | 34 |
| 5 Anhang | 37 |

1. Einleitung

Die automatische Generierung von Inhalten wie Texten, Bildern oder Modellen ist heutzutage Standard in vielen Bereichen der Industrie. Um solche Inhalte mit vordefinierten Parametern zu erstellen, werden oft zwei Methoden zur Generierung verwendet. AI's (Künstliche Intelligenzen) wie DALL-E2, Midjourney und Algorithmen wie die Patch Synthese. Vorteile dieser Werkzeuge ist es, dass sie in kurzer Zeit qualitativ hochwertige Resultate generieren können und auch, wie oben erwähnt, vordefinierte Parameter als Input erhalten können, um die Resultate für ihren Gebrauch anzupassen. In dieser Bachelorarbeit wird auf den von Maxim Gumin erstellten Wave-Function-Collapse Algorithmus, dessen Herkunft, Funktionsweise und Anwendungsfälle eingegangen. Zuerst werden gängige nicht interaktive Textursynthesen wie die Pixel basierende Textursynthese, Pyramid basierende Textursynthese und der Patch basierende Textursynthese erläutert und wie Wave-Function-Collapse sowie andere Textursynthesen davon inspiriert wurden. Daraufhin wird das Konzept zum Lösen von Constraint-Satisfaction-Problemen (CSP) und die Funktionsweise vom WFC Algorithmus, solche Probleme zu lösen, dargestellt. Anhand eines Beispiels wird im Detail die Funktionsweise zum Lösen solcher CSP's aufgezeigt. Wave-Function-Collapse als Algorithmus für Procedural-Content-Generation (PCG) wird mit der Model Synthese von Paul Merrell verglichen und deren Unterschiede genauer betrachtet. Es wird deutlich, dass der Wave-Function-Collapse Algorithmus sowie die Model Synthese beide sowohl in ihrer Verwendung von Pixelgenauer / diskreten Synthese sowie im Lösen von CSP ähnliche Methoden verwenden und beide sich dadurch als PCG's verwenden lassen. Dadurch kann der Output dieser Algorithmen interaktiv als Content verwendet werden wie beispielsweise für Videospiele etc.

2. Textursynthesen im Vergleich

Es gibt viele Möglichkeiten Textursynthese mit Algorithmen zu erzielen. Die meisten dieser Methoden basieren auf demselben Grundprinzip, aus kleineren Input-Images größere oder gleich große Output-Images zu generieren. Nach D.Gomathi und Rajvi Shah [1, S.1], kann die Textursynthese wie folgt definiert werden.

Ziel einer Textursynthese ist es aus einer Texturprobe eine neue Textur zu generieren, die „wenn sie von einem menschlichen Beobachter wahrgenommen wird, durch denselben zugrundeliegenden Prozess erzeugt wird.“ Das Ergebnis muss der Texturprobe ähnlich sein aber dennoch in der Wahrnehmung genügend Variation enthalten. Eine solche Synthese ist nicht möglich, wenn man die Eingabetextur einfach mehrfach kachelt. Dadurch erhält man keine „sauberen“ Übergänge, und die einzelnen Blöcke sind klar erkennbar (siehe Abbildung 2.1).

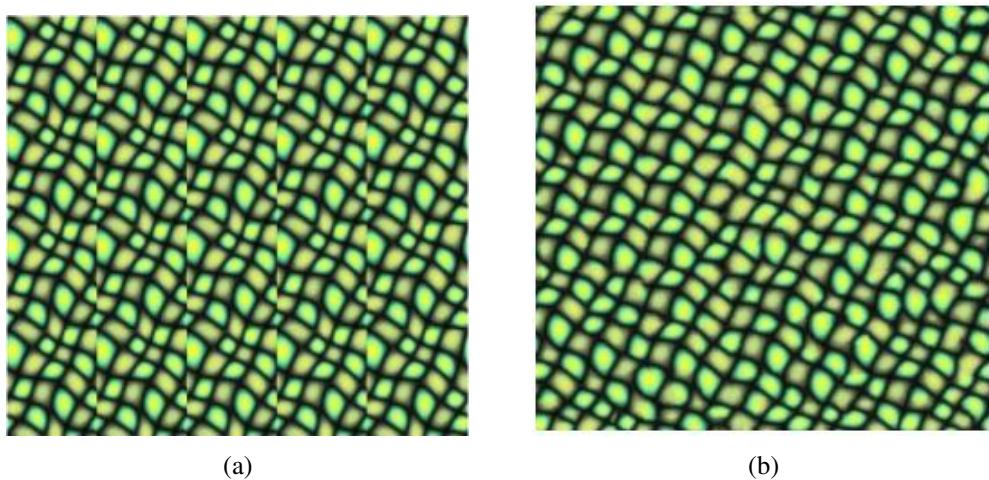


Abbildung 2.1: (a) Blockartige Textur, (b) Textursynthese

Dadurch ergeben sich folgende Metriken, die erfüllt werden müssen.

- Bei gegebener Texturprobe, generiere eine neue Textur, die der Probe gleicht.
 - Die neue Textur kann eine beliebige Größe haben, die vom Benutzer festgelegt wird.
 - Es sollen keine sichtbaren Übergänge, Artefakte oder fehlerhafte Kanten sichtbar sein.

- Dasselbe Muster soll nicht mehrfach in der neuen Textur vorkommen. [1, S.2]

Im Folgenden soll auf drei in ihrem Prinzip und ihrer Wirkungsweise unterschiedliche Textursyntheseverfahren eingegangen werden, um damit verschiedene Ansätze von Synthesen aufzuzeigen. Da eine vertiefte Diskussion für diese Arbeit nicht relevant ist, werden hierbei lediglich die groben Prinzipien / Hintergrundinformationen geklärt.

2.1 Pixel basierende Textursynthese

Bei dieser Methode werden neue Texturen Pixel für Pixel generiert. Jeder neue Pixelwert wird von seinen lokalen Nachbarn festgelegt. Diese Verfahren verwenden meistens Markow-Netzwerke (*Markov Random Field*),¹ die relativ gute Resultate liefern mit wenig Rechenlast. Markov Random Fields Methoden beurteilen jeden Pixel nach einer kleinen Menge von Nachbarn. Voraussetzung hierfür ist, dass das Input-Image stationär und lokal ist. Ein Image wird als stationär bezeichnet, wenn unter korrekter Fenstergröße, jeder betrachtete Bereich ähnlich zueinander aussieht. Lokal ist ein Image dann, wenn jeder Pixel allein von seinen Nachbarn bestimmt werden kann [1].

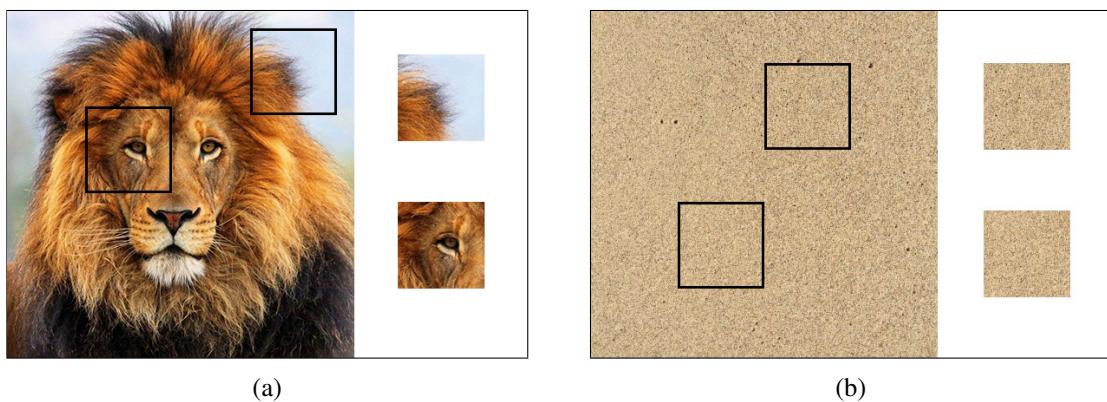


Abbildung 2.2: (a) Nicht stationär und lokal, (b) stationär und lokal.

Unterschiedliche Bereiche einer Textur sehen sich immer ähnlich (siehe Abbildung 2.2 (b)). Dies ist nicht der Fall für normale Images wie wir bei Abbildung 2.2 (a) erkennen können. Zudem ist es möglich jeden Pixel in (b) allein durch seine benachbarten Pixel zu bestimmen. Diese Attribute bezeichnet man als Stationär und Lokal [1]. Im Folgenden ist die Funktionsweise eines Algorithmus basierend auf Markov Random Fields nach Efros und T. Leung dargestellt: [2]

¹Auf Markow-Netzwerke und Markow-Ketten wird am Ende dieses Abschnitts genauer eingegangen.

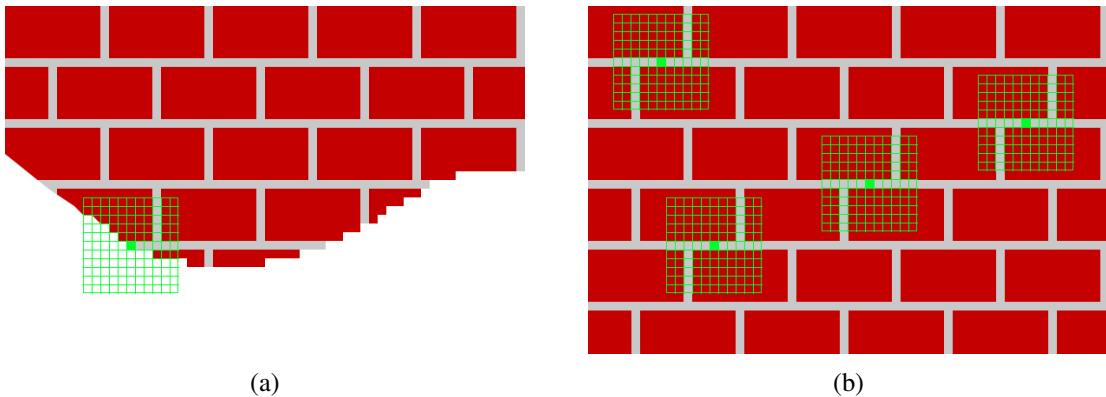


Abbildung 2.3: (a) Pixelsynthese, (b) Sampling des Input-Images.

- Zuerst wird vom Input-Image eine Teiltextrur einer bestimmten Größe (z.B. 5×5 Pixel Fenster) ausgewählt. Von diesem Feld aus werden Spiralförmig neue Pixel generiert.
 - Für jeden Pixel der betrachtet wird, wird ein Fenster einer selbst bestimmten Größe zentral über das Pixel gelegt. Die Größe des Fensters muss nach Größe der einzelnen Elemente der Textur gewählt werden (siehe Abbildung 2.3 (a)).
 - Mit dieser Gruppe von Pixel (der Zentrale Pixel und alle seiner Nachbarn im Fenster) werden nun alle im Input-Image ähnlichen N Kandidaten gesucht (siehe Abbildung 2.3 (b)).
 - Danach wird zufällig aus einer dieser möglichen Kandidaten einer ausgewählt und der betrachtete Pixel im Output wird aus diesem Kandidaten kopiert.
 - Dieser Prozess wiederholt sich so lange, bis alle nicht bekannten Pixel generiert wurden [1, S.4].

Markow-Netzwerk und Markov-Kette

Wie bereits erwähnt basieren einige Textursynthesen auf sogenannten Markow-Netwerke oder Markov-Ketten (*Markov Chain*). In diesem Abschnitt wird kurz erklärt, um was es sich genau bei diesen beiden Begriffen handelt.

Ein Markow-Netzwerk kann als Generalisierung einer Markow-Kette im 3D-Raum angesehen werden. Eine Markow-Kette dient dazu Wahrscheinlichkeiten zukünftiger Ereignisse anzugeben. Sie basiert auf der Theorie, dass bereits begrenzte Informationen über den vergangenen Zustand eines Systems ausreichen, um Prognosen für die zukünftige Entwicklung des Systems zu erstellen. Die Markow-Kette, oder auch Markow-Kette erster Ordnung,

beschreibt konkret: "Der zukünftige Zustand des Prozesses ist nur durch den aktuellen Zustand bedingt und wird nicht durch vergangene Zustände beeinflusst." [3]

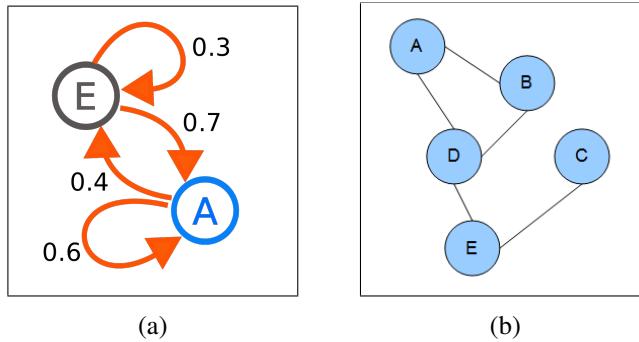


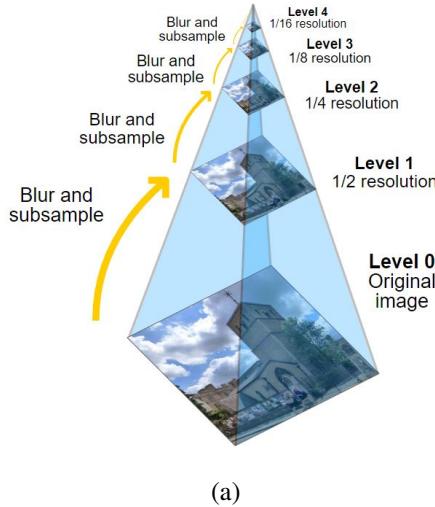
Abbildung 2.4: (a) Markow-Kette mit 2 möglichen Zuständen und deren Wahrscheinlichkeiten [3]. Beispiel: Falls zustand E , dann gibt es eine 30% Chance das der Zustand E wieder Eintritt oder eine 70% Chance das A Eintritt. Falls wir den Zustand A haben dann sind es 60% für A und 40% für E . (b) Ein Markow-Netzwerk mit seinen Abhängigkeiten [4]. Beispiel: A ist abhängig von B und D . B von A und D . D von A , B und E . E von D und C . Und zuletzt ist C nur von E abhängig.

In einer Markow-Kette erster Ordnung hängt der Zustand nur vom vorhergehenden Zustand ab, während bei einer Markow-Kette höherer Ordnung, somit ein Markov-Netzwerk, jeder Zustand von seinen Nachbarn in einer von mehreren Richtungen abhängt [4]. Ein Markov-Netzwerk kann als Feld oder Graph von Zufallsvariablen dargestellt werden, wobei die Verteilung jeder Zufallsvariablen von den benachbarten Variablen abhängt, mit denen sie verbunden ist (siehe Abbildung 2.4).

2.2 Pyramid basierte Textur Synthese

Bei der Pyramid-Methode wird das Verfahren der Bildpyramide verwendet. Hierbei werden aus dem Input-Image mehrere Output-Images in verschiedenen Auflösungen mithilfe von Glättung und Downsampling generiert (siehe Abbildung 2.5) [5].

Zudem wird ein Bildrauschen, (*Noise-Image*) der i.d.R uniform weiß ist, verwendet. Das Noise-Image wird dann durch Histogram-Matching und der Image-Pyramid so verändert, dass es dem Input-Image ähnlich ist.



(a)

Abbildung 2.5: (a) Bildpyramide

Histogram-Matching ist die Generalisierung des Punktoperators, mehr spezifisch, der Histogrammäqualisation. Bei der Histogrammäqualisation (auch Histogrammausgleich, Histogrammegalisierung oder Histogrammequalisierung genannt) werden die Kontraste von Grauwertbildern derart verbessert, sodass diese über eine bloße Kontrastverstärkung hinausgeht. Dabei wird die Gleichverteilung mithilfe der Grauwertverteilung berechnet, damit der gesamte zur Verfügung stehende Wertebereich optimal ausgenutzt wird [6]. Bei dem Fall der Pyramid-Methode nimmt der Algorithmus ein Input-Image und bildet ein bestimmtes Histogramm, indem das Bild mit mehreren Nachschlagetabellen verarbeitet wird.

Die beiden Nachschlagetabellen Tabellen sind:

- Die kumulative Verteilungsfunktion (*cumulative distribution function (CDF)*) eines Bildes und
- die inverse kumulative Verteilungsfunktion eines Bildes.

Die CDF ist eine Nachschlagetabelle, die das Intervall [0,256] auf das Intervall [0,1] abbildet. Die inverse CDF ist eine Nachschlagetabelle, die von [0, 1] auf [0, 256] zurückführt. Sie wird (mit linearer Interpolation) neu abgetastet, sodass die Stichproben gleichmäßig auf dem Intervall [0, 1] verteilt sind [5].

Während der Algorithmus weiter iteriert, beginnt das Noise-Image dem Input-Image zu ähneln wie wir Abbildung 2.6 (c) erkennen können. Der Prozess stoppt, wenn eine ausreichende Ähnlichkeit erreicht ist oder eine festgelegte Anzahl von Iterationen erreicht wurde.

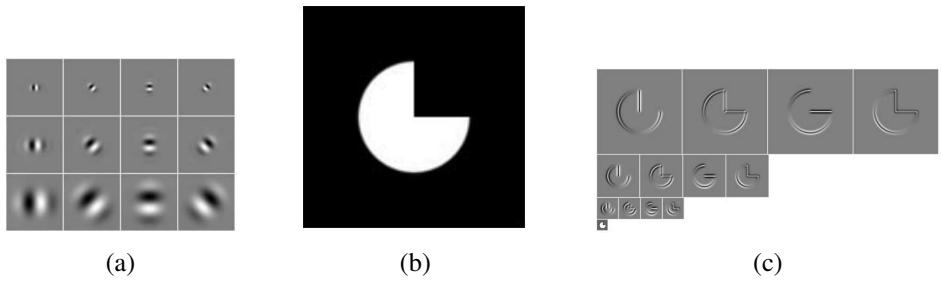


Abbildung 2.6: (a) Projektion der Bildpyramide, (b) Input-Image, (c) Teilband-Bilder vom Input-Image.

Die Pyramid-Methode ist eine weitere übliche Synthesemethode die, im Detail, sehr komplex ist und für das weitere Verständnis in dieser Arbeit keine Relevanz hat. Daher wird sie nicht näher betrachtet.

2.3 Patch basierte Textursynthese

Die Patch basierte Textursynthese (*auch Quilten genannt*) ist eine Erweiterung der Pixel basierten Textursynthese. Hier werden statt einzelnen Pixeln ganze Felder (*Patches*) verglichen und generiert (siehe Abbildung 2.7). Dabei werden die Patches mithilfe ihrer Nachbarn bestimmt und ausgewertet. Dies hat zur Folge, dass sich die Qualität und die Geschwindigkeit des Algorithmus erhöht.

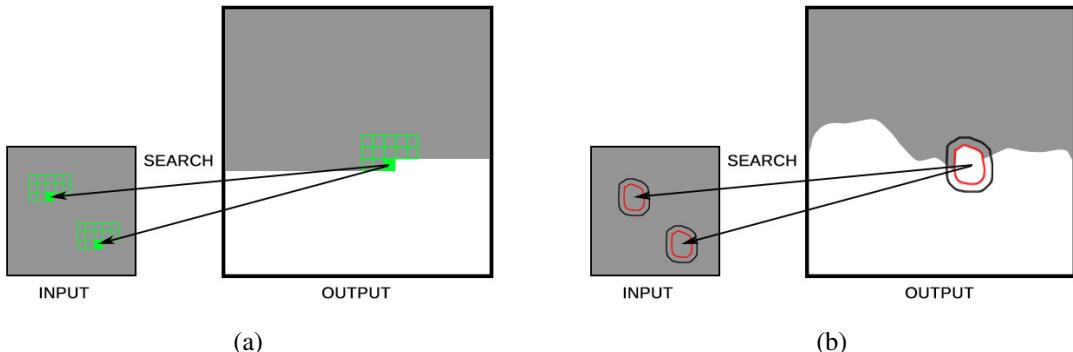


Abbildung 2.7: (a) Pixel basiert, (b) Patch basiert.

Ein Problem dieser Synthese im Vergleich zur Pixel basierten Synthese ist, dass sich hier die neuen Patches mit bereits vorhandenen Patches überschneiden. Es gibt viele Methoden, um dieses Problem zu lösen (siehe Abbildung 2.8). Eine davon ist es, Patches mit verschiedenen Größen zu verwenden, damit die Konfliktbereiche zwischen den Patches durch das

Phänomen der visuellen Maskierung (*Visual Masking*) reduziert werden. Insbesondere bei stationären Texturen erreicht diese Methode gute Ergebnisse [7].

Die Funktionsweise des Algorithmus nach D.Gomathi und Rajvi Shah:

1. Generierung des ersten Patches in der oberen-linken Ecke des Output-Images. Der Patch wird zufällig aus dem Input-Image ausgewählt.
2. Von links-nach-rechts und von oben-nach-unten werden im Output-Image folgende Aufgaben ausgeführt.
 - Auswahl des nächsten hinzuzufügenden Feldes aus dem Input-Image aus einem der am besten passenden Patches.
 - Berechnen der Fehlerfläche zwischen diesem neuen Patch und seinem Überlappungsbereich mit bereits verarbeiteten Patches.
 - Berechnen des Pfades mit den geringsten Kosten durch die Fehlerfläche, um die Patch-Grenze zu bestimmen, und fügen Sie dann den neuen Patch zum Output-Image hinzu (siehe Abbildung 2.8 (c)).

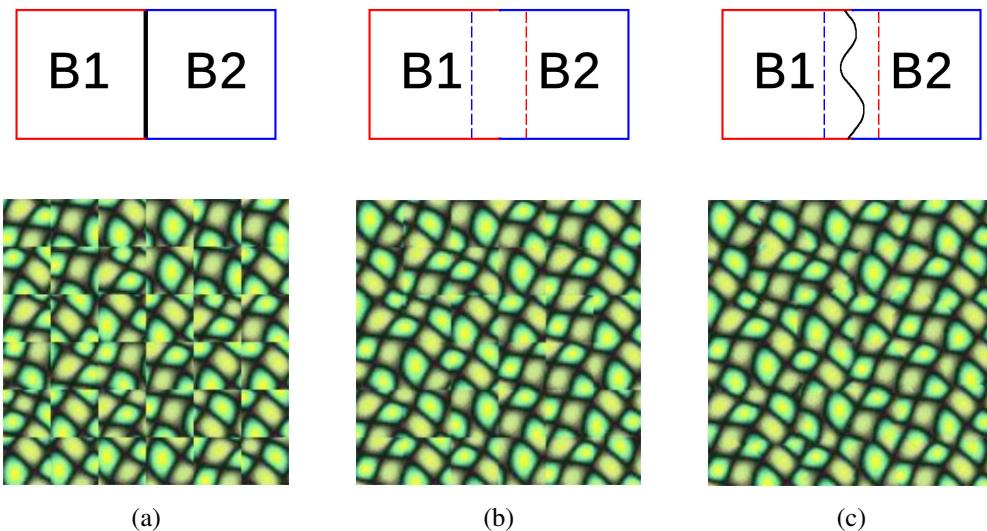


Abbildung 2.8: (a) Zufällige Patch Anordnung, (b) Patches eingeschränkt durch Nachbarn, (c) Minimal Fehler Randschnitt. Nach Alexei A. Efros und William T. Freeman [7].

3. Wave-Function-Collapse

Wichtig bei allen Textursynthesen ist, dass die Muster des Output-Images immer lokal ähnlich oder gleich dem des Input-Images sein müssen. Das wird größtenteils dadurch erzielt, dass aus dem Input-Image kleinere Subimages, Patches oder Pixel extrahiert werden. Bei den Verfahren, bei denen die lokale Ähnlichkeit nicht 1-zu-1 bzw. pixelgenau stattfindet, werden die Pixel und deren Farbwert oft nach Grundlage der Abstandsmetrik (z.B. dem euklidischen Abstand von Pixelfarbvektoren) beurteilt. Solche Verfahren finden meistens in der visuellen Computergrafik Anwendung. Diese Methodiken haben große Nachteile im Gegensatz zu Algorithmen, bei denen das lokale Muster des Outputs pixelgenau dem Input-Image gleicht. Gerade für PCG (Procedural-Content-Generation) kann die Pixelgenauigkeit von großem Nutzen sein, da dadurch Abgrenzungen der Pixel innerhalb des Output-Images klar definiert sein können [8]. Von allen oben beschriebenen Textursynthese-Methoden ist der WFC der Patch basierten Methode am ähnlichsten.

Gumin hat von Paul Merrell's Arbeit inspirieren lassen, obwohl dieser sich hauptsächlich mit der Generierung von 3D-Modellen beschäftigte. Bei Merrell's Verfahren werden die Modelle mithilfe von bereits erstellten Bausteinen zusammengesetzt. Das ist dahingehen von Vorteil, da sich in vielen Textursynthesen die Übergänge der Pixel vermischen und sich somit Artefakte bilden können. Dieses Verhalten ist bei WFC und dem Verfahren von Paul Merrell nicht möglich, da es sich um eine diskrete Synthese handelt. Jedes lokale Muster ist immer im Input wiederzufinden (siehe Abbildung 3.1) [8]–[10].

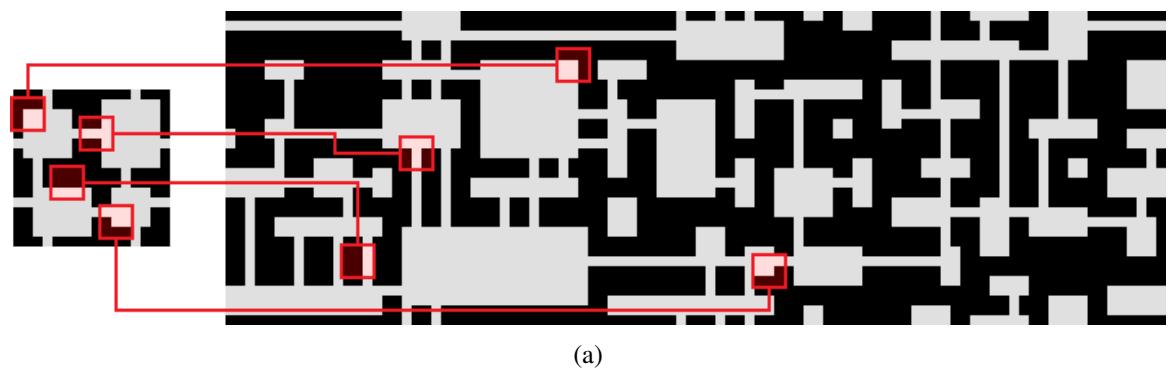


Abbildung 3.1: (a) WFC generiertes Muster.

3.1 Constraint-Satisfaction-Problem (CSP)

Der WFC von Gumin ist lose an der Quantenmechanik angelehnt. Das liegt daran, dass bei der Synthese von WFC in jeder Zelle des $N \times N$ Output-Images theoretisch jedes Muster / jeder Pixelwert vorkommen kann, bevor dies final festgelegt werden. Dieser Zustand nennt sich *Superposition* (siehe Abbildung 3.2). Jede Zelle hat mehrere Eigenwerte (*eigenstates*) und somit auch eine maximale Entropie bzw. einen maximalen Informationsgehalt. Auch hat jede Zelle bestimmte Regeln, die sich aus deren Nachbarzellen ergeben und erfüllt werden müssen. Denn nicht jede Zelle kann an jeder Position auftauchen, sobald bereits eine Zelle ein Eigenwert erhält (siehe Patch basierte Textursynthese. Nicht jeder Patch kann an einer beliebigen Position sein, da sie sich den Nachbarn anpassen müssen, damit die Synthese nahtlos ist.) Sobald eine Zelle bekannt wird (*Observation*) und nur einen Eigenwert besitzt, wird die Entropie aller anderen Zellen angepasst. "Nach dieser Auffassung ist der "collapse" der Wellenfunktion kein physikalischer Prozess und spiegelt lediglich eine Aktualisierung unserer Informationen über das System wider." [11, S.5, 2.2 The wave function] Da der WFC Algorithmus genau auf diese Art und Weise die Synthese durchführt, kann der WFC auch als Löser für solche Bedingungserfüllungsprobleme (*Constraint-Satisfaction-Problem*) verwendet werden.

Grundsätzlich beschreiben CSP's Gruppen von Objekten, denen Variablen zugeteilt sind. Diesen Objekten sind Regeln, sogenannte (*constraints*) auferlegt, die erfüllt werden müssen. Jeder dieser Objekte hat zu Beginn eine Superposition und kann somit grundsätzlich jeden Wert enthalten. Die Aufgabe von Algorithmen zum Lösen von CSP's (*solver*) ist es einen Zustand (*State*) zu finden, in dem alle constraints erfüllt sind und jedem Objekt nur noch ein Wert zugeordnet ist [12]. Solche Probleme finden sich oft bei der Künstlichen Intelligenz und bei Operations Researchs wieder. Im Fall von WFC sind die Objekte, denen die Variablen zugeteilt sind, die einzelnen Bereiche im Output-Image. Jedem dieser Bereiche muss ein lokales Muster aus dem Input zugeordnet werden. Immer, wenn einem Bereich ein Wert zugeordnet wird, werden auch die benachbarten Bereiche damit beeinflusst (*Propagation*). Der Gesamtprozess, wenn sich eine Gruppe aus Superpositionen mit mehreren Eigenwerten zu einem einzelnen Eigenwert aufgrund von Interaktion mit der Außenwelt (*Observation*) reduziert, nennt sich Wave-Function-Collapse [11]. Während dem Prozess einen gültigen State für das CSP zu finden, so kann es immer Situationen geben, in dem es mehrere gültige Optionen / Eigenwerte für ein Objekt gibt. Wenn eine solche Situation auftritt, dann haben verschiedene Solver verschiedene Lösungsansätze. Einige Algorithmen wählen zufällig einen der möglichen Werte aus den momentan zulässigen Optionen. Bei diesem Ansatz kann es sein, dass der Algorithmus nicht auf einen Zustand kommen kann,

bei dem alle constraints erfüllt werden können. In so einem Fall gibt es Rücksetzverfahren (*Backtracking*), bei dem der Algorithmus zu seinem letzten Ergebnis zurückfällt und einen anderen Wert für die Variable setzt, um den ungültigen Zustand zu verlassen und diesen neu zu berechnen. Andere Algorithmen verwenden zusätzliche Heuristiken abgesehen von den bereits bekannten constraints, um die Möglichkeit eines ungültigen Zustandes zu reduzieren [8].

3.1.1 CSP-Beispiel an Sudoku

Es gibt viele verschiedene CSP Probleme in der Wissenschaft. Eines der bekanntesten Probleme, welches Menschen nahezu täglich unbewusst lösen, ist Sudoku. Sudoku ist ein CSP welcher am besten auch die Funktionsweise von Gumin's WFC aufzeigt, da die Constraints der Probleme sehr ähnlich sein können. Wenn ein Sudoku Feld ohne initiale Zahlen betrachtet (*unobserved*) wird, kann in jedem einzelnen Objekt, in diesem Fall auch Zelle, theoretisch jede Zahl vorhanden sein (*Superposition*) [13].

| | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 |
| 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 |
| 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 |
| 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 |
| 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 |
| 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 |
| 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 |
| 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 |
| 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 |
| 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 |
| 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 |
| 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 |
| 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 |
| 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 |
| 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 |
| 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 |
| 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 | 4 5 6 |
| 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 | 7 8 9 |

(a)

Abbildung 3.2: (a) Sudoku Feld unbeobachtet in einer Superposition.

Die Regeln (*constraints*) dieses CSP sind wie folgt:

1. Jede Zeile, Spalte und jedes Quadrat (je 9 Felder) muss mit den Zahlen 1-9 ausgefüllt werden.

2. Keine Zahl innerhalb der Zeile, Spalte oder des Quadrats darf sich wiederholen.

In Abbildung 3.3 erkennen wir den ersten Schritt der initiale *Observation*, dabei wird das erste Feld mit seinen vielen Eigenwerten auf einen festen Wert kollabiert. Dadurch treten für die benachbarten Zellen die Constraints ein, die bestimmen, welche Werte diese noch einnehmen können [13].

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 6 | 4 | 6 | 4 | 6 | 4 | 6 | 4 | 6 | 4 | 6 | 4 | 6 | 4 | 6 | 4 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 |

(a)

Abbildung 3.3: (a) Sudoku Feld mit dem ersten Input.

Nach dem ersten Input reduziert sich die Entropie aller Zellen in jeder betroffenen Zeile, Spalte und Quadrat bzw. ihre Anzahl an möglichen Eigenwerten reduziert sich. Als nächsten Schritt könnte jetzt eine beliebige Zelle ausgewählt und diese auf einen ihrer zufälligen möglichen Eigenwerte gesetzt werden. Allerdings besteht dabei immer die Möglichkeit, dass der Prozess einen Zustand erreicht, an dem kein gültiger Zustand erreicht werden kann. Um die Wahrscheinlichkeit dieses Szenarios zu reduzieren, können Heuristiken ein-

gesetzt werden, um eine Auswahl zu treffen die zu einem gültigen Endzustand führt. In diesem Fall wird eine Zelle ausgewählt, die bereits eine sehr niedrige Anzahl an möglichen Eigenwerten hat, bzw. eine sehr niedrige Entropie besitzt. Dadurch ist die Wahrscheinlichkeit geringer, dass eine Zelle auf einen nachteiligen Wert gesetzt wird, der später zu einem ungültigen Endzustand führt (siehe Abbildung 3.4) [13].²

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | | | | |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | | |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | | |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | | |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | | |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 2 | 1 | 2 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 3 | | |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 3 | 4 | 4 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 2 | 1 | 2 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 3 | | |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 4 | 4 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 2 | 3 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | | |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 | 2 | 1 | 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 | | |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 1 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | | | |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | |
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 2 | 3 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | 4 | 5 | 6 | | |
| 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | 7 | 8 | 9 | |

(a)

Abbildung 3.4: (a) Sudoku Feld mit mehreren festgelegten Zellen. Blau markierte Zellen sind mögliche Kandidaten für nächsten Zyklus.

Dieser Zyklus wiederholt sich so lange, bis alle Zellen ausgefüllt sind.

²Dies ist nicht zwingend der Fall für WFC. Dazu später mehr.

3.2 WFC und die Model Synthese

Wie bereits erwähnt, wurde der WFC Algorithmus von Gumin durch Paul Merrell's diskreter Model Synthese inspiriert. Die Model Synthese wurde durch die Patch basierte Textur-synthese inspiriert und ist, wie WFC, ebenfalls ein Lösungsalgorithmus für CSP's.

Eine Textursynthese kann in vielen Bereichen angewendet werden. Ausgenommen, wenn im Input-Image feste Strukturen vorkommen oder eine Synthese im 3D-Raum generiert werden soll, dann sind nicht alle Synthesemethoden anwendbar. Diese Probleme wollte Paul Merrell mit seiner Model Synthese lösen [9].

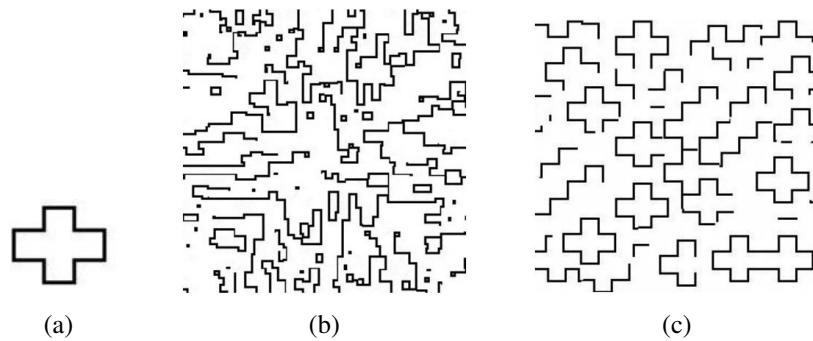


Abbildung 3.5: (a) Input Struktur, (b) Alexei A. Efros and Thomas K. Leung 1999, (c) Kwatra et al., 2005 [9].

Viele dieser gängigen Textursynthesen scheitern an Strukturen, da diese Aufgelöst werden nicht als geschlossenes Objekt erkannt werden. In Abbildung 3.5 sind genannte Schwachstellen von den anderen Textursynthesen klar erkennbar.

Paul Merrell erkannte, dass viele natürliche und künstliche Objekte aus sich immer wiederholenden Komponenten bestehen. Dies führte zu dem Ansatz, mit ganzen Komponenten, statt mit einzelnen Pixel zu arbeiten. Dadurch können neue Objekte sowohl im 2D als auch im 3D-Raum generiert werden (siehe Abbildung 3.6 (c)) [9].

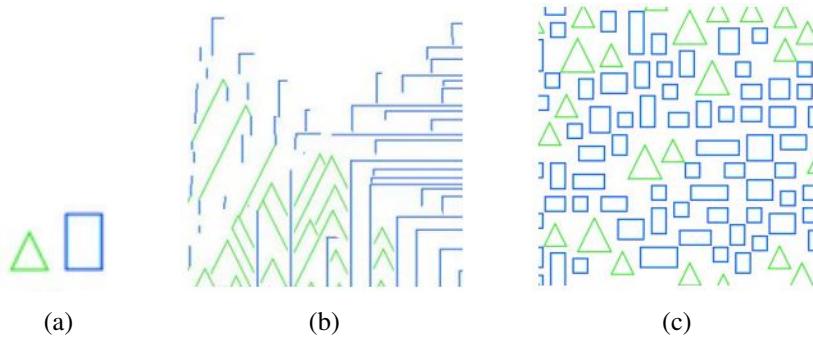


Abbildung 3.6: (a) Input Struktur, (b) Alexei A. Efros and Thomas K. Leung, (c) Paul Merrell Model Synthese [9].

Anders als bei Sudoku, gilt bei der Model Synthese jedoch die Regel der Adjazenz-Bedingung (*adjacency constraint*). Die Adjazenz-Bedingung stellt sicher, dass alle Teile des Modells nahtlos zusammenpassen und dass das neue Modell dem Input ähnlich ist (siehe Abbildung 3.7).

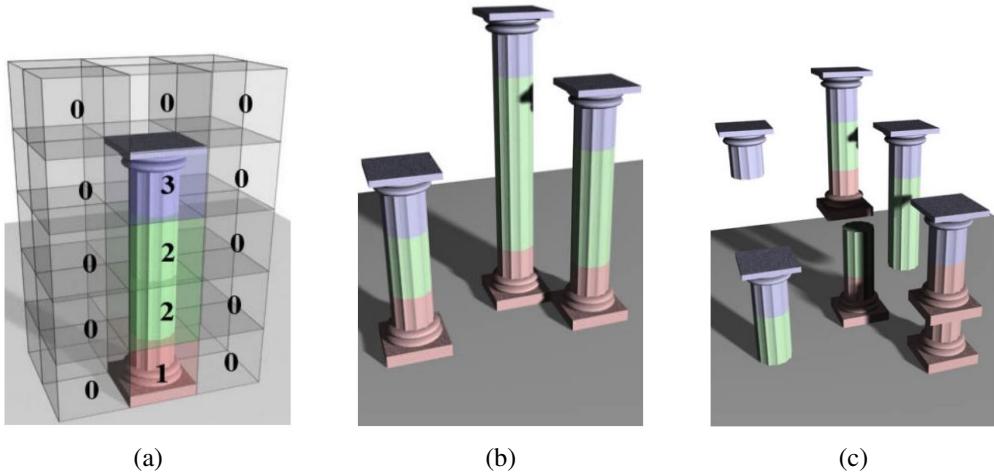


Abbildung 3.7: (a) 3D-Modell in Komponenten aufgeteilt, (b) Komponente nahtlos zusammengesetzt, (c) Komponente halten Adjazenz-Bedingungen nicht ein [9].

Der grundsätzliche Ablauf der Model Synthese im 2D-Raum ist wie folgt:

1. Erstelle Module (siehe Abbildung 3.8 (a)).
2. Wähle eine Zelle unter eventueller Berücksichtigung von Heuristiken.
3. Kollabiere die Zelle auf einen Eigenwert.
4. Errechne neue mögliche Eigenwerte der benachbarten Zellen (*Propagation*) (siehe Abbildung 3.8 (b)).
5. Wiederhole Schritte 2 bis 4.

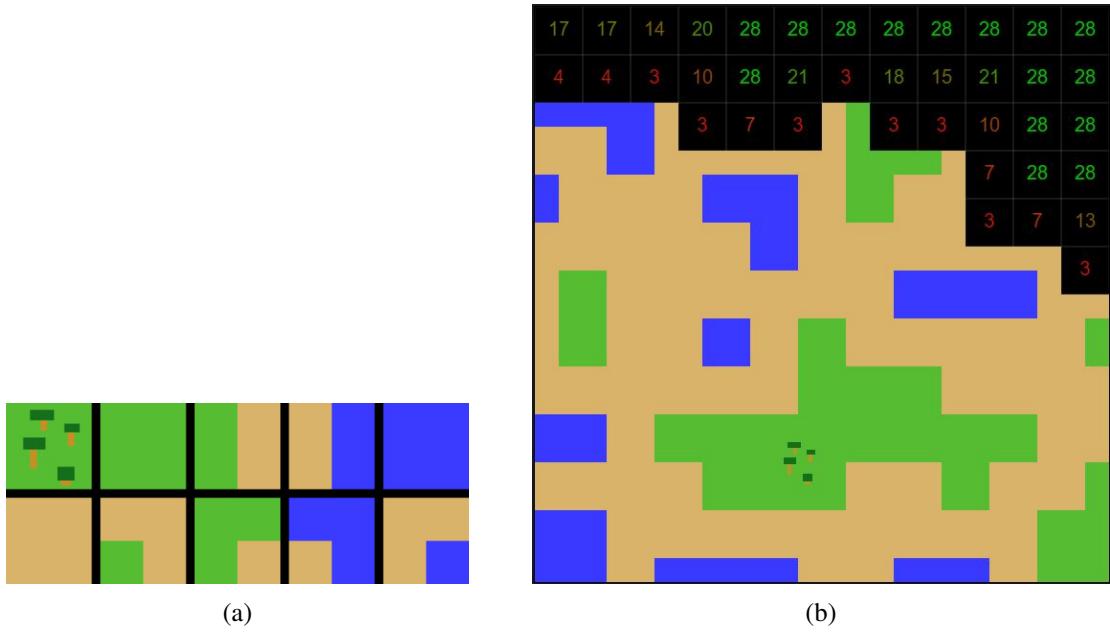


Abbildung 3.8: (a) Module, (b) Möglicher Output mit möglichen Optionen pro Zelle.

An diesem Punkt ist es wichtig zu erwähnen, dass es sich hierbei bereits um das “Simple Tile Model” des WFC Algorithmus handelt.

3.2.1 Unterschiede zwischen WFC und Model Synthese

Tatsächlich ist der WFC Algorithmus von Gumin und die Model Synthese von Merrell, in der Ausführung der oben genannten Schritte, nahezu identisch. Der Unterschied zwischen den beiden Algorithmen besteht lediglich in der Implementierung der einzelnen Schritte, sowie in zusätzlichen Optimierungen. Laut Paul Merrell gibt es mehrere Unterschiede in der Implementierung. Ein Unterschied ist die Reihenfolge der Auswahl der nächsten Zelle. Während WFC nach der niedrigsten Entropie Heuristik seine Zelle auswählt, wählt die Model Synthese nach der Scanline-Methode, in der erst Reihe für Reihe durch das Modell / Bild iteriert wird, seine Zellen aus. Die Zellauswahl von WFC verursacht mehr Fehlerzustände bei großen Output-Images als die von Paul Merrell’s Model Synthese. Auch die Komplexität des Inputs kann dazu beitragen, dass WFC nicht zu einem Ergebnis kommt. Obwohl das ebenfalls der Fall bei Merrell’s Implementierung ist, ist es bei der Model Synthese wesentlich wahrscheinlicher, dass ein gültiges Ergebnis zustande kommt, da die Model Synthese Fehlerbehandlungen besitzt, die bei WFC fehlen.

Einer dieser Fehlerbehandlungen ist die Blockweise Generierung vom Output-Image. In der Model Synthese wird der Output blockweise generiert und nicht komplett in einem Durchgang. Das ist für die Generierung von großen Outputs, vor allem bei 3D-Modellen,

von großer Bedeutung, da dadurch die Fehlerquote bei einigen Inputs signifikant reduziert wird. WFC teilt sein CSP nicht in kleinere Blöcke auf, da der Output von WFC in der Regel kleiner ist, und die Verarbeitung im 2D-Raum wesentlich einfacher. Dennoch kann das WFC von einer solchen Implementierung profitieren, wenn man WFC im 3D-Raum verwenden möchte oder größere Outputs erzielen möchte [14].

| Name | width x height | Model Syn Total (s) | WFC Total (s) | Model Syn Synthesis (s) | WFC Synthesis (s) | WFC Success Rate | WFC Trials |
|------------|----------------|---------------------|---------------|-------------------------|-------------------|------------------|------------|
| Summer | 100 x 100 | 3.741 | 116.3 | 0.969 | 114.5 | 0.2 % | 1,000 |
| Summer | 200 x 200 | 25.144 | – | 13.869 | > 1,354 | 0 % | 1,000 |
| Castle | 100 x 100 | 0.628 | – | 0.557 | > 1,365 | 0 % | 10,000 |
| Castle | 200 x 200 | 4.683 | – | 4.399 | > 1,365 | – | – |
| Knot Dense | 100 x 100 | 0.310 | 16.04 | 0.156 | 15.94 | 2.7% | 1,000 |
| Knot Dense | 200 x 200 | 1.758 | – | 1.173 | > 1,877 | 0 % | 1,000 |
| Knot TE | 100 x 100 | 0.210 | – | 0.098 | > 882 | 0 % | 10,000 |
| Knot TE | 200 x 200 | 1.086 | – | 0.687 | > 882 | – | – |
| Knot T | 100 x 100 | 0.180 | 2.60 | 0.071 | 2.56 | 10.4 % | 1,000 |
| Knot T | 200 x 200 | 0.909 | – | 0.532 | > 1,253 | 0 % | 1,000 |
| Knot CE | 100 x 100 | 0.868 | 9.80 | 0.772 | 9.76 | 1.8 % | 1,000 |
| Knot CE | 200 x 200 | 1.055 | – | 0.673 | > 787 | 0 % | 1,000 |
| Rooms | 100 x 100 | 0.557 | 1.30 | 0.544 | 1.29 | 45.9 % | 1,000 |
| Rooms | 200 x 200 | 4.136 | 200.8 | 4.049 | 200.7 | 1.8 % | 1,000 |
| Red Dot | 200 x 200 | 8.205 | – | 8.194 | > 7,061 | 0 % | 10,000 |
| Shew1 | 200 x 200 | 15.603 | – | 15.593 | > 1,757 | 0 % | 1,000 |
| Cat | 400 x 400 | 279.9 | – | 279.8 | > 14,802 | 0 % | 1,000 |

Tabelle 3.1: Paul Merrell's Vergleich zwischen WFC und seiner Model Synthese [14].

In Tabelle 3.1 wird die Erfolgsquote sowie die benötigte Zeit zur Generierung der genannten Inputs verglichen. Die Resultate können in Abbildungen 5.1 bis 5.4 betrachtet werden. Der WFC kommt aufgrund der Größe des Outputs als auch der Komplexität des Inputs nicht immer auf gültige Ergebnisse. Es wird deutlich, dass die Model Synthese fast immer zu einem erfolgreichen Resultat kommt, da die Größe der einzelnen Blöcke einfach reduziert werden kann, um die Erfolgsquote zu steigern [14].

Neben weiteren Unterschieden, wie der Laufzeit und der Erweiterbarkeit der beiden Algorithmen, gibt es noch einen weiteren Unterschied den Gumin's WFC implementiert. Das sogenannte Overlapping Model, das in der Originalversion von Gumin vorkommt, ist das, was WFC initial bekannt gemacht hat.

3.2.2 Simple Tile Model und Overlapping Model

Anders als bei dem Simple Tile Model von WFC, bei dem einzelne Module für die Synthese bereitgestellt werden, benötigt das Overlapping Model von WFC ein komplettes Input-Image, das daraufhin analysiert wird. Aus dieser Analyse werden die Module automatisch generiert. Zudem haben sich die Module bei dem Simple Tile Model nicht überschnitten, wie es bei dem Overlapping Model der Fall ist. Das hat den Vorteil, dass die Module direkt aus dem Input errechnet werden können (siehe Abbildung 3.9 und 3.10). Zudem kann der Output dem Input ähnlicher sein, da die Module näher aneinander liegen [14].

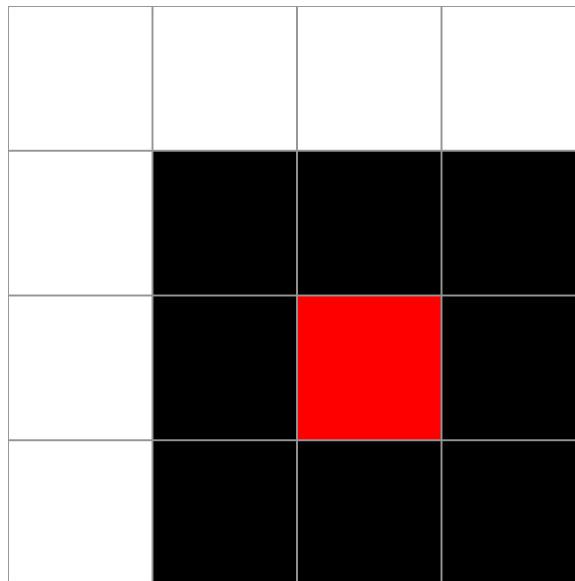


Abbildung 3.9: 4×4 Pixel Red-Maze Beispiel Input [8].

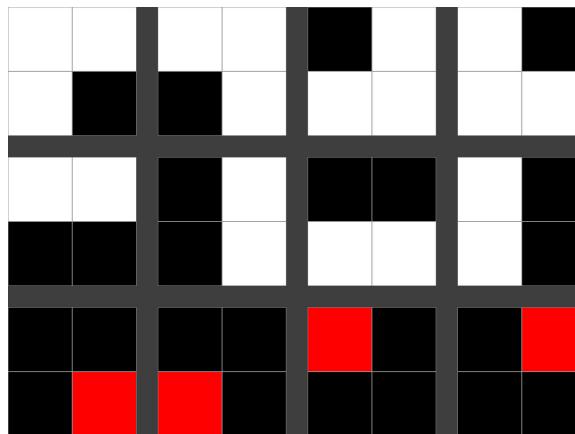


Abbildung 3.10: Alle Red-Maze Module der Größe $N = 2$ mit Spiegelungen und Rotationen [8].

Bei einem $N = 2$ Patch gibt es insgesamt 9 verschiedene Möglichkeiten, wie diese übereinander liegen können (siehe Abbildung 3.11). (Wenn $N = 3$ dann $(2(N - 1) + 1)^2 =$

36 offsets.) Der WFC Algorithmus legt eine Indexstruktur an, die die Möglichkeiten beschreibt, wie die Muster nebeneinander platziert werden können. Bei diesem Modell enthält der Index die vorberechnete Anzahl an gültigen Modulen, die an einem anderen Index mit x,y -Offset platziert werden dürfen. Bei dem in Kapitel 3.1.2 beschriebenen Sudoku-Beispiel, könnte dies angewendet werden, indem jede Zelle im Feld eine bestimmte x,y Koordinate besitzt, damit durch diese iteriert werden kann. Sobald ein Feld einen Wert besitzt, werden in allen benachbarten Zellen Module entfernt, die nicht nutzbar sind [8].

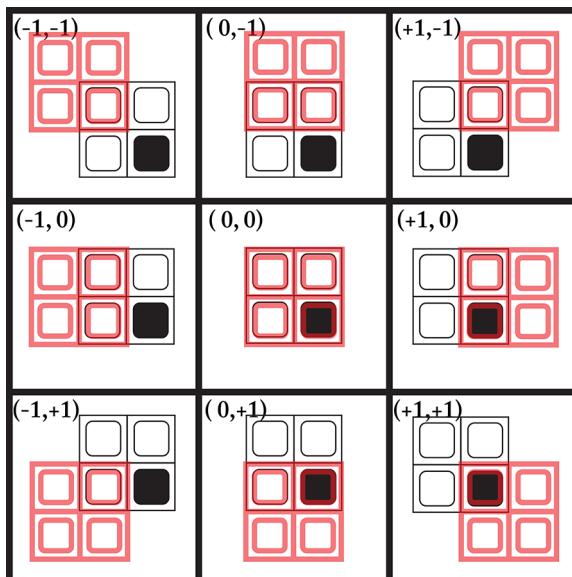


Abbildung 3.11: Die 9 Möglichkeiten wie die Module übereinander liegen können [8].

In Abbildung 3.12 erkennen wir einen Ausschnitt einer einzelnen Zelle mit allen möglichen Modulen, die an ihren benachbarten x,y -Offsets erlaubt sind, dargestellt. Dabei ist bei einem Offset mit den Werten $(0,0)$ (kein Offset) immer nur der betrachtete Patch nutzbar (bzw. die Zelle mit bereits einzelnen Eigenwert). Als Beispiel Offset $(-1,0)$. Da sind nur Module möglich, die Rechts zwei weiße Pixel besitzen damit sie sich nahtlos mit den zwei weißen Pixel bei $(0,0)$ überlappen können [8].

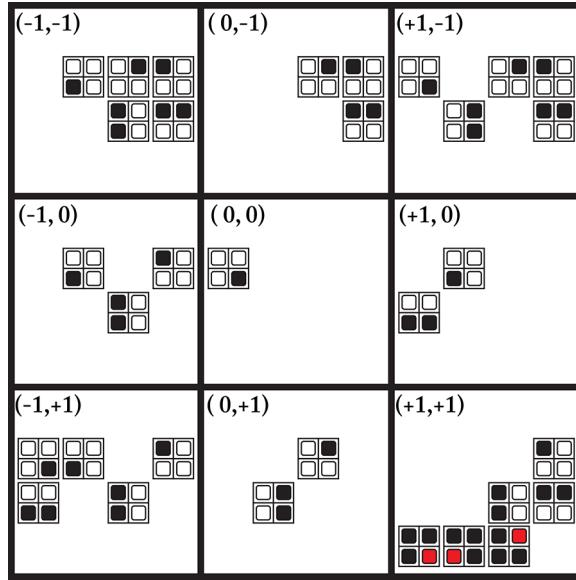


Abbildung 3.12: Ausschnitt einer Zelle. Von 12 möglichen Modulen sind nur die gültigen Optionen für die benachbarten Zellen mit x, y -Offset möglich [8].

In diesem Beispiel wurden mit $N = 2$, dies Entspricht in diesem Fall einer Fenstergröße von zwei Pixeln, die einzelnen Module aus dem Input generiert. Die Fenstergröße wird von dem Entwickler festgelegt und hat direkten Einfluss auf den Output. Falls $N = 1$ gesetzt wird, und somit nur einzelne Pixel verarbeitet werden, dann befinden wir uns wieder beim Simple Tile Model. Allerdings funktioniert $N = 1$ in diesem Fall nicht, da jeder Pixel zu einem Patch wird, da diese nicht mehr überlappt werden können, es sei denn $N = 1$ repräsentiert eine Menge an Pixeln. Bei einer zu kleinen Fenstergröße kann es vorkommen, dass der Output nicht dem Input ähnelt. Bei einer zu großen Fenstergröße kann der Output dem Input zu ähnlich sein, da ganze Gruppen von Strukturen kopiert und eingefügt werden. Daher hat die Wahl der Größe, Einfluss auf das Ergebnis (siehe Abbildung 3.13 und 3.14).

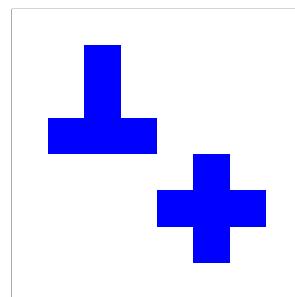


Abbildung 3.13: Beispiel Input-Image (8×8 Pixel) für ein Overlapping Model.

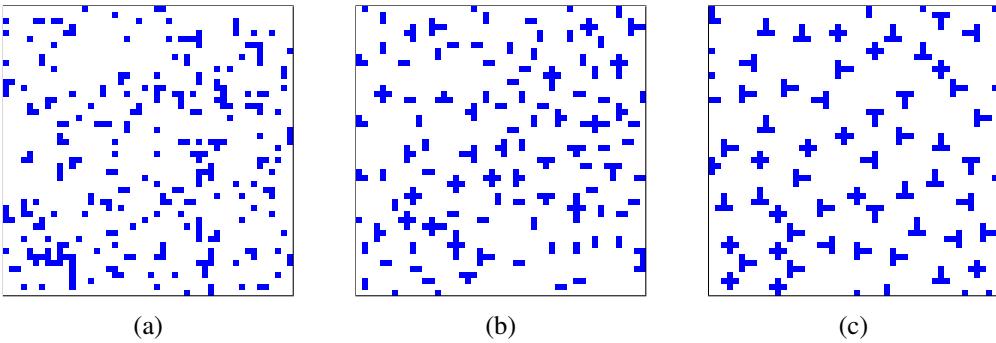


Abbildung 3.14: (a) Output mit $N = 2$, (b) Output mit $N = 3$, (c) Output mit $N = 4$. Beispiele generiert mit Kchapelier's Overlapping Model.³

Es ist wichtig anzumerken, dass viele der hier gezeigten Texturen auch mit aktuellen Textursynthese-Algorithmen generiert werden können. Der Vorteil von Model Synthese und WFC im Vergleich zu anderen Synthesen ist, dass der generierte Content auch Interaktiv genutzt werden kann, da wir bei diesen Verfahren der Entwickler die gesamte Kontrolle über die lokalen Muster des Outputs hat und das Endergebnis alle auferlegten Constraints erfüllt. Gerade der Simple Tile Model Algorithmus bietet sich deswegen für PCG an, da der Entwickler die Kontrolle über die einzelnen Module hat, die dann für das Output zusammengesetzt werden. Dadurch, und das keine unvorhersehbaren Artefakte entstehen können, qualifiziert den WFC für den Einsatz in der Spiele-Entwicklung. Wird statt einem lokalen und stationären Input (siehe Abbildung 2.2) bspw. ein Foto verwendet, dann würden die anderen Synthesemethoden, im Gegensatz zur Model Synthese und WFC, passendere Ergebnisse liefern, sofern diese mit der Verwendung von KI dafür ausgelegt sind. Es sei erwähnt, dass der Input für WFC nicht “strikt” eine Textur wie in Abbildung 2.2 (b) sein muss. Diese Texturen sollten allerdings Selbstähnlichkeit besitzen und nur wenige unterschiedliche Pixelfarben verwenden. Bereits ein Tag nach der Veröffentlichung von Gumin’s WFC am 30. September 2016 haben viele Entwickler begonnen mit diesem Algorithmus zu experimentieren [8]. Ein Grund für die große Beliebtheit von Gumin’s WFC Algorithmus ist neben der bereits erwähnten Pixelgenauigkeit und die damit einhergehende Möglichkeit den Output Interaktiv zu nutzen, auch die Echtzeit Generierung des Outputs. Viele PCG’s Methoden variieren bei der Generierung ihres Outputs in ihrer Laufzeit. Dies führt dazu, dass große Teile des Outputs sofort generiert werden können, dafür aber der Abschluss aufgrund von komplexen constraint solving Algorithmen nicht gleichmäßig entsteht [8]. Da WFC nach der niedrigsten Entropie Heuristik seine nächste Zelle auswählt und im Overlapping Model alle benachbarten Zellen, die nicht kollabiert sind, als gemischte Pixelwerte aller möglichen Optionen innerhalb der Zellen darstellt, führt dies zu einer ästhetisch wahrgenommenen Generierung des Outputs. Abbildung 3.15 zeigt die erste kollabierte Zelle im

Overlapping Model mit dem Red-Maze als Input und Abbildung 3.16 weitere Zustände des Zyklus nach der niedrigst Entropie Heuristik.⁴

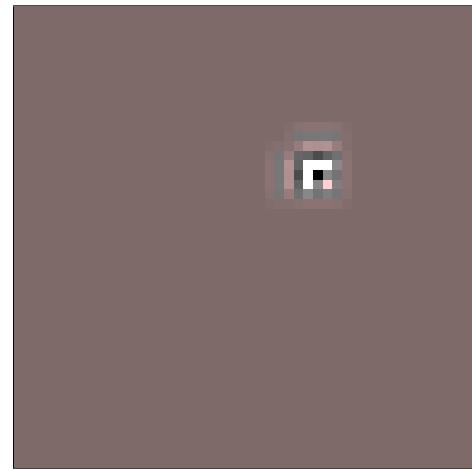


Abbildung 3.15: Resultat des ersten Zyklus mit dem Red-Maze Input. Da am Anfang die Entropie überall gleich ist, wird die Anfangszelle zufällig gewählt. Wenn der Algorithmus bereits die möglichen Module der benachbarten Zellen ausgewertet hat, dann werden die Zellen mit allen Farben der verwendbaren Modulen eingefärbt (die Pixelfarben werden dann gemischt). Der direkt umliegende Bereich der Anfangszelle nicht in einer einheitlichen Farbe dargestellt, da es dort bereits weniger Optionen gibt die verwendet werden können [8].

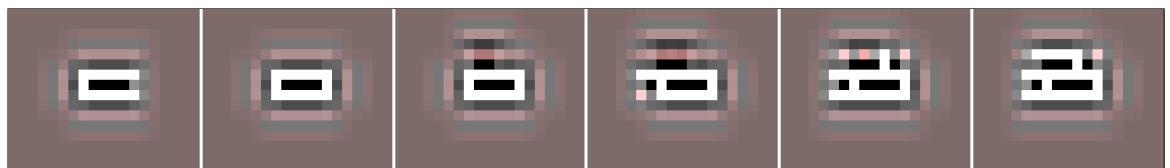


Abbildung 3.16: Die nächsten Zyklen für das Red-Maze Input [8].

⁴Maxim Gumin's erster WFC tweet "Procedural generation from a single example by wave function collapse <https://github.com/mxgmn/WaveFunctionCollapse>"
<https://twitter.com/ExUtumno/status/781833475884277760>
und Danny Wynne's "3d tile placement with WFC. This algorithm is amazing. Inspired by @OskSta and based on @ExUtumno work #screenshotsaturday #gamedev #indiedev"
<https://twitter.com/dwtw/status/810166761270243328> [8]

3.3 Beispiel Implementierung

In diesem Kapitel wird eine einfache Implementierung des WFC Algorithmus im Simple Tile Model, stellenweise als Pseudocode als auch als Programmcode, dargestellt. Diese Implementierung wird dann um zwei Optimierungen erweitert. Auf eine dieser Optimierungen wird mit Programmcode genauer erläutert, während die andere nicht implementiert, aber auf dessen Vorteile eingegangen wird.

```
1  init() {
2      ladeModule();
3      erstelleOutputZellen();
4      fuelleOutputZellen();
5      while(!alleZellenKollabiert) {
6          sortiereZellenNachEntropie();
7          kollabiereZelleMitNiedrigsterEntropy();
8          for(jedeZelleImOutput) {
9              berechneNachbarOptionen();
10         }
11         zeichneKollabierteZellen();
12     }
13 }
```

Zuerst werden die selbst erstellten Module geladen und in jeder einzelnen Zelle des Outputs werden alle Module hinzugefügt. Dadurch erstellen wir ein CSP in der jede Zelle eine *Superposition* besitzt. In den Zeilen 5 bis 11 wird der Hauptteil von WFC ausgeführt. Dort werden erst alle Zellen nach ihrer Entropie sortiert, sodass dann zufällig aus einer der niedrigsten Zellen ausgewählt werden kann. Diese Zelle wird dann auf einen ihrer möglichen Eigenwerte kollabiert. Damit haben wir den initialen *Observation* Schritt. Anschließend beginnt die *Propagation*. Hier wird durch jede nicht kollabierte Zelle Iteriert und die neuen möglichen Eigenwerte nach den bestimmten *Constraints* errechnet. Diese Zellen werden mit diesen neuen Eigenwerten aktualisiert.

Um die Optimierung, die gleich vorgestellt wird, zu verdeutlichen wird auf die Funktion `ladeModule()` genauer eingegangen.

```
1  ladeModule() {
2      modul[0]=new Tile(0,new Image(),[0,1,4],[0,1,2],[0,2,3],[0,3,4]);
3      modul[1]=new Tile(1,new Image(),[2,3],[3,4],[0,2,3],[0,3,4]);
4      modul[2]=new Tile(2,new Image(),[0,1,4],[3,4],[1,4],[0,3,4]);
5      modul[3]=new Tile(3,new Image(),[0,1,4],[0,1,2],[1,4],[1,2]);
6      modul[4]=new Tile(4,new Image(),[2,3],[0,1,2],[0,2,3],[1,2]);
7  }
```

Die `ladeModule()` Funktion dient als Mapping der einzelnen Module. Hier werden jedem Modul zusätzlich noch alle anderen Module, die an dieses Modul angrenzen können, festgelegt. Die Reihenfolge der Optionen beginnt von Oben und rotiert im Uhrzeigersinn herum, damit alle vier Richtungen abgedeckt sind (siehe Abbildung 3.17). Das Problem an dieser Implementierung ist das mit Zunahme der Anzahl der Module auch die Komplexität und die Größe des Mappings zunimmt [15].

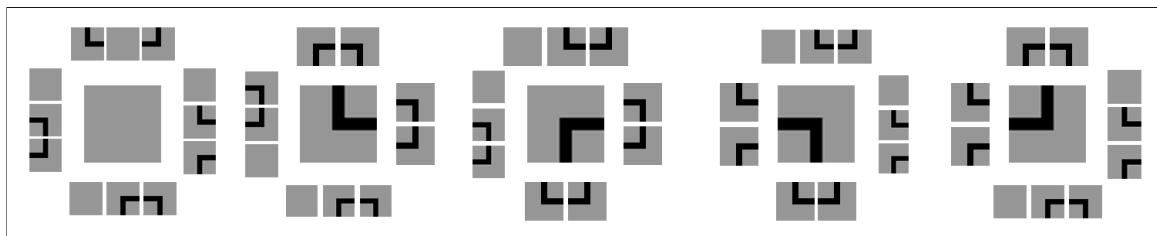


Abbildung 3.17: Visualisierung eines Mappings der Module.

Um das Mapping zu vereinfachen können sogenannte “Sockets” verwendet werden. Anstelle selbst zu entscheiden, welche Module an welche Module angeschlossen werden dürfen, werden den Modulen an jeder Kante Sockets zugewiesen. Dadurch können die Module selbst entscheiden welche Module an diese gesetzt werden dürfen, indem die Sockets miteinander verglichen werden und nur Module mit identischen Sockets eine gültige Verbindung sind [15].

```
1  ladeModule() {
2      modul[0]= new Tile(0,'/maze/0.png','AAA','AAA','AAA','AAA');
3      modul[1]= new Tile(1,'/maze/1.png','ABA','ABA','AAA','AAA');
4      modul[2]= new Tile(2,'/maze/2.png','AAA','ABA','ABA','AAA');
5      modul[3]= new Tile(3,'/maze/3.png','AAA','AAA','ABA','ABA');
6      modul[4]= new Tile(4,'/maze/4.png','ABA','AAA','AAA','ABA');
7  }
```

Die Sockets werden hier ebenfalls beginnend von Oben und rotierend im Uhrzeigersinn zugewiesen. Es muss darauf geachtet werden das die Sockets konsistent für alle Module nach demselben Prinzip zugewiesen werden. In diesem Fall werden sie strikt um Uhrzeigersinn ungeachtet der natürlichen Schreibrichtung zugewiesen (siehe Abbildung 3.18). Durch die Verwendung von Sockets lässt sich die Komplexität und somit die Erstellung des Mappings stark reduzieren [15].

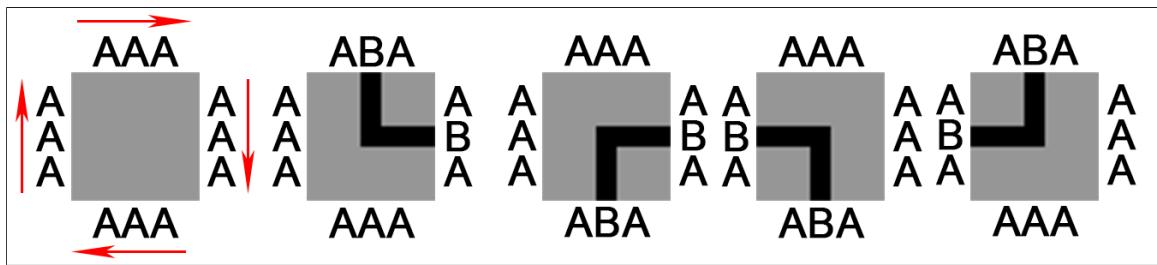


Abbildung 3.18: Visualisierung eines Mappings mit Sockets.

Wenn Sockets verwendet werden, dann müssen auch dazu passende Module verwendet werden. Werden einfarbige Module verwendet, wie z.B. Module für Wald, Wiese, Strand und Meer, dann können die Sockets das Mapping verkomplizieren anstatt zu vereinfachen. In diesen Fällen benötigen die Module Verbindungsmodule und Eckmodule, um alle nötigen Kombinationen zu ermöglichen (siehe Abbildung 3.8 (a)). Dadurch verändert sich der Output dahingehend, das “halbe” Terrains zustande kommen.

Eine weitere Optimierung vorgeschlagen von Boris ist, dass nur die Zellen auf Constraints geprüft werden, die durch den Propagation Vorgang zu anderen Ergebnissen führen können. Das bedeutet, wenn eine Zelle aktualisiert wird, dann fügen wir sie einer Warteschlange ausstehender Zellen hinzu. Dann werden die ausstehenden Zellen einer nach der anderen aus der Warteschlange entfernt und auf ihren Nachbarzellen geprüft. Sollten die Nachbarzellen dazu führen, dass weitere Zellen aktualisiert werden müssen, dann werden diese ebenfalls zu Warteschlange hinzugefügt. Dieser Vorgang wird so lange wiederholt bis die Warteschlange leer ist. Dadurch haben wir sichergestellt, dass jeder Constraint vollständig ausgewertet wurde, aber wir überprüfen nie ein Constraint, es sei denn, eine der mit ihr verbundenen Zellen hat ihren Eigenwert geändert. Diese Optimierung erhöht die Geschwindigkeit von WFC maßgeblich [13].

3.3.1 Weitere Implementierungen von WFC

Joseph Parker war einer der ersten Entwickler, der WFC verwendet hat. In seinem in der Unity Engine entwickeltem Spiel *Proc Skater 2016* verwendet er den Algorithmus, um einzigartige 3D Skateparks zu generieren. In Abbildung 3.19 dargestellt seine selbst erstellten Blöcke, aus denen die Karte generiert werden soll (Abbildung 3.20), anstatt sie automatisch aus einem Input zu analysieren [16]. J. Parker zufolge benutzt er das Simple Tile Model, als auch das Overlapping Model für den Skatepark.⁵

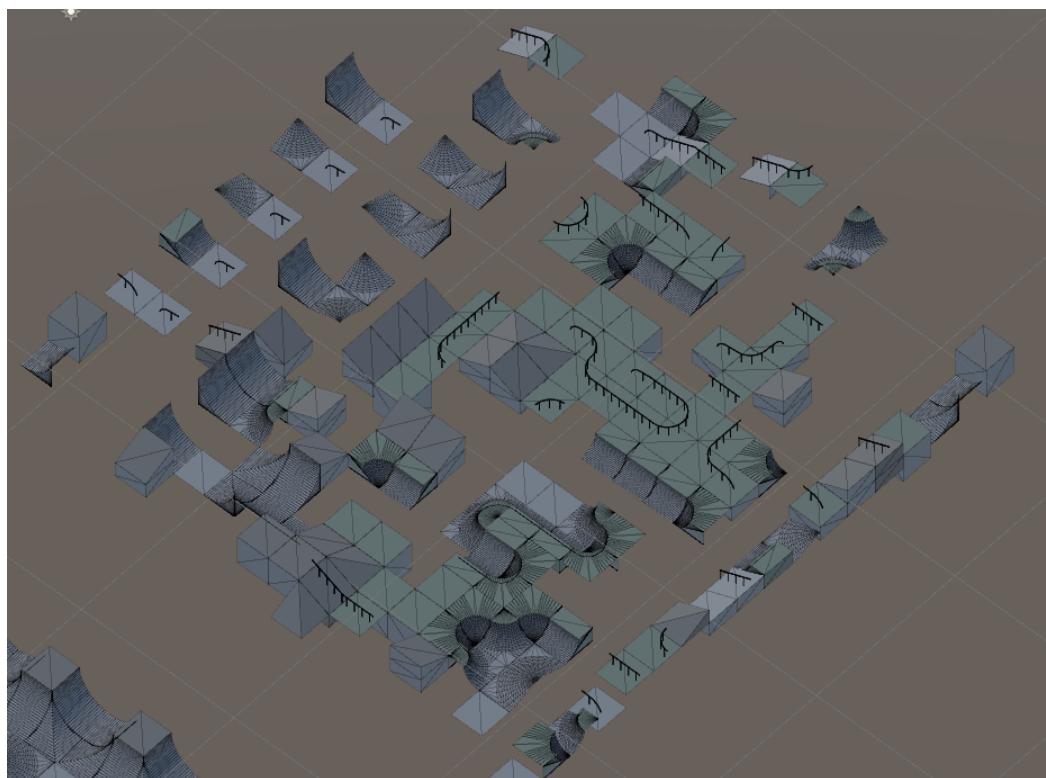


Abbildung 3.19: Blöcke zu Generierung des Skateparks in *Proc Skater 2016* [16].

Die adjacency constraints für diese Blöcke werden aus einer ruleset .xml ausgelesen und angewendet.

⁵Persönliche Kommunikation, 23. Mai 2023

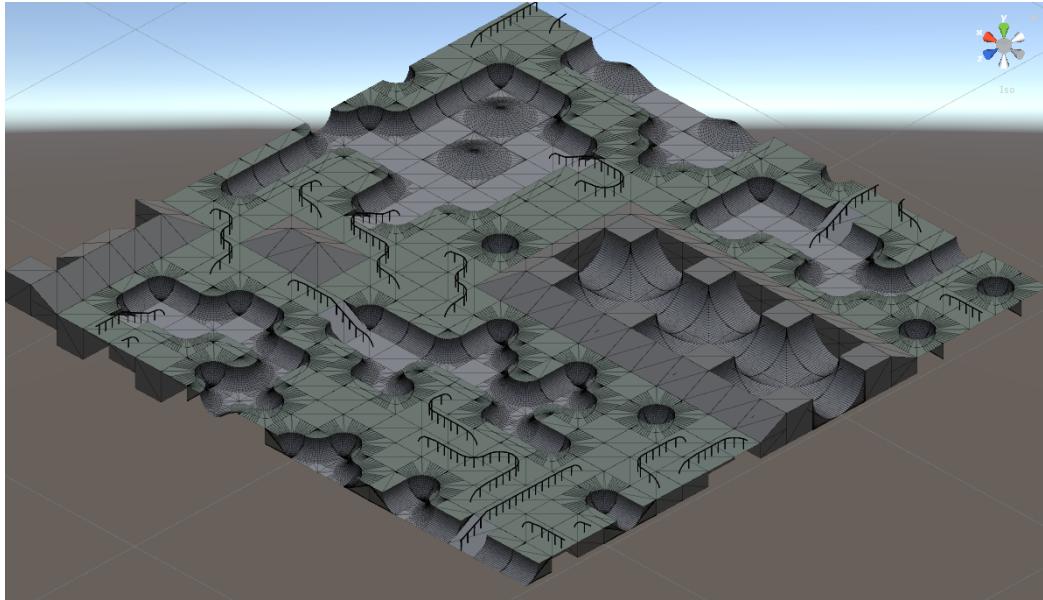


Abbildung 3.20: Typisches Level in *Proc Skater 2016* [16].

Das Overlapping Model verwendete er um den umliegenden Hintergrund (*backdrop*) für seinen Skatepark in Abbildung 3.21 zu erstellen. Weiter erklärte Parker, dass das Tile Model und das Overlapping Model sich jeweils für unterschiedliche Bereiche gut anwenden lassen. “Das Tile Model ist besser für Straßen, Labyrinthe oder alles mit formalen Übergängen. Overlapping Model für Plattform artige Situationen, in denen man eher eine Art Markow-Kette für die Größe der Features haben möchte.”⁵

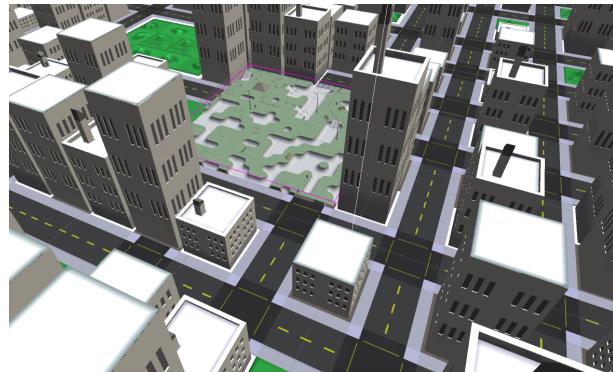


Abbildung 3.21: Overlapping Model für das Backdrop in *Proc Skater 2016* [16].

Eine weitere kommerzielle Implementierung von WFC ist *Caves of Qud*, entwickelt von Freehold Games. Laut Brian Bucklew, einem der Entwickler bei Freehold Games, verwendet *Caves of Qud* einen Mehrfachdurchlauf (*multipass*) von WFC. Dadurch können größere Komplexitäten bei der Generierung von Karten erreicht werden wie in Abbildung 3.22. Ein weiterer Vorteil, der sich durch WFC ergeben hat, ist die einfache Handhabung. Sobald das zugrunde liegende System vorhanden ist, kann jeder Entwickler Input-Images einspielen und spielbaren Content generieren.⁶ [8]

⁵Persönliche Kommunikation, 23. Mai 2023

⁶Aus Forums-Korrespondenz <https://forums.somethingawful.com/showthread.php?threadid=3563643&userid=68893&perpage=40&pagenumber=23#post467126402>

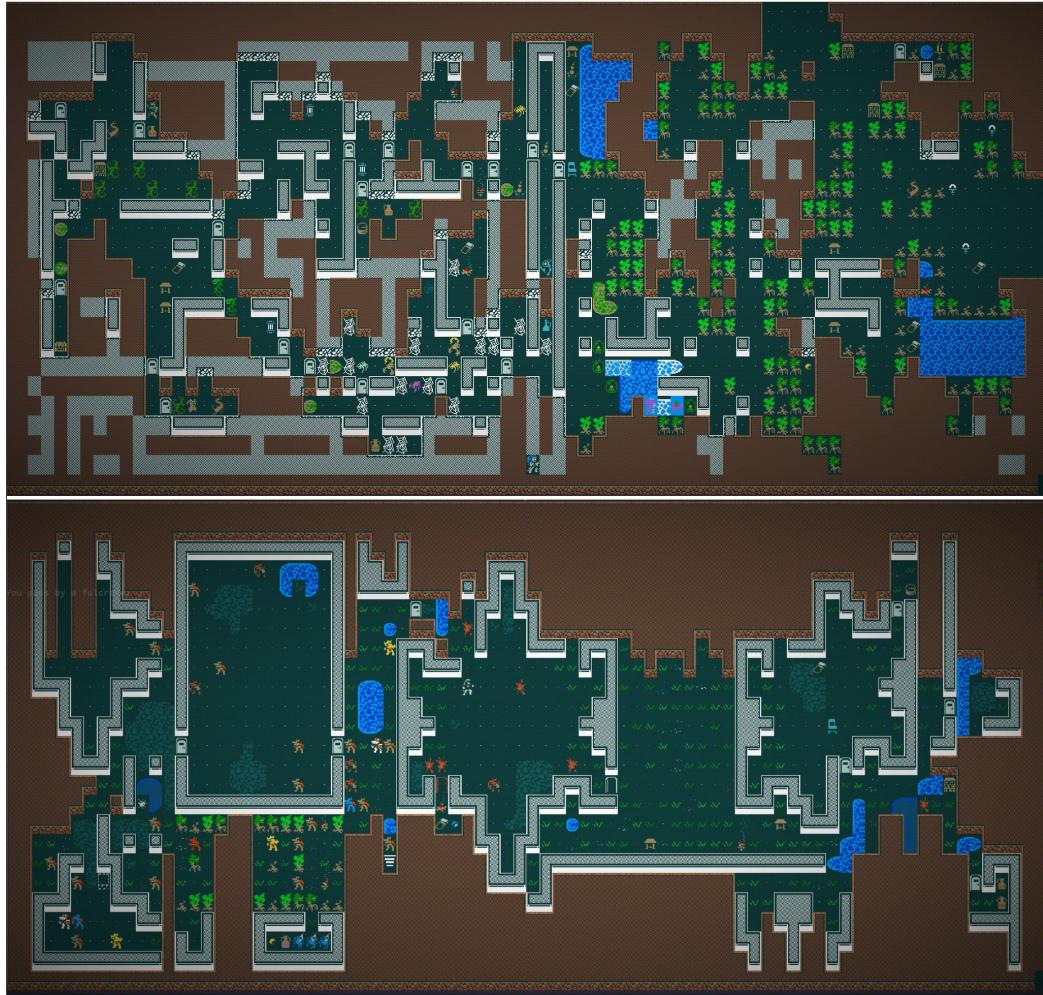


Abbildung 3.22: Zwei Spielbare Karten von *Caves of Qud* [8].

Wave-Function-Collapse kann nicht nur auf flachen Ebenen mit quadratischen Zellen verwendet werden [13]. Oskar Stålberg verwendete WFC nicht nur im 2D und 3D-Raum, sondern auch auf irregulären Rastern. In einem Beispiel implementiert Stålberg WFC auf einer Sphäre mit einem Dreiecks-Raster.⁷

⁷Beispiele auf seiner Website <https://oskarstalberg.tumblr.com/> oder seinem Twitter <https://twitter.com/OskSta/status/784847588893814785>

4. Fazit

Der Wave-Function-Collapse Algorithmus von Gumin sowie die Model Synthese von Paul Merrell haben mehr Gemeinsamkeiten als Unterschiede. Das liegt daran, dass beide Algorithmen denselben grundlegenden Ablauf für das Lösen von Constraint-Satisfaction-Problem verwenden. Beide Algorithmen kann man als jeweils eine Seite derselben Münze bezeichnen. Wave-Function-Collapse ist essenziell ein Verfahren um einzelne Elemente, ob Blöcke, Texte, Bilder etc. nach bestimmten Regeln aneinander zu setzen. Dadurch lassen sich mit beiden Algorithmen fehlerfreie Systeme, die gegebenenfalls Interaktiv verwendet werden können, automatisch generieren. Die Unterschiede zwischen den beiden Algorithmen sind hauptsächlich in der Implementierung der einzelnen Ausführungsschritte sowie in zusätzlichen Optimierungen zu finden. Der WFC von Gumin verwendet das Overlapping Model, um die Generierung der Bausteine / Module zu vereinfachen. Die dadurch dichtere Platzierung der einzelnen Zellen führt zu einem einheitlicherem Output, kann aber zur Folge haben das Artefakte entstehen, sollte die Fenstergröße für die Analyse nicht korrekt gewählt worden sein. Die Model Synthese von Paul Merrell hingegen besitzt Fehlerbehandlungen wie die blockweise Generierung bei der Ausführung des Algorithmus, um größere Outputs zu erzielen und die Fehlerrate zu reduzieren. Beide Implementierungen können um weitere Funktionalitäten erweitert werden, um dieselben oder mehr Funktionalitäten zu erhalten.

Literatur

- [1] D. Gomathi und R. Shah, *Texture Synthesis*, Dez. 2009.
- [2] A. A. Efros und T. K. Leung, „Texture Synthesis by Non-parametric Sampling“, in *IEEE International Conference on Computer Vision*, Corfu, Greece, Sep. 1999, S. 1033–1038.
- [3] Wikipedia, *Markow-Kette — Wikipedia, The Free Encyclopedia*, <http://de.wikipedia.org/w/index.php?title=Markow-Kette&oldid=233283844>, [Online; accessed 26-May-2023], 2023.
- [4] Wikipedia, *Markov model — Wikipedia, The Free Encyclopedia*, <http://en.wikipedia.org/w/index.php?title=Markov%20model&oldid=1146951281>, [Online; accessed 26-May-2023], 2023.
- [5] D. J. Heeger und J. R. Bergen, „Pyramid-Based Texture Analysis/Synthesis“, in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, Ser. SIGGRAPH '95, New York, NY, USA: Association for Computing Machinery, 1995, S. 229–238, ISBN: 0897917014. DOI: [10.1145/218380.218446](https://doi.org/10.1145/218380.218446). Adresse: <https://doi.org/10.1145/218380.218446>.
- [6] T. Lehmann, W. Oberschelp, E. Pelikan und R. Repges, *Bildverarbeitung für die Medizin - Grundlagen, Modelle, Methoden, Anwendungen*. Berlin Heidelberg New York: Springer-Verlag, 2013, ISBN: 978-3-642-60487-4.
- [7] A. A. Efros und W. T. Freeman, „Image Quilting for Texture Synthesis and Transfer“, in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, Ser. SIGGRAPH '01, New York, NY, USA: Association for Computing Machinery, 2001, S. 341–346, ISBN: 158113374X. DOI: [10.1145/383259.383296](https://doi.org/10.1145/383259.383296). Adresse: <https://doi.org/10.1145/383259.383296>.
- [8] I. Karth und A. M. Smith, „WaveFunctionCollapse is constraint solving in the wild“, *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017.
- [9] P. C. Merrell, „Model synthesis“, Diss., The University of North Carolina at Chapel Hill, 2009.

- [10] M. Gumin, *Wave Function Collapse Algorithm*, Version 1.0, Sep. 2016. Adresse: <https://github.com/mxgmn/WaveFunctionCollapse>.
- [11] H. Zinkernagel, „Niels Bohr on the wave function and the classical/quantum divide“, *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, Jg. 53, S. 9–19, Feb. 2016. DOI: [10.1016/j.shpsb.2015.11.001](https://doi.org/10.1016/j.shpsb.2015.11.001). Adresse: <https://doi.org/10.1016%2Fj.shpsb.2015.11.001>.
- [12] C. Lecoutre, „Constraint Networks: Techniques and Algorithms“, 2009.
- [13] B. the Brave, *Wave Function Collapse Explained*, Accessed: 2023-04-13. Adresse: <https://www.boristhebrave.com/2020/04/13/wave-function-collapse-explained/>.
- [14] P. C. Merrell, „Comparing Model Synthesis and Wave Function Collapse“, Diss., Juli 2018.
- [15] D. Shiffman, *Wave Function Collapse*, Accessed: 2023-03-30. Adresse: <https://thecodingtrain.com/challenges/171-wave-function-collapse>.
- [16] O. M. Joseph Parker Ryan Jones, *Proc Skater 2016*, <https://selfsame.itch.io/unitywfc>, Accessed: 2023-05-26. Adresse: <https://arcadia-clojure.itch.io/proc-skater-2016>.

5. Anhang

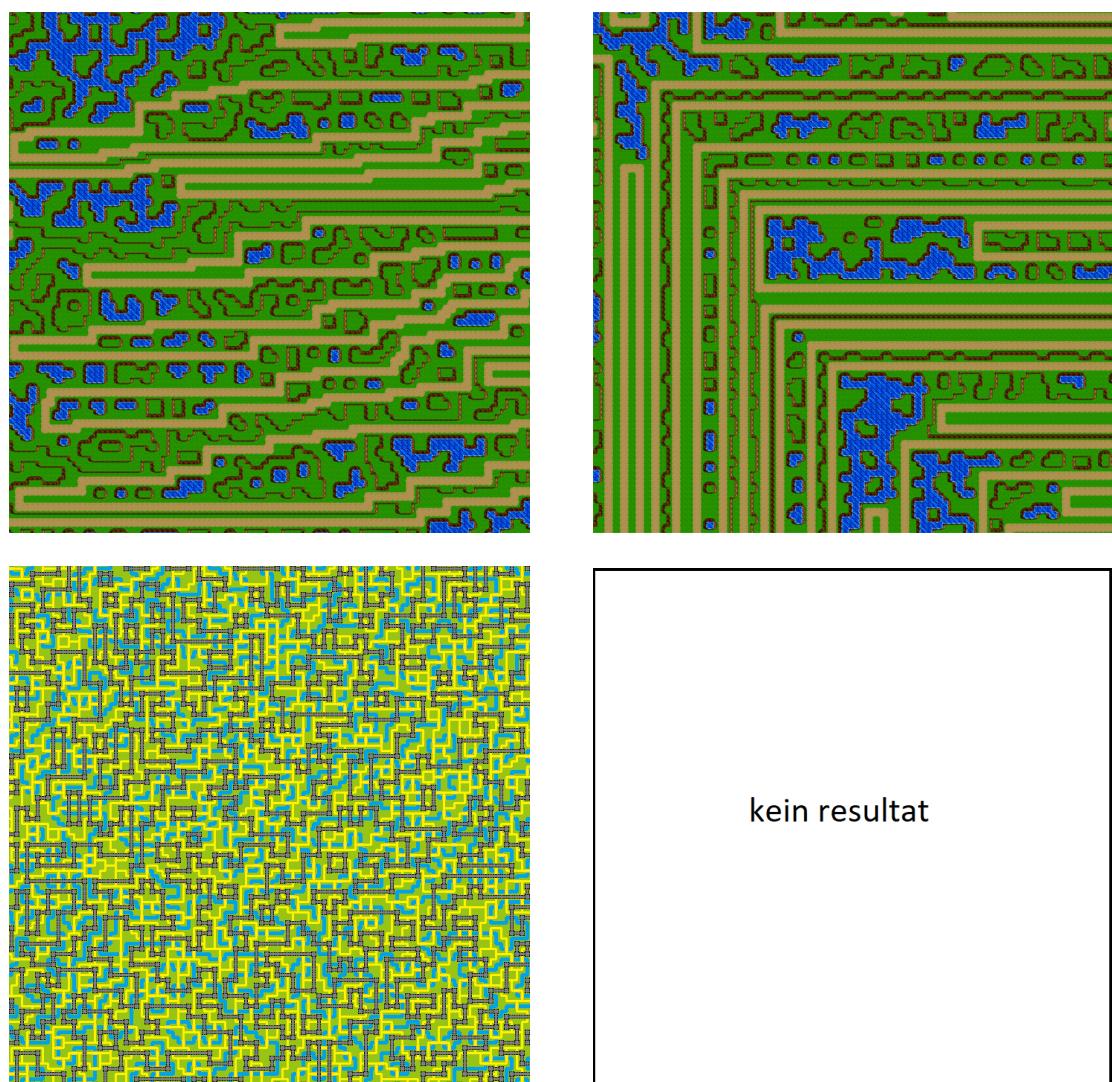


Abbildung 5.1: Texturen Links mit Model Synthese generiert. Texturen Rechts mit WFC. [14, S.10].

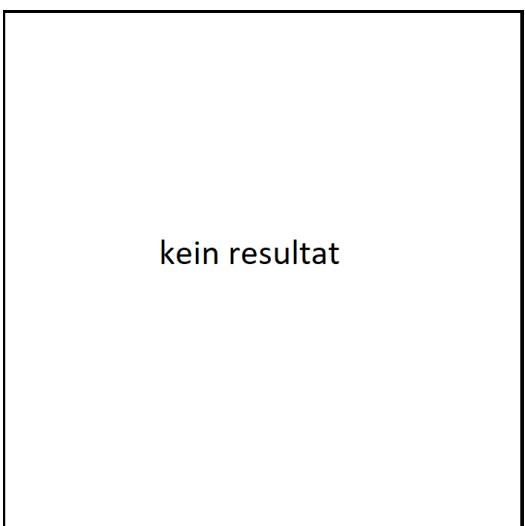
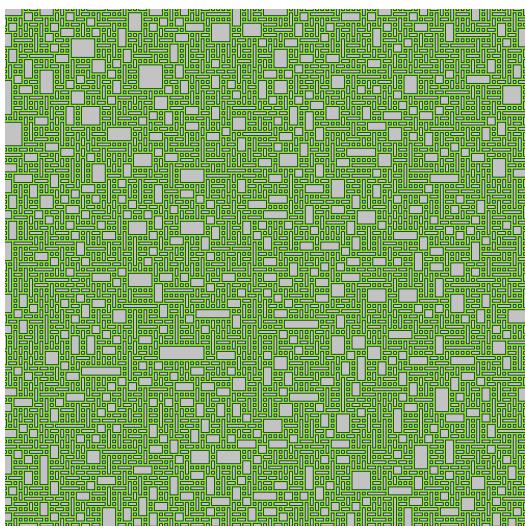
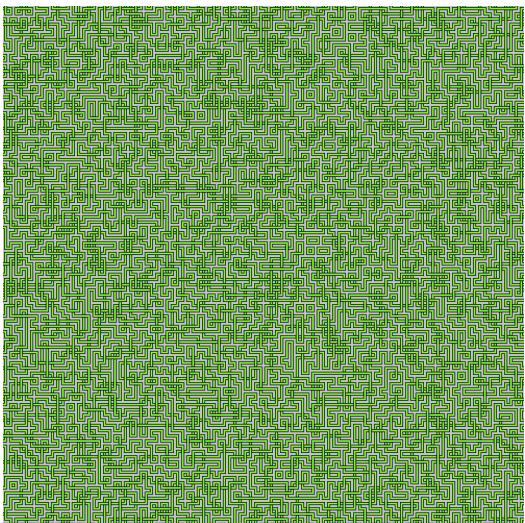
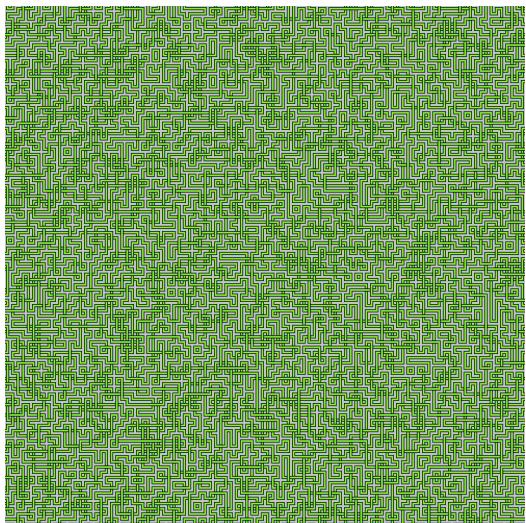


Abbildung 5.2: Texturen Links mit Model Synthese generiert. Texturen Rechts mit WFC. [14, S.11].

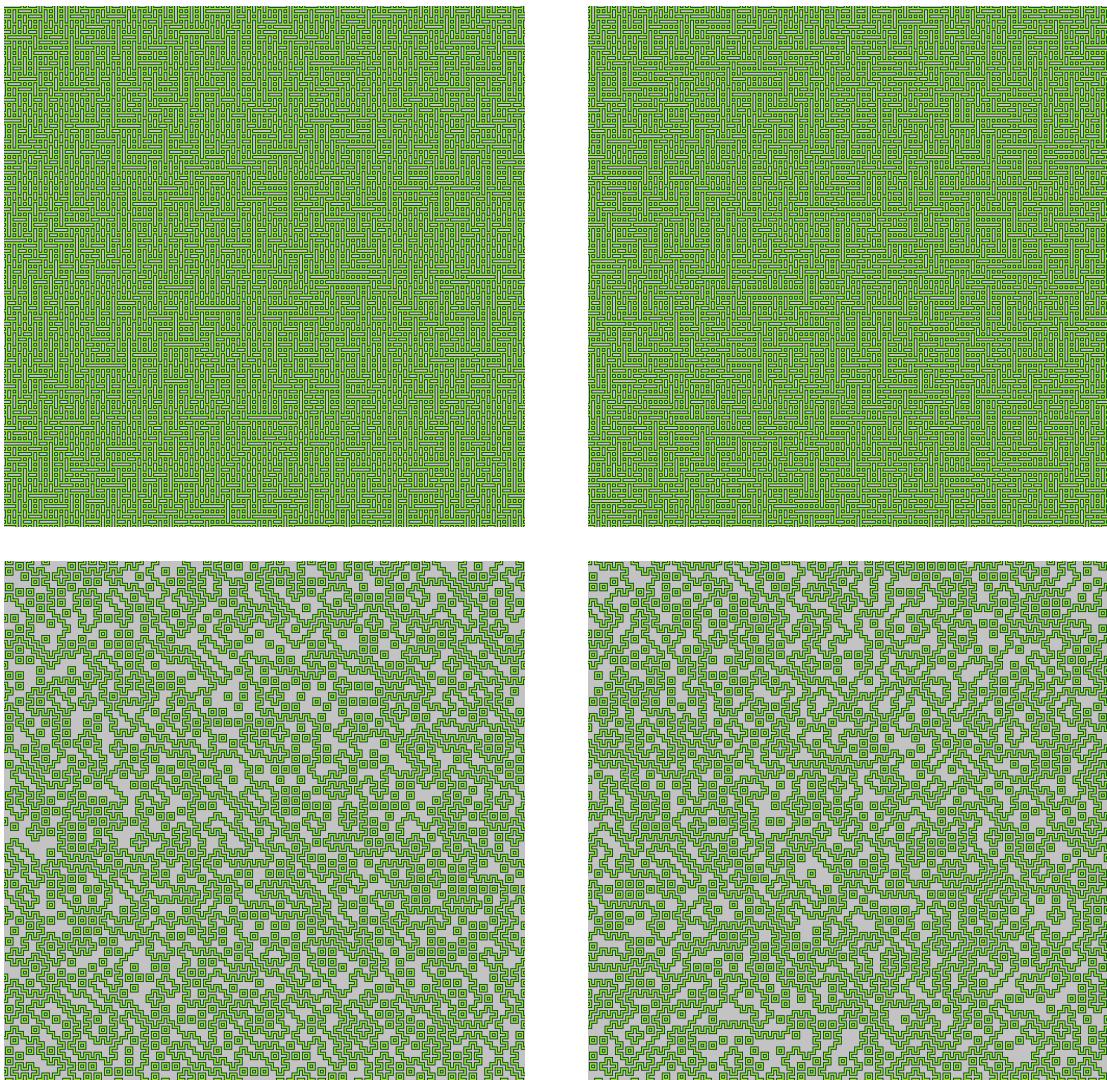
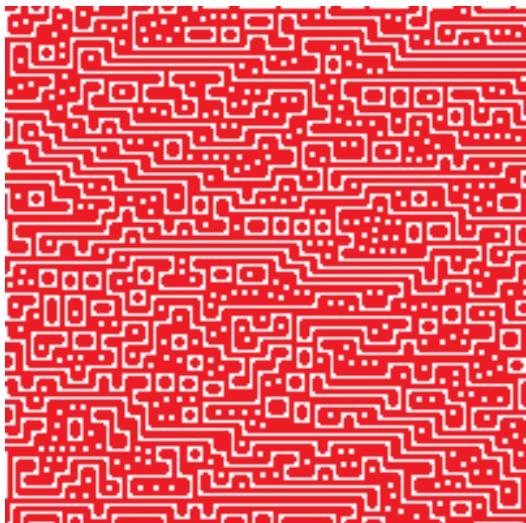
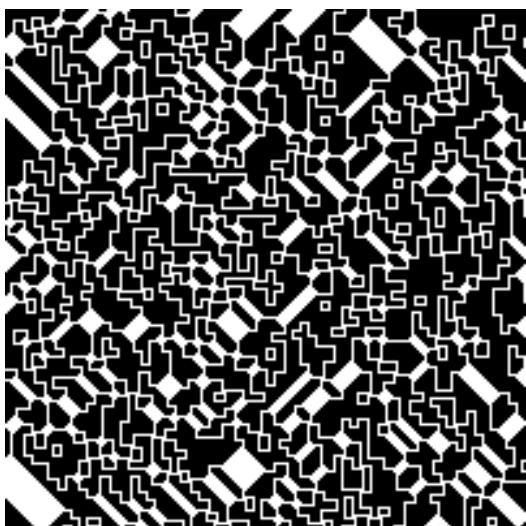


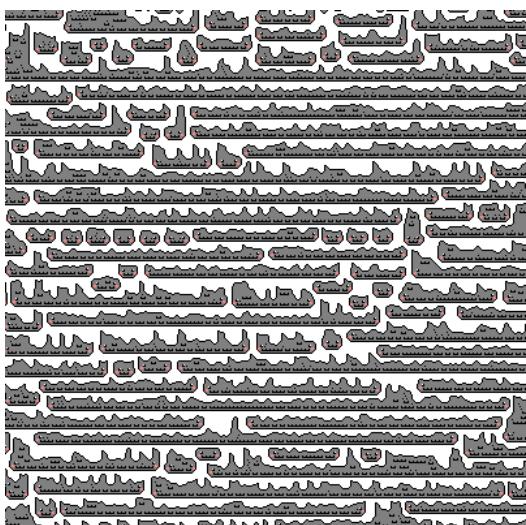
Abbildung 5.3: Texturen Links mit Model Synthese generiert. Texturen Rechts mit WFC. [14, S.12].



kein resultat



kein resultat



kein resultat

Abbildung 5.4: Texturen Links mit Model Synthese generiert. Texturen Rechts mit WFC.
[14, S.13].