Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Wave Function Collapse

Tommé Nöll

Ausarbeitung im Rahmen des Fachseminars zur GameDevWeek im Sommersemester 2020

Trier, 31.08.2020

Inhaltsverzeichnis

1

Einleitung

Die prozedurale Generierung wird in vielen Anwendungsbereichen verwendet. Besonders in Spielen sind oft die Erzeugung von Levelstrukturen aus gegebenen Bausteinen oder die Textursynthese relevant.

Wave Function Collapse ist ein von Maxim Gumin entwickelter Algorithmus für die prozedurale Generierung, welcher durch das gleichnamige Konzept in der Quantenphysik inspiriert wurde. Er kann unter anderem verwendet werden um aus simplen Texturen größere Versionen dieser Textur zu generieren (Abb. 1.2), oder aus Einzelteilen Objekte und Level (Abb. 1.1) zu erzeugen, wobei vorgegebene Regeln strikt eingehalten werden.

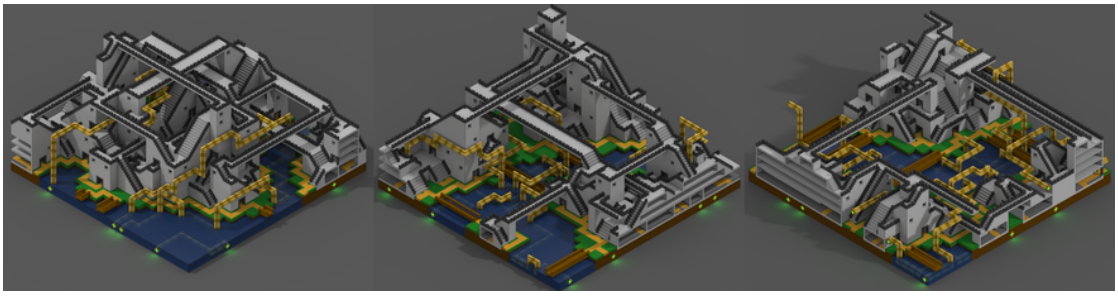

Abb. 1.1. 3D-Modell, generiert mit WFC. Bildquelle: [Gum]

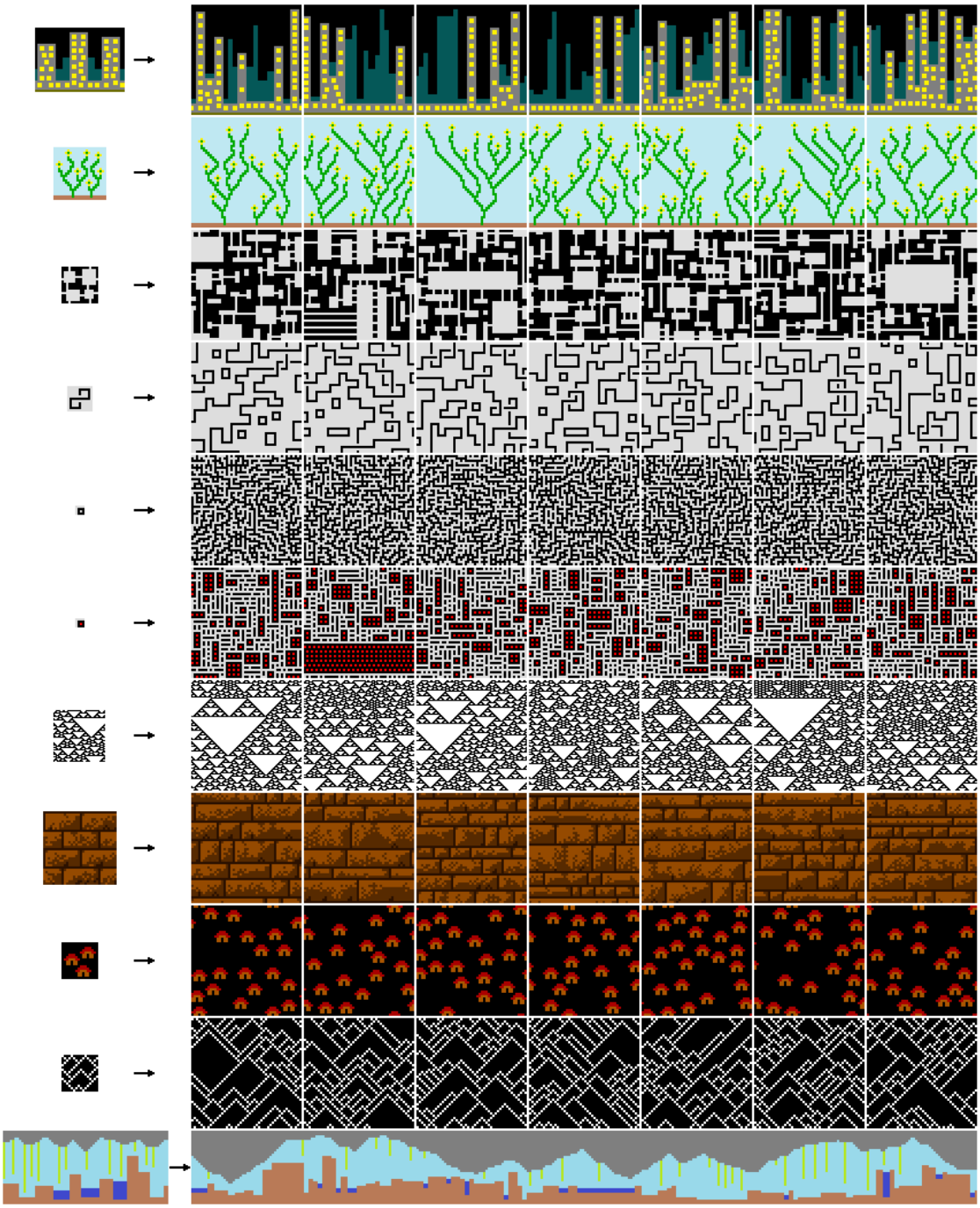

Abb. 1.2. Ergebnisse der Textursynthese anhand verschiedener Eingabemuster. Bildquelle: [Gum]

2

Funktionsweise

Der Wave-Function-Collapse-Algorithmus ist in [Gum] beschrieben. Falls nicht anders erwähnt, beziehen sich die Aussagen in diesem Kapitel daher auf diese Quelle.

Das Ziel von Wave Function Collapse ist die Anordnung von Einzelbausteinen anhand von Regeln. Beides wird dazu entweder vorgegeben oder aus einem Bild ausgelesen. Die Regeln sind für gewöhnlich Festlegungen über die möglichen Nachbarn der einzelnen Bausteine.

Zunächst wird ein Gitter erzeugt, in das die Bausteine hineingesetzt werden können. Da noch nicht bekannt ist, wo welcher Baustein hingesetzt werden soll, wird zunächst jeder Baustein in jedes Feld eingetragen. Das bedeutet, dass in den Feldern jeweils für alle Bausteine die Möglichkeit besteht, dass am Ende dieser Baustein in das Feld gesetzt wird.

Dies entspricht grob dem Konzept der Superposition aus der Quantenphysik: Ein Teilchen in einer Superposition hat nicht, wie in der klassischen Physik, einen bestimmten Zustand, sondern eine Wahrscheinlichkeitsverteilung mehrerer möglicher Zustände [Dir88], oft als Wellenfunktion dargestellt. Erst durch Observation kollabiert diese Wellenfunktion ("Wave Function Collapse") zu einem bestimmten Zustand.

Das Konzept der Observation ist ebenfalls Bestandteil des Algorithmus. In einem Feld wird einer der möglichen Bausteine ausgewählt und fest in dieses Feld eingesetzt. Anschließend werden die Auswirkungen dieser Festlegung auf die anderen Felder bestimmt und die Felder entsprechend aktualisiert, so dass die nun unmöglichen Bausteine nicht mehr in den Feldern eingetragen sind. Dieser Schritt

wird als Propagation bezeichnet. Diese Abfolge von Observation und Propagation wird wiederholt, bis für alle Felder ein Baustein festgelegt ist.

2.1 Auswahl des Feldes für die Observation

Wenn eine Observation durchgeführt werden soll, muss dafür zunächst ein Feld ausgewählt werden. Ein mögliches Auswahlkriterium hierfür ist das der wenigsten möglichen Bausteine in einem Feld. Dieses Feld läuft am ehesten Gefahr, dass dort keine Möglichkeiten mehr übrig bleiben, d. h. ein Widerspruch erreicht wird. Durch Observation dieses Feldes wird ein Baustein darauf festgelegt, und das restliche Gitter muss sich an diese Festlegung anpassen und danach richten.

Falls den Bausteinen keine Gewichtungen zugewiesen sind, entspricht das Maß der wenigsten möglichen Bausteine in einem Feld dem Maß der niedrigsten Entropie. Letzteres drückt allgemeiner aus, wie viel Information noch in dem Feld enthalten ist, ist also geringer je mehr Information bereits bekannt ist. Durch die Berücksichtigung der Gewichte sorgt das Maß der niedrigsten Entropie zusätzlich dafür, dass die Häufigkeit der Bausteine im Endergebnis näher an den Gewichtungen liegt, weshalb es besser für den Algorithmus geeignet ist.

2.2 Propagation

Ausgehend von dem durch die Observation festgelegten Feld passt sich das restliche Gitter an die Änderung an. Dazu werden zunächst die Möglichkeiten der Nachbarfelder bestimmt. Falls sich diese durch die Observation geändert haben, müssen auch dessen Nachbarn wiederum überprüft werden. Diese Aktualisierungen setzen sich durch das Gitter fort, bis sie keine Änderung mehr verursachen.

2.3 Widerspruch

Es kann vorkommen, dass durch eine Observation ein Widerspruch erreicht wird. Das heißt, dass ein anderes Feld keine möglichen Zustände mehr hat. In einem sol-

chen Fall kann einfach das Gitter wieder geleert und der Algorithmus neu gestartet werden [Hea18].

Alternativ könnte, durch Überführen bereits festgelegter Felder zurück in eine Superposition, versucht werden, wieder einen widerspruchsfreien Zustand zu erreichen (Backtracking) [Hea18]. Beispielsweise könnte die letzte Observation rückgängig gemacht werden, da diese anscheinend zum Widerspruch führte. Von diesem Widerspruchsfreien Zustand aus wird dann der Algorithmus fortgesetzt. Es kann allerdings auch sein, dass bereits vor dieser vorangegangenen Observation keine Möglichkeit mehr bestand, das Gitter auszufüllen, und der Widerspruch nur noch nicht zu erkennen war. In diesem Fall könnte erst durch Rückgängigmachen einer unbekannten Anzahl Schritte eine Lösung gefunden werden. Da Widersprüche ohnehin relativ selten auftreten, ist ein Neustart oft sinnvoller.

2.3.1 Auswirkungen des Bausteinsatzes

Ob und wie häufig Widersprüche auftreten, ist abhängig vom ausgewählten Bausteinsatz. Die Frage, ob aus einem gegebenen Bausteinsatz nichttriviale[1] Anordnungen generiert werden können, ist NP-schwer. Aus diesem Grund kann man für gewöhnlich keine Garantien über das Widerspruchsverhalten eines gegebenen Bausteinsatzes liefern. Im Allgemeinen sinkt die Wahrscheinlichkeit für einen Widerspruch jedoch, je mehr mögliche Nachbarn es pro Kante eines Bausteins gibt. Die Anzahl der Auswahlmöglichkeiten wird dadurch allerdings an jedem Punkt erhöht, weshalb mehr zufällige Entscheidungen notwendig werden. Durch den höheren Zufallsanteil enthält das Ergebnis weniger interessante Strukturen.

2.4 Regeln

Die Regeln, welche vom Algorithmus eingehalten werden müssen, bestehen typischerweise aus der Festlegung der möglichen Nachbarn eines Bausteins. Es sind

[1] Trivial sind beispielsweise Anordnungen der Größe 1×1, oder, im Falle der Textursynthese, das ursprüngliche Bild, aus dem die Bausteine stammen.

aber auch andere Regeln möglich. Beispielsweise könnten für bestimmte Elemente feste Anzahlen vorgegeben werden, welche vom Algorithmus eingehalten werden müssen. Eine andere Möglichkeit ist, einzelne Felder im Ergebnis im Vorhinein zu bestimmen, in dem sie von Anfang an auf den entsprechenden Baustein festgelegt werden. So kann auch eine Anwendung umgesetzt werden, in welcher der Nutzer das Bild interaktiv mitgestalten kann.

2.5 Textursynthese

Falls anstatt von Bausteinen ein einzelnes Referenzbild vorgegeben ist, ist eine Vorverarbeitung nötig. Dazu muss ein Wert n festgelegt werden, welcher die Größe der Einzelbausteine bestimmt. Alle im Bild vorkommenden Muster der Größe $n \times n$ werden ausgelesen und als Bausteine gespeichert, wobei auch die Anzahl der Vorkommnisse aufgezeichnet wird. Diese wird im Algorithmus als Gewichtung der Bausteine verwendet, damit die Häufigkeitsverteilung des Ergebnisbildes derer des Referenzbildes möglichst ähnelt. Im Verlauf des Algorithmus werden die Bausteine dann überlappend platziert, sodass die Positionen zweier Nachbarn sich nur um einen Pixel unterscheiden. Als Regel gilt entsprechend, dass sie in den überlappenden Pixeln identisch sein müssen. Dieser Ansatz wird als überlappendes Modell bezeichnet.

3

Implementierung

Im Folgenden wird eine Implementierung von WFC beschrieben, welche im Rahmen dieser Arbeit entwickelt wurde. Der Algorithmus kommt unoptimiert sehr schnell an seine Grenzen, vor allem in Hinsicht auf die Größe des Gitters. Durch verschiedene Optimierungen kann aber eine starke Verbesserung der Laufzeit erzielt werden. Obwohl die Implementierung nur zur Illustration dient, wurde Wert auf Performanz gelegt, da für die meisten Anwendungsfälle eine unoptimierte Lösung nicht ausreicht.

3.1 Datenstruktur

Für die Implementierung muss zunächst eine Datenstruktur erstellt werden, welche die Möglichkeiten in den Feldern repräsentiert. Die Felder könnten direkt als Arrays umgesetzt werden, welche die möglichen Bausteine enthalten. Hier sollte allerdings von Anfang an die Performanz in Anbetracht gezogen werden. Indem die möglichen Bausteine nicht beispielsweise als Elemente in einem Array implementiert werden, sondern als einzelne Bits in einer gemeinsamen Variable, lässt sich der Platzbedarf massiv verringern, was aufgrund begrenzter Speichergeschwindigkeiten und Caching auch einen großen Einfluss auf die Laufzeit hat. Außerdem müssen, um die Möglichkeiten in einem Feld zu aktualisieren, alle Möglichkeiten der Nachbarn überprüft werden. Durch die Bit-Repräsentierung ist dies durch bitweise Operatoren umsetzbar, sodass immer bis zu 64 Möglichkeiten in einem einzigen Schritt verglichen werden können.

Für die Zuordnung der einzelnen Bits zu den Bausteinen werden die Bausteine in einem globalen Array festgehalten. Wenn das n-te Bit in einem Feld gesetzt ist, besteht für dieses Feld die Möglichkeit, dass es später auf den n-ten Baustein im Bausteinarray festgelegt wird.

3.2 Hauptschleife

Nachdem in allen Feldern alle Möglichkeiten eingetragen, das heißt alle Bits auf 1 gesetzt wurden, wird die Hauptschleife betreten, welche aus wiederholter Observation (Festlegung) eines Feldes und anschließender Propagation dieser Änderung besteht. Der Algorithmus ist abgeschlossen, wenn alle Felder festgelegt sind, oder ein Feld keine möglichen Bausteine mehr hat.

3.2.1 Observation

Zunächst muss das Feld ausgewählt werden, welches observiert werden soll. Dazu wird die Entropie jedes Feldes berechnet. Die Entropieberechnung läuft durch alle Gewichte g der möglichen Bausteine in einem Feld, und berechnet darüber $\sum g$ und $\sum(g * log(g))$. Die Shannon-Entropie E des Feldes lässt sich anschließend mit der Formel 3.1 berechnen [Hea18].

$$E = log(\sum g) - \sum(g * log(g))/\sum g \tag{3.1}$$

Das Feld mit der niedrigsten Entropie wird zufällig, unter Berücksichtigung der Gewichte, auf einen Zustand festgelegt. Im nächsten Schritt muss das Gitter an diese Änderung angepasst werden.

3.2.2 Propagation

Die Änderung eines Feldes kann für die Nachbarfelder bedeuten, dass dort Bausteine unmöglich werden, diese Felder sich also auch ändern. Deren Nachbarn müssen anschließend ebenfalls überprüft werden. Auf diese Weise setzen sich die Änderungen rekursiv durch das Gitter fort, bis in dem betrachteten Feld keine

Änderungen mehr geschehen. Es bietet sich also eine rekursive Implementierung an. Alternativ kann eine iterative Umsetzung die zu überprüfenden Positionen in einer Datenstruktur sammeln, von welcher aus sie nacheinander abgearbeitet werden.

Die rekursive Implementierung dieses Schrittes wäre weniger performant als eine iterative Umsetzung. Ein Grund hierfür ist, dass Funktionsaufrufe aufwändiger sind als Schleifendurchläufe, und Rekursion deshalb im Allgemeinen langsamer ist als Iteration. Hinzu kommt in diesem Fall noch, dass bei der Rekursion eine doppelte Überprüfung eines Feldes vorkommen kann, die bei der iterativen Umsetzung vermeidbar gewesen wäre. Ein solcher Fall ist in Abbildung 3.1 zu sehen. Durch Verwendung einer Datenstruktur, in welcher Mehrfacheintragungen nicht erlaubt sind, wäre das Feld dort nur einmal überprüft worden. Doppelte Überprüfungen kommen nur noch vor, wenn sich zwischen diesen ein Nachbar geändert hat. In dieser Arbeit wurde die Propagation deshalb iterativ umgesetzt.

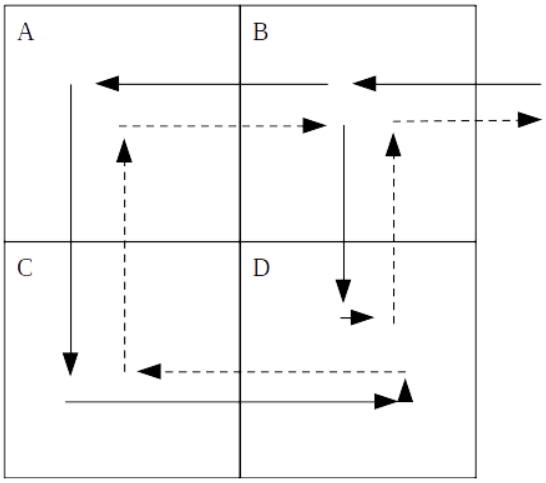

Abb. 3.1. Relevanter Ausschnitt aus einer vermeidbaren doppelten Überprüfung des Feldes D. Funktionsaufrufe sind als durchgezogene Pfeile dargestellt, das Zurückkehren aus diesen als gestrichelt. Obwohl D bereits von C aus überprüft wurde, und sich danach kein benachbartes Feld geändert hat, wird D auch von B aus geprüft.

Um zu überprüfen, welche Bausteine in einem Feld möglich sind, müssen die Nachbarn betrachtet werden. Die Regeln legen fest, welche Bausteine an einen bestimmten Baustein angrenzen können. In der Implementierung hat jeder Baustein für jede Seite eine „Edgemask“, welche die möglichen Nachbarn als einzelne Bits darstellt. Um für ein Feld die möglichen Nachbarn an einer Seite zu bestimmen, müssen die Edgemasks dieser Seite über alle möglichen Bausteine kombiniert werden. Wie in Kapitel 3.1 erwähnt, kann diese Kombination dank der bitweisen Darstellung durch einen bitweisen Operator erfolgen, in diesem Fall ein logisches Oder. Für ein Feld werden die vier kombinierten Edgemasks der Nachbarn mit einem logischen Und verknüpft, woraus die aktualisierten Möglichkeiten für dieses Feld resultieren.

3.3 Bausteinsatz und Regeln

Für die Anwendung wurde ein Satz aus 7 quadratischen Bausteinen der Größe 9x9 Pixel erstellt, welcher in Abbildung 3.2 zu sehen ist. Die Pixelgröße der Bausteine ist für die Anwendung fast vollkommen irrelevant, da die Grafiken während des eigentlichen Algorithmus nicht verwendet werden.

Der Regelsatz ergibt sich aus der Vorgabe, dass die Kanten der benachbarten Bausteine übereinstimmen müssen. Die möglichen Nachbarn an jeder Kante werden für alle Bausteine in der Vorverarbeitung bestimmt. Dazu werden die gegenüberliegenden Kanten zweier Bausteine pixelweise verglichen. Falls sie übereinstimmen, werden sie gegenseitig als mögliche Nachbarn abgespeichert. Als Datenstruktur hierfür dienen, einheitlich mit der Repräsentationen der Möglichkeiten in einem Feld, einzelne Bits in einer Variable.

3.3.1 Ergebnisse

Durch Kombination der Bausteine zu Teilbausteinsätzen lassen sich unterschiedliche Arten von Bildern generieren, was in Abbildung 3.3 dargestellt wird. Ohne am Algorithmus etwas zu ändern lassen sich sehr unterschiedliche Ergebnisse erzielen.

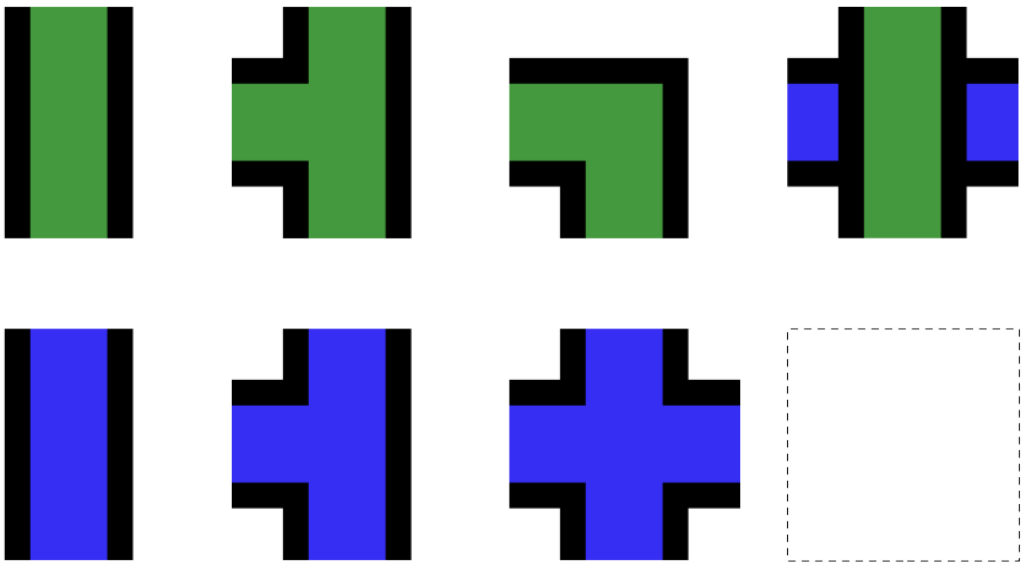

Abb. 3.2. Der verwendete Bausteinsatz. Das gestrichelt umrandete Quadrat stellt einen komplett weißen Baustein dar. Die Bausteine sind jeweils quadratisch mit einer Größe von 9x9 Pixeln.

Besonders sticht der vierte Bausteinsatz heraus. Zwischen den blauen Bausteinen bilden sich Rechtecke, welche von grünen Bausteinen gefüllt werden können. Diese Strukturen entstehen dadurch, dass es im ausgewählten Bausteinsatz keine blauen Kurven oder Kreuzungen zwischen den Farben gibt. So kann keine blaue Linie in ein Rechteck hinein führen, ohne bis zur gegenüberliegenden Seite fortzufahren, und somit ein oder zwei neue Rechtecke zu bilden. Daraus folgt, dass Gruppen grüner Bausteine immer von perfekten blauen Rechtecken umschlossen sein müssen.

Da es recht schwierig ist, mit den gegebenen grünen Bausteinen ein Rechteck genau auszufüllen, treten bei größeren Gittern immer öfter Widersprüche auf, was bei den anderen drei Bausteinsätzen fast nie geschieht. Außerdem kommt es häufig vor, dass ein Bild vollkommen ohne blaue Bausteine generiert wird.

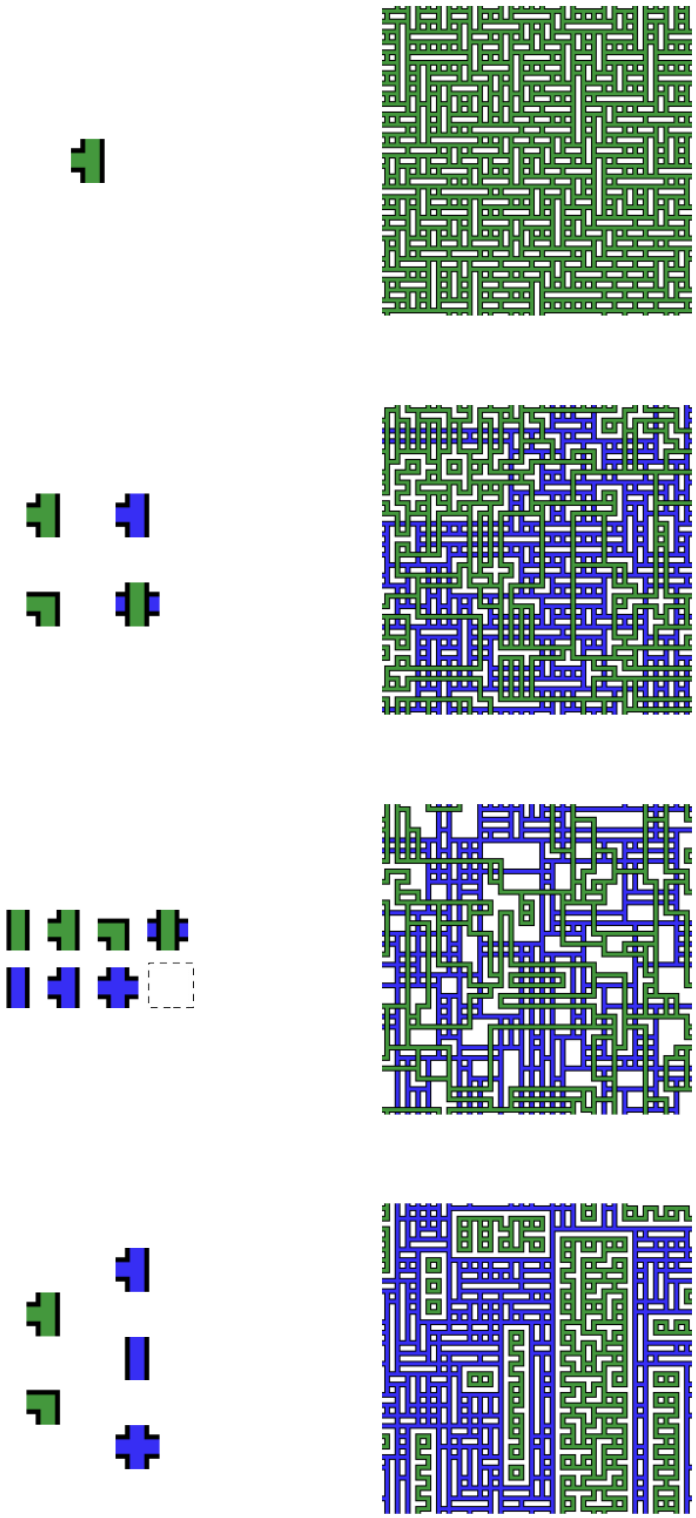

Abb. 3.3. Verschiedene Teilbausteinsätze und typische sich daraus ergebende Bilder (Größe: 30x30 Bausteine).

4

Optimierung

Der Algorithmus enthält viele ineinander geschachtelte Schleifen. Eine Observation betrachtet alle n Felder, und iteriert jeweils durch alle Bits der Variable, welche die Möglichkeiten darstellt, um die Entropie zu berechnen. Die Laufzeit der Propagation ist aufgrund ihrer rekursiven Natur schwer zu bestimmen, für gewöhnlich macht die Observation aber den deutlich größeren Anteil an der Laufzeit aus. Im schlimmsten Fall muss jedes Feld durch Observation bestimmt werden.

Unter Vernachlässigung des Propagationsschrittes lässt sich die Laufzeit also als $O(n^2 * b)$ abschätzen, mit n als Anzahl der Felder, und b als Anzahl der Bausteine. n ist dabei selbst quadratisch (bei einem zweidimensionalen Gitter) oder kubisch (im dreidimensionalen) in der Seitenlänge. Wie der Tabelle 4.1 zu entnehmen ist, führt dies bei wachsender Gittergröße zu einem schnellen Anstieg der Laufzeit. Für größere Gitter ist daher eine Optimierung nötig.

4.1 Caching der Entropien

Die erste und wichtigste Optimierung ist das Caching der Entropiewerte (auch in [Gum] umgesetzt). Hierzu wird der berechnete Entropiewert für jedes Feld abgespeichert. Durch einen booleschen Wert können einzelne Entropiewerte als „dirty“ markiert werden, um anzuzeigen, dass der alte Wert ungültig ist und neu berechnet werden muss. Nach der Neuberechnung wird die Markierung wieder entfernt.

Optimierung	Gitter-größe	Entropiebe-rechnungen	Berechnungen kombinierter Edgemasks	Laufzeit (ms)
Keine Optimierung	30x30	332 100	17 092	363,7
	50x50	2 507 500	47 880	2 430,1
	70x70	9 069 900	98 740	9 027,6
Caching der Entropien	70x70	13 849	98 336	171,8
	150x150	64 142	469 608	1 166,9
	300x300	255 757	1 689 768	9 780,2
C. d. komb. Edgemasks	70x70	15 793	10 195	88,4
	150x150	72 237	122 160	770,2
	300x300	286 863	470 348	8 328,5
Schrump. Entropie-Array	70x70	10 951	25 782	72,0
	150x150	49 768	122 062	568,5
	300x300	196 987	470 315	5 681,6
inline checkEntropy	300x300			3 463,0
	600x600			49 110,3

Tabelle 4.1. Rechenaufwände mit verschiedenen Optimierungen. Die wichtigsten Unterschiede sind farblich hervorgehoben. Der Bausteinsatz bestand aus den vier Rotationen des T-Blocks. Die Anzahlen der Funktionsaufrufe sind jeweils aus einem einzelnen Durchlauf, und können daher nur zur Einschätzung der Größenordnung dienen. Die Laufzeiten sind Durchschnitte von zehn Durchläufen, gemessen auf einem Computer mit AMD Ryzen 2700X 3,7GHz-Prozessor und 2x8GB DDR4-3000 CL16-18-18-38-Arbeitsspeicher.

Alternativ zu einem separaten booleschen Wert kann die Entropie auch auf einen festgelegten ungültigen Wert gesetzt werden, um zu zeigen, dass eine Neuberechnung vonnöten ist. Voraussetzung hierfür ist, dass ein ungültiger Wert existiert.

Eine Dirty-Flag ist nur dann nötig, wenn sich die möglichen Bausteine im Feld geändert haben. Im Observationsschritt wird nur ein Feld verändert. Dessen Entropie wird dabei auf unendlich gesetzt, damit es nicht mehr für die Observation aus-

gewählt werden kann. Eine Dirty-Flag ist also nicht mehr nötig, und kann folglich nur bei einer Änderung im Propagationsschritt gesetzt werden.

Durch das Caching der Entropien wird die Anzahl der nötigen Entropieberechnungen im Beispielfall der Gittergröße 70x70 (siehe Tabelle 4.1) um 99,8% reduziert, wodurch sich die Geschwindigkeit der gesamten Berechnung auf etwa das 53-fache erhöht.

4.2 Caching der kombinierten Edgemasks

Das Prinzip des Caching lässt sich auch auf die kombinierten Edgemasks anwenden, welche zur Bestimmung der Möglichkeiten in den Nachbarfeldern benötigt werden. Bei der Änderung eines Feldes werden die vier Edgemasks gleichzeitig invalidiert, aber jede für sich bei Bedarf neu berechnet. Die Anzahl nötiger Berechnungen der Edgemasks wurde auf etwas über ein Viertel reduziert, wodurch die Laufzeit um ca. 15% sank[1].

4.3 Schrumpfendes Entropie-Array

Durch Zuhilfenahme von Profiling-Werkzeugen zeigte sich, dass ein großer Teil der Laufzeit mit dem Bestimmen des niedrigsten Entropiewertes zugebracht wurde. Zur Optimierung sollen daher bereits abschließend betrachtete Felder aus dem Array entfernt werden.

Zuvor entsprach ein Entropiewert an einer Indexposition im Entropie-Array demjenigen Feld mit der gleichen Indexposition im Gitter. Da die Felder nicht entfernt werden können, sondern nur die Entropien, ist diese Zuordnung nun nicht mehr gültig, und muss anders umgesetzt werden. Hierzu wurde in dem Entropien-Array zu jeder Entropie auch die Indexposition des entsprechenden Feldes gespeichert. Außerdem muss das Array mit einer dynamischen Datenstruktur ersetzt werden. In C++ eignet sich hierfür die „Vector“-Klasse. Für das Array der dirty-Flags wird die Entsprechung der Indizes mit dem Gitter beibehalten.

[1] Aussage bezieht sich auf den Beispielfall 300x300, siehe Tabelle 4.1

Nach dieser Optimierung müssen im Observationsschritt nur noch die Entropien der Felder betrachtet werden, welche noch nicht auf einen bestimmten Baustein festgelegt sind. Die Laufzeit wurde hierdurch um ca. 32% reduziert[2]

4.4 Inlining der checkEntropy-Funktion

Die letzte Optimierung in dieser Arbeit ist nur ein kleines Detail, welches allerdings eine dramatische Auswirkung auf die Laufzeit hatte. In der Implementierung wird eine simple Helferfunktion namens „checkEntropy“ verwendet, welche den Entropiewert eines Feldes zurückliefert, und diesen dabei selbst neu berechnet, falls er als dirty markiert ist. Dank des Cachings ist letzteres in einem Gitter der Größe 300x300 nur noch ca. 200 000 mal nötig. Die checkEntropy-Funktion wurde im gleichen Durchlauf hingegen ungefähr 1,4 Milliarden mal aufgerufen. Obwohl die Funktion sehr simpel ist, wird durch diese immense Anzahl an Aufrufen trotzdem einiges an Laufzeit für sie verwendet.

Funktionsaufrufe sind allgemein mit einem Aufwand verbunden, der für sich genommen zwar klein ist, bei Anzahlen in dieser Größenordnung jedoch einen signifikanten Anteil ausmacht. So müssen die Parameter für den Aufruf auf dem Stack platziert und der Stack selbst verwaltet werden [LMG+18]. Daher ist in diesem Fall weniger der Inhalt der Funktion ausschlaggebend, als der Funktionsaufruf an sich.

In C++ gibt es das Schlüsselwort „inline“, welches dem Compiler anzeigt, dass, statt einem gewöhnlichen Funktionsaufruf, es bevorzugt werden soll, den Inhalt der Funktion direkt an der Stelle des Funktionsaufrufs einzufügen [ISO17]. Somit wird der Mehraufwand des Aufrufs eingespart. Im Gegenzug kann die Dateigröße des Programms wachsen.

Durch Inlining der checkEntropy-Funktion wurde die Laufzeit um ca. 39% reduziert[2]. Hierbei ist anzumerken, dass ein Großteil des Zeitgewinnes, welcher durch die Optimierung des Schrumpfenden Entropie-Arrays erzielt wurde, durch den

[2] Aussage bezieht sich auf den Beispielfall 300x300, siehe Tabelle 4.1

Aufwand des Funktionsaufrufs begründet war. Bei Weglassen dieser Optimierung, aber mit Inlining der checkEntropy-Funktion, wird für das 300x300-Gitter eine Laufzeit von 3857,8 ms erzielt.

4.5 Fazit

Nach dem sehr dramatischen Performanzgewinn durch das Caching der Entropien konnte die Laufzeit noch einmal auf knapp über ein Drittel reduziert werden. An dem Fall 600x600 in Tabelle 4.1 lässt sich erkennen, dass die vierfache Feldanzahl ungefähr zu der 16-fachen Laufzeit führt. Das entspricht weiterhin dem vor der Optimierung berechneten Wachstumsverhalten von $O(n^2 * b)$. Die Optimierung scheint sich also nur durch einen konstanten Faktor in der Laufzeit ausgewirkt zu haben.

Trotz der hauptsächlich auf den Observationsschritt ausgerichteten Optimierungen ist dieser momentan für 98,5% der Laufzeit verantwortlich. Eine weitere mögliche Optimierung, welche auf diesen Schritt abzielt, wäre das Sortieren des Entropie-Arrays. So müsste es nicht mehr für jede Observation komplett durchlaufen werden, da der erste Eintrag der niedrigste wäre. Die Sortierung könnte bei einer Änderung inkrementell durch Vertauschungen lokal bei dem geänderten Wert beibehalten werden. Ob mehr Zeit eingespart würde, als durch die Sortierungen verloren geht, ist allerdings unklar, weshalb in dieser Arbeit auf eine Umsetzung verzichtet wurde.

5

Constraintprogrammierung

Der Kernbestandteil des Wave-Function-Collapse-Algorithmus sind die Regeln für die Anordnung der Bausteine. Diese bieten sich neben der beschriebenen Implementierung auch für eine Umsetzung mithilfe von Constraintprogrammierung an. Es handelt sich hierbei um eine Form der deklarativen Programmierung. Anders als bei der imperativen Programmierung, in welcher das Programm durch eine Abfolge von durchzuführenden Schritten implementiert wird, wird bei der deklarativen Programmierung nur das Problem selbst definiert. Anschließend wird es es an einen allgemeinen Löser (solver) übergeben, welcher das Ergebnis berechnet.

Im speziellen Fall der Constraintprogrammierung wird das Problem modelliert durch ([Apt03]):

1. eine Reihe von Variablen, für welche Werte gefunden werden sollen
2. Domänen/Wertebereiche für die Variablen
3. Beschränkungen (Constraints), welche in der Lösung gelten müssen

Die Beschränkungen können eine einzelne Variable betreffen oder Beziehungen zwischen mehreren darstellen.

5.1 Löser

Löser sind unabhängig von einem bestimmten Problem, da sie den gesamten Lösungsraum durchsuchen. Sie tragen mögliche Werte in die Variablen ein, bis sich ein Widerspruch ergibt. Dann werden eine bzw. mehrere Entscheidungen rückgängig gemacht, und andere Werte eingetragen. Die Suche funktioniert also

hauptsächlich durch Backtracking. Der Lösungsraum kann auch als Baum modelliert werden, so dass jeder Knoten die Entscheidung für eine Variable darstellt, und jedes Blatt eine Lösung. Die Suche mit Backtracking ist dann eine Tiefensuche.

Da der Lösungsraum sehr groß ist, ist eine Optimierung notwendig. Weit verbreitet sind hier constraint propagation (Beschränkungspropagation), welches der Propagation in WFC entspricht, und Heuristiken für die Auswahl der Variablen und der darin einzutragenden Werte [KS17].

5.2 Vergleich mit WFC

Es bilden sich einige Ähnlichkeiten zum Wave-Function-Collapse-Algorithmus heraus. Dieser verwendet ebenfalls eine Heuristik zur Auswahl der Felder, um Widersprüche zu vermeiden. Nach der Festlegung eines Feldes findet in beiden Ansätzen ein Propagationsschritt statt, um die Auswirkungen auf die restlichen Variablen zu bestimmen. Im Unterschied zur Constraintprogrammierung wird in WFC aber keine Heuristik zur Auswahl des Wertes (Bausteins) verwendet, sondern auf eine gewichtet zufällige Häufigkeitsverteilung im Ergebnis abgezielt. Außerdem ist Backtracking ein fester Teil von Constraintprogrammierung, da diese nicht Problemspezifisch ist, und daher widerspruchsfreie Lösungen möglicherweise sehr schwer zu finden sind. Im Gegensatz hierzu reicht im Anwendungsfall von WFC typischerweise die Propagation dafür aus, Widersprüche zu vermeiden, weshalb für den seltenen Fall eines Widerspruchs ein Neustart (für gewöhnlich, abhängig von den Beschränkungen) genügt.

Eine tiefergehende Gegenüberstellung der beiden Ansätze, sowie eine Umsetzung des Problems mit Constraintprogrammierung, ist in [KS17] beschrieben.

6

Anwendungen

Wave Function Collapse liefert einen neuartigen Ansatz zur prozeduralen Generierung. Um den Algorithmus zu verstehen ist keine komplizierte Mathematik nötig. Da er aber wenig bis gar keine Konfiguration benötigt, kann er auch gut ohne ein tiefes Verständnis verwendet werden. Aus diesen Gründen hat er das Interesse vieler Entwickler geweckt, welche ihn für verschiedene Anwendungsfälle eingesetzt und weiterentwickelt haben.

Eine extensive Liste von Anwendungen, Implementierungen in verschiedenen Sprachen, Experimenten, Demos und Anderem lässt sich auf [Gum] finden.

6.1 Levelgenerierung

Oskar Stålberg passte den Algorithmus darauf an, mit ungewöhnlichen Gitterformen wie kugelförmigen [Sta16] und unregelmäßigen Gittern [Sta19] umzugehen, und verwendete ihn im 2018 veröffentlichten Spiel Bad North zur Erzeugung von Leveln. Der Algorithmus ist außerdem das Kernelement seines aktuellen Projektes Townscaper, in welchem aus Eingaben des Spielers interaktiv Städte erzeugt werden.

Für Unity wurden zwei Implementierungen als Packages veröffentlicht [Par] [tes], erstere von Joseph Parker, der den Algorithmus auch in seinen Spielen Proc Skater 2016, Swapland (2017) und Bug with a Gun (2018) verwendete.

6.2 Textursynthese

In Rodina wird die Textursynthese zur prozeduralen Dekoration von Wänden verwendet [rod18]. Caves of Qud erzeugt Texturen als Zwischenergebnisse für die Generierung von Levels [Buc19].

7

Zusammenfassung

Es wurde der Wave-Function-Collapse-Algorithmus vorgestellt, dessen Funktionsweise erklärt und eine Implementierung dargestellt. Durch eine ausführliche Optimierung wurde gezeigt, dass der Algorithmus so weit beschleunigt werden kann, dass er in Echtzeitanwendungen auch für das zufällige Generieren mit größeren Gittern Anwendung finden kann. Für einen Ausblick wurde der Algorithmus mit dem eng verwandten Konzept der Constraintprogrammierung in Bezug gesetzt, und dargestellt, wie der Algorithmus in der Praxis Anwendung gefunden hat.

Wave Function Collapse demonstriert eine ungewöhnliche Herangehensweise, in der es, statt zu einem leeren Gitter einzelne Elemente hinzuzufügen, aus einem vollen Gitter mit allen Möglichkeiten die Elemente Stück für Stück entfernt. Der Algorithmus eröffnet so neue Möglichkeiten, vor allem was das Erzeugen von komplexer Geometrie angeht. Da sein Verhalten sehr Datengetrieben ist, und keine komplexe Konfiguration erfordert, hat er breiten Anklang unter Entwicklern gefunden.

Literaturverzeichnis

Apt03. Apt, Krzysztof R.: *Principles of constraint programming*. Cambridge University Press, Cambridge ; New York, 2003. OCLC: ocm52358589.

Buc19. Bucklew, Brian: *Dungeon Generation via Wave Function Collapse*. `https://www.youtube.com/watch?v=fnFj3dOKcIQ`, Oktober 2019. Abrufdatum: 25.08.2020.

Dir88. Dirac, P. A. M.: *The principles of quantum mechanics*. Number 27 in *The international series of monographs on physics*. Clarendon Press, Oxford, January 1988. OCLC: 836973857.

Gum. Gumin, Maxim: *WaveFunctionCollapse git repository*. `https://github.com/mxgmn/WaveFunctionCollapse/`. Abrufdatum: 08.08.2020.

Hea18. Heaton, Robert: *The Wavefunction Collapse Algorithm explained very clearly*. `https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/`, December 2018. Abrufdatum: 08.08.2020.

ISO17. ISO/IEC JTC 1/SC 22, Committee for Programming languages, their environments and system software interfaces: *ISO/IEC 14882:2017, Standard for Programming Language C++*, December 2017.

KS17. Karth, Isaac and Adam M. Smith: *WaveFunctionCollapse is constraint solving in the wild*. In *Proceedings of the International Con-*

ference on the Foundations of Digital Games - FDG '17, pages 1–10, Hyannis, Massachusetts, 2017. ACM Press.

LMG+18. Lu, H. J., Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell: *System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0*. December 2018.

Par. Parker, Joseph: *unity-wave-function-collapse by selfsame*. https://selfsame.itch.io/unitywfc. Abrufdatum: 25.08.2020.

rod18. *Rodina 1.3.4: Buildings upgrade!* https://steamcommunity.com/games/314230/announcements/detail/3369147113795750369, September 2018. Abrufdatum: 25.08.2020.

Sta16. Stalberg, Oskar: *Oskar stålberg auf twitter über wfc auf einer kugel*. https://twitter.com/OskSta/status/784847588893814785, October 2016. Abrufdatum: 25.08.2020.

Sta19. Stalberg, Oskar: *Oskar stålberg auf twitter über wfc auf einem unregelmäßigen gitter*. https://twitter.com/OskSta/status/1164926304640229376, August 2019. Abrufdatum: 25.08.2020.

tes. *Tessera procedural tile based generator*. https://assetstore.unity.com/packages/tools/modeling/tessera-procedural-tile-based-generator-155425. Abrufdatum: 25.08.2020.