

Anwendung und Beurteilung des Wave Function Collapse Algorithmus im Kontext der prozeduralen Generierung eines Dun- geons

Application and Evaluation of the Wave Function Collapse Algorithm
in the Context of Procedural Dungeon Generation

Marcel Kühn

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr. Christoph Lürig

Ort, 5.12.2023

Kurzfassung

Die folgende Arbeit behandelt das Generieren eines Dungeons unter Verwendung des Wave Function Collapse Algorithmus und bewertet dabei dessen Eignung dafür. Dazu wird erklärt wie Dungeons alternativ generiert werden können und Beispiele anderer Verfahren der prozeduralen Generierung werden vorgestellt. Der WFC-Algorithmus wird danach in seiner Funktionsweise vorgestellt. Dabei werden Vorteile, wie dessen vielseitige Einsetzbarkeit, und Nachteile, wie dessen Komplexität aufgezeigt. Danach wird besprochen, wie dieser Algorithmus zum Generieren eines Dungeons umgesetzt werden kann. Dabei werden spezifisch für diese Arbeit definierte Anforderungen dargelegt, welche das Generieren eines realistischen Dungeons fördern sollen. Die Umsetzung dieser Anforderungen wird erklärt, sowie aufgezeigt welche Probleme der Algorithmus für manche dieser Anforderungen besitzt. Für diese Probleme werden Lösungen vorgestellt, welche im Zuge dieser Arbeit entwickelt wurden, sowie die Umsetzung aller Anforderungen in einem Beispielprogramm erklärt. Anschließend wird die Eignung des WFC-Algorithmus zum Generieren von Dungeons bewertet, sowie eventuelle Weiterentwicklungen und Transfermöglichkeiten der Erkenntnisse und entwickelten Lösungen vorgeschlagen.

Abstract

This Paper covers the generation of a Dungeon by the usage of the wave function collapse algorithm and evaluates its suitability for the context. For that it is explained how Dungeons can be generated alternatively, for which alternative algorithms of procedural generation are presented. Then the functionality of the wave function collapse algorithm will be explained. In the process advantages, like the wide possible usage, and disadvantages, as the complexity of the WFC-algorithm are shown. Afterwards its discussed how the WFC-Algorithm can be implemented for the generation of a Dungeon. At that specific requirements for the generation of a dungeon are introduced, which were defined for this paper with the purpose of generating more realistic Dungeons with a higher probability. The implementation of these requirements will be explained, as well as specific problems of the WFC-algorithm for the implementation of some of these requirements are shown. Solutions for these Problems, which were developed for this Paper, are explained. Then it will be explained how all requirements were implemented in a sample program. Afterwards the suitability of the WFC-algorithm for the generation of Dungeons is evaluated and possible further developments as well as transfer options of the insights and developed solutions are proposed.

Inhaltsverzeichnis

1	Einleitung	7
1.1	Motivation	7
1.2	Ziel der Arbeit	7
2	Möglichkeiten zum Erstellen eines Dungeons und prozedurale Generierung	8
2.1	Prozedurale Möglichkeiten zum Erstellen von Dungeons	8
2.1.1	Zellulare Automaten	8
2.1.2	Binäre Raumpartitionierung.....	9
2.2	Wang Tiles	10
2.3	Gründe zur Wahl des Wave Function Collapse Algorithmus	10
3	Erklärung des Wave Function Collapse Algorithmus.....	11
3.1	Wie funktioniert der Algorithmus	11
3.1.1	Wählen geeigneter Tiles.....	12
3.1.2	Ablauf des Algorithmus	12
3.2	Vor und Nachteile des Algorithmus	14
3.2.1	Vorteile.....	14
3.2.2	Nachteile	14
4	Theoretisches und praktisches Vorgehen	17
4.1	Vorgehen zum prozeduralen erzeugen eines Dungeons.....	17
4.2	Umsetzungen eines realistischen Dungeon durch WFC und Probleme	17
4.2.1	Erstellen eines Dungeons aus verschiedenen Elementen.....	18
4.2.2	Erstellen eines verbundenen Hauptpfades	18
4.2.3	Erstellen von weiteren Objekten wie Gegenständen oder Gegnern.....	19
4.2.4	Erstellen von Gewichtungen für Elemente und Objekte.....	19
4.2.5	Erstellen eines Dungeons indem alle Pfade verbunden sind.....	19
4.2.6	Erstellen eines Dungeon als Pfadproblem	19
4.3	Möglichkeiten zum Lösen dieses Pfadproblems	20
4.4	Gefundene und Entwickelte Anpassungen	21
4.4.1	Festlegen eines verbundenen Hauptpfades durch preprocessing	21
4.4.2	Einschränken des Algorithmus (Constraints based)	22
4.4.3	Nachträgliche Korrektur (Post Processing).....	24
5	Umsetzung im Beispielprogramm.....	27
5.1	Verwendung von Unity	27
5.2	Umsetzung des WFC-Algorithmus im Beispielprogramm zum Realisieren der Anforderungen	28
5.2.1	Einbringen verschiedener Dungeon Elemente mit verschiedenen Objekten	28
5.2.2	Implementieren eines verbundenen Hauptpfades	29
5.2.3	Implementierung von gewichteten Tiles	29

5.2.4	Implementierung zum Generieren verbundener Pfade (Pfadzwang Ansatz)	
	30	
5.2.5	Implementierung zum Generieren verbundener Pfade (Gruppenkombinierung).....	30
5.3	Weitere verwendete Klassen des Beispielprogramms.....	31
6	Erkenntnisse.....	32
6.1	Eignung des WFC-Algorithmus zum Erstellen eines Dungeons	32
6.2	Zukünftige mögliche Anwendungen und Weiterführungen.....	33
7	Zusammenfassung.....	35
	Literatur.....	36
	Glossar	38
	Erklärung der Kandidatin / des Kandidaten	39

Abbildungsverzeichnis

Abbildung 2.1.1: Beispiel zum Aufteilen einer Ebene äquivalent zum Binärbaum, Quelle (Börje Santen, 2015) S.14	9
Abbildung 2.1.2: Beispiel des verbinden von Räumen durch Binäre Raumpartitionierung, Quelle (Börje Santen, 2015) S.16-17	9
Abbildung 3.1.1: Beispiel texturbasierter WFC mit $N=3$, Quelle (Maxim Gumin, 2023)	11
Abbildung 3.1.2: Beispiel erzeugen aus beschränkten Eingabetiles, Quelle (Maxim Gumin, 2023)	12
Abbildung 3.1.3: Beispiel constraint propagation anhand Sudoku, Quelle (Boris, Wave Function Collapse Explained, 2023)	13
Abbildung 3.2.1: Beispiel WFC-Algorithmus mit vorgegebenen Tiles, Quelle (Boris, Wave Function Collapse tips and tricks, 2023)	14
Abbildung 3.2.2: Beispiel einer erzeugten Stadt ohne sinnvolle Verbindungen von Straßen, Quelle (Robert Heaton, 2023)	15
Abbildung 3.2.3: Veranschaulichung des Aufteilens einer Ebene in Teilebenen, Quelle (Yuhe NIE, 2023) S.7	16
Abbildung 4.2.1: Beispiel von Tiles zum Bilden eines Ganges, Raums oder Wand	18
Abbildung 4.2.2: Beispiel des Erstellens eines Dungeons mit Gang, Raum und Wand Tiles .	18
Abbildung 4.2.3: Beispiel 2 verschiedener Tiles die zusammenpassen. 0 steht für Wände, 1 für Gänge und 2 für Räume	18
Abbildung 4.2.4: Beispiel Tiles zum Einfügen von Objekten	19
Abbildung 4.3.1: Beispiel erzeugen eines Levels ohne und mit Stretch Space, Quelle (Hugo Scurtie, 2018)	21
Abbildung 4.4.1: Beispiel eines Dungeon mit WFC-Algorithmus mit Hauptpfad	22
Abbildung 4.4.2: Beispiel des Generierens ohne Kantenvermeidung (links) und mit (rechts)	23
Abbildung 4.4.3: Beispiel des Generierens und auffüllen eines Dungeon mit Phadzwang	24
Abbildung 4.4.4: Beispiel für das Erkennen von Pfadgruppen und Hauptpfad. Hauptpfad = rot	25
Abbildung 4.4.5: Beispiel Schrittweises suchen nach benachbarten Pfadgruppen von Extremstellen aus	25
Abbildung 4.4.6: Beispiele einer für Gruppenkombinierung unlösbaren Generierung	26

Abbildung 4.4.7: Beispiel eines mit Gruppenkombinierung erstellten Dungeons nach
Vorgaben 27

1 Einleitung

1.1 Motivation

In den Bereichen der Informatik gibt es schon lange Algorithmen zum zufälligen Erstellen von gewünschten Ausgaben. Solche Algorithmen der prozeduralen Generierung wirkten stets faszinierend auf mich, da es theoretisch keine Grenzen für ihre Anwendungen gibt. Praktisch werden diese hauptsächlich in Bereichen der Computergrafik zum Erstellen von Texturen und 3D Objekten, sowie im Bereich von Computerspielen verwendet. Es gibt auch Versuche diese Konzepte der prozeduralen Generierung in anderen Bereichen, wie beispielsweise der Architektur, anzuwenden. Da Algorithmen der prozeduralen Generierung meist relative zufällig ihre Ausgaben generieren, sind diese jedoch für viele Anwendungen nicht geeignet oder müssen individuell angepasst werden. Ein junger Algorithmus in diesem Gebiet ist der Wave Function Collapse Algorithmus, fortlaufend WFC-Algorithmus genannt. Dieser Algorithmus ist seit seinem Erscheinen 2016 als Teil der Forschungsgebiete der Informatik häufig behandelt worden, da dieser viele Eigenschaften besitzt, welche eine breite Anwendung des Algorithmus erlauben.

Daher wurde im Zuge dieser Arbeit entschieden in der Unity Engine ein Tool zu erstellen, welches Dungeons unter Anwendung des WFC-Algorithmus prozedural erstellt und dabei bestimmte Kriterien erfüllt. Dadurch soll der WFC-Algorithmus auf seine Eignung für solche Anwendungen überprüft werden, sowie Lösungen für Probleme gefunden werden, welche aus der zufälligen Natur dieses prozeduralen Algorithmus entstehen. Erkenntnisse über Probleme und Lösungen des WFC-Algorithmus in diesem Kontext sollten dabei auf andere Anwendungen weitgehendst übertragbar sein und somit möglichst Erkenntnisse für weitere Anwendungen des Algorithmus liefern. Dies könnte eine breitere Anwendung des Algorithmus in noch fremden Bereichen fördern.

1.2 Ziel der Arbeit

Diese Arbeit verfolgt das Ziel eine Anwendung zu entwickeln, welche unter Verwendung des WFC-Algorithmus zufällig erzeugte Dungeon Level erzeugt, welche jedoch immer einen verbundenen Hauptpfad zwischen einem Start und Endpunkt besitzen. Außerdem sollte möglichst eine Methode gefunden werden, durch die keine alternativen Pfade entstehen, die nicht mit dem Hauptpfad verbunden sind, um einen verwendbaren und realistischeren Dungeon zu erzeugen. Dies ist durch das zufällige Vorgehen des WFC-Algorithmus, ohne Anpassungen, nicht möglich. Daher ist ein Auseinandersetzen mit den Problemen und versuchen des finden von Lösungen ein essenzielles Ziel.

Diese Arbeit beschäftigt sich daher damit wie der WFC-Algorithmus zum Erzeugen eines Dungeons umgesetzt werden kann, welche möglichen Anpassungen vorgenommen werden können um ein möglichst zusammenhängendes Konstrukt zu erzeugen, sowie welche Probleme und Lösungen im Rahmen des Erzeugens eines Dungeons durch den WFC-Algorithmus bestehen. Anschließend wird die Eignung des Algorithmus zum Generieren von Dungeons bewertet.

2 Möglichkeiten zum Erstellen eines Dungeons und prozedurale Generierung

Ein Dungeon kann versimpelt auf zwei Arten erstellt werden, manuell von einer Person wie einem Designer, oder durch einen Algorithmus der prozeduralen Generierung. Dabei gibt es eine Vielzahl von Algorithmen der prozeduralen Generierung, welche primär in Spielen, beispielsweise zum Erstellen von Leveln wie Dungeons, oder dem Erstellen von Texturen eingesetzt werden. Der Vorteil des prozeduralen Erstellens von Dungeons liegt dabei in der Möglichkeit eine Vielzahl verschiedener Ausgaben in kurzer Zeit zu schaffen. Dies ermöglicht eine hohe Anzahl unterschiedlicher Level. Außerdem ist es dadurch möglich Anwendungen, welche solche Ausgaben verwenden, ohne den Einsatz eines Designers zu entwickeln. Dadurch können Entwickler und Unternehmen den Einsatz solcher einsparen. Zudem können Anwendungen geschaffen werden, welche Ausgaben wie Dungeons oder Texturen in einer Anzahl generieren, welche manuell, ohne das Verwenden eines Algorithmus, zeitlich nicht möglich wäre.

2.1 Prozedurale Möglichkeiten zum Erstellen von Dungeons

Es gibt viele prozedurale Verfahren, welche zum Generieren eines Dungeons verwendet werden können. Zwei bekannte Verfahren sind dabei Zellulare Automaten und die Binäre Raumpartitionierung.

2.1.1 Zellulare Automaten

Das Verfahren der Zellularen Automaten wurde erstmal 1970 bekannt durch von Jhon Horton Conway erstellte „Game of Life“ (vgl. (Börje Santen, 2015) S.18). Bei Zellularen Automaten existiert ein Netz mit Zellen, das eine Ebene darstellt, welche durch die Länge ihrer Kanten $M \times N$ aufgespannt wird. In seiner simpelsten Form sind die Einträge des Netzes als Pixel einer Ebene zu betrachten. Diese Einträge können verschiedene Objekte wie Pfade, Wände und Mauern darstellen, sind meist jedoch binär mit zwei verschiedenen möglichen Objekten. Beim Initialisieren der Ebene wird jedem Eintrag des Netzes ein zufälliger Wert zugewiesen. Dieser Zeitpunkt ist als t definiert. Dadurch ist jedes Element mit zufälligen Nachbarn umgeben. Die weiteren Schritte $t + 1$ verändern die Einträge von Zellen abhängig von ihrem vorigen Zustand. Dazu werden Regeln definiert, welche abhängig von den Nachbarn einer Zelle diese beeinflussen. Zunächst wird festgelegt welche Zellen als Nachbarn gelten und dann bestimmt, wie sich eine Zelle abhängig von ihren Nachbarn verändert. Beispielsweise kann eine Zelle mit dem Wert Null zu dem Wert Eins verändert werden, sollten mindestens 5 von 9 benachbarten Zellen mit einer Eins gefüllt sein. Dabei wird abhängig von der gewünschten Ausgabe von einem Anwender festgelegt welche Zellen als Nachbarn gelten und wann diese eine Zelle verändern. Da Zellen in Ecken und an Kanten weniger potenzielle Nachbarn besitzen sind für diese Sonderregeln zu definieren. Dieses Verfahren kann somit Dungeons erstellen, garantiert jedoch keine Verbindung zwischen Elementen und ermöglicht keinen hohen Detailgrad durch das Zufügen komplexer Objekte wie Türen und Items innerhalb des Dungeons (vgl. (Börje Santen, 2015) S.18-25).

2.1.2 Binäre Raumpartitionierung

Die Binäre Raumpartitionierung ist ein Verfahren der prozeduralen Generierung aus dem Jahr 1969 von Robert A. Schumacher (vgl. (Börje Santen, 2015) S.14), welches auf dem Prinzip des binären Aufteilens von Baumdiagrammen basiert. Dabei wird ein Raum in verschieden große Teilräume aufgeteilt. Die Aufteilung geschieht durch Hyperebenen, welche den Raum trennen. Diese Hyperebenen sind im eindimensionalen Raum ein Punkt, im zweidimensionalen Raum eine Gerade und im dreidimensionalen Raum eine Ebene (vgl. (Börje Santen, 2015) S.14). Durch diese Aufteilung entstehen kleinere Teilebenen, welche äquivalent zu einem Binärbaum aufgeteilt werden. Die neu entstandenen Teilebenen werden erneut entlang des Binärbaums aufgeteilt (siehe Abbildung 2.1.1).

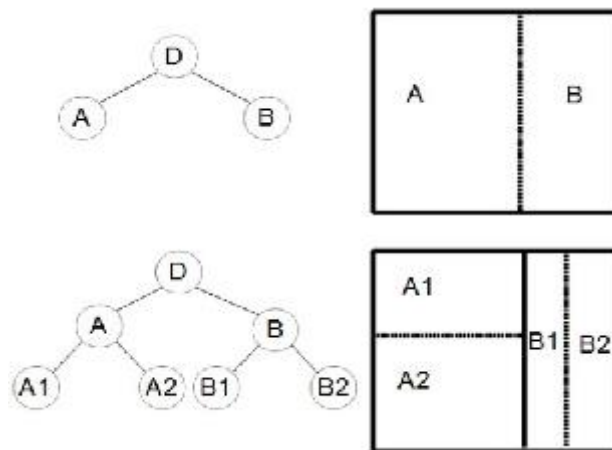


Abbildung 2.1.1: Beispiel zum Aufteilen einer Ebene äquivalent zum Binärbaum, Quelle (Börje Santen, 2015) S.14

Nachdem die Ebene in der gewünschten Anzahl von Durchläufen in mehrere Teilebenen unterteilt wurde, werden die Teilebenen mit Rechtecken variabler Größe gefüllt. Diese Rechtecke stellen die Räume des Dungeons dar und dürfen die Grenzen ihrer jeweiligen Teilebene nicht berühren. Im letzten Schritt werden die Räume nacheinander verbunden, indem der Binärbaum rückwärts abgelaufen wird. Dabei werden die Blätter der jeweiligen Knoten einer Ebene des Baums verbunden, wodurch Gänge zwischen diesen entstehen. Das Verbinden der Blätter aller jeweiligen Knoten einer Ebene des Baumes wird wiederholt, bis der oberste Knoten des Baums erreicht ist. Nach der Ausführung dieses Vorgehens sind alle Räume des Dungeons verbunden (vgl. (Börje Santen, 2015) S.15-17) (siehe Abbildung 2.1.2).

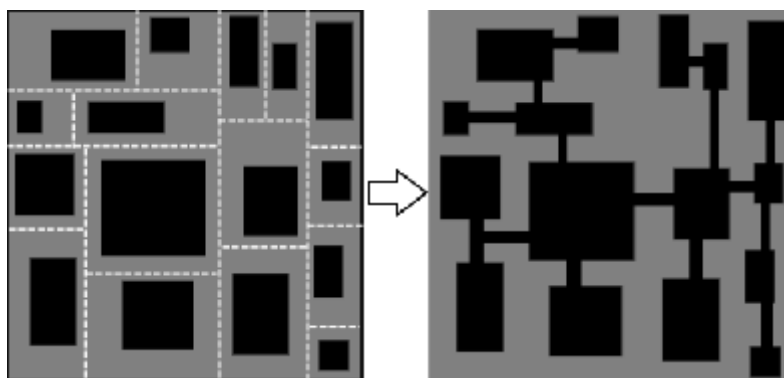


Abbildung 2.1.2: Beispiel des Verbinden von Räumen durch Binäre Raumpartitionierung, Quelle (Börje Santen, 2015) S.16-17

Durch das Verfahren sind Dungeons mit verbundenen Räumen effizient zu realisieren, jedoch platziert dies lediglich Räume innerhalb der Teilebenen und verbindet diese auf direktem Wege. Ein Dungeon mit erhöhten Detailgrad der Räume, sowie komplexeren verschlungenen Gängen ist demnach schwer umsetzbar.

2.2 Wang Tiles

Ein Verfahren der prozeduralen Generierung welches meist zum Erzeugen von Texturen verwendet wird ist das Anwenden von Wang Tiles. Diese werden im Folgenden erklärt, da bestimmte Prinzipien auf den WFC-Algorithmus übertragbar sind.

Wang Tiles sind quadratische Tiles, welche über einen Farbcode ihrer Kanten zusammengesetzt werden. Diese Zusammensetzung geschieht in einer $M \times N$ großen Ebene, in welcher die Tiles von einer Ecke ausgehend Zeile für Zeile eingesetzt werden. Zum Verwenden von Wang Tiles wird ein Tilesset benötigt in welchem für jede Wang Tile eine andere Wang Tile existiert, deren Kanten zusammenpassen. Dabei ist zu beachten das Wang Tiles nicht gedreht werden. Daher muss für jede Tile mit einem Farbcode x an ihrer rechten Kante eine Tile mit demselben Farbcode x an ihrer linken Kante existieren, sodass diese nebeneinander platzierbar sind. Dies gilt ebenfalls für die obere und untere Kante einer Tile (vgl. (Michael F. Cohen, 2003) S.2-3). Ist ein geeignetes Tilesset gewählt kann der Algorithmus Zeile für Zeile über die Ebene iterieren und dabei zufällig eine Wang Tile aus dem Tilesset auswählen, welche an die Kanten ihrer benachbarten, bereits eingefügten Tiles, angelegt werden kann. Demnach entscheidet der Algorithmus welche Tile eingesetzt wird in dem dieser überprüft welche Tiles von den Kanten benachbarter Tiles zugelassen werden.

2.3 Gründe zur Wahl des Wave Function Collapse Algorithmus

Der WFC-Algorithmus ist ein Algorithmus der prozeduralen Generierung, welcher 2016 von Maxim Gumin entwickelt wurde. Dieser Entscheidet über die Einschränkungen an den Kanten von Tiles, welche Tiles aus einem Tilesset innerhalb einer Eben platziert werden. Dadurch weist der WFC-Algorithmus Ähnlichkeiten zum Wang Tile Algorithmus auf, mit welchen ich in der Vergangenheit agierte. Da das Prinzip der Wang Tiles eine komplette Ebene durch Regeln des Platzierens von Tiles, welche die Kanten der Tiles zum Anwenden der Regeln verwenden interessant und effektiv ist, erschien der WFC-Algorithmus als eine lehrreiche Erweiterung. Zudem ist der Wang Tile Algorithmus stark eingeschränkt und zum Generieren von Dungeons ungeeignet, der WFC-Algorithmus erscheint hingegen vielseitiger einsetzbar.

Außerdem ist der WFC-Algorithmus durch sein erscheinen im Jahre 2016 ein neuer Algorithmus der prozeduralen Generierung und damit sicherlich noch nicht vollständig erforscht. Dies bietet Möglichkeiten neue Anpassungen zu entwickeln und den Algorithmus aufgeschlossen zu betrachten. Die theoretischen ungeklärten Möglichkeiten des Algorithmus, zusammen mit vertrauten Ansätzen, machen ein auseinandersetzen mit diesem daher erstrebenswert.

Zudem ist das Feld der prozeduralen Generierung zwar bereits viel erkundet, jedoch sind die meisten Algorithmen nur für bestimmte Anwendungen geeignet. Der WFC-Algorithmus stellt daher die Frage seiner möglichen Anwendungsgebiete.

3 Erklärung des Wave Function Collapse Algorithmus

Der WFC-Algorithmus ist ein einschränkungsbasierter Algorithmus, welcher 2016 von Maxim Gumin entwickelt wurde. Dieser kann mit einer kleinen Anzahl an Eingabeobjekten, welche aus Tiles bestehen, unter Einhalten von Einschränkungen, eine nahezu beliebig große Ausgabe erzeugen. Dabei gibt es einen texturbasierten, auch Overlapping Model genannten, und einen tilebasierten, auch Simple Tiled Model genannten Ansatz. Diese Arbeit verwendet in ihrer Ausführung den tilebasierten Ansatz.

3.1 Wie funktioniert der Algorithmus

Bei dem texturbasierten Ansatz wird original nach Maxim Gumin's Beitrag eine Bitmap erzeugt, welche nur Muster vorweist, welche in der Eingabebitmap vorhanden sind. Bei diesem texturbasierten Ansatz werden gleich große Rechtecke erstellt, wobei die Größe dieser Rechtecke $M \times N$ Pixel groß ist und diese aus allen Teilrechtecken der Eingabebitmap erzeugt werden. Der Wert für N und M wird zu Anfang festgelegt. Bei einem Wert von $N = M = 3$ ist das festgelegte Rechtecke 3×3 Pixel groß. Nun wandert der Algorithmus Zeile für Zeile von einer Ecke der Eingabebitmap aus in einen Pixel großen Schritten über die gesamte Eingabebitmap, und erzeugt dabei jeweils $M \times N$ große Rechtecke mit den jeweiligen Pixeln. Dadurch ergibt sich eine Menge von 3×3 Pixel großen Tiles, welche gespeichert werden, um mit diesen durch den WFC-Algorithmus eine Ausgabebitmap zu erzeugen. Für jede erstellte Tile wird ebenfalls bestimmt welche Tiles benachbart zu dieser sein dürfen. Die Ausgabe besteht nur aus den zuvor erzeugten $M \times N$ großen Rechtecken (siehe Abbildung 3.1.1) (vgl. (Maxim Gumin, 2023)).



Abbildung 3.1.1: Beispiel texturbasierter WFC mit $N=3$, Quelle (Maxim Gumin, 2023)

Durch das Erstellen der $M \times N$ großen Rechtecke wurden die zu verwendenden Tiles erzeugt. Ob Tiles benachbart sein dürfen und zusammengesetzt werden können wird über ihre jeweiligen Kanten entschieden. Dies geschieht indem nur bestimmte Tiles an den jeweiligen Kanten als Nachbar erlaubt sind und somit die Nachbarn an den Kanten eingeschränkt werden. Demnach weist der WFC-Algorithmus Ähnlichkeiten zu anderen prozeduralen Algorithmen wie dem Wang-Tile-Algorithmus auf, da beide über die Kanten zweier Tiles bestimmen, ob diese zusammengefügt werden können (vgl. (Maxim Gumin, 2023)).

Bei einem tilebasierten Ansatz werden die benötigten Tiles zuvor manuell erstellt sowie für deren Kanten manuell eingetragen welche Tiles angrenzen können. Da sowohl das Overlapping Model, als auch das Simple Tiled Model letztlich mit Tiles und deren Kanten agieren,

wobei das Overlapping Model lediglich diese Tiles aus einem Beispiel erzeugt und das Simple Tiled Model vom Benutzer erstellte Tiles verlangt, wird im restlichen Verlauf dieser Arbeit von dem Simple Tiled Model ausgegangen (vgl. (Yuhe NIE, 2023) S.2-3).

3.1.1 Wählen geeigneter Tiles

Beim WFC-Algorithmus ist es wichtig eine Tilemap zu wählen, bei der mit den vorgegebenen Tiles ein größeres Konstrukt gebildet werden kann. Dazu müssen die Kanten verschiedener Tiles aneinanderpassen. Es ist jedoch nicht nötig, dass jede Tile an jede andere angrenzen kann, da der Algorithmus auch mit starken Einschränkungen meist Ausgaben erzeugen kann. Für jede Tile die verwendet werden soll muss jedoch mindestens eine andere Tile existieren, mit der es übereinstimmende Kanten gibt. Dadurch können auch Tiles mit nur wenigen potenziellen Nachbarn häufig im Output vorkommen.

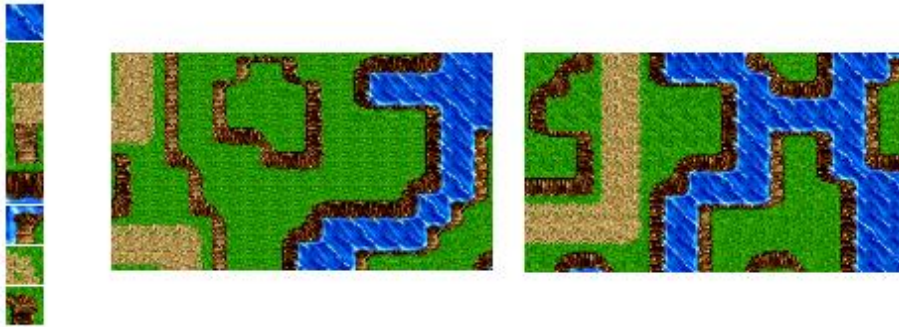


Abbildung 3.1.2: Beispiel erzeugen aus beschränkten Eingabetiles, Quelle (Maxim Gumin, 2023)

3.1.2 Ablauf des Algorithmus

Der WFC-Algorithmus besteht nach dem Erstellen der Tiles aus fünf Schritten.

1. Initialisieren einer Liste aller Eingabetiles
2. Erstellen eines Arrays mit den Dimensionen der Ausgabe
3. Füllen aller Einträge des Arrays jeweils mit der Liste aller möglichen Eingabetils
4. Finden eines Eintrags des Arrays mit der niedrigsten Entropy über eins und diesen kollabieren. Update alle übrigen Einträge durch diese Veränderung
5. Wiederhole Schritt 4. bis alle Einträge kollabiert sind und der Algorithmus beendet oder in einem nicht lösbaren Zustand ist (vgl. (Maxim Gumin, 2023))

Im ersten Schritt werden alle Eingabetiles in einer Liste gespeichert. Dabei werden falls nötig vorhandene Tiles gedreht, um Eingabetiles in jeder nötigen Ausrichtung zu erhalten. Jede Tile speichert dabei welche Tiles an ihren jeweiligen Kanten zulässig ist.

Der zweite Schritt erzeugt die Ebene oder den Raum, in welchem die Ausgabe gespeichert wird. Bei einer 2-dimensionalen Ausgabe wird demnach das Array als eine Ebene definiert, welches durch eine X und Y Länge festgelegt wird.

Im dritten Schritt werden alle Elemente dieser 2-dimensionalen Ebene jeweils mit der Liste aller Eingabetiles aus Schritt eins gefüllt. Nun enthält jeder Eintrag der Ebene alle möglichen Tiles die in diesen Eintrag eingefügt werden können. Da zu Anfang des Verfahrens jedes Element leer ist, kann daher jedes Element mit jeder Tile gefüllt werden. Demnach ist zum Erstellen einer 2-dimensionalen Ebene dieser Schritt vorstellbar wie ein 3-dimensionales Objekt, bei welchem über der Ebene alle möglichen Tiles in der Z-Achse über den Einträgen der Ebene liegen. Diese Menge der Einträge wird nun im nächsten Schritt für jeden Eintrag der Ebene nacheinander auf eine Tile reduziert. Die Menge aller Einträge eines Elements des Arrays stellt demnach alle für diesen Eintrag möglichen Tiles dar, die Menge aller möglichen Tiles

wird Entropy genannt, und diese verändert sich für viele Einträge stetig im Verlauf des Algorithmus. Welche Tiles in einem Eintrag eingesetzt werden können hängt von den Tiles seiner Nachbarn ab. Wenn die Liste eines Eintrags die Tiles A, B und C enthält, dann müssen alle Nachbarn in ihrer Liste mindestens eine Tile enthalten, welche an der jeweiligen Kante die Tiles A, B und C zulassen. Sobald beispielsweise der rechte benachbarte Eintrag keine Tile mehr enthält an dessen linker Kante Tile B sein kann, müsst Tile B aus der Liste aller möglichen Einträge entfernt werden.

Schritt vier wendet nun den eigentlichen WFC an. Dieser Schritt wird so lange wiederholt, bis alle Einträge nur noch ein Element und damit eine Entropy von eins besitzen, da eine Entropy von eins bedeutet, dass dieser Eintrag bereits ein finales Ergebnis besitzt. Zu Anfang dieses Schrittes wird überprüft welche Einträge des Arrays die niedrigste Entropy über eins besitzen. Von diesen Einträgen des Arrays mit der niedrigsten Entropy über eins wird zufällig ein Eintrag ausgewählt und dann zufällig eine Tile aus der Liste aller Tiles dieses Eintrags selektiert. Alle Tiles außer die Ausgewählte werden aus der Liste entfernt, sodass nun absolut eine Tile in diesem Eintrag gespeichert wird. In diesem Moment wird bezeichnet, dass die Wave Function kollabiert „eng. collapsed“ ist. Nun müssen alle Nachbarn des kollabierten Eintrags überprüft werden, da es möglich ist, dass die Listen bestimmter Nachbarn Tiles enthalten, welche nicht mehr an diesen Koordinaten eingefügt werden können, da die Kanten der kollabierten Tile eventuell nicht alle Nachbarn zulässt, die zuvor von der Menge der Liste zugelassen worden. Sollten Änderungen an benachbarten Einträgen vorgenommen werden, müssen deren Nachbarn ebenfalls aus denselben Gründen überprüft werden. Dies zieht sich fort bis keine Veränderungen an Nachbarn mehr entstehen und wird constraint propagation genannt, da die Informationen von einem Eintrag zum nächsten aufgrund von Einschränkungen verbreitet werden (vgl. (Maxim Gumin, 2023)).

Als fünfter Schritt wird der vierte Schritt nun wiederholt, bis alle Einträge kollabiert sind oder ein unlösbarer Konflikt entsteht. Wenn alle Einträge eine Entropy von eins besitzen, ist die Ausgabe fertig.

Dieses Vorgehen, und vor allem Schritt vier können vereinfacht an einem Beispiel des bekannten Rätselspiels Sudoku dargestellt werden. Bei Sudoku gibt es vorgegebene Zahlen, die beschränken welche Zahlen in den übrigen Kästen möglich sind. Solange wie ein Kasten nicht bestimmt ist, ist eine Menge an Zahlen in diesem Kasten möglich. Die Menge aller Zahlen eines Kastens wird dabei von den schon eingesetzten Zahlen beeinflusst. Zunächst werden demnach alle möglichen Zahlen in einen Kasten vermerkt, und diese dann durch ein Betrachten der bereits eingesetzten Zahlen eingeschränkt. Bei jedem entfernen von Zahlen aus einer Menge möglicher Zahlen werden dadurch auch benachbarte Mengen eingeschränkt, es findet demnach constraint propagation statt (siehe Abbildung 3.1.3). Nun wird wie beim WFC-Algorithmus aus dem Kasten mit den wenigsten möglichen Zahlen eine Zahl zufällig ausgewählt. Durch das Auswählen dieser Zahl verändert sich erneut welche Zahlen in den übrigen Kästen möglich sind. Dies wird so lange fortgeführt, bis das Sudoku gelöst ist (vgl. (Boris, Wave Function Collapse Explained, 2023)).

12 34	4	12 34	12 34
12 34	2	3	12 34
12 34	12 34	12 34	3
4	3	12 34	2

12 3	4	12 3	12 3
12 34	2	3	12 34
12 34	12 34	12 34	3
4	3	12 34	2

1 3	4	12 1	
1	2	3	1 4
12 1		1 4	3
4	3	1	2

1 3	4	12	1
1	2	3	1 4
12	1	1 4	3
4	3	1	2

Abbildung 3.1.3: Beispiel constraint propagation anhand Sudoku, Quelle (Boris, Wave Function Collapse Explained, 2023)

3.2 Vor und Nachteile des Algorithmus

3.2.1 Vorteile

Ein erster Vorteil des WFC-Algorithmus ist, dass dieser sehr vielseitig einsetzbar ist und somit prozedurale Generierungen in vielen Anwendungsbereichen möglich macht. Prozedurale Generierungen sparen Zeit und Kosten und sind daher ein überlegenswertes Vorgehen.

Außerdem ist der WFC-Algorithmus sehr zuverlässig, wodurch es nur selten zu Fällen kommt, in denen der Algorithmus es nicht schafft, eine erfolgreiche Ausführung abzuschließen (vgl. (Maxim Gumin, 2023)).

Der WFC-Algorithmus kann aufgrund seines Verfahrens des kollabieren des Feldes mit der niedrigsten Entropy und der constraint propagation ebenfalls Ausgaben vervollständigen, in denen zunächst bestimmte Tiles in Einträgen manuell festgelegt wurden. Dadurch kann ein Anwender bestimmte Tiles oder Muster vorgeben und der Algorithmus vervollständigt dann den Restlichen Output. Somit kann der WFC-Algorithmus zum Vervollständigen von Texturen, Game Leveln und mehr verwendet werden (vgl. (Maxim Gumin, 2023)) (siehe Abbildung 3.2.1).

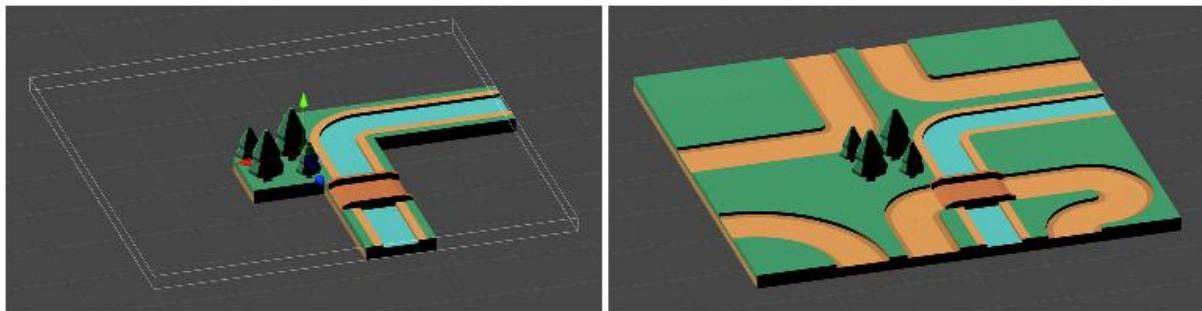


Abbildung 3.2.1: Beispiel WFC-Algorithmus mit vorgegebenen Tiles, Quelle (Boris, Wave Function Collapse tips and tricks, 2023)

Des Weiteren bietet der WFC-Algorithmus viele Anpassungsmöglichkeiten, an denen seit dessen Erscheinen in 2016 weitgehend geforscht wird. Dabei wurden viele Methoden vorgeschlagen, um den WFC-Algorithmus zu optimieren. Beispielsweise wurden Ansätze wie Backtracking oder Graphen-basiertes WFC, das das Navigieren in 3-dimensionalen Generationen erlaubt, vorgeschlagen (vgl. (Yuhe NIE, 2023) S.2).

3.2.2 Nachteile

Ein erster Nachteil des WFC-Algorithmus entsteht daraus, dass dieser ein Algorithmus zur prozeduralen Generierung ist. Dadurch dass dieser Algorithmus zufällig entscheidet, welche Tiles kollabiert werden, entstehen häufig unrealistische Ausgaben, die für viele Anwendungen ungeeignet sind. Daher muss der WFC-Algorithmus häufig auf seine jeweilige Anwendung angepasst werden. Dabei ist die jeweilige Anwendung mit ihren zu erfüllenden Zielen zu beachten. Der WFC-Algorithmus kann eine Stadt generieren, diese sieht jedoch meist unrealistisch aus, da Gebäude und Straßen zufällig platziert werden, ohne diese sinnvoll zu verbinden (siehe Abbildung 3.2.2). Außerdem kann der Algorithmus scheitern, und somit kein Ergebnis liefern, auch wenn dies sehr selten geschieht.



Abbildung 3.2.2: Beispiel einer erzeugten Stadt ohne sinnvolle Verbindungen von Straßen, Quelle (Robert Heaton, 2023)

Des Weiteren definierte Maxim Gumin bei seiner Vorstellung des WFC-Algorithmus die Frage, ob eine Bitmap für sein Overlapping Model Ausgaben erzeugt, die nur solche $M \times N$ Muster enthält, welche in der Eingabe Bitmap enthalten sind als NP-hartes Problem. Dies macht es unmöglich eine schnelle Lösung zu finden die immer vollständig ausgeführt werden kann. Dadurch ist es für den Anwender schwer zu wissen, ob eine Menge an Eingabetiles immer erfolgreich den WFC-Algorithmus abschließt oder scheitern kann (vgl. (Maxim Gumin, 2023)).

Ein großes Problem in der praktischen Anwendung des Algorithmus kann dessen Laufzeit darstellen. Der Algorithmus besitzt zwei Eingabegrößen, einmal die Menge aller Tiles eines Tilesets T , und die Menge aller Einträge welche durch $M \times N$ bestimmt wird. Die Menge aller Tiles des Tilesets T wird in folge $|T| = d$ genannt. Die Menge aller Einträge des Arrays wird bestimmt aus $M \times N = n$. Da jeder Eintrag kollabiert werden muss läuft der Algorithmus immer über alle Einträge n . Beim Kollabieren eines Eintrages müssen alle benachbarten Einträge aktualisiert werden. Diese Aktualisierung kann im schlimmsten Fall über alle anderen Einträge stattfinden, wodurch für jeden Eintrag über alle anderen Einträge iteriert werden müsse und somit $O(n * n) = O(n^2)$. Bei diesem aktualisieren muss jede Tile der Liste eines Eintrags betrachtet werden, um zu entscheiden, ob diese erhalten bleiben darf. Im schlimmsten Fall sind alle möglichen Tiles d in einem Eintrag enthalten. Dadurch müsse beim Aktualisieren jedes Eintrags über jede Tile iteriert werden und somit $O(n^2 * d)$. Da für jede erlaubte Tile eines Eintrags jede vorhandene Tile eines anderen Eintrags überprüft werden muss und im schlimmsten Fall beide Einträge das vollständige Tilesset d darstellen können, wäre der Algorithmus im schlimmsten Fall in $O(n^2 * d^2)$. Dieses Vergleichen von Tilesets kann jedoch für einen Eintrag und somit ein Tilesset im schlimmsten Fall über alle Einträge n durch constraint propagation iteriert werden, wodurch der Algorithmus unter dieser Betrachtung als maximal $O(d^n)$ angesehen werden kann.

Die Arbeit von Yuhe NIE definiert den WFC-Algorithmus als einen einschränkungserfüllenden Algorithmus welcher binäre Kantenprobleme löst, backtracking nutzt, um akzeptable Lösungen zu generieren und AC-3 für Optimierungen verwendet. Unter diesen Rahmenbedingungen definieren diese den WFC-Algorithmus als $O(d^n + n^2 * d^3)$ (vgl. (Yuhe NIE, 2023) S.3).

Die wichtigste Unterscheidung zwischen der oben hergeleiteten Komplexität und der von (Yuhe NIE, 2023) ist die exponentielle Steigerung. Diese lässt sich wie zuvor angedeutet dadurch erklären, dass für jede Tile eines Eintrags über jede Tile eines benachbarten Eintrags iteriert wird, und diese Iteration im schlimmsten Fall über alle Einträge ausgeführt wird, ausgehend von dem Tileset eines Eintrags. Dadurch muss das Tileset d nicht nur einmal mit sich selbst multipliziert werden, wofür d^2 steht, sondern im schlimmsten Fall für jeden Eintrag mit einem anderen Tileset verglichen werden, wodurch die Komplexität im schlimmsten Fall bei $O(d^n)$ liegt. Dadurch kann die Laufzeit des Algorithmus in bestimmten Anwendungen zu Problemen führen, vor allem wenn der schlimmstmögliche Fall eintritt.

Die Laufzeit bei großen Ausgabeflächen $F(X \times Y)$ mit hohen Werten für X und Y kann verbessert werden, indem die Ebene in n viele Ausgabeflächen aufgeteilt wird. Dazu wird die gesamte Ebene welche sich aus $X \times Y$ ergibt in n viele gleichgroße Teilebenen unterteilt, welche ein Netz bilden. Diese Teilebenen überlappen an ihren Kanten mit der jeweils benachbarten Teilebene, sodass diese an ihren Kanten zueinander die gleichen Tiles enthalten. Dazu werden die Teilebenen im Netz nacheinander von einer Ecke ausgehend diagonal durch den WFC-Algorithmus ausgefüllt. Dadurch kann die nächste Teilebene an Kanten mit einer benachbarten Teilebene die Kante mit den Tiles ihres anliegenden Nachbars ausfüllen. Nachdem alle Teilebenen durch den WFC-Algorithmus ausgefüllt sind, überlappen die Kanten jeder Teilebene mit ihren Nachbarn, da diese an den Kanten zueinander die gleichen Tiles besitzen. Die Teilebenen werden zuletzt zusammengefügt, indem die überlappenden Tiles übereinandergelegt werden, sodass alle Teilebenen zusammenpassen. Dadurch wird eine Ebene $E(X \times Y)$ mit großen Werten für X und Y in mehrere kleinere Ebenen $E_{t_1 \text{ bis } n}(A \times B)$, mit kleineren Werten für A und B unterteilt (vgl. (Yuhe NIE, 2023) S.6-7)(siehe Abbildung 3.2.3). Dieses Verfahren bietet jedoch nur bei Ebenen $E(X \times Y)$ mit großen Werten für X und Y eine wesentliche Verbesserung der Laufzeit. Daher ist ein Aufteilen kleiner Ebenen nicht ratsam.

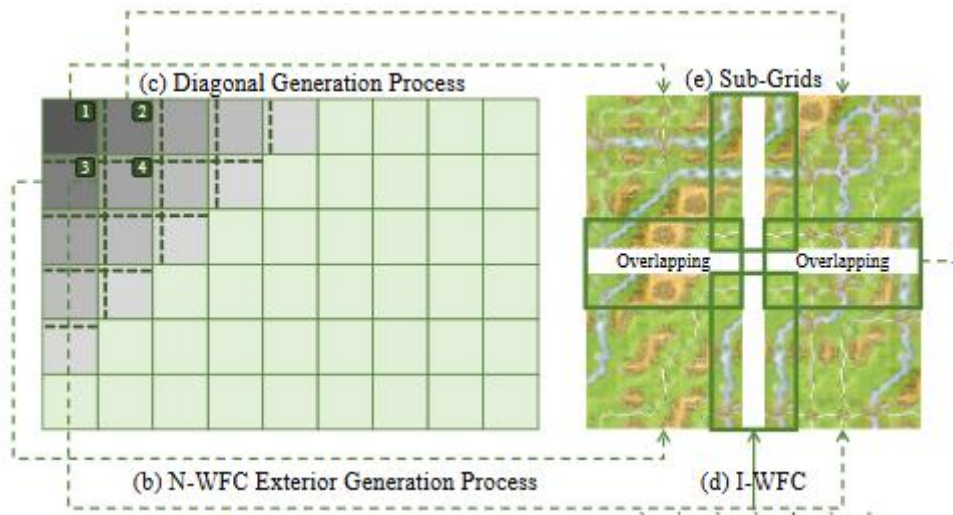


Abbildung 3.2.3: Veranschaulichung des Aufteilens einer Ebene in Teilebenen, Quelle (Yuhe NIE, 2023)

4 Theoretisches und praktisches Vorgehen

In Kapitel 3 wurde erklärt, wie der WFC-Algorithmus funktioniert, sowie manche der Vorteile und Nachteile dessen. Um einen Dungeon mit Hilfe des WFC-Algorithmus zu erstellen werden nun bestimmte Anpassungen vorgenommen und erklärt.

4.1 Vorgehen zum prozeduralen erzeugen eines Dungeons

Um einen Dungeon zu erstellen werden die einzelnen Punkte aus 3.1.2 praktisch umgesetzt. Dazu müssen Datenstrukturen erstellt werden, welche die wichtigen Daten aller Tiles enthalten, wie deren Typ und dessen erlaubte Nachbarn an ihren jeweiligen Kanten. Die Menge dieser Strukturen wird dann jeweils in jeden Eintrag eines Arrays eingefügt, welches die $E(XxY)$ Ebene der Ausgabe darstellt. Auf diesem Array, welches in jedem Element jede dort mögliche Tile speichert, wird nun jedes Element des Arrays kollabiert, und die daraus entstandenen Änderungen angepasst, bis der Algorithmus beendet wird, wie in Kapitel 3.1.2 erklärt. Ein solcher durch WFC erzeugter Dungeon wäre jedoch komplett zufällig generiert, und könnte daher keine sinnvoll passierbaren Wege garantieren. Zudem würden viele einzelne Pfade generiert werden, die nicht miteinander verbunden sind.

Die Dungeons die für diese Arbeit generiert werden sollen bestimmte Anforderungen erfüllen, welche den WFC-Algorithmus sinnvoll erweitern. Diese Anforderungen sollen einen realistischeren und für Anwendungen geeigneteren Dungeon erzeugen. Diese Anforderungen sind im Folgenden erklärt.

1. Der Dungeon soll aus verschiedenen Elementen von Wänden, Räumen und Gängen erzeugt werden.
2. Es soll möglich sein den erzeugten Dungeon von einem Startpunkt zu einem Endpunkt durch einen Hauptpfad zu durchqueren.
3. Es sollen verschiedene Objekte innerhalb des Dungeons bei der Generierung des Dungeons miterzeugt werden. Objekte wie Gegenstände und Gegnertypen.
4. Es soll versucht werden Gewichtungen für die verschiedenen Typen von Dungeon Elementen sowie deren Inhaltsobjekten zu verwirklichen.
5. Es wird nach einer Möglichkeit gesucht den WFC-Algorithmus in einer Weise anzupassen, dass dieser trotz seiner zufälligen Generierung keine Gänge außerhalb des Hauptpfades generiert, welche nicht erreichbar sind.

Die weiteren Unterkapitel beschäftigen sich damit ob diese Anpassungen möglich sind, und wie diese im jeweiligen Fall umgesetzt werden.

4.2 Umsetzungen eines realistischen Dungeon durch WFC und Probleme

Um einen realistischen Dungeon zu erzeugen, müssen bestimmte bereits genannte Anforderungen eingehalten werden, welche in diesem Kapitel erläutert werden. In allen weiteren Kapiteln wird angenommen, dass zum Erstellen eines Dungeons nur Tiles verwendet werden die einen Gang, einen Raum oder eine Wand darstellen können (siehe Abbildung 4.2.1). Durch diese Typen von Tiles können vollständige Dungeons erstellt werden (siehe Abbildung 4.2.2).



Abbildung 4.2.1: Beispiel von Tiles zum Bilden eines Ganges, Raums oder Wand



Abbildung 4.2.2: Beispiel des Erstellens eines Dungeons mit Gang, Raum und Wand Tiles

4.2.1 Erstellen eines Dungeons aus verschiedenen Elementen

Einen Dungeon zu erzeugen, welcher aus verschiedenen Elementen wie Räumen, Gängen und Wänden besteht ist mit dem WFC-Algorithmus ohne Anpassungen durchführbar, da der WFC-Algorithmus alle Eingabetiles abhängig von ihren Einschränkungen effektiv platziert. Um einen Dungeon zu erstellen, welcher aus verschiedenen Elementen besteht, werden die nötigen Tiles erstellt und dabei die passenden Einschränkungen gewählt. Es ist dabei lediglich wichtig die Elemente richtig zu wählen und richtig zu bezeichnen. Bei richtigem Bezeichnen der Tiles und der Zulässigkeit ihrer Kanten kann der WFC-Algorithmus Dungeons aus solchen Elementen erstellen. Solange die Kanten von verwendeten Tiles zusammenpassen können somit verschiedene Tile Typen in einem Dungeon verwendet werden (siehe Abbildung 4.2.3).

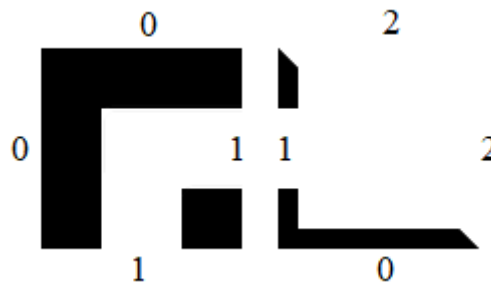


Abbildung 4.2.3: Beispiel 2 verschiedener Tiles die zusammenpassen. 0 steht für Wände, 1 für Gänge und 2 für Räume

4.2.2 Erstellen eines verbundenen Hauptpfades

Das Erstellen eines verbundenen Hauptpfades ist ohne Anpassungen nicht möglich, da der WFC-Algorithmus sämtliche Tiles zufällig auswählt sowie immer die Tile mit der niedrigsten Entropy kollabiert. Selbst wenn ein Startpunkt und ein Endpunkt festgelegt werden, und diese Tiles beispielsweise vom Typ eines Ganges sein müssen, ist nicht garantiert, dass diese verbunden werden. Dies liegt daran, dass der WFC-Algorithmus häufig viele kleine, nicht miteinander verbundene Pfade erstellt. Durch das Kollabieren der Tile mit der niedrigsten Entropy über eins werden häufig Tiles kollabiert, welche nicht an einem bereits bestehenden Pfad liegen. Durch das zufällige Auswählen der Tiles werden solche neuen Pfade gelegentlich durch die Auswahl von Endpunkten geschlossen, bevor diese einen anderen Pfad erreichen. Dadurch wird ein Pfad, der den Start oder Endpunkt durchquert, häufig beendet, bevor der jeweils andere Punkt erreicht wird. Eine entwickelte Lösung wird in Kapitel 4.4.1 besprochen.

4.2.3 Erstellen von weiteren Objekten wie Gegenständen oder Gegnern

Dies wird ohne Anpassungen am WFC-Algorithmus umgesetzt. Dazu werden beim vorherigen Festlegen der Tiles ebenfalls Tiles erzeugt welche Elemente wie Gegner oder Gegenstände enthalten (siehe Abbildung 4.2.4). Solche Tiles können die gleichen Einschränkungen ihrer jeweiligen Kanten besitzen, wodurch diese in ihrer relativen Häufigkeit so häufig auftreten würden wie eine äquivalente Tile ohne Objekt. Alternative können diese Tiles andere Einschränkungen ihrer Kanten besitzen, sodass beispielsweise eine Tile mit Objekten nur neben einer Tile ohne Objekte erzeugt werden kann. Als andere Möglichkeit der Einschränkung kann nur für bestimmte Tiles eine äquivalente Tile mit Objekten erstellt werden, sodass die gesamte Anzahl normaler Tiles größer ist, und somit Tiles mit Objekten seltener erstellt werden. Durch ein intelligentes Wählen der Eingabetiles kann der einschränkungsbasierte WFC-Algorithmus demnach Objekte ohne Anpassungen einfügen.



Abbildung 4.2.4: Beispiel Tiles zum Einfügen von Objekten

4.2.4 Erstellen von Gewichtungen für Elemente und Objekte

Der WFC-Algorithmus ist in seiner Grundlegenden Form nicht auf Gewichtungen ausgelegt, kann durch diese jedoch ergänzt werden, ohne den Algorithmus in seiner grundlegenden Funktionsweise zu verändern oder zu korrigieren. Der WFC-Algorithmus kann, wie in Kapitel 4.2.1 beschrieben, verschiedene Typen von Elementen für seine Ausgabe verwenden, wenn diese als Teil der Eingabetils erstellt werden. Dasselbe gilt für Objekte, wie in Kapitel 4.2.3 beschrieben. Dadurch sind alle Elemente und Objekte ein Teil der Eingabetils, welche als Menge aller Tiles $|T| = d$ in jedes Element des Arrays eingefügt werden. Da beim Kollabieren eines Elements des Arrays eine Tile zufällig aus der Restmenge $r \subseteq d$ ausgewählt wird, kann diese zufällige Auswahl durch Gewichte ergänzt werden. Dabei wird die Wahrscheinlichkeit eine der übrigen Tiles auszuwählen durch die Gewichte ihres jeweiligen Typs, sowie Objektes angepasst.

4.2.5 Erstellen eines Dungeons indem alle Pfade verbunden sind

Da der WFC-Algorithmus zufällig auswählt welche Tile kollabiert wird, sowie welche Tile ausgewählt wird, kann der WFC-Algorithmus, wie bereits in Kapitel 4.2.2 erwähnt, ohne Anpassungen nicht garantieren, dass alle Pfade verbunden sind. Daher müssen Anpassungen am WFC-Algorithmus vorgenommen werden. Diese werden in den weiteren Kapiteln besprochen.

4.2.6 Erstellen eines Dungeon als Pfadproblem

Ein grundlegendes Problem des WFC-Algorithmus zur Generierung eines Dungeons besteht darin, dass dieser durch seine zufällige Auswahl von Tiles meist nicht verbundene Pfade erzeugt. Ein funktionierender realistischer Dungeon muss daher den Algorithmus auf eine Weise anpassen, welche garantiert, dass alle Pfade des Dungeons verbunden sind und ist demnach ein Pfadproblem. Das Verwenden des WFC-Algorithmus im Kontext eines Pfadproblemen kann in vielen Anwendungen angetroffen werden. Dem Erstellen von Raumverteilungen zur Generierung von Häusern für die Architektur, dem Pfad in einem Dungeon oder lediglich dem Erzeugen realistischer Wege in einer erzeugten Landschaft. Da ein Dungeon nach den Anforderungen dieser Arbeit Räume und Gänge enthalten und realistisch verbinden muss, sollten gefundene Prinzipien daher auf andere Anwendungen übertragbar sein.

4.3 Möglichkeiten zum Lösen dieses Pfadproblems

Diese Arbeit versucht den WFC-Algorithmus zum Generieren eines Dungeons anzupassen, daher wurde versucht möglichst wenige Änderungen am Algorithmus vorzunehmen, welche seine funktionelle Arbeitsweise beschränken, oder zu spezifisch und somit schwer übertragbar sind. Dazu wurden zunächst einige Funktionen festgelegt, welche den Algorithmus in seiner Vorgehensweise ausmachen und versucht diese möglichst nicht zu beschränken.

- Der Algorithmus erstellt eine zufällige Ausgabe abhängig von einer beschränkten Eingabe, welche jedoch Sinn ergibt, abhängig von den gewählten Einschränkungen.
- Der Algorithmus kollabiert zufällig ein Element mit der niedrigsten Entropy über eins, wodurch eine hohe Erfolgsquote entsteht.
- Die endgültigen Tiles aller Elemente werden vom Algorithmus zufällig aus einer Restmenge aller möglichen Tiles ausgewählt.
- Nachdem alle Elemente kollabiert sind ist eine fertige funktionsfähige Ausgabe entstanden.
- Der Algorithmus verwendet constraint propagation.

Der WFC-Algorithmus erzeugt zufällige Ausgaben, welche jedoch keine lokalen Garantien besitzen, wie die Verbindung von erzeugten Pfaden. Dieser Umstand entsteht direkt aus der Funktion des Algorithmus zufällig innerhalb der Ebene ein Element zu kollabieren, und dessen Tile zufällig auszuwählen.

Zum Lösen des Pfadproblems kann ein einschränkungsbasierter Ansatz verwendet werden, welche die Eingabetils selbst, oder deren Auswahl einschränkt. Außerdem sind Methoden des vorigen und nachträglichen Bearbeitens (pre- and postprocessing) möglich. Beispiele zum Anpassen des WFC-Algorithmus für ein Pfadfindungsproblem bei einem texturbasierten Ansatz werden in der Arbeit von (Hugo Scurtie, 2018) aufgezeigt. Als Methoden der nachträglichen Bearbeitung wird beispielsweise Path Filtering oder Path Smoothing vorgestellt. Path Filtering überprüft die Ausgabe und entfernt dabei kleinere, nicht für eine Anwendung ideale Pfade, indem kleine Pfade erkannt und entfernt werden (vgl. (Hugo Scurtie, 2018) S.3). Ein solches Vorgehen der nachträglichen Bearbeitung ist funktionell, muss jedoch meist abhängig vom Kontext, in dem der WFC-Algorithmus verwendet wird, individuell angepasst werden und erfüllt somit bestimmte, der zuvor in diesem Kapitel definierten, Funktionen nicht.

Als weiterer Ansatz kann der WFC-Algorithmus durch weitere Einschränkungen das Pfadproblem lösen. Dabei werden Einschränkungen gewählt die ungewünschte zufällige Generierungen eindämmen oder ganz verhindern. Bei einem texturbasierten Ansatz ist eine mögliche Lösung nach der Arbeit von (Hugo Scurtie, 2018) das Einführen eines „stretch space“, welcher das Generieren von Pfaden und Wänden einschränkt, indem dieser ein Gebiet zwischen Pfaden und Hindernissen darstellt. Die Wahl geeigneter Einschränkungen kann somit ein Entstehen funktionierender Pfade wahrscheinlicher gestalten, indem Wände nur um einen stretch space entstehen (siehe Abbildung 4.3.1)(vgl. (Hugo Scurtie, 2018) S.2-5). Diese Methode ist, abhängig vom verwendeten Tileset einer Anwendung, nicht direkt auf jeden tilebasierten Ansatz übertragbar, zeigt jedoch, dass ein einschränken des WFC-Algorithmus das Pfadproblem angehen kann.

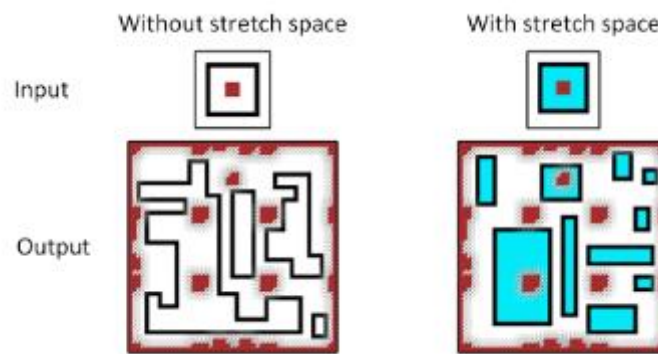


Abbildung 4.3.1: Beispiel erzeugen eines Levels ohne und mit Stretch Space, Quelle (Hugo Scurtie, 2018)

Solche Ansätze, die das Pfadproblem lösen, erfüllen jedoch nie alle in diesem Kapitel definierten Funktionen. Es konnten im Zuge dieser Arbeit für einen tilebasierten Ansatz des WFC-Algorithmus keine Ansätze gefunden oder entwickelt werden, die stets alle in diesem Kapitel definierten Funktionen erhalten. Gefundene Lösungen dieser Arbeit verwenden stets das für diese Arbeit in Kapitel 4.2 vorgestellte Tilesset. Demnach stellen Lösungen einen Fokus auf das Verbinden von Gängen und Räumen, die durch den WFC-Algorithmus erzeugt werden, sodass ein verbundenes Netz entsteht.

4.4 Gefundene und Entwickelte Anpassungen

Alle im folgenden vorgestellten Anpassungen wurden als Lösungen zum umsetzen eines Dungeons unter Verwendung des WFC-Algorithmus selbst entwickelt und zur Veranschaulichung in einem Beispielsprogramm umgesetzt. Die Umsetzung betrachtet dabei die Ränder des Arrays, in welchem der Dungeon generiert wird, als Wände. Bei deren Entwicklung wurde versucht alle Funktionen aus Kapitel 4.3 zu wahren, oder dargestellt weshalb diese nicht vollständig einhaltbar sind. Zudem wurde durch diese Anpassungen versucht unter Verwendung des WFC-Algorithmus alle Anforderungen die in Kapitel 4.1 erwähnt wurden umzusetzen, indem die jeweiligen Anforderungen die nach Kapitel 4.2 noch ausstanden ergänzt werden. Dies betrifft das Garantieren eines verbundenen Hauptpfades, sowie ein Verbinden aller generierten Pfade. Eine Verbindung aller generierten Pfade wäre immer gewährleistet, wenn keine Wandtiles verwendet werden, da in diesem Fall nur Pfade erzeugt werden. Daher gehen sämtliche folgende Ansätze von einer Verwendung aller drei Typen von Tiles aus.

4.4.1 Festlegen eines verbundenen Hauptpfades durch preprocessing

Dieser Ansatz behandelt das Problem, dass ein verbundener Hauptpfad durch einen unangepassten WFC-Algorithmus nicht gesichert ist. Zu dessen Lösung wird ein Ansatz des preprocessing verwendet, welcher im Zuge dieser Arbeit entwickelt und umgesetzt wurde. Zunächst wird auf einem Array, welches äquivalent zum Array der Ebene der Ausgabe ist, ein Start- und Endpunkt zufällig bestimmt. Zwischen diesen Punkten wird ein verbundener Pfad ermittelt. Um einen realistischen Dungeon zu erstellen, wird dabei kein Pfadalgorithmus wie A* verwendet, da solche den kürzesten Weg ermitteln. Der Dungeon sollte keinen direkten, sondern einen möglichst zufälligen Pfad besitzen. Demnach wird ein dritter Punkt eingeführt, welcher als Verbindungspunkt zwischen Start und Ende dient und zufällig im Array bestimmt wird. Zufügen weiterer Verbindungspunkte erhöht die Verstrickung des Hauptpfades, beeinflusst jedoch mehr Elemente des Ausgangsarrays. Der Startpunkt wird mit dem dritten Punkt und anschließend dieser mit dem Endpunkt verbunden. Die Koordinaten aller Elemente des Pfades im Array wer-

den separat gespeichert, zusammen mit den eingehenden und ausgehenden Kanten der Koordinaten. Diese Kanten bestimmen welche Richtungen einen eingehenden und ausgehenden Pfad besitzen, und somit an einen verbundenen Gang oder Raum grenzen.

Zwischen dem dritten und vierten der in Kapitel 3.1.2 benannten Schritte wird das Array der Ebene, dessen Elemente jeweils alle möglichen Tiles beinhalten, angepasst. Dazu wird jedes Element der Ebene mit den gleichen Koordinaten wie ein Element des Hauptpfades durch die eingehenden und ausgehenden Kanten des Pfades angepasst. Dies bedeutet, wenn für ein Element E mit den Koordinaten m und n ebenfalls ein Element im Hauptpfad H mit den gleichen Koordinaten im äquivalenten Array existiert, und dieses Element $H_{m,n}$ an der oberen Kante K_0 einen ausgehenden Pfad, und an der unteren Kante K_2 einen eingehenden Pfad hat, dass die Kanten des Elementes $K_0, K_2(H_{m,n})$ dort mit einem Pfad verbunden sind. Mit dem definierten Tilesatz von Gängen, Räumen und Wänden, wobei Gänge und Räume Teil eines Pfades sein können, darf die Teilmenge aller Eingabetiles des Elementes $H_{m,n}$ für die Kanten $K_0, K_2(H_{m,n})$ keine Wände als Nachbarn zulassen. Demnach werden im Array der Ebene des WFC-Algorithmus bei allen Elementen für die gilt, dass im äquivalenten Array ein Element mit den gleichen Koordinaten im Hauptpfad existiert alle Kanten angepasst, die aufgrund des Pfades keine Wände zulassen. Beim Ausführen des vierten Schrittes des WFC-Algorithmus wird dadurch ein verbundener Hauptpfad garantiert, da keine Elemente im Pfad zugelassen sind, welche an einer Kante innerhalb des Pfades eine Wand zulassen.

Der WFC-Algorithmus wird nicht eingeschränkt, da der Algorithmus um vorgegebene Konstrukte herum eine Ausgabe ergänzen kann (siehe Abbildung 3.2.1). Außerdem bleiben alle Funktionen aus Kapitel 4.3 erhalten, da kein absolutes Konstrukt vorgegeben wird, sondern die Teilmengen bestimmter Elemente lediglich reduziert. Jedoch können nicht alle Anforderungen aus 4.1 erfüllt werden, da weitere Pfade abseits des Hauptpfades erzeugt werden, welche nicht mit diesem verbunden sind (siehe Abbildung 4.4.1).



Abbildung 4.4.1: Beispiel eines Dungeon mit WFC-Algorithmus mit Hauptpfad

4.4.2 Einschränken des Algorithmus (Constraints based)

Dieser Ansatz behandelt das Problem des Erzeugens mehrerer, nicht mit dem Hauptpfad verbundener Pfade. Ein einschränkungsbasierter Ansatz fügt der Ausführung des WFC-Algorithmus weitere Einschränkungen zu, damit bestimmte Generierungen häufiger, oder garantiert, auftreten. Diese Einschränkungen können in den Einschränkungen der Tiles, deren Kanten oder der Auswahl eines Elementes in der Ebene erfolgen. Die im Folgenden beiden erklärten Ansätze werden vom Beispielprogramm umgesetzt.

Der WFC-Algorithmus kollabiert stehts ein Element mit der geringsten Entropy über eins. Die Umsetzung der Ränder des Arrays der Ebene als Wände schränkt alle Elemente an den Rändern ein. Ein Element $E_{0,0}$, welches in der oberen linken Ecke des Arrays liegt kann keine Pfade an den oberen und linken Kanten K_0K_3 zulassen. Mit der Menge aller Eingabetiles d und der Teilmenge $d_{w(K_0,K_3)}$, welche alle Tiles enthält die bei K_0, K_3 eine Wand zulassen ist die Teilmenge $d_o(E_{0,0})$, welche alle möglichen Tiles für $E_{0,0}$ darstellt $d_o(E_{0,0}) = d \setminus d_{w(K_0,K_3)}$ die Menge aller Eingabetiles ohne solche die eine Wand an K_0K_3 zulassen. Dadurch ist die Entropy der Elemente an den Rändern des Arrays niedriger. Der Algorithmus kollabiert

dadurch meist die Elemente von außen nach innen, wodurch häufig neue Pfade an Rändern des Arrays erstellt werden (siehe Abbildung 4.4.2).

Der erste einschränkungs-basierte Ansatz verhindert ein Kollabieren der Elemente an den Rändern des Arrays, solange das Kollabieren anderer Elemente möglich ist, und wird im folgenden Kantenvermeidung genannt. Dadurch werden Elemente neben bereits kollabierten Elementen häufiger kollabiert, da diese durch constraint propagation eine meist reduzierte Entropy besitzen. Kollabiert ein Element neben einem bereits vorhandenen Pfad, steigt die Wahrscheinlichkeit, dass dieses Element mit einem Element des existierenden Pfades verbunden wird. Diese Einschränkung verhindert das Erzeugen unverbundener Pfade nicht, reduziert jedoch dessen Wahrscheinlichkeit (siehe Abbildung 4.4.2).

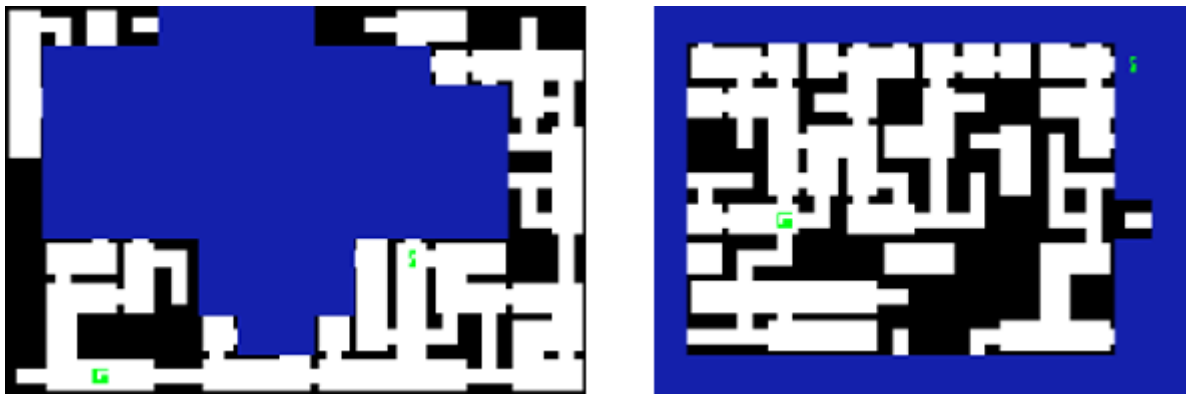


Abbildung 4.4.2: Beispiel des Generierens ohne Kantenvermeidung (links) und mit (rechts)

Der zweite Ansatz schränkt ein welche Elemente des Arrays der Ebene kollabiert werden, indem nur Elemente kollabieren, welche an einem eingehenden Pfad liegen und wird im folgenden Pfadzwang genannt. Außerdem garantiert dieser Ansatz nur eine Verbindung zwischen einem Start und Endpunkt unter Verwendung eines vergleichbaren Ansatzes wie in Kapitel 4.4.1 erklärt wurde. Dazu wird zunächst ein Element kollabiert, welches einen Pfad bildet und mit dem Hauptpfad verbunden wird. Demnach darf dieses erste Element keine Tile des Typs Wand sein. Als Auswahl der ersten zu kollabierenden Tile bietet sich der Startpunkt an, welcher immer zu einer Gang oder Raum Tile kollabiert und Teil des Hauptpfades ist. Schritt vier der in Kapitel 3.1.2 erklärten Schritte muss eingeschränkt werden. Dazu wird nicht ein Element mit der kleinsten Entropy über eins kollabiert, sondern ein Element mit der kleinsten Entropy über eins welches zusätzlich mindestens ein benachbartes Element mit einer Entropy von eins besitzt, bei welchem dieser Nachbar zudem einen eingehenden Pfad zu diesem Element bildet. Demnach werden für ein durch die Entropy ausgewähltes Element $E_{m,n}$ nacheinander seine Nachbarn $E_{m+1,n}, E_{m-1,n}, E_{m,n+1}, E_{m,n-1}$ betrachtet. Sobald ein benachbartes Element eine Entropy von eins vorweist, wird dessen eingehende Kante überprüft. Bei einem benachbarten Element welches über dem Element, und damit an dessen oberen Kante K_0 liegt, ist die eingehende Kante des benachbarten Elementes dessen untere Kante K_2 . Wenn eine der eingehenden Kanten $K_3(E_{m+1,n}), K_1(E_{m-1,n}), K_2(E_{m,n+1}), K_0(E_{m,n-1})$ keine Wand darstellt bedeutet dies, dass ein Pfad zum Element $E_{m,n}$ geht. Sobald erkannt wird, dass ein Pfad zu einem Element geht, wird das Element kollabiert. Tritt dies für keines der benachbarten Elemente ein, geht kein Pfad auf das Element zu und der Vorgang wird für ein anderes Element mit der niedrigsten Entropy über eins durchgeführt, bis ein Element kollabiert wird. Eine effektive Methode dazu ist eine Liste aller Elemente mit der niedrigsten Entropy über eins zu speichern. Diese Liste wird aktualisiert, sofern das Kollabieren eines Elementes die Entropy anderer Elemente beeinflusst. Sobald Elemente der Liste zugefügt werden, wird überprüft, ob diese geeignete Nachbarn besitzen und nur im Falle dessen das Element in die Liste eingetragen. So

wird ein Eintragen ungeeigneter Elemente verhindert. Diese Methode garantiert das Generieren eines verbundenen Pfades, indem ein existierender Pfad nach den Prinzipien des WFC-Algorithmus erweitert wird, bis dieser beendet ist. Der Pfad wächst durch das Array der Ebene. Jedoch können nicht alle Elemente durch diese Methode kollabiert werden, sodass die übrigen Elemente durch Wände aufgefüllt werden (siehe Abbildung 4.4.3).

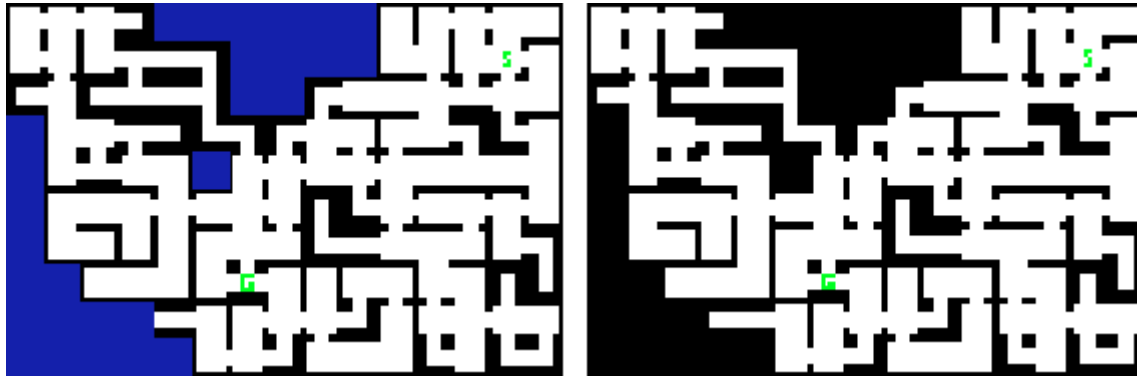


Abbildung 4.4.3: Beispiel des Generierens und auffüllen eines Dungeons mit Phadzwang

Diese Methode erfüllt alle Anforderungen nach Kapitel 4.1, der Einfluss von Gewichtungen für Wände wird jedoch reduziert, da diese nicht an einem eingehenden Pfad erscheinen können. Gewichtungen für Räume und Gänge bleiben jedoch vollständig funktionell. Die definierten Funktionen aus Kapitel 4.3 bleiben jedoch nicht alle erhalten, da dieser Ansatz nicht garantiert ein Element mit der niedrigsten Entropy über eins kollabiert, sondern ein Element mit der niedrigsten Entropy über eins aus einer Teilsumme aller Elemente. Zudem wird durch das Auffüllen des Arrays mit Wandtiles nicht jedes Element zufällig auf eine geeignete Tile kollabiert.

4.4.3 Nachträgliche Korrektur (Post Processing)

Dieser Ansatz behandelt ebenfalls das Generieren mehrerer, nicht mit dem Hauptpfad verbundener Pfade. Ein Ansatz der nachträglichen Korrektur, allgemein post processing genannt, versucht den WFC-Algorithmus selbst nicht zu verändern und lediglich nachträglich entstandene Fehler zu beseitigen. Für die Generierung eines verbundenen Dungeons werden entstandene Pfade, welche nicht mit dem Hauptpfad verbunden sind, nachträglich mit dem Hauptpfad verbunden. Dazu wurde ein eigenständiger Ansatz entwickelt, welcher im folgenden Gruppenkombinierung genannt wird.

Der Ansatz der Gruppenkombinierung speichert jeden entstandenen Pfad als eine eigene Gruppe. Dazu wird innerhalb von Schritt 4 des WFC-Algorithmus aus Kapitel 3.1.2 beim Kollabieren jedes Elementes überprüft, ob die festgelegte Tile ein Pfadelement ist. Dies ist weiterhin gegeben, wenn die Tile keine Tile des Typs Wand, sondern Gang oder Raum ist. Ist die Tile ein Pfadelement wird wie in Kapitel 4.4.2 beschrieben überprüft ob benachbarte Elemente bereits zu Tiles mit eingehenden Pfaden kollabiert sind. Sofern noch kein benachbartes Element zu einer Tile mit einem eingehenden Pfad kollabiert ist, grenzt die kollabierte Tile an keinen Pfad an. Daher wird eine neue Liste erzeugt, in welche die Koordinaten der Tile eingefügt werden. Diese Liste repräsentiert einen Pfad, speichert alle Elemente des Pfades und wird im folgenden Pfadgruppe genannt. Zudem speichert jedes kollabierte Element zu welcher Pfadgruppe dieses gehört.

Wird für das kollabierte Element eine benachbarte Tile erkannt, welche einen eingehenden Pfad bildet, wird das Element der existierenden Pfadgruppe seines Nachbarn zugefügt, so wie für dieses Element ebenfalls gespeichert, welcher Pfadgruppe dieses zugefügt wurde.

Besitzt ein kollabiertes Element mehr als einen Nachbarn, welcher durch seine Tile einen eingehenden Pfad bildet, werden die Pfadgruppen durch das kollabierende Element verbunden. Dazu wird überprüft, ob die beiden Nachbarn verschiedenen Pfadgruppen angehören. Sind die Pfadgruppen gleich wird das Element der ersten Pfadgruppe zugefügt. Bei verschiedenen Pfadgruppen wird das Element der ersten Pfadgruppe zugefügt, alle Tiles aus der zweiten Pfadgruppe entfernt und der ersten zugefügt, für alle übertragenen Tiles aktualisiert welcher Pfadgruppe diese angehören und die entleerte Pfadgruppe entfernt. Dieses Vorgehen garantiert durch die Prinzipien des WFC-Algorithmus am Abschluss dessen, dass alle existierenden Pfade in Gruppen gespeichert und deren enthaltene Tiles bekannt sind. Zudem ist das Vorgehen einfach in den WFC-Algorithmus integrierbar, sodass keine aufwändigen Pfadalgorithmen im Nachhinein verwendet werden, um die einzelnen Pfadgruppen zu erkennen. Wird als erstes der Startpunkt kollabiert und durch diesen eine Pfadgruppe erstellt, ist diese Gruppe der Hauptpfad. Die folgende Abbildung 4.4.4 veranschaulicht das Speichern der Pfade in Pfadgruppen.

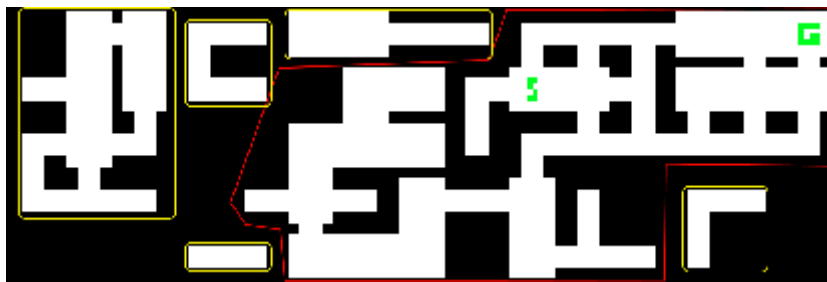


Abbildung 4.4.4: Beispiel für das Erkennen von Pfadgruppen und Hauptpfad. Hauptpfad = rot

Der nächste Schritt verbindet die einzelnen Pfade. Dazu speichern die Pfadgruppen zusätzlich ihre jeweiligen Extremstellen. Für jede Pfadgruppe mit enthaltenen Elementen $E_{m,n}$ werden die Koordinaten der vier Extremstellen $E_{m,n_{min}}$, $E_{m,n_{max}}$, $E_{m_{min},n}$, $E_{m_{max},n}$ welche am weitesten oben, unten, links und rechts sind gespeichert. Beim zufügen einer Tile in eine Gruppe wird überprüft, ob diese Tile eine der vorigen Extremstellen übersteigt und in diesem Fall die jeweilige Extremstelle aktualisiert. Nach beenden des WFC-Algorithmus wird der Reihe nach jede Pfadgruppe mit ihrer jeweils nächsten Gruppe verbunden. Dazu wird von den Extremstellen einer Pfadgruppe jeweils in die Richtung der jeweiligen extremstelle schrittweise das Array der Ebene überprüft. Dabei sind die Grenzen des Arrays der Ebene die Grenzen der Schritte (siehe Abbildung 4.4.5).

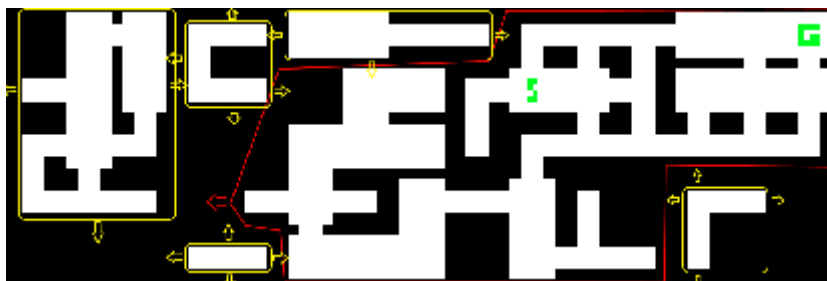


Abbildung 4.4.5: Beispiel Schrittweises suchen nach benachbarten Pfadgruppen von Extremstellen aus

Sobald die erste Extremstelle eine Tile entdeckt, welche keine Wand und somit ein Pfadelement ist werden alle Wandtiles zwischen den Pfadelementen entfernt. Diese Elemente werden jeweils mit einer Teilmenge aller Eingabetiles gefüllt, welche keine Tiles enthalten deren Kanten entlang des Pfades der Extremstellen Wände enthalten. Demnach werden, wenn eine Extremstelle rechts von sich eine Pfadgruppe entdeckt, von der Extremstelle $E_{m,n}$ ausgehend alle Tiles

$E_{m,n++}$ entfernt, bis das entdeckte Pfadelement erreicht wird. Diese geleerten Elemente zwischen den Pfadelementen werden mit der Teilmenge $d_{pf} = d \setminus d_{w(K_1, K_3)}$ aufgefüllt. Die beiden Pfadelemente, welche jeweils über den zu erstellenden Pfad zwischen diesen verbunden werden, werden ebenfalls entfernt und mit Teilmengen aufgefüllt. Diese Teilmengen enthalten jeweils alle Elemente der Menge der Eingabetiles d , welche keine Wand an den Kanten zulassen, die in die Richtung des jeweils anderen Pfadelementes zeigen. Die restlichen Kanten der Tiles dieser Teilmenge entsprechen den Kanten der jeweils entfernten Pfadelemente. Danach wird Schritt 4 des WFC-Algorithmus wiederholt, um die Elemente des Verbindungspfades mit den neuen Teilmengen zu kollabieren. Dabei entsteht durch die erklärte Auswahl der Teilmengen garantiert eine Verbindung zwischen den Pfaden. Beim Kollabieren der Elemente des Verbindungspfades werden die Pfadgruppen nach demselben Prinzip von zuvor zusammengefügt. Dieses Vorgehen wird wiederholt, bis ein Pfad übrig ist oder keine Pfadgruppe einen anderen Pfad entdecken kann.

Die folgende Abbildung zeigt zwei Beispiele für den einzigen Fall welcher mir einfiel in welchem der Algorithmus keine verbundenen Pfade erzeugt. Dies geschieht dadurch, dass bestimmte Pfadgruppen sich gegenseitig nicht entdecken können, da diese abgespalten angeordnet sind und der Algorithmus daher vorzeitig beendet wird. Damit dies geschieht ist jedoch eine sehr bestimmte Generierung notwendig, welche in keinem Test der Gruppenkombinierung auftrat. Die Gelben Kästen stellen Pfadgruppen dar, in welchen sich Pfade befinden, die roten Pfeile das Suchverfahren, welches von den jeweiligen Extrempunkten aus schrittweise vorgeht. Weiße Flächen stellen Wände dar (siehe Abbildung 4.4.6).

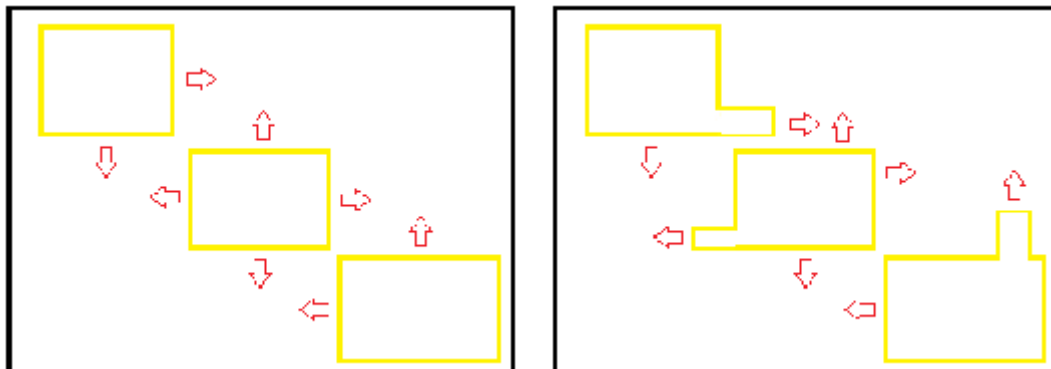


Abbildung 4.4.6: Beispiele einer für Gruppenkombinierung unlösbaren Generierung

Dieser Ansatz erzeugt daher nahezu garantiert einen verbundenen Pfad (siehe Abbildung 4.4.7), kann dies jedoch nicht zu 100% garantieren. Der WFC-Algorithmus selbst ist auch nicht zu 100% garantiert abzuschließen, weshalb dies als akzeptabel angenommen wird. Außerdem ist für diesen Ansatz ein voriges Generieren eines Hauptpfades nicht nötig, da dieser stets alle Pfade verbindet. Der Start und Endpunkt müssen jedoch als Pfadelement festgelegt werden. Der Algorithmus erfüllt demnach alle Anforderungen aus Kapitel 4.1, behält jedoch nicht unbedingt alle Funktionen aus Kapitel 4.3, da der Algorithmus nicht nach dem ersten Abschließen aller Schritte des WFC-Algorithmus einen komplett verbundenen Dungeon erzeugte, sondern im Nachhinein durch post processing korrigiert wurde. Jedoch wurde diese Korrektur durch erneutes Kollabieren von Elementen durchgeführt. Dadurch sollte das Verfahren auf andere Anwendungen des WFC-Algorithmus mit Pfadfindungsproblemen übertragbar sein.

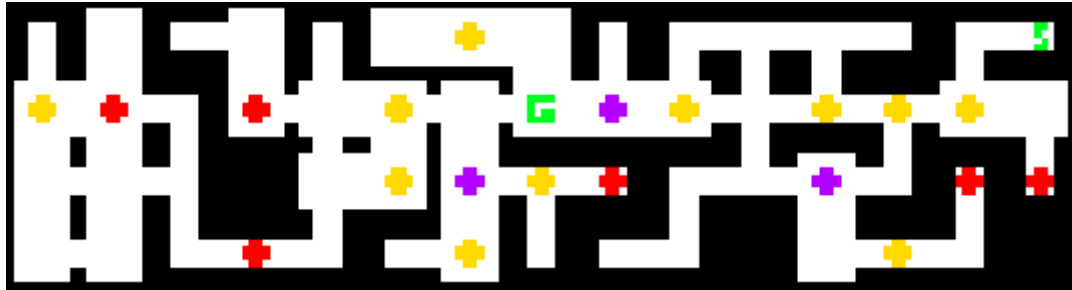


Abbildung 4.4.7: Beispiel eines mit Gruppenkombinierung erstellten Dungeons nach Vorgaben

5 Umsetzung im Beispielprogramm

Wie bereits in den vorigen Kapiteln angedeutet wurde im Zuge dieser Arbeit ein Beispielprogramm umgesetzt, welches die Umsetzung eines Dungeons durch den WFC-Algorithmus tätigt, sowie dessen angesprochene Probleme angeht und veranschaulicht. Diese Umsetzung ermöglicht eine Veranschaulichung aller in Kapitel 4.4 genannten Anpassungen. Die genauere praktische Umsetzung dieser Anpassungen wird im folgenden Kapitel anhand umsetzender Klassen erklärt.

5.1 Verwendung von Unity

Das Programm, welches zur Veranschaulichung und Auseinandersetzung mit dem WFC-Algorithmus und dessen Vorteilen, sowie Nachteile entwickelt wurde, ist in Unity umgesetzt. Unity ist eine seit 2005 bestehende Entwicklungsumgebung für 2D und 3D Anwendungen, welche primär zur Gestaltung und Entwicklung von Computerspielen entwickelt wurde. Durch ihre ideale Umsetzung von 2D und 3D Anwendungen wird diese jedoch nicht nur in Spielen, sondern einer Vielzahl von Anwendungen wie Simulatoren, Architekturprogrammen und anderen Anwendungen, welche im zwei- oder dreidimensionalen Raum agieren angewendet. Dabei bietet Unity Möglichkeiten des interagieren von Objekten innerhalb dieses Raumes in der Form von Code, meist in C#, sowie durch visuelle Komponenten (vgl. (Adam Tuliper, 2014)). Dabei wird in Unity der Visuelle Ausgang in einer Szene dargestellt. Innerhalb dieser Szene sind Gameobjekte platzierbar, welche Objekte der Szene, Platzhalter für Skripte oder andere Funktionen wie eine Kamera darstellen können. Diese Objekte enthalten stets eine Komponente, durch welche ihre jeweilige Position in der Szene bestimmt wird, und können durch weitere Komponenten wie Skripte oder Texturen erweitert werden.

Für die Umsetzung dieser Arbeit wurde Unity als Entwicklungsumgebung gewählt, da die visuelle Ausgabe leicht darstellbar, sowie das Tileset der Eingabetiles einfach zu verwalten ist. Zudem sind die Skripte zum Anwenden und Anpassen des WFC-Algorithmus durch die Systeme von Unity, wie dem Komponentensystem, einfach mit dem Tileset der Eingabetiles verwendbar, wodurch problemlos eine gewünschte Ausgabe geschaffen wird. Die UI-Systeme von Unity ermöglichen dazu ein einfaches und funktionelles Erstellen einer intuitiven Benutzererfahrung.

5.2 Umsetzung des WFC-Algorithmus im Beispielprogramm zum Realisieren der Anforderungen

Das Beispielprogramm setzt den WFC-Algorithmus nach seinen benötigten Schritten um. Dabei findet die komplette Ausführung des WFC-Algorithmus in der Klasse `Make_Doungen` statt. Innerhalb dieser Klasse führt die Methode `public TileData[,] CreateLevel([...])` die in Kapitel 3.1.2 benannten Schritte aus. Diese einzelnen Schritte wiederum werden in ihren jeweiligen Methoden nach den zuvor erklärten Prinzipien ausgeführt. Um den WFC-Algorithmus auszuführen, müssen die relevanten Daten einer jeden Eingabetile in einer verwendbaren Struktur gespeichert werden, wozu sich ein `Struct` eignet. Die Art der Daten, die für das Ausführen einer Anwendung notwendig sind, welche den WFC-Algorithmus verwendet, ist abhängig von der jeweiligen Anwendung und sollten daher individuell angepasst werden. Im Falle dieses Beispielprogrammes ist der `Struct TileData` als eine geeignete Repräsentation aller Tiles verwendet. Die Eingabetiles des Algorithmus werden dabei aus einer vorgefertigten `Tilemap` bezogen, sowie in der Methode `void CreateListOfAllDoungenObjects()` für Tiles welche drehbar sind weitere Tiles dynamisch aus der `Tilemap` erzeugt. Für tilebasierte Anwendungen des WFC-Algorithmus ist eine solche dynamische Ergänzung für mögliche Tiles zu empfehlen, da dadurch die Menge der zuvor zu erstellenden Tiles reduziert wird. Da Algorithmen der prozeduralen Generierung häufig zum Sparen von Aufwänden und Kosten verwendet werden sollten Ähnliche Möglichkeiten stets beachtet werden.

Die wichtigsten Methoden, welche die `constraint propagation`, und somit den vierten Schritt aus Kapitel 3.1.2 umsetzen, sind die Methoden `void MakeLvlWithWaveFunctionCollapse()`, `void UpdateNeighboursOfThis(int x, int y, DirectionsOf bdir)` und die ergänzende Methode `void UpdateNeighbourOfThisAfterRemove([...])`. Diese setzen die zuvor für Schritt 4 beschriebenen Vorgehensweisen um. Dabei ist zu empfehlen für die Gesamte Ebene der Ausgabe zuvor ein 2D Array zu erstellen, dessen Elemente Listen von `TileData` Objekten enthalten. Dadurch wird auf diesem Array die `constraint propagation` ausgeführt. Die Entropy eines jeden Element ist die Länge der jeweiligen Liste. Es ist vorteilhaft die Menge aller Elemente mit der kleinsten Entropy über eins in einer separaten Liste abzuspeichern. Diese Liste wird ergänzt oder aktualisiert sollte durch ausgeführte `constraint propagation` ein weiteres Element der aktuell kleinsten Entropy über eins entsprechen, oder eine kleinere Entropy über eins vorweisen. Das Verwalten einer solchen Liste erspart das Suchen der Elemente mit der kleinsten Entropy über eins.

5.2.1 Einbringen verschiedener Dungeon Elemente mit verschiedenen Objekten

Wie in den Kapiteln 4.2.1 und 4.2.3 erklärt sind diese Anforderungen ohne Anpassungen des WFC-Algorithmus und ohne spezifische Anpassungen im Code umsetzbar. Das Beispielprogramm wählt eine geeignete Menge von Eingabetiles in seiner `Tilemap`, welche sowohl sämtliche Typen von Dungeon Elementen enthält, sowie diese Elemente in verschiedenen Ausführungen mit den jeweiligen Objekten umsetzt. Zur Umsetzung eines WFC-Algorithmus basierten Programmes zur Erstellung von Dungeons muss bei einer angemessenen Auswahl von Eingabetiles lediglich eine geeignete Repräsentation dieser implementiert werden. Diese geeignete Repräsentation wird vom `TileData struct` durchgeführt.

```
public struct TileData
{
    public Sprite Sprite;
    public int top;    // gibt an ob eine Richtung einen Gang, Mauer oder
//offenen Raum hat
    public int right; // 0 steht für kein Durchgang(wand), 1 für gang,
//und 2 für offene Fläche (Raum) in dieser Richtung
```

```

public int down;
public int left;
public DoungenObjType doungeObjTypeInt;
public GangRaumWand gangRaumWandInt;
public int rotateEulerValue;
[...]
}

```

5.2.2 Implementieren eines verbundenen Hauptpfades

Das in Kapitel 4.4.1 beschriebene Verfahren wird im Beispielprogramm umgesetzt. Die zwei verantwortlichen Methoden sind `void CreateFirstStartToEndPath(int x, int y)` und `void UpdatePlaneWithStartToEndPath()`.

Die erste Methode erzeugt zwei Arrays, welche jeweils eine Ebene von identischer Größe des Levels aufspannen. Diese speichern jeweils für alle Elemente, die Teil des Hauptpfades sind, deren jeweils eingehende und ausgehende Richtung an den Koordinaten des Elementes. Der Hauptpfad wird dabei schrittweise nach dem Verfahren aus Kapitel 4.4.1 ermittelt.

Die zweite Methode aktualisiert das Array des Levels, indem alle Elemente des Arrays mit den beiden äquivalenten Arrays verglichen werden. Dabei werden inkompatible Tiles aus der Menge der Eingabetiles entfernt, sofern die äquivalenten Arrays eine eingehende oder ausgehende Richtung für dieses Element vorgeben. Die entfernten Tiles jedes Elements sind alle Tiles, welche eine eingehende oder ausgehende Kante mit einer Wand blockieren.

Dieses Verfahren ist simple zu implementieren, bei einem Ansatz mit Gewichtungen ist jedoch zu beachten, dass die zweite Methode nicht ausgeführt wird, sollten die Gewichte für Gänge und Räume bei null stehen. In diesem Fall dürfen nur Tiles des Typs Wand platziert werden. Solche Tiles erfüllen den Typ Wand an jeder ihrer Kanten. Die zweite Methode entfernt jedoch Tiles mit Wandkanten. Demnach könnte der WFC-Algorithmus nicht erfolgreich beendet werden, da die Teilmengen mancher Elemente keine Tiles enthalten, welche zu ihren jeweiligen Nachbarn zulässig sind.

5.2.3 Implementierung von gewichteten Tiles

Wie in Kapitel 4.2.4 erklärt können Gewichtungen für Tiles des WFC-Algorithmus eingeführt werden, indem die Wahrscheinlichkeiten zur Auswahl einer Tile beim Kollabieren eines Elementes beeinflusst werden. Das kollabieren eines Elementes wird in der Methode `void CollapseRandomLowestEntropy()` umgesetzt. Es wird aus der Liste aller Elemente mit der niedrigsten Entropy über eins zufällig ein Element ausgewählt. Aus der Liste der Tiles des Elementes, welche eine Teilmenge aller Eingabetiles enthält, welche alle an dieser Position zulässigen Tiles darstellt, wird beim WFC-Algorithmus zufällig eine Tile ausgewählt. Das Beispielprogramm beeinflusst diese zufällige Auswahl durch Gewichte.

Die Gewichte werden während der Ausführung des Programms vom Anwender im UI festgelegt und an die Klasse beim Generieren eines Dungeons übermittelt. Jede Tile wird auf ihren Typ, ihr enthaltenes Objekt, sowie den Typen ihrer Kanten innerhalb der Methode überprüft. Dabei wird ein finales Gewicht für die jeweilige Tile aus ihren jeweiligen Gewichten aufsummiert. Zudem wird ein Dictionary verwendet, im Beispielcode `tmpStoredDORepre` genannt, welches jede Tile, mit einem Integer Wert der aufsummierten Gewichte der Tiles, als Schlüssel speichert.

```

if ((weightType * weightGangRaumWand) > 0)
{
    totalWeight += (weightType * weightGangRaumWand * cornerWeight);
    tmpStoredDORepre.Add(totalWeight, DORepre);
    tmpListToChoseTile.Add(totalWeight);
}

```

Letztlich wird zufällig eine Zahl zwischen Null und der endgültigen Summe der Gewichte aller Tiles ausgewählt. Die Tile, deren Schlüssel der Erste größer oder gleich dem gewählten Wert ist, wird ausgewählt. Da die Schlüssel sich aus den aufsummierten Gewichten ihrer Tiles ergeben, welche Gewichte von unterschiedlicher Größe besitzen, ergeben sich verschiedenen große Abstände zwischen den Schlüsseln. Der Abstand eines Schlüssels bestimmt, wie wahrscheinlich dieser ausgewählt wird. Es wird eine zufällige und gewichtete Auswahl nach der Methode von Kapitel 4.2.4 getroffen wobei der Typ einer Tile, sowie dessen enthaltenes Element berücksichtigt wird. Jedoch ist es wichtig für jedes Objekt jeweils eine Wandtile zu erstellen und diese als Wand eines solchen Objektes zu deklarieren, da ansonsten Wände nur als normale Tiles gelten, und seltener in Dungeons mit vielen nicht normalen Tiles generiert werden.

5.2.4 Implementierung zum Generieren verbundener Pfade (Pfadzwang Ansatz)

Der Ansatz des Pfadzwangs nach Kapitel 4.4.2 beschränkt die Auswahl von Elementen nach ihrer Entropy. Das Beispielprogramm überprüft die Entropy eines Elements in der Methode `void ExecuteEntropyPositions(int x, int y)`. Dabei wird bei Änderungen der Teilmenge eines Elements dessen Entropy überprüft und wenn nötig die Liste aller Elemente mit der niedrigsten Entropy über eins angepasst.

Beim Pfadzwangsverfahren wird zunächst in der Methode `void ProcessEntropyOfPosition(int x, int y)` entschieden, ob die Entropy eines Elementes überprüft wird. Dazu wird die Methode `bool PositionHasConnectedNeighbour(int x, int y)` aufgerufen, welche nach den in Kapitel 4.4.2 beschriebenen Vorgehensweisen des Pfadzwangsverfahrens zurückgibt, ob ein benachbartes Element mit einem Pfadeingang existiert.

5.2.5 Implementierung zum Generieren verbundener Pfade (Gruppenkombinierung)

Das Verfahren der Gruppenkombinierung, welches in Kapitel 4.4.3 erläutert wird, besteht grundlegend aus zwei Schritten. Dem Erkennen und Speichern der Pfadgruppen, sowie dem nachträglichen Verbinden dieser Pfade.

Der erste Schritt erstellt zunächst ein Array mit der *X* und *Y* Achsenlänge des Levelarrays. In diesem wird für jede Tile gespeichert welcher Pfadgruppe sie angehört. Die Methode `void InitiateStartPathGroup()` kollabiert den Startpunkt damit die erste erzeugt Pfadgruppe den Hauptpfad darstellt. Das erstellen und aktualisieren der Pfadgruppen geschieht in der Methode `void AddTileToPathGroup(int x, int y)`, welche durch andere Methoden aufgerufen wird, sobald ein Element auf eine Tile kollabiert. Dabei wird die aus Kapitel 5.2.4 bekannte Methode `bool PositionHasConnectedNeighbour(int x, int y)` aufgerufen um zu überprüfen ob ein Nachbar mit eingehendem Pfad existiert. Geht kein Pfad auf das kollabierte Element zu wird eine neue Pfadgruppe durch die Methode `PathTilesGroup MakeNewPathGroup(int x, int y, PathTilesGroup p)` erstellt. Ist ein eingehender Pfad vorhanden wird die Tile der Pfadgruppe der ersten entdeckten Pfadgruppe zugefügt. Bei mehr als einem eingehenden Pfad werden die Pfadgruppen in `void MergePathGroups(int id1, int id2)` kombiniert, sofern diese unterschiedliche Gruppen darstellen. Dabei bleibt der Hauptpfad stets erhalten. Das Ausführen dieser Methoden stellt sicher, dass jede Tile einer Pfadgruppe angehört, sowie diese Gruppen eindeutig unverbundene Pfade repräsentieren, wie in Kapitel 4.4.3 dargestellt.

Im zweiten Schritt werden die Pfade verbunden. Dazu wird nach dem Kollabieren aller Elemente die Methode `void ConnectPathGroupsInto1()` ausgeführt. Diese Methode sucht von jeder Pfadgruppe, die nicht die Hauptpfadgruppe darstellt, in alle vier Richtungen von ihren jeweiligen Extrempunkten nach anderen Pfadgruppen, wie in Kapitel 4.4.3 erläutert. Wird eine andere Pfadgruppe entdeckt werden die jeweiligen Pfade mit einem direkten Weg durch die Methode `void ConnectTilesAndGroups([...])` verbunden. Dazu wird schrittweise von der Extremstelle der Gruppe bis zur entdeckten Tile der anderen Gruppe jede Tile entfernt und durch

eine Teilmenge aller zulässigen Tiles ersetzt. Durch die Wahl geeigneter Teilmengen ermöglichen diese das Bilden eines direkten Pfades zwischen den beiden Gruppen durch eine erneute Ausführung des WFC-Algorithmus und verbinden die Pfade. Das Einfügen von geeigneten Teilmenge zum Kollabieren der Elemente geschieht in der Methode `void UpdatePossibleTilesToConnectPaths([...])`. Nachdem zwischen zwei Pfaden die Tiles mit geeigneten Teilmengen ersetzt wurden wird der WFC-Algorithmus durch aufrufen der Methode `MakeLvl-WithWaveFunctionCollapse()` ausgeführt und somit die Pfade verbunden und die Menge aller Pfadgruppen aktualisiert. Dies ist essenziell da der Algorithmus beendet wird so bald keine Pfadgruppen, außer der Hauptpfadgruppe, übrig sind. Ist eine Pfadgruppe nicht in der Lage einen Pfad zu entdecken steigt ein Zähler, welcher die Anzahl erfolgloser Suchen darstellt und die nächste Pfadgruppe sucht nach Pfaden. Alternativ wird der Algorithmus abgebrochen sollte keine Gruppe in der Lage sein eine andere Pfadgruppe zu finden. Der unten gezeigte Code stellt diese beiden Abbruchbedingungen dar, wobei der Index für die Anzahl erfolgloser Suchen steht. Wie in Kapitel 4.4.3 erklärt ist es möglich für alle Gruppen keinen Pfad zu finden, jedoch sehr unwahrscheinlich.

```
if (!nothingFound)
{
    pgIndex = 0;
}
if (nonMainPathGroups.Count == 0 || pgIndex == nonMainPathGroups.Count)
{
    break;
}
```

5.3 Weitere verwendete Klassen des Beispielprogramms

Wie in Kapitel 5.2 erklärt wird die Logik des WFC-Algorithmus und dessen Anpassungen in der Klasse `Make_Doungen` umgesetzt. Zum umsetzen eines Dungeons durch den WFC-Algorithmus in Unity werden allerdings weitere Klassen verwendet, welche im Folgenden erklärt sind.

`MyGlobalSelfmadeNamespace` ist ein Namespace, welcher alle wichtigen Datenstrukturen verwaltet, die innerhalb des Beispielprogramms verwendet werden. Die wichtigsten Datenstrukturen sind dabei `TileData` und `PathTileGroups`. Die Verwendungen beider sind in den Kapiteln 5.2 und 5.2.5 erklärt.

Die Klasse `Tile_Library` wird zum Erstellen eines Scriptable Objects verwendet, welches zum Verwalten der Menge aller manuell erstellten Eingabetiles verwendet wird, welche als `TileData` Strukturen gespeichert werden.

`Menu_Script` ist eine Klasse, welche die Verwaltung des Programms tätigt. Diese Klasse implementiert die Steuerung der Kamera und verwaltet die Aktionen der Buttons und Slider des Menüs. Zudem entnimmt diese Klasse den jeweiligen Slidern und Feldern die Gewichte, Erstellungstypen und Ausmaße, welche zur Erstellung des Dungeons benötigt werden. Diese gibt sie an `Make_Doungen` weiter und erhält ein Array zurück, welches den fertigen Dungeon darstellt.

Die Klasse `Create_Grafik_Doungen` erstellt die graphische Ausgabe des Dungeons. Dazu nimmt diese das von `Make_Dungeon` erstellte Array, welches den Dungeon mit all seinen Einträgen darstellt von der Klasse `Menu_Script` entgegen. Die logische Darstellung des Dungeons durch das Array wird dann von dieser Klasse in eine grafische Ausgabe umgewandelt. Dazu werden die Sprites der `TileData` Objekte abhängig von der Größe des Dungeons skaliert und an den richtigen Positionen innerhalb der Szene platziert.

6 Erkenntnisse

Diese Arbeit hat sich mit den Möglichkeiten, Vorteilen und Nachteilen des WFC-Algorithmus auseinandergesetzt und dabei versucht Techniken zu finden, welche eine Anwendung des WFC-Algorithmus in Dungeons ermöglicht und optimiert. Diese sollten dabei versuchen die Ausgaben realistischer zu gestalten. Dabei wurde versucht Ansätze zu definieren, welche eine Übertragung gefundener Techniken auf andere Anwendungen erleichtert. Im Folgenden wird die Eignung des WFC-Algorithmus zum Generieren von Dungeons beurteilt, sowie Mögliche Übertragbarkeiten und Fortführungen besprochen.

6.1 Eignung des WFC-Algorithmus zum Erstellen eines Dungeons

Der WFC-Algorithmus kann zum Erstellen eines realistischen Dungeons verwendet werden. Dazu müssen jedoch meist abhängig von der Anwendung Individuelle Anpassungen an diesem vorgenommen werden.

Ein bekanntes Problem des WFC-Algorithmus ist, dass es schwierig ist Details effektiv in eine Endausgabe einzubringen, da diese meist nicht im gewünschten Maße auftreten. Dies wird Overfitting genannt (vgl. (Brian Bucklew, 2022)). Ein tilebasierter Ansatz zum Generieren eines Dungeons kann Details durch eine effektive Wahl von Einschränkungen, sowie einer Anpassung des WFC-Algorithmus zum Verwenden von Gewichten umsetzen. Die verschiedenen Typen von Dungeonobjekten, welche in dieser Arbeit verwendet wurden, fügen einem Dungeon Details wie bestimmte Gegner oder Gegenstände zu und können in ihrer Menge über Gewichte geregelt werden. Durch eine Anpassung Gewichte zuzufügen kann demnach eine detailreichere und realistischere Generierung eines Dungeons unter Verwendung des WFC-Algorithmus ermöglichen. Die Art der Gewichte ist jedoch entscheidend für die Ausgabe. Daher muss ein Anwender bewusst entscheiden welche Unterschiede in Tiles gewählt und somit gewichtet werden.

Die generelle Erzeugung eines Dungeons unter Verwendung des WFC-Algorithmus ist zufällig und daher werden meist alle Tiles innerhalb des Levels willkürlich erzeugt. Ist es gewünscht, dass in einem Dungeon bestimmte Räume nur neben bestimmten anderen Räumen entstehen, bestimmte Objekte nur in bestimmten Räumen auftreten oder bestimmte Objekte nur neben bestimmten Räumen und Objekten erscheinen, ist dies möglich. Dies muss jedoch durch eine genau Bestimmung der Eingabetiles festgelegt werden, indem deren Einschränkungen an ihren jeweiligen Kanten für die gewünschte Ausgabe passend gewählt wird, sowie nur ausgewählte Tiles mit Objekten angefertigt werden. Dadurch kann gewährleistet werden, dass bestimmte Tiles nur in der Nähe bestimmter anderer auftreten. Deren Position innerhalb des Levels ist jedoch weiterhin zufällig. Eine genauestens gewählte Menge von Eingabetiles mit ihren jeweiligen Einschränkungen kann durch Gewichtungen ergänzt werden, um gewünschte Ausgaben zu erzeugen. Ein solches Vorgehen benötigt nicht viele Anpassungen, jedoch präzise Überlegungen über die zu verwendenden Tiles. Dies kann viel Zeit kosten, sowie zukünftige Anpassungen erschweren oder Fehlerquellen darstellen.

Das bedeutendste Problem des WFC-Algorithmus ist das Pfadfindungsproblem. Selbst wenn Teile von Levels ideal erstellt werden, ist dies nutzlos, wenn diese nicht miteinander verbunden sind. Daher müssen Anpassungen am WFC-Algorithmus zum Generieren von Dungeons vorgenommen werden. Da das Pfadfindungsproblem zur Verbindung aller Teile eines Levels innerhalb eines Dungeons jedoch meist ähnlich ablaufen sollte, ist es möglich einen allgemeinen

Algorithmus zu entwickeln der auf verschiedene Dungeons anwendbar ist. Ein Algorithmus wie der in dieser Arbeit entwickelte Algorithmus der Gruppenkombinierung, welcher die Prinzipien der Generierung des WFC-Algorithmus für seine eigene Funktionalität verwendet, sollte daher mehrere, mit dem WFC-Algorithmus generierte Dungeons anpassen können. Daher ist das Generieren von Dungeons mit dem WFC-Algorithmus, bei welchen jede begehbare Tile verbunden ist möglich, jedoch müssen Anpassungen am WFC-Algorithmus vorgenommen werden. Zudem ist es eine effiziente Methode zum Erstellen von Dungeons, da nach einmaligen entwickeln eines solchen Algorithmus eine Vielzahl von Dungeons generiert werden kann.

Ein großes Problem zum Generieren von Dungeons durch den WFC-Algorithmus kann die Laufzeit darstellen. Wird der WFC-Algorithmus verwendet, um in Echtzeit beispielsweise für ein Spiel ein Level zu erzeugen dürfte dieses nicht zu große Maße besitzen, da der WFC-Algorithmus aufgrund seiner Komplexitätsklasse im schlimmsten Fall bei Ausgaben $E(XxY)$ mit großen Werten für X und Y eine lange Ausführungszeit besitzt. Bei entsprechend großen Ebenen kann die Laufzeit jedoch durch ein Aufteilen der Ebene, wie in Kapitel 3.2.2 besprochen, verbessert werden. Laufzeitprobleme, welche aus einer großen Menge von Eingabetiles entstehen sind hingegen schwerer zu lösen und können das Erstellen von detailreichen Dungeons einschränken.

Abschließend ist der WFC-Algorithmus zum Erstellen von Dungeons geeignet, jedoch nicht in jedem Fall gleich zu betrachten. Verschiedene Anforderungen brauchen verschiedene Anpassungen oder bewusst gewählte Arten von Eingabetiles. Solche Anpassungen des WFC-Algorithmus, sowie Überlegungen zum Auswählen geeigneter Eingabetiles, brauchen eingehend Zeit, können dann jedoch eine Vielzahl von Dungeons schaffen, welche gewählte Ausgaben erzeugen. Daher ist der WFC-Algorithmus geeignet zum Schaffen von Dungeons, vor allem wenn eine Vielzahl dieser benötigt wird. Soll jedoch nur eine geringe Menge von Dungeons erstellt werden, ist eine manuelle Erstellung wahrscheinlich effizienter.

6.2 Zukünftige mögliche Anwendungen und Weiterführungen

Der WFC-Algorithmus ist ein Algorithmus der prozeduralen Generierung und daher ist eine gewisse Zufälligkeit der Ergebnisse immer zu erwarten. Der Algorithmus kann jedoch durch besprochene und gezeigte Anpassungen mit einem geeigneten Tileset viele Anforderungen erfüllen. Vor allem die Erkenntnisse über das Verwenden von Gewichtungen sowie den Methoden zum erzeugen verbundener Pfade könnte in der Generierung von Dungeons, sowie anderen Anwendungen Vorteile schaffen. Dabei können Gewichte in Anwendungen einen höheren Detailgrad ermöglichen, ohne die Ausgabe mit Details zu überladen. Die Methoden zum Verbinden von Pfaden, insbesondere der Algorithmus der Gruppenkombinierung kann vom WFC-Algorithmus erstellte Konstrukte zusammenfügen. Diese müssen nicht ein Pfad sein, sondern lediglich Objekte welche als Gruppen zusammengefasst verstanden werden können. Beispielsweise kann mit Implementierungen des WFC-Algorithmus zum Ergänzen von Städten in Software zum Erstellen von Stadtplanungen experimentiert werden. Dazu kann der WFC-Algorithmus in angegebenen Bereichen leere Flächen durch Mengen ausgewählter Objekte ergänzen. Dieser kann dabei um bereits eingefügte Objekte herum arbeiten, da der WFC-Algorithmus mit vorgegebenen Tiles funktionsfähig ist. Zudem kann durch das garantierte Verbinden von Pfaden gewährleistet werden, dass Straßen, Wohnblocks oder Grünflächen verbunden sind. Indem für Tiles solcher Gruppen geeignete Einschränkungen an ihren Kanten eingestellt sind, kann eine realistischere Generierung erzeugt werden, und Gewichte können Details wie Bänke auf Gehwegen in gewünschten maßen zufügen.

Die Verfahren dieser Arbeit versuchen stets eine realistischere Ausgabe zu erzeugen. Mögliche Weiterentwicklungen dieser Arbeit können daher weitere Algorithmen zur Verbindung von Pfaden schaffen oder die in dieser Arbeit entwickelten optimieren. Zudem kann durch eine Kombination aus Gewichten, einer Vielzahl an Tiles mit stark eingeschränkten Kanten, und einer funktionierenden Pfadverbindung versucht werden Ausführungen des WFC-Algorithmus zu schaffen, welche nicht willkürlich aussehende Ausgaben schaffen. Dadurch könnten bestimmte Elemente sehr gruppiert, jedoch stets verbunden platziert werden. Dies kann realistische Anwendungen außerhalb von Spielen, wie Architektursoftwares, dazu bewegen mehr auf den WFC-Algorithmus zur Vervollständigung von entwickelten Objekten, zurückgreifen.

7 Zusammenfassung

In dieser Arbeit wurde der Wave Function Collapse Algorithmus auf seine Eignung zum Erstellen eines Dungeons untersucht. Dazu wurden zunächst Möglichkeiten einen Dungeon zu erzeugen vorgestellt, andere Algorithmen der prozeduralen Generierung gezeigt und der WFC-Algorithmus erklärt. Dabei wurde der Ablauf des Algorithmus aufgezeigt, sowie Vorteile und Nachteile des WFC-Algorithmus erklärt. Im nächsten Kapitel wurde gezeigt, wie ein Dungeon durch den WFC-Algorithmus erstellt werden kann. Dazu wurden zunächst Anforderungen aufgelistet, welche ein erzeugter Dungeon erfüllen sollte und dann, wie diese Anforderungen unter Verwendung des WFC-Algorithmus umgesetzt werden können. Dabei konnten nicht alle Anforderungen ohne Anpassungen des Algorithmus erreicht werden. Es wird erklärt, warum diese Anforderungen nicht ohne Anpassungen umsetzbar sind, sowie anschließend welche Anpassungen entwickelt wurden und wie diese funktionieren. Dabei stellt vor allem der Umstand, dass der WFC-Algorithmus stets Dungeon mit vielen unverbundenen Pfaden erzeugt das Hauptproblem dar. Dieses wird durch entwickelte Verfahren gelöst, welche erklärt werden. Danach wird die Implementierung der Verfahren und Ansätze im Beispielprogramm erläutert und dieses erklärt. Abschließend wird die Frage beantwortet, ob der WFC-Algorithmus zum Generieren von Dungeons geeignet ist, sowie mögliche Übertragungen der entwickelten Verfahren auf andere Anwendung besprochen. Dabei stellt eine mögliche Weiterentwicklung der Ergebnisse dieser Arbeit die Weiterentwicklung und Optimierung der genannten entwickelten Verfahren da, welche eine realistischere Ausgabe des WFC-Algorithmus erstreben. Demnach könnte eine mögliche Kombination verschiedener genannter Ansätze und Verfahren zum erstellen realistischer Ausgaben, den WFC-Algorithmus eventuell attraktiver für realistische Anwendungen außerhalb von Spielen und Texturgenerierungen gestalten.

Literatur

- (n.d.). Retrieved from sldo.github.io/ Procedural Dungeon Genrator:
<https://sldo.github.io/procedural-dungeon/>
- Adam Tuliper. (2014, August). *microsoft.com*. Retrieved from <https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/august/unity-developing-your-first-game-with-unity-and-csharp>
- Boris. (2023). Retrieved from Wave Function Collapse Explained:
<https://www.boristhebrave.com/2020/04/13/wave-function-collapse-explained/>
- Boris. (2023). Retrieved from Wave Function Collapse tips and tricks:
<https://www.boristhebrave.com/2020/02/08/wave-function-collapse-tips-and-tricks/>
- Boris. (2023). Retrieved from Path Constraints Tutorial:
https://www.boristhebrave.com/permanent/20/01/tessera_docs_2/articles/path.html
- Börje Santen. (2015, 08 5). Retrieved from Techniken der Prozeduralen Generierung für Dreidimensionale Dungeons in Videospielen: <https://reposit.haw-hamburg.de/bitstream/20.500.12738/7974/1/BachelorArbeit.pdf>
- Brian Bucklew. (2022). Retrieved from Youtube/ Game Development Conference:
<https://www.youtube.com/watch?v=AdCgi9E90jw>
- Christian Mills. (2021). Retrieved from Notes on Dungeon Generation via WaveFunctionCollapse: <https://christianjmill.com/posts/dungeon-generation-via-wavefunctioncollapse-notes/>
- Donald, M. (2023). *Youtube*. Retrieved from Superpositions, Sudoku, the Wave Function Collapse algorithm.: <https://www.youtube.com/watch?v=2SuvO4Gi7uY>
- Dykeman, I. (n.d.). Retrieved from ijdykeman.github.io:
<https://ijdykeman.github.io/ml/2017/10/12/wang-tile-procedural-generation.html>
- Filip Hedman und Martin Hakansson. (2023). *Representing video game style with procedurally generated content. How wave function collapse can be used to represent style in video games*. Retrieved from <https://www.diva-portal.org/smash/get/diva2:1786164/FULLTEXT01.pdf>
- Hugo Scurti. (2023). Retrieved from hugoscurti /: <https://github.com/hugoscurti/path-wfc>
- Hugo Scurtie, C. V. (2018). Retrieved from Generating Paths with WFC:
<https://arxiv.org/pdf/1808.04317.pdf>
- Maxim Gumin. (2023, 12 2). Retrieved from mxgmn /WaveFunctionCollapse:
<https://github.com/mxgmn/WaveFunctionCollapse>
- Michael F. Cohen, J. S. (2003). Retrieved from uni-konstanz.de/ Wang Tiles for Image and Texture Generation: <https://kops.uni-konstanz.de/server/api/core/bitstreams/f74fb7e3-333f-43ac-91a7-9cbd26527d60/content>
- Oskar Stalberg. (n.d.). Retrieved from <https://oskarstalberg.com/game/wave/wave.html>
- Robert Heaton. (2023). Retrieved from robertheaton.com/ The Wavefunction Collapse Algorithm explained very clearly: <https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/>

Yuhe NIE, S. Z. (2023). *Extended Wave Function Collapse Algorithm to Large.Scale Content Generation*. Retrieved from <https://arxiv.org/pdf/2308.07307.pdf>

Glossar

WFC	(Wave Function Collapse) ist ein einschränkungsbasierter Algorithmus zur prozeduralen Generierung, der von Maxim Gumin entwickelt wurde.
Tile	Eine Tile ist ein rechteckiges Objekt, welches eine bestimmte grafische Information enthält. Beispielsweise kann ein Stück eines Ganges drauf dargestellt sein. Tiles können weitere Informationen wie ihre zugelassenen Nachbartiles enthalten.
Tilemap	Eine Objekt das eine Menge von Tiles speichert/ darstellt.

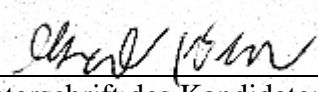
Erklärung des Kandidaten

- ☒ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.
- ☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist im Kapitel „Verantwortliche“ zu Beginn der Dokumentation aufgeführt.

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Name der Mitverfasser:
.....

05.12.2023
Datum



Unterschrift des Kandidaten