Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Bachelor-Thesis zur Erlangung des akademischen Grades B.Sc.

Synthese generierter und handgebauter Welten mittels WaveFunctionCollapse

---

Kevin Hagen
Matr.-Nr.: 2270985

Erstprüfer: Prof. Dr.-Ing. Sabine Schumann
Zweitprüfer: Prof. Gunther Rehfeld

Hamburg, 02.04.2019

Inhaltsverzeichnis

Abstract

The aim of this bachelor thesis was to test the WaveFunctionCollapse algorithm with regard to its suitability for the generation of procedural worlds and to highlight its eligibility as a tool for game and level designers. Existing traditional algorithms as well as level design fundamentals were investigated. In addition, a level generator based on WFC was developed. Afterwards, the insights gained at the beginning were used to evaluate the results of the generator and the suitability of the algorithm for the synthesis of generated and handcrafted worlds.

It has been shown that the WFC definitely has potential for use in real production. Furthermore, it was found that an implementation of the SimpleTiled Model is better suited than the Overlapping Model.

The thesis is aimed at both beginners and advanced students in dealing with the topic of PCG.

Zusammenfassung

Das Ziel der vorliegenden Bachelorarbeit war es, den WFC-Algorithmus bezüglich seiner Fähigkeit zur Generierung prozeduraler Welten zu prüfen und herauszustellen, inwiefern er sich als Tool für Game- und Leveldesigner eignet. Dabei wurden sowohl bestehende, traditionelle Algorithmen als auch Leveldesign Grundlagen untersucht. Außerdem wurde ein eigener, auf WFC basierender Levelgenerator entwickelt. Danach wurden die eingangs gewonnenen Erkenntnisse genutzt, um die Ergebnisse des Generators und die Eignung des Algorithmus zur Synthese generierter und handgebauter Welten zu bewerten.

Es hat sich gezeigt, dass der WFC definitiv Potenzial zum Einsatz in einer realen Produktion besitzt. Außerdem wurde festgestellt, dass sich eine Implementierung des SimpleTiled Models besser eignet, als das Overlapping Model.

Die Thesis richtet sich sowohl an Einsteiger als auch an Fortgeschrittene im Umgang mit der Thematik der PCG.

1 Vorwort

1.1 Motivation

Die Spieleindustrie ist so stark gewachsen, dass täglich eine Vielzahl verschiedener Spiele auf dem Markt erscheint. Derartiger Wettbewerb sorgt dafür, dass die Anforderungen an ein erfolgreiches Computerspiel steigen. Von grafischer und ästhetischer Qualität, über innovatives Gameplay, bis hin zu abwechslungsreichem Inhalt gibt es alle möglichen Skalen, auf denen sich Spiele heutzutage messen. Letzteres Kriterium ist besonders spannend, da abwechslungsreicher Inhalt dafür sorgen kann, dass sich ein spielerisch immer gleiches Spiel bei jedem Durchlauf anders für den Spieler anfühlt. Gibt es immer (oder zumindest oft genug) unterschiedliche Waffen, Gegner, Welten oder diverse andere Spielinhalte, so kann der gleiche Core-Gameplay-Loop jedes Mal ein anderes Erlebnis vermitteln [Lee, 2014; Portnow, 2015].

Die Erstellung grafischer Assets, ausbalancierter Gegenstände und spielbarer Welten, ist jedoch ein umfangreicher und kostspieliger Prozess. Häufig wird ein riesiges Entwicklerteam und/oder sehr viel Zeit für ein einzelnes Projekt benötigt. Der aktuelle Produktionszyklus ist allerdings eher kurzlebig. Meist haben auch weder Triple-A Firmen noch Indie-Entwickler die Zeit und das nötige Budget, um eine solche Vielzahl an Inhalten manuell zu erstellen [Lambe, 2012; Portnow, 2015]. Deshalb wird oft auf die prozedurale Contentgenerierung (PCG) zurückgegriffen, bei der die Erstellung bestimmter Spielinhalte von einem Algorithmus übernommen wird [Lee, 2014].

Die Programmierung dieser Algorithmen ist eine faszinierende Aufgabe, da sie den Programmierer vor immer neue Herausforderungen stellen. Die Anforderungen an die Generierung variieren von Spiel zu Spiel, es gibt also kein einheitliches Rezept. Das Balancing und die Spielbarkeit, sowie weitere Rahmenbedingungen des Spiels müssen im Auge behalten werden. Deshalb genügt es nicht, zufällig oder pseudo-zufällig Werte und Gegenstände zu generieren [Portnow, 2015]. Gerade bei der Levelgenerierung entstehen oft sichtbar zufällige Landschaften bzw. Umgebungen. Das ist keine zwangsläufig schlechte Eigenschaft, kann aber deutlich die Immersion schwächen [Portnow, 2015]. Um dem entgegenzuwirken, versuchen einige Spiele sich an der Synthese handgebauter und prozeduraler Welten.

1.2 Ziel und Struktur der Arbeit

Diese Arbeit beschäftigt sich hauptsächlich mit der Synthese generierter und handgebauter Level. Es wird analysiert, wie Spiele dies aktuell lösen und inwiefern sich der Algorithmus *WaveFunctionCollapse (WFC)* eignet, um Game- und Leveldesignern mehr Kontrolle über den Prozess der prozeduralen Generierung zu bieten. Ziel ist die prototypenhafte Entwicklung eines auf WFC basierenden 3D-Levelgenerators mit integrierten Tools für einen Leveldesigner.

Da sich diese Arbeit sowohl an Fortgeschrittene als auch Einsteiger im Umgang mit prozeduraler Generierung und Leveldesign richtet, widmet sich der erste Teil der Thesis einer Einführung in beide Themen. Dafür werden erst grundlegende Begriffe der PCG definiert und das prägende Spiel *Rogue* betrachtet. Um die PCG besser einordnen zu können, folgt eine Vorstellung spiele- und nicht spielebezogener Einsatzbereiche, sowie dreier häufig verwendeter Techniken. Das Kapitel endet mit einer Pro-Kontra-Analyse über den Einsatz von PCG in einer Spieleproduktion. Das dritte Kapitel führt die Grundlagen der *Constraint Satisfaction Probleme (CSP)* und den WFC-Algorithmus ein. Im Anschluss wird die Levelgenerierung als Constraint Satisfaction Problem (CSP) formuliert, bei dem WFC als Solver dient. In Kapitel vier werden Leveldesign Prinzipien aufgestellt, anhand derer beispielhaft drei Spiele analysiert werden, die generierte und handgebaute Elemente synthetisieren. Auf Basis dieser Erkenntnisse werden die Anforderungen für den eigens entwickelten Generator aufgestellt. Das fünfte Kapitel stellt das zugrundeliegende Konzept des entwickelten Levelgenerators, *KHollapse*, vor. In Kapitel sechs folgen Details zur Implementierung und Erläuterungen zu Abweichungen vom Konzept. Im letzten Kapitel wird die entwickelte Software mit den in Kapitel drei aufgestellten Anforderungen abgeglichen, um die Leitfrage der Thesis zu beantworten. Das Ende bildet ein Fazit über die gewonnenen Erkenntnisse.

1.3 Danksagung

Ein besonderer Dank gilt allen Korrekturleser/innen, die meine Arbeit immer wieder geduldig und aufmerksam auf Fehler geprüft und zahlreiche Fragebögen zur Verbesserung meiner Arbeit beantwortet haben. Weiterhin möchte ich mich bei Maxim Gumin für die persönliche Korrespondenz und Unterstützung bedanken. Weiterhin möchte ich Jason Coffi für die Anfertigung von 3D-Modellen für diese danken.

Diese Arbeit ist meinen Eltern gewidmet, die mich stets unterstützt und ermutigt haben, meine Ziele und Träume zu verfolgen. Meinem Vater, der meine Leidenschaft zu Videospielen und IT teilte und mich bereits in jungen Jahren an die faszinierende Technik der Computer heranführte, gebührt ein besonderer Dank. Auch wenn du diese Arbeit aufgrund deines plötzlichen Ablebens nicht mehr lesen kannst, weiß ich, wie stolz du auf mich wärst.

2 Einführung in die prozedurale Contentgenerierung

Dieses Kapitel dient dazu, einen Überblick über die PCG im Allgemeinen zu schaffen. Zu Beginn wird der Begriff erklärt und definiert, um ihn im weiteren Verlauf der Thesis zu nutzen. Als nächstes wird erläutert, welche Relevanz das Spiel *Rogue* für die PCG hat und wie aus ihm ein eigenes Spielgenre wurde. Der nachfolgende Abschnitt setzt sich mit den Einsatzgebieten der PCG auseinander. Im Zuge dessen werden einige interessante Anwendungsbeispiele genannt, die für die Generierung von Welten aber nicht zwangsläufig nötig sind. Sie dienen lediglich dem besseren Verständnis. Im Anschluss werden häufig genutzte Techniken erklärt, um das Wissen zu vertiefen. Abschließend folgt eine Pro-Kontra-Analyse über den Einsatz von PCG in der Produktion.

2.1 Begriffserklärung und Definition

Neben dem Begriff PCG taucht im deutschsprachigen Raum der Begriff der „prozeduralen Synthese“ auf. Beide Begriffe werden oftmals synonym füreinander verwendet. Im Laufe dieser Arbeit wird der Einheitlichkeit halber allerdings nur der Begriff PCG verwendet.

Zur Analyse der Wortbedeutung ist der als zweites genannte Begriff allerdings hilfreich. Prozedural heißt laut *duden.de*: „verfahrensmäßig; den äußeren Ablauf einer Sache betreffend“ [Duden Online, 2018b]. Das Wort Synthese hat seinen Ursprung im Spätlateinischen (synthesis) und taucht später im altgriechischen Wortschatz als „syntithénai“ auf. Es besteht aus den beiden Silben „sýn“ (zusammen) und „tithénai“ (setzen, stellen, legen) und bedeutet soviel wie „zusammensetzen, -stellen, -fügen“ [Duden Online, 2018c]. Die prozedurale Synthese beschreibt demnach ein Verfahren bzw. einen Ablauf, wie „Etwas“ zusammengefügt wird.
Unter Berücksichtigung des Begriffs der PCG, kann das Wort „Content“ mit in Betracht gezogen werden, welches übersetzt „qualifizierter Inhalt“ bedeutet [Duden Online, 2018a]. Zusammen mit den vorigen Kenntnissen lässt sich daraus schlussfolgern, dass das weiter oben erwähnte „Etwas“ qualifizierten Inhalt meint. Prozedurale Synthese, bzw. PCG, ist dementsprechend die verfahrensmäßige Zusammenstellung (Generierung) qualifizierter Inhalte.

In der Informatik können diese Inhalte die verschiedensten Entitäten sein:

- Texturen
- 3D Modelle
- Sounds/Musik
- virtuelle Welten
- Gegenstände (bzw. ihre Eigenschaften)
- u.v.m. [Beca, 2017; Lee, 2014]

Ein zusätzliches Kriterium der PCG ist die Generierung der Inhalte ohne manuelle Einflüsse. Stattdessen wird sie von einem Algorithmus übernommen.

> „In general computing terms, procedural generation is any technique that creates data algorithmically as opposed to manually. It may also be called random generation, but this is an over-simplification: although procedural algorithms incorporate random numbers, they are never truly random, or at least not as random as that term implies.“ [Beca, 2017]

In diesem Zitat von S. Beca wird weiterhin deutlich, dass Pseudozufallszahlengeneratoren (engl. Pseudorandom number generator (PRNG)) relevant sind.[1] Bereits wenige Parameter sollen dafür sorgen, dass eine große Menge unterschiedlicher Inhalte generiert werden kann [Lee, 2014].

Neben diesen Definitionen gibt es noch die Idee, prozedurale Generierung von PCG abzugrenzen. Die Internetseite „Procedural Content Generation Wiki“ schreibt dazu:

> „This wiki uses the term procedural content generation as opposed to procedural generation: the wikipedia definition of procedural generation includes using dynamic as opposed to precomputed light maps, and procedurally generated textures, which while procedural in scope, do not affect game play in a meaningful way.“ [Doull, 2014]

Die Unterscheidung besteht darin, inwiefern sich generierte Inhalte auf das Gameplay auswirken. Für die Community der Wiki heißt dies, dass beispielsweise die Generierung von Texturen zur Laufzeit keine PCG ist, weil sie lediglich eingesetzt wird, um Speicher zu sparen und nicht, um das Gameplay zu beeinflussen. Diese Unterscheidung ist für den weiteren Verlauf der Arbeit nicht relevant, sei hier aber der Vollständigkeit halber erwähnt. Interessierte Leser finden auf der genannten Seite weitere Informationen zu der Unterscheidung.[2]

[1] PRNG generieren keine echten Zufallszahlen im Sinne der Informatik, weil sie auf einem sogenannten Seed (Ausgangswert) basieren. Mit dem gleichen Seed wird die gleiche Sequenz an Pseudo-Zufallszahlen erzeugt.

[2] Beispielsweise auf: `http://pcg.wikidot.com/pcg-algorithm:procedural-generation`

2.2 Rogue und sein Einfluss auf die PCG

Das Computerspiel *Rogue: Exploring the Dungeons of Doom*, kurz genannt *Rogue*, wurde ursprünglich Anfang der Achtzigerjahre von Michael Toy und Glenn Wichmann entwickelt. Im Laufe der Zeit stießen noch Jon Lane und Ken Arnold zum Team dazu [Wichmann, 1997].[3] Das Gameplay besteht darin, durch einen Kerker zu laufen, in dem etliche Monster auf den Spieler lauern. Ziel ist es, die Monster zu besiegen und den letzten Korridor zu erreichen, um dort das magische Amulett Yendors zu bergen. Grafisch basiert das Spiel auf ASCII-Zeichen. Es wird über Tastatureingaben, sowie einige zusätzliche Kommandos gesteuert. Der Ablauf ist komplett rundenbasiert, das Verstreichen echter Zeit spielt keine Rolle. Eine weitere Besonderheit des Spiels stellt der permanente Tod (engl. Permadeath) dar [EPYX Inc., 1985].[4]

Abbildung 2.1: Typisches Level aus dem Spiel *Rogue* [Barton and Loguidice, 2009]

Abbildung 2.1 zeigt ein klassisches Level aus dem Spiel. Der Held wird durch das @-Zeichen repräsentiert. Ihm gegenüber steht ein Kobold, symbolisiert durch ein *K*. Sie befinden sich gerade in einem Kampf miteinander [Barton and Loguidice, 2009].

Damalige Adventure-Games lieferten stets den gleichen, statischen Inhalt und boten insbesondere für die Entwickler keine Überraschungen. Dies brachte die beiden auf die Idee, die Spielwelt prozedural generieren zu lassen:

> „We decided that with Rogue, the program itself should 'build the dungeon', giving you a new adventure every time you played, and making it possible for even the creators to be surprised by the game."
> [Wichmann, 1997]

[3] Der Artikel *A Brief History of Rogue* gibt weitere Auskunft über den Verlauf der Entwicklung.
[4] Stirbt der Spieler, so verliert er seinen Charakter und den daran gebundenen Fortschritt.

Populär wurde das Spiel, als es ein Teil der Version 4.2 der BSD UNIX Distribution wurde. Sie war der damalige Standard an Universitätscomputern, was dem Spiel dazu verhalf sich wie ein Lauffeuer auf diversen Campussen zu verbreiten.

> „Over the next 3 years, Rogue became the undisputed most popular game on college campuses.“ [Wichmann, 1997]

Obwohl *Rogue* nie kommerziellen Erfolg hatte, hat es bleibende Eindrücke hinterlassen. Der entscheidende Faktor sei die prozedurale Generierung der Welt:

> „But I think Rogue's biggest contribution, and one that still stands out to this day, is that the computer itself generated the adventure in Rogue. Every time you played, you got a new adventure. That's really what made it so popular for all those years in the early eighties.“ [Wichmann, 1997]

Das neue Genre: Rogue-like

Als *Rogue* entwickelt wurde, war die Idee der PCG noch neu.[5] Sowohl Spieler als auch Entwickler waren fasziniert von den Konzepten und Ideen, die das Spiel ineinander vereint. Permanenter Tod, generierte Level, rundenbasierter Spielablauf und viele Weitere. Als logischer Schluss dieser Beliebtheit entstanden weitere Spiele mit gleichen oder ähnlichen Ansätzen, wie z.B. *Moria*, *Hack*, *Larn* oder *Ancient Domains of Mystery* [Barton and Loguidice, 2009]. Neben diesen unmittelbaren Nachfolgern hatte es auch Einflüsse auf einige spätere Spiele, wie z.B. *Blizzards Diablo* [Barton and Loguidice, 2009].

Diese Einflüsse werden heute in einem Genre zusammengefasst, den *Rogue-like*-Games. Es gibt mehrere Versuche zu klassifizieren, was genau ein *Rogue-like* ist. Den wohl Bekanntesten stellt die „Berlin Interpretation“ dar. Darin werden die Elemente des Spiels *Rogue* bezüglich ihrer Relevanz für das Genre eingestuft [Lait, 2008].

Heutige Spiele klassifizieren sich oftmals bereits als *Rogue-like*, wenn sie eine Kombination aus Permadeath und prozedural generierter Welten implementieren.[6] Die Spiele, auf die im Abschnitt 4.3 genauer eingegangen wird, bedienen sich alle der Konzepte dieses Genres und lassen sich daher, je nach Strenge, als *Rogue-like* bzw. *Rogue-lite* klassifizieren. Ihnen allen gemeinsam ist die prozedurale Generierung der Spielwelt. Mit dieser Innovation hat *Rogue* einen entscheidenden Meilenstein gesetzt.

2.3 Aktuelle Einsatzbereiche

Die Vielseitigkeit der PCG macht die Erstellung virtueller Inhalte über Quellcode nicht nur für Informatiker oder Spieleentwickler interessant. In diesem Abschnitt

[5] *Rogue* war nicht das erste Spiel, welches PCG nutzte. Laut dem Guinnes-Buch war es *Elite*

[6] Manchmal wird dafür auch der Begriff *Rogue-lites* verwendet. Da sie nur einige der ursprünglichen Elemente implementieren, stellen sie eine „Lite-Version“ dar.

werden verschiedenste Anwendungsfälle der PCG gezeigt. Die prozedurale Levelgenerierung wird hier nicht erwähnt, sie wird später detailliert thematisiert.

2.3.1 Texturen

Ein bekannter Anwendungsfall für prozedural generierte Texturen ist die Einsparung von Speicherplatz auf der Festplatte. Das 2003 erschienene Spiel *.kkrieger* reduziert die Größe des Builds drastisch, indem es die Texturen nicht per Pixel speichert, sondern über ihre Kreationshistorie. Weiterhin werden die Meshes erst zur Laufzeit aus primitiven Formen, wie Boxen oder Zylindern, erstellt und deformiert. Dadurch erreicht *.kkrieger* eine Build-Size von ca. 96 kB, statt 200-300 MB [StrategyWiki, 2016].

Prozedurale Texturen können z.B. auch als Heightmaps für Terrain dienen. Oft werden sie durch sogenannte Rausch-Funktionen generiert (vgl. Abschnitt 2.4.3).

2.3.2 Vegetation

Die Generierung von Vegetation findet oft unter der Verwendung von *Lindenmayer-Systemen* (auch *L-Systeme*) statt. Sowohl bekannte Spiele als auch Filme machen davon Gebrauch [Shaker et al., 2016]. Oftmals wird dabei auf bestehende, professionelle Software wie *SpeedTree* zurückgegriffen [IDV, 2014a]. *SpeedTree* verleiht Filmen wie *Iron Man 3*, *The Lone Ranger*, *Star Trek: Into Darkness*, *The Great Gatsby*, *The Wolf of Wall Street* und vielen mehr, prozedural generierte Pflanzen und Bäume [CGW Magazine, 2014; Williams, 2015]. Als erster relevanter Auftritt im Bereich Film gilt die Anwendung von *SpeedTree* in dem 2009 erschienenem Film *Avatar*. Da es ursprünglich für die Verwendung in Computerspielen entwickelt wurde, zählen ebenfalls eine Menge prominenter Spiele, wie beispielsweise *The Witcher 3: Wild Hunt* oder *Battlefield 4* zu den Anwendern der Software [Williams, 2015].

Abbildung 2.2: Mit *SpeedTree* generierte Vegetation [IDV, 2014b]

2.3.3 Musik

Auch Musik lässt sich prozedural synthetisieren, obwohl die (sinnvollen) Anwendungsfälle hier seltener sind. Viele prozedurale Soundgeneratoren erzeugen sehr zufällig klingende Musikstücke, weshalb sie oft als erfolglos gelten [Alexander, 2015].

Als erfolgreiches Beispiel gilt der algorithmische Soundtrack zum Spiel *No Man's Sky*. Die Algorithmen, die zur Generierung der Welt verwendet werden, haben auch Einfluss auf die begleitende Hintergrundmusik [Seppala, 2016]. *No Man's Sky* ist allerdings nicht das einzige Beispiel: Ein GitHub Nutzer hat unter dem Pseudonym *pernyblom* das Tool „Abundant Music", einen prozeduralen Musik-Editor, auf seinem Blog veröffentlicht [Nyblom, 2012]. Auf der Video-Plattform YouTube gibt es einige Hörbeispiele.[7]

2.3.4 Gegenstände

In vielen Spielen werden Gegenstände prozedural generiert, d.h. ihre Eigenschaften werden zur Laufzeit zusammengesetzt. Es gibt keine vorige Planung aller möglichen Gegenstände im Spiel, sondern lediglich eine Anzahl möglicher Kompositionen verschiedener Gegenstände. Oftmals geht dies mit der prozeduralen Generierung des Gegenstandnamens einher. Er kann z.B. auf Seltenheitsgrad, Art und Eigenschaften des Gegenstands basieren. Zu jeder Charakteristik des Gegenstands wird zufällig ein Wort gewählt, welches im Vorfeld als Assoziation zu dieser bestimmt wurde. Spiele wie die *Diablo*- oder die *Borderlands*-Reihe implementieren solche Systeme für die Erstellung von Waffen und weiterem Loot[8] (vgl. Abb. 2.3).

(a) Magischer Gegenstand aus Diablo *(b) Waffe in Borderlands*

Abbildung 2.3: Beispiele für prozedural generierte Gegenstände in Spielen [Victusmetuo, 2012; WarBlade, 2009]

[7] Zum Beispiel folgendes: `https://www.youtube.com/watch?v=Iyz_rXXCmqw`
[8] Sammelbare Gegenstände, die z.B. durch das Besiegen von Feinden erlangt werden

2.4 Häufige Techniken zur prozeduralen Generierung

Im Folgenden werden drei PCG-Techniken vorgestellt. Sie werden häufig in verschiedenen Szenarien, aber vor allem in der Spieleentwicklung, eingesetzt. Für jeden dieser Algorithmen wurde eine Demo entwickelt, die dazu dient, den Algorithmus und seine Parameter besser zu verstehen. Sie sind online verfügbar.[9]

2.4.1 Drunkard's Walk

Eine sehr einfache Methode, um Dungeon-ähnliche Strukturen zu generieren, bietet der sogenannte *Random Walk*-Algorithmus. Im deutschsprachigen Raum wird auch von der „stochastischen Irrfahrt" gesprochen [Wikipedia, 2018]. Zur Veranschaulichung wird häufig die Metapher eines Betrunkenen genutzt, der willkürlich für eine bestimmte Zeit in eine Richtung läuft und sie zwischendurch ändert, wodurch diese Algorithmen auch als *Drunkard's Walk* bekannt sind [Wikipedia, 2018].

Ein einfacher, eindimensionaler *Drunkard's Walk* besteht aus n zufälligen Schritten in einer Dimension [Wikipedia, 2018]. Die Wahrscheinlichkeit vorwärts zu gehen, beschreibt dabei p, entsprechend ist $q = 1 - p$, die Wahrscheinlichkeit rückwärts zu gehen. Da die Wahrscheinlichkeit der Schritte nicht von vorigen Schritten abhängig ist, bilden sie einen Bernoulli-Prozess.

Für die Generierung zweidimensionaler Dungeons kann der Algorithmus folgendermaßen adaptiert werden [Read, 2014]:

1. Initialisiere ein mit Wänden gefülltes Level-Gitter.
2. Platziere einen Agenten (Tunneler) auf einer zufälligen Position des Gitters, markiere diese als Boden.
3. Wähle zufällig eine der vier Haupthimmelsrichtungen ($n = s = e = w = \frac{1}{4}$) und bewege dich dorthin.
4. Markiere das neue Feld als Boden.
5. Wiederhole Schritt 3 und 4, bis die gewünschte Menge des Dungeons ausgegraben wurde.

Diese Variante garantiert einen vollkommen verbundenen Level, da sie auf einem einzigen Walker mit Schrittlänge = 1 basiert. Außerdem enthalten die Resultate einen variablen Mix schmaler Pfade und offener Räume [Read, 2014]. Der Algorithmus stellt eine Menge Parameter bereit (Wahrscheinlichkeiten für jede Richtung, Levelgröße, Füllrate, ...) und kann außerdem durch Methoden wie den *Lévy Flight*[10] variiert werden [Shiffman, 2012].

[9] `https://kevinhagen.github.io/ba_thesis/pcgdemos/`

[10] *Drunkard's Walk* mit variabler Schrittgröße; modelliert beispielsweise das Verhalten eines Raubvogels bei der Futtersuche

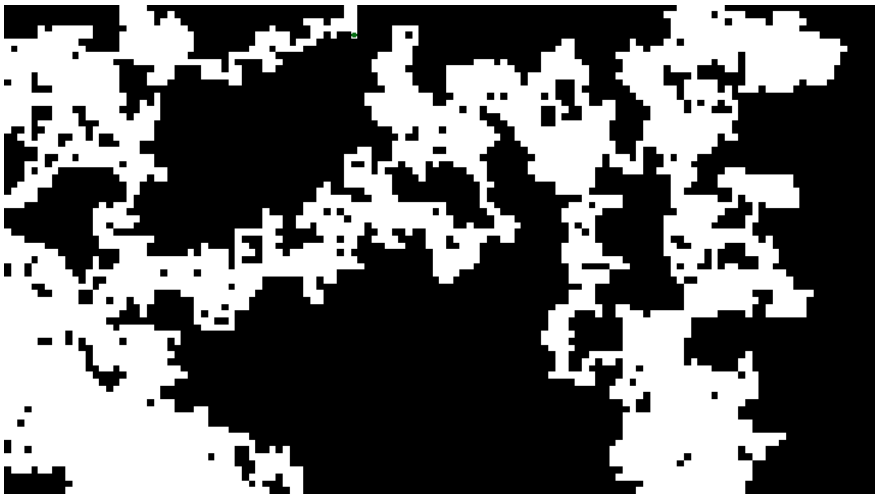


Abbildung 2.4: Beispielhafter Dungeon generiert mit *Drunkard's Walk*, eigene Grafik

2.4.2 Zelluläre Automaten

Zelluläre Automaten (ZA) dienen zur Simulation und Modellierung von Prozessen, wie beispielsweise der Verbreitung von Schadstoffen oder Epidemien. Trotz ihrer simplen Struktur können sie komplexes, dynamisches Verhalten widerspiegeln [Shiffman, 2012]. In der Spieleentwicklung finden sie vor allem für die Generierung höhlenartiger Systeme Anwendung [TheSheep, 2016]. Sie wurden 1940 erstmals von *Stanislaw M. Ulam* und *John von Neumann* vorgestellt. Später entwickelte *John H. Conway* das auf ZA basierende *Game of Life*.[11] Die modernsten Forschungsansätze stammen aus *Stephen Wolframs* 2002 erschienenem Buch *A New Kind of Science* [Shiffman, 2012].

ZA haben folgende Eigenschaften [Shiffman, 2012]:

- n-dimensionaler Zellularraum R, einer (unendlichen) Menge geordneter Zellen C
- Endliche Nachbarschaft N, typischerweise die angrenzenden Zellen
- Endliche Zustandsmenge Q, wobei oftmals $Q = \{0, 1\}$
- Übergangsfunktion $\delta : Q_{zelle} \times Q_{nachbar1} \times Q_{nachbar2} \cdots \times Q_{nachbar\ n} \rightarrow Q_{zelle}$

Der einfachste ZA ist eindimensional, hat die Zustände $Q = \{0, 1\}$ und die Nachbarschaft beschreibt eine Zelle auf dem Gitter, sowie die beiden direkt angrenzenden Zellen. Die Zustände heißen oft auch „tot“ oder „lebendig“. Der Zustand aller Zellen eines ZA zum Zeitpunkt t heißt Konfiguration. Bei ZA heißt eine Konfiguration Generation und beschreibt mit Gen_t die Konfiguration zum Zeitpunkt t [Shiffman, 2012]. Tabelle 2.1 zeigt einen einfachen ZA in einer zufälligen Anfangskonfiguration.

[11] Eine Simulation künstlichen Lebens auf einem zweidimensionalem Gitter

0	0	1	0	1	1	0	1

Tabelle 2.1: Zufällige Ausgangskonfiguration Gen_0 eines ZA [Shiffman, 2012]

Da die Nachbarschaft einer Zelle von sich selbst und zwei weiteren Zellen abhängt, gibt es für sie genau acht Zustände, die sich binär darstellen lassen:

000 001 010 011 100 101 110 111

Tabelle 2.2: Binärdarstellung aller acht Zustände [Shiffman, 2012]

Für die Übergangsfunktion Q braucht jeder Zustand ein definiertes Ergebnis, z.B.:

000	001	010	011	100	101	110	111
↓	↓	↓	↓	↓	↓	↓	↓
0	1	0	1	1	0	1	0

Tabelle 2.3: Zuweisung für die Übergangsfunktion Q [Shiffman, 2012]

Mit dieser Regel kann die Konfiguration Gen_0 aus Tabelle 2.1 in die Generation Gen_1 transferiert werden. Wichtig ist, dass die Zustände der Zellen in der nächsten Generation von denen der jetzigen Generation abhängig sind. Es darf also kein Zustand geändert werden, bevor die neuen Zustände aller Zellen bekannt sind. Nachbarn am Rand des Gitters werden als „tot“ gewertet. [Shiffman, 2012]:

Gen_0	0	0	1	0	1	1	0	1
Gen_1	0	1	0	0	1	1	0	0

Tabelle 2.4: Übergang von einer Generation zur nächsten [Shiffman, 2012]

Mit acht Zuständen lassen sich $2^8 = 256$ verschiedene Regeln bilden. Nicht alle generieren sinnvolle Resultate. Jede Regel ist dabei, wie in Tabelle 2.3, durch eine acht Bit Binärzahl darstellbar. Da diese Regeln nach ihrer Dezimalzahl benannt werden, heißt die oben angewendete Regel „Regel 90“. Es wird i.d.R. mit einer Anfangskonfiguration gestartet, bei der die mittlere Zelle „lebendig“ und alle anderen „tot“ sind. Bei der Anwendung von „Regel 90“, entsteht das *Sierpiński Dreieck* [Shiffman, 2012].

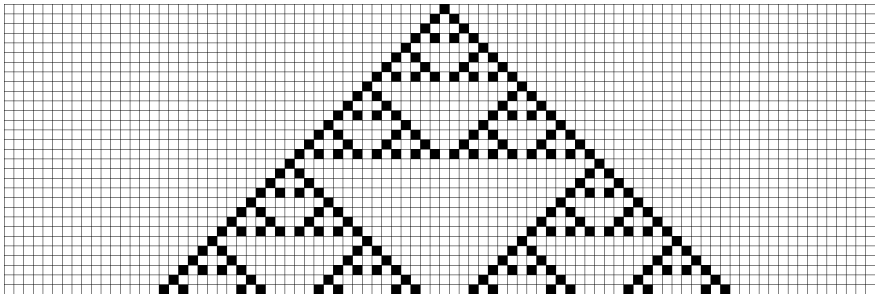
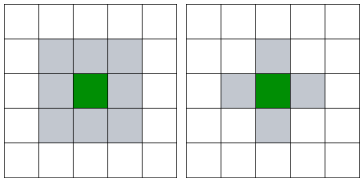


Abbildung 2.5: *Sierpiński Dreieck* nach 30 Generationen (1D-ZA in 2D-Visualisierung) [Shiffman, 2012]

Für zweidimensionale ZA bleibt das Prinzip das Gleiche, aber die Definition der Nachbarschaft ändert sich. Die Häufigsten sind die, in Abb. 2.6 illustrierten, *Moore*- und *Von Neumann*-Nachbarschaften. Auch hierfür können unterschiedliche Übergangsfunktionen definiert werden [Shiffman, 2012].



(a) *(b)*

Abbildung 2.6: *Moore*- (a) und *Von Neumann*-Nachbarschaft (b), eigene Grafik

In der Spieleentwicklung werden am häufigsten zweidimensionale ZA zur Generierung von Höhlen verwendet. Die Anfangskonfiguration wird zufällig mit Wand- bzw. Boden-Tiles[12] gefüllt. Dann wird schrittweise die *Moore*-Nachbarschaft mit einer 4-5 Regel angewendet. Konkret heißt dies, dass ein Tile zu einer Wand wird, wenn es entweder schon eine Wand ist und vier oder mehr Nachbarn auch Wände sind oder es noch keine Wand ist und fünf oder mehr Nachbarn Wände sind. Jede Iteration gleicht die Tiles dabei seinen Nachbarn an, sodass das zufällig generierte „Rauschen" reduziert wird [TheSheep, 2016]. ZA generieren jedoch häufig voneinander isolierte Höhlen, weshalb es eines zusätzlichen Algorithmus' bedarf, der für die Verbundenheit der Räume sorgt (bspw. Flood Fill) [TheSheep, 2016].

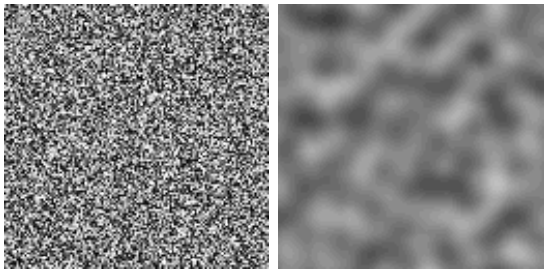
Neben diesem Ansatz können ZA stark modifiziert werden, Abschnitt 4.3.2 liefert hierfür ein Beispiel. Weitere Variationsmöglichkeiten schlägt *Shiffman* in Kapitel 7 seines Buches *The Nature of Code* vor [Shiffman, 2012].

Abbildung 2.7: Durch ZA generierte Höhle mit mehreren unverbundenen Abschnitten, eigene Grafik

[12]Ein Tile ist eine grafische Repräsentation von Raumgeometrie

2.4.3 Perlin Noise

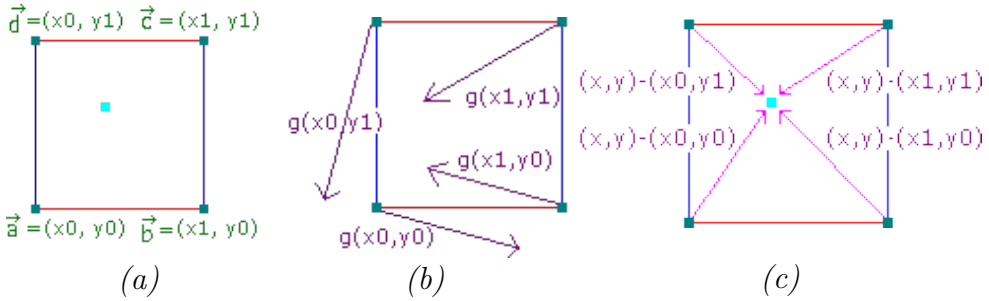
Perlin Noise ist eine kohärente Rausch-Funktion, die ursprünglich von *Ken Perlin* entwickelt wurde, um Texturen zur Laufzeit zu erstellen [Perlin, 1999]. Kohärent heißt dabei, dass die generierten Werte jeweils im Zusammenhang zu ihren umliegenden Werten stehen [Zucker, 2001]. Dies sorgt für weichere Übergänge. Es eignet sich zur Synthese von Feuer-, Wasser-, Holz-, Marmor-, oder Wolkentexturen, zur Erstellung dreidimensionalen Terrains, sowie vieler anderer Zwecke [Perlin, 1999]. Abbildung 2.8 zeigt klassisches Rauschen und *Perlin Noise* im Vergleich [Zucker, 2001].



(a) *(b)*

Abbildung 2.8: Nicht-kohärentes Rauschen (a) vs. *Perlin Noise* (b) [Zucker, 2001]

Gesucht ist der Rauschwert r eines Punktes $\vec{p} = \left(x; y\right)$ auf einem Gitternetz mit integralen[13] Koordinaten. Für alle Punkte $(x; y)$ gilt: Liegen sie innerhalb eines Quadrates Q_{xy} im Koordinatensystem, dann sind jeweils $x, y \in \mathbb{R} \setminus \mathbb{Z}$, ansonsten $x, y \in \mathbb{Z}$ [Zucker, 2001]. Für die Berechnung wird $\vec{p}$ zunächst in seinem Quadrat Q_{xy} lokalisiert. Die vier Eckpunkte von Q_{xy} werden als $\vec{a} = \left(x_0; y_0\right)$, $\vec{b} = \left(x_1; y_0\right)$, $\vec{c} = \left(x_1; y_1\right)$ und $\vec{d} = \left(x_0; y_1\right)$ bezeichnet (vgl. Abb. 2.9a). Über eine Gradienten-Funktion $G(\vec{x})$ wird jedem Eckpunkt ein Gradientenvektor $\vec{g}(x_a; y_b)$ mit der Länge 1 zugewiesen (vgl. Abb. 2.9b). Danach wird von jedem Eckpunkt ein Vektor zu $\vec{p}$ berechnet, was durch die Subtraktion des jeweiligen Punktes von $\vec{p}$ geschieht (vgl. Abb. 2.9c) [Zucker, 2001].



(a) *(b)* *(c)*

Abbildung 2.9: Zuerst wird $\vec{p}$ lokalisiert (a), dann allen Eckpunkte von Q_{xy} ihre Gradientenvektoren zugewiesen (b) und anschließend die Vektoren zwischen den Eckpunkten und $\vec{p}$ berechnet (c) [Zucker, 2001]

[13] ganzzahlig

Die Gradienten-Funktion $G(\vec{x})$ bestimmt für ein beliebiges, ganzzahliges Koordinatenpaar $(x_a; y_b)$ einen Gradientenvektor. Dabei wird von einer Menge gleichverteilter Vektoren der Länge 1 auf dem Einheitskreis (Sphäre in 3D) einer pseudo-zufällig ausgewählt. Aus Performanzgründen kann der Gradientenvektor zu jedem möglichen Koordinatenpaar vorberechnet werden. Dafür wird eine Permutationstabelle P angelegt, in der jedem Wert von 0-255 ein anderer Wert des gleichen Intervalls zugewiesen wird. Außerdem wird eine Tabelle G berechnet, die 256 pseudo-zufällige Gradientenvektoren enthält. Nun kann mit $g(i, j) = G[(i + P[j]) \, mod \, 256]$ der Gradientenvektor berechnet werden. Da die Operation $mod \, 256$ auch durch ein bitweises & mit 255 durchgeführt werden kann, bleibt nur das Nachschlagen in einer Lookup-Tabelle. Daraus folgt, dass sich die Gradientenvektoren alle 256 Einheiten pro Dimension wiederholen, allerdings sind diese Wiederholungen sehr unauffällig [Zucker, 2001].

Im nächsten Schritt wird der Einflussfaktor jedes einzelnen Gradientenvektors $\vec{g}(x_a; y_b)$ auf den Punkt $\vec{p}$ berechnet. Dafür wird das Skalarprodukt des Gradientenvektors und des Vektors zwischen seinem assoziierten Eckpunkt zu $\vec{p}$ gebildet [Zucker, 2001]:

$$\begin{aligned} s = G(\vec{a}) \cdot (\vec{p} - \vec{a}) &= \vec{g}(x_0, y_0) \cdot ((x, y) - (x_0, y_0)) \\ t = G(\vec{b}) \cdot (\vec{p} - \vec{b}) &= \vec{g}(x_1, y_0) \cdot ((x, y) - (x_1, y_0)) \\ u = G(\vec{c}) \cdot (\vec{p} - \vec{c}) &= \vec{g}(x_0, y_1) \cdot ((x, y) - (x_0, y_1)) \\ v = G(\vec{d}) \cdot (\vec{p} - \vec{d}) &= \vec{g}(x_1, y_1) \cdot ((x, y) - (x_1, y_1)) \end{aligned} \tag{2.1}$$

Wie aus der Gleichung hervorgeht, werden die vier Zahlen s, t, u, v lediglich durch die Gradientenfunktion $G(\vec{x})$ und die Position des Punktes $\vec{p}$ im Quadrat bestimmt. Damit *Perlin Noise* deterministisch für jeden Punkt auf dem Gitter ist, muss $G(\vec{x})$ also keine echt- sondern pseudo-zufälligen Gradientenvektoren liefern [Zucker, 2001].

Damit nun der Rauschwert im Punkt $\vec{p}$ berechnet werden kann, wird der gewichtete Durchschnitt der vier Skalarprodukte gebildet. Die Gewichte werden durch eine Glättungs-Funktion $s(x) = 3x^2 - 2x^3$ bearbeitet, um visuell ansprechendere Ergebnisse zu erzielen [Zucker, 2001]. Sie gibt für $0 <= x <= 1$ Werte im Intervall $[0, 1]$ aus. Dabei wird aus dem Input x generell ein Wert, der näher an 0 bzw. 1 liegt als vorher, außer bei $x = 0.5$ [Zucker, 2001]. Da der Punkt $\vec{p}$ innerhalb des Quadrates Q liegt, ergeben $x - x_0$ und $y - y_0$ immer einen Wert zwischen 0 bis 1, sie sind somit als Input geeignet. Anhand dieser beiden Inputs werden die Gewichte S_x und S_y berechnet:

$$\begin{aligned} S_x &= 3 * (x - x_0)^2 - 2 * (x - x_0)^3 \\ S_y &= 3 * (y - y_0)^2 - 2 * (y - y_0)^3 \end{aligned} \tag{2.2}$$

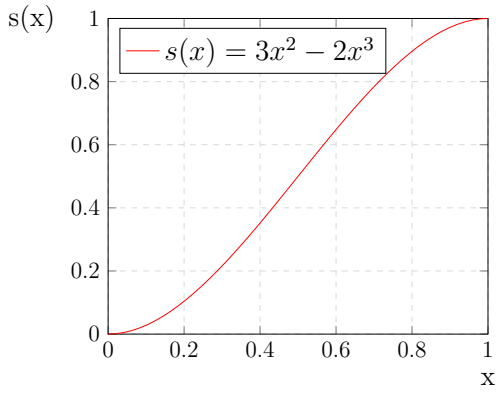


Abbildung 2.10: Glättungs-Funktion, eigene Grafik

Sie sind das Gewicht der Skalarprodukte t und v. Die fehlenden Gewichte können mit $1 - S_x$ bzw. $1 - S_y$ berechnet werden. *Ken Perlin* nutzt lineare Interpolation zur Berechnung des gewichteten Durchschnitts [Gustavson, 2005; Zucker, 2001]:

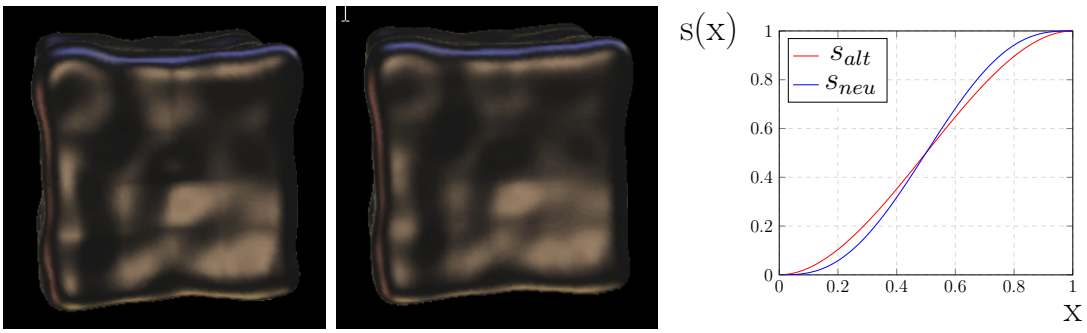
$$\begin{aligned} f(x) &= a * (1 - x) + b * x \\ &= a + x * b - x * a \\ &= a + x * (b - a) \end{aligned} \tag{2.3}$$

$$\begin{aligned} a &= s + S_x * (t - s) \\ b &= u + S_x * (v - u) \\ c &= a + S_y * (b - a) \end{aligned} \tag{2.4}$$

Mit der Formel aus Gleichung 2.3 wird zuerst der Durchschnitt a der Skalarprodukte s und t (Eckpunkte $\vec{a}$, $\vec{b}$), dann der Durchschnitt b von v und u (Eckpunkte $\vec{c}$, $\vec{d}$) und abschließend der Durchschnitt c von a und b gebildet (vgl. Gleichung 2.4). Dabei ist c der gesuchte Rauschwert r am Punkt $\vec{p}$ [Zucker, 2001]. Diese Berechnung wird für alle möglichen Punkte eines 2D-Gitters durchgeführt, um eine 2D-Noisemap zu erhalten. Mit ihr kann z.B. Terrain generiert werden, wobei die Höhe des Rauschwertes die Höhe des Terrains beschreibt [Lague, 2016]. Des Weiteren kann *Perlin Noise* in n-Dimensionen angewandt werden. Anstelle von vier umliegenden Gitterpunkten werden 2^n berücksichtigt und die Anzahl der zu berechnenden, gewichteten Summen ist $2^{(n-1)}$. Die Komplexität des Algorithmus beträgt $O(2^{(n-1)})$, weshalb er ca. ab der 5. Dimension unperformant wird.

Verbesserungen des originalen Perlin Noise

Im Jahr 2002 publizierte *Ken Perlin* über ACM[14] das Paper *Improving Noise*, worin er zwei Verbesserungsvorschläge zu *Perlin Noise* gibt. Die Glättungs-Funktion $s(x) = 3x^2 - 2x^3$ hat in der zweiten Ableitung, $s''(x) = 6 - 12x$, weder bei $x = 0$ noch bei $x = 1$ eine Nullstelle. Dies führte zu Diskontinuitäten im Shading (vgl. Abb. 2.11a). Die Funktion $s(x) = 6x^5 - 15x^4 + 10x^3$ erfüllt diese zusätzliche Bedingung und entfernt somit die visuellen Artefakte (vgl. Abb. 2.11b & c) [Perlin, 2002].



(a) *(b)* *(c)*

Abbildung 2.11: Ergebnisse der alten (a) und neuen (b) Glättungs-Funktion (c) im Vergleich [Perlin, 2002]

[14]Eine Non-Profit-Organisation, die regelmäßig computerwissenschaftliche Paper veröffentlicht

Eine zweite Änderung betrifft die Wahl der Gradientenvektoren. Sie sind, wie bereits erwähnt, auf einer Einheitssphäre gleichverteilt. Da das kubische Gitter eines Würfels aber entlang seiner Achsen eine andere Länge hat, als entlang seiner Diagonalen, tendiert dies dazu, an einigen Stellen zu verklumpen und führt somit anomal hohe Werte herbei [Perlin, 2002]. Deshalb nutzt *Perlin* folgendes Set von Vektoren:

(1,1,0)	(-1,1,0)	(1,-1,0)	(-1,-1,0)
(1,0,1)	(-1,0,1)	(1,0,-1)	(-1,0,-1)
(0,1,1)	(0,-1,1)	(0,1,-1)	(0,-1,-1)
(1,1,0)	(-1,1,0)	(0,-1,1)	(0,-1,-1)

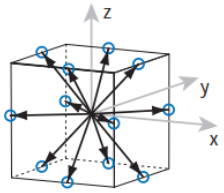


Tabelle 2.5: Zwölf Vektoren (unterste Reihe wiederholt sich) die vom Zentrum eines Würfels zu seinen Kanten zeigen [Gustavson, 2005; Perlin, 2002]

Die Gradienten-Funktion $G(\vec{x})$ arbeitet weiterhin mit einer Look-Up-Tabelle. Durch die Hash-Werte sind sie ausreichend, um zufällig wirkende Resultate zu erzeugen [Perlin, 2002]. Die gleiche Tabelle erzeugt demnach immer gleiche Werte. Für die zweidimensionale Variante können statt der zufälligen Vektoren einfach acht oder 16 auf dem Einheitskreis gleichverteilte Vektoren gewählt werden [Gustavson, 2005].

Eine weitere Verbesserung zum *Perlin Noise* stellt der, ebenfalls von *Ken Perlin* entwickelte, *Simplex Noise*-Algorithmus dar. Er beseitigt beispielsweise die Problematik, dass *Perlin Noise* für alle $x, y \in \mathbb{Z}$ den gleichen Wert ergibt und ist außerdem performanter [Gustavson, 2005].

Weitere Bearbeitung mittels Oktaven

Perlin Noise liefert durch seine Funktionsweise Resultate, die nicht immer detailliert genug sind, um alle Unregelmäßigkeiten der Natur ausreichend abzudecken. Terrain beispielsweise besteht aus großen Bergen, etwas kleineren Hügeln, Felsen, die wiederum kleiner sind und nochmal kleineren Steinen. Diese Details können zwar durch die Amplitude und die Frequenz des *Perlin Noise* gesteuert, aber nicht kombiniert werden. Als Lösung werden mehrere Schichten (Oktaven) *Perlin Noise* übereinandergelegt und addiert. Jede Oktave sollte dabei detaillierter werden. Hierfür werden zwei Parameter eingeführt: Lückenhaftigkeit und Beständigkeit. Die Lückenhaftigkeit steigert die Zunahme der Frequenz von Oktave zu Oktave, wodurch der Detailgrad steigt. Die Beständigkeit beschreibt die Verringerung der Amplitude, also wie stark der Einfluss einzelner Oktaven auf das Gesamterscheinungsbild ist.

Hierbei sei angemerkt, dass die Implementierung von Oktaven nicht dem ursprünglichen Gedanken des Algorithmus entspricht. Es handelt sich dabei lediglich um eine Methode, die Ergebnisse des *Perlin Noise* weiterzuverarbeiten und bessere Resultate zu erzielen.

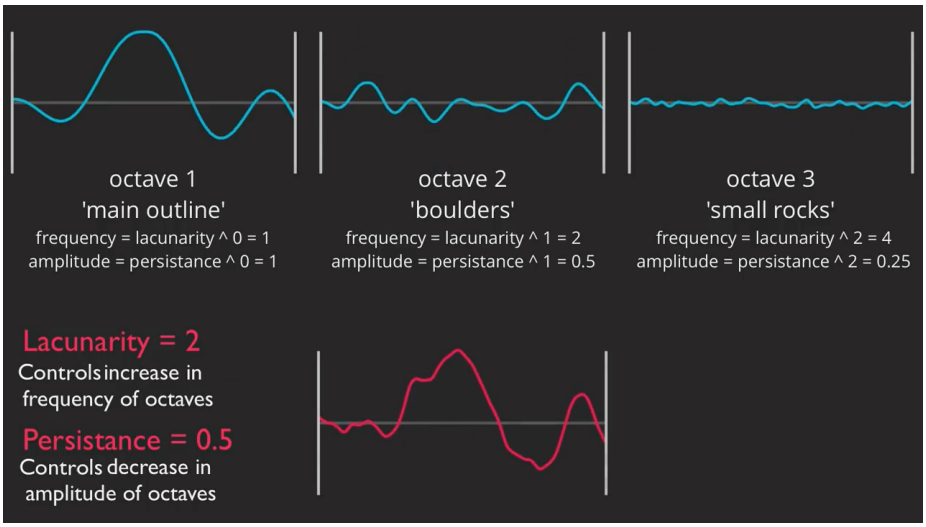


Abbildung 2.12: Drei Oktaven ergeben ein detaillierteres *Perlin Noise* [Lague, 2016]

2.5 Nutzen prozeduraler Contentgenerierung

PCG wird mittlerweile seit über 30 Jahren in diversen Bereichen angewendet (vgl. Abschnitt 2.3). Gerade in der Spieleentwicklung kann sie ein nützliches Tool sein. Im Folgenden werden die Vor- und Nachteile der PCG diskutiert, um ein tiefgehendes Verständnis zu vermitteln, wann die Anwendung eben dieser sinnvoll ist.

Ursprünglich wurde PCG in Spielen wie *Rogue* verwendet, um Festplattenspeicher einzusparen und trotz begrenzter Hardware eine Vielzahl verschiedener Level zur Laufzeit zur Verfügung zu stellen [Hill, 2008]. Auch neuere Spiele, wie das in Abschnitt 2.3.1 erwähnte *.kkrieger*, nutzen diese Technik, um die Build-Size drastisch zu reduzieren. *Dominic Guay*, Technical Director des Spiels *Far Cry 2*, nennt einen weiteren Sparfaktor: „The first big draw of generating procedural content is to save on cost and time“ [Remo, 2008]. Mit einem ausgereiften Generator, beispielsweise für die Erstellung von Leveln, können auf Knopfdruck beliebig viele Entwürfe angefertigt werden, deren Herstellung ansonsten sehr kostspielig wäre. In *Introversion Software's* Spiel *Subversion* wurde ein Generator implementiert, der prozedural Städte generiert. All diese Städte per Hand zu erstellen, hätte laut Schätzung der Entwickler tausende Personenstunden gekostet, die Entwicklung des Generators dauerte jedoch nur eine Woche [Introversion Software, 2007]. Ein guter Generator bietet mehrere Parameter, deren Anpassung seine Resultate maßgeblich verändern kann. Für viele Spiele ist diese Variation entscheidend, um die Replayability[15] zu steigern [Beca, 2017]. *Beca* führt hier als Beispiele die Level-Generierung in *Spelunky* (vgl. Abschnitt 4.3.1) und das Waffensystem aus *Borderlands* (vgl. Abschnitt 2.3.4) an. Als weiteren Pluspunkt sieht

[15]Beschreibt den Wiederspielwert eines Spiels. Spiele mit statischem Content schneiden hier meist schlecht ab, weil nach einmaligem Durchspielen bereits alles bekannt ist.

er, dass prozedurale Techniken auch in Form von Tools nützlich seien. So könne auch ein Spiel, welches letztlich gar keine prozeduralen Gameplay-Features beinhaltet, beispielsweise prozedural Bäume (oder andere Assets) erstellen, die dann per Hand in der Welt platziert werden, wodurch wiederum Zeit und Kosten gespart werden. Ein solches Tool ist z.B. *SpeedTree* (vgl. Abschnitt 2.3.2).
Guay bezeichnet die Möglichkeiten prozeduraler Techniken außerdem als Erschließung neuer Innovationsfelder. Die intensive Auseinandersetzung mit Prozessen und Algorithmen führe zu einem tieferen Verständnis der eigenen Tätigkeiten, was für die Entwicklung interessante Möglichkeiten böte [Remo, 2008].

Da PCG in der Regel sehr programmierlastig ist, verschiebt sich eine große Menge des anfallenden Aufwands vom Art- auf das Programmier-Departement. Je nach Stärken und Schwächen des Entwicklerteams kann dies ein Vor- oder Nachteil sein. Außerdem ist ein Generator, der theoretisch unendlich Variationen liefert, dementsprechend schwer zu testen [Beca, 2017; Hill, 2008]. *Guay* bezeichnet PCG als „Testing-Alptraum“ und teilt eine beispielhafte Erfahrung:

> „If I'm tweaking a jungle procedurally, maybe I'll just tweak it in my test map [...]. But when I integrate it into the game [...] it might cause problems, and we won't find those problems until QA uncovers them.“ [Remo, 2008]

Oft wird fälschlicherweise angenommen, dass ein Generator den Leveldesigner ersetzen kann. Es braucht zwar keinen Designer, der den Level von A-Z plant, allerdings gibt es ausreichend Konfigurationsmöglichkeiten, um das Game-Design zu steuern [Beca, 2017].[16] Diese Parameter können zu ungeahnter Komplexität führen: Das Feintuning teilweise abstrakter Parameter ist nicht immer intuitiv [Remo, 2008]. Die besagte Komplexität sollte auch bei der Entwicklung im Auge behalten werden. Sowohl Programmierer als auch Designer müssen stets gewährleisten, dass der erstellte Inhalt das Spiel nicht unspielbar macht [Beca, 2017]. Ein fataler Fehler wäre z.B. ein Level ohne lösbaren Pfad. Schlecht implementierte PCG führt oftmals zu redundanten Resultaten, die den Spielspaß mindern. *The Elder Scrolls II: Daggerfall* ist hierfür ein anschauliches Beispiel. Es warb mit einer Welt in der zweifachen Größe Großbritanniens, stellte sich durch die Repetitivität jedoch als Enttäuschung heraus [Hill, 2008; Remo, 2008]. Außerdem ist eine handgebaute Welt immer leichter an eine spezielle Atmosphäre bzw. Geschichte anzupassen als ein Generator [Portnow, 2015].

Ob prozedurale Techniken für die eigene Produktion sinnvoll sind, muss also individuell betrachtet werden. Richtig eingesetzt sind sie ein nützliches Werkzeug, um die Produktions-Pipeline zu verbessern und für spannendes, abwechslungsreiches Gameplay zu sorgen. Gleichzeitig birgt PCG auch ganz eigene Problematiken und passt schlichtweg nicht zu jedem Projekt.

[16] Einen guten Überblick über prozedurales Game-Design verschafft *Olaf Bulas'* Bachelor-Thesis mit dem Titel *Linearität in Spielen mit prozeduraler Synthese*

3 Levelgenerierung als CSP

Da der für diese Arbeit entwickelte Generator auf einem Constraint Solver basiert, wird in diesem Kapitel die Levelgenerierung als Constraint Satisfaction Problem betrachtet. CSPs beschreiben Probleme durch Regeln und Verbote auf einer abstrakten Ebene. Constraint Solver können diese ohne domänenspezifisches Wissen lösen. Der erste Abschnitt liefert die nötigen Grundlagen zu CSPs. Er basiert auf *Russell* und *Norvigs* „Artificial Intelligence: A Modern Approach“ (Kapitel 6) [Russell and Norvig, 2016]. Nach der Einführung der Grundlagen folgt eine Erläuterung des WFC-Algorithmus, dem des Generators zugrundeliegenden Solver. Abschließend wird gezeigt, wie dieser Algorithmus zur Levelgenerierung genutzt werden kann.

3.1 Constraint Satisfaction Probleme

3.1.1 Begriffe und Definition

Ein CSP P besteht aus einem Tripel {X, C, D}, wobei:

- X, eine Menge von Variablen X_1 bis X_n und
- C, eine Menge von Constraints C_1 bis C_m, mit C_i = {Reichweite; Relationen}
- Jede Variable X_i besitzt eine nicht leere Domäne D_i, mit einer Menge möglicher Werte für diese Variable. D = Menge aller Domänen.

Jedes Constraint definiert einen Geltungsbereich, bestehend aus einer Teilmenge der Variablen und Relationen, die für sie erlaubten Wertekombinationen. Der Zustand von P wird durch eine Zuordnung von Werten für alle oder einige Variablen $X_i = v_i, X_j = v_j, ...$ beschrieben. Erfüllt eine Zuordnung alle Constraints, heißt sie konsistent. Bei einer vollständigen Zuordnung, wurde jeder Variable ein Wert zugewiesen. Eine vollständige, konsistente Zuordnung zu P heißt auch Lösung von P.

Im „Australian Map“-Problem sollen Australiens Regionen auf einer Karte rot, grün oder blau eingefärbt werden, wobei benachbarte Regionen nicht die gleiche Farbe haben dürfen. Beschreibt man dieses Problem als CSP, sind die Variablen die Regionen {*WA*, *NT*, *Q*, *NSW*, *V*, *SA*, *T*} und haben jeweils die Domäne {rot, grün, blau}. Die Regionen *WA* und *NT* erlauben die Kombinationen (rot, grün), (rot, blau), (grün, rot), (grün, blau), (blau, rot), (blau, grün) oder kürzer $WA \neq NT$. Die Menge aller Constraints ist demnach: {($WA \neq NT$), ($WA \neq SA$), ($NT \neq Q$), ($NT \neq SA$), ($SA \neq Q$), ($SA \neq NSW$), ($SA \neq V$), ($Q \neq NSW$), ($NSW \neq V$)}. Tasmania taucht nicht auf, da es als Insel keine Nachbarregion hat (vgl. Abb. 3.1a). Die Visualisierung des Problems als Graph heißt Constraint-Graph, wobei Knoten die Variablen und Kanten die Constraints repräsentieren (vgl. Abb. 3.1b).

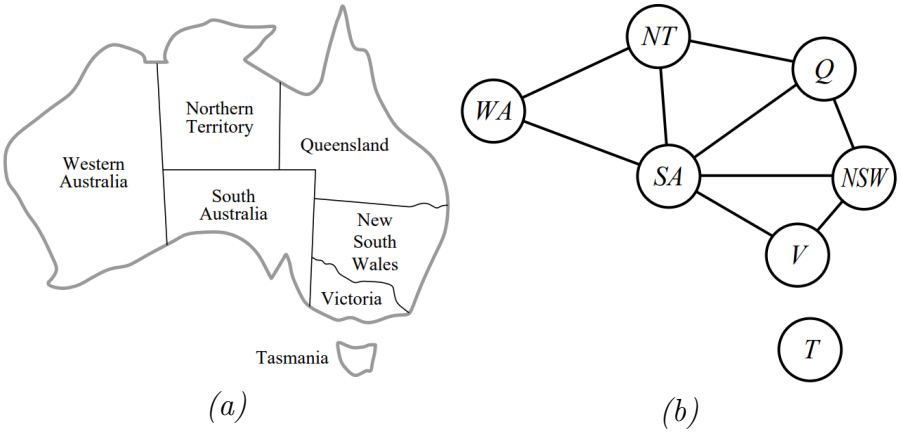


Abbildung 3.1: (a) Karte Australiens mit seinen sieben Bundesstaaten/Territorien und (b) die Repräsentation als Constraint-Graph [Russell and Norvig, 2016]

Die simpelsten CSPs haben diskrete Variablen mit endlichen Domänen, weshalb sie auch *finite Constraint Satisfaction Probleme (FCSP)* heißen. FCSPs beinhalten die sogenannten boolschen CSP, dessen Variablen „true“ oder „false“ sind. Sie umfassen einige NP-vollständige[17] Probleme, wie z.B. 3SAT. Ihr Laufzeitverhalten ist $O(d^n)$, mit $d = maximaleDomänengröße$ und $n = AnzahlVariablen$. Im schlimmsten Fall sind sie dementsprechend nicht in weniger als exponentieller Zeit lösbar. Trotzdem lösen universelle CSP oft komplexere Probleme als universelle Suchalgorithmen.

Constraints werden in unäre, binäre und Constraints höherer Ordnung eingeteilt. Unäre Constraints schränken eine Variable ein ($SA \neq grün$), während binäre zwei in Relation setzen ($SA \neq NT$). Besteht ein CSP nur aus unären und binären Constraints, heißt es binäres CSP und ist als Constraint-Graph darstellbar. Constraints höherer Ordnung involvieren drei oder mehr Variablen, z.B. $Alldiff(SA, NT, WA)$. Dieses Constraint verlangt, dass die drei Regionen alle eine andere Farbe besitzen und besteht aus folgenden binären Constraints: $SA \neq NT$, $SA \neq WA$, $NT \neq WA$. Jedes Constraint höherer Ordnung kann in eine Sammlung binärer überführt werden.

Bisher ging es um harte Constraints, dessen Verstoß zum Ausschluss einer potenziellen Lösung führt. Neben diesen gibt es noch Präferenz-Constraints, die eine bevorzugte Lösung suchen. Zur Veranschaulichung dient die Planung eines Stundenplans: Herr X präferiert Vormittagsunterricht, während Frau Y die Nachmittagsstunden vorzieht. Wird gegen dieses Constraint verstoßen, indem Herr X um 15 und Frau Y um 10 Uhr unterrichten, gilt die Zuordnung als Lösung, ist aber nicht optimal für das Problem. Dies kann als Pfadkosten, z.B. zwei Punkte bei Nachmittags- und einen bei Vormittagsstunden für Herrn X kodiert werden, um Optimierungsprobleme zu lösen.

[17] Umfasst die NP-Schweren Probleme der Klasse NP. Kurz gesagt: Diese Probleme sind vermutlich nicht effizient lösbar.

3.1.2 Lösen eines CSP

Die Problembeschreibung als CSP ermöglicht es, Heuristiken und Methoden zu formulieren, die unabhängig von der problemspezifischen Domäne und stattdessen sehr generisch agieren. Ein Algorithmus zum Lösen eines CSP, ein Constraint Solver, nutzt dies aus und löst durch eine einheitliche Prozedur eine Vielzahl von Problemen.

Dazu wird das CSP als Suchproblem formuliert. Der Startzustand ist eine leere Zuordnung {}, in der alle Variablen unbestimmt sind. Die Nachfolger-Funktion weist iterativ jeder unbestimmten Variable einen Wert zu, vorausgesetzt er führt keine Inkonsistenzen hervor. Der Zieltest prüft, ob eine vollständige Zuordnung vorliegt. Die Pfadkosten werden auf eine Konstante (z.B. 1) festgesetzt. Eine Lösung des Suchproblems muss eine Lösung des CSPs sein, d.h. wenn n Variablen vorliegen, tritt sie auf der Tiefe n im Suchbaum auf. Außerdem lässt sich der Suchbaum nicht weiter als bis zu dieser Tiefe ausdehnen. CSPs haben maximal d^n mögliche Zuordnungen und sind kommutativ, d. h. die Reihenfolge der Wertezuweisung ist irrelevant. Deshalb wird die Tiefensuche oft zur Lösung von CSPs verwendet.

Backtracking

Backtracking ist eine Technik, bei der jeweils einer Variablen zur Zeit ein Wert zugewiesen wird und wenn sie auf Inkonsistenzen trifft, wird der Fehler zurückverfolgt. Sie wird von Suchalgorithmen wie der Tiefensuche benutzt.

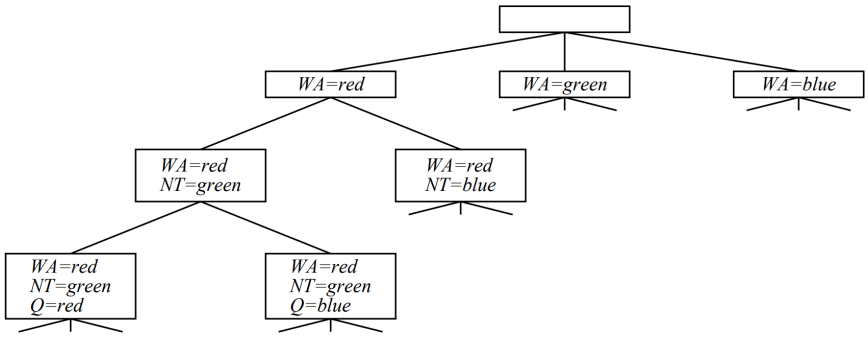


Abbildung 3.2: Teil des erzeugten Suchbaumes (*Australian Map-Problem*) [Russell and Norvig, 2016]

Im zweiten Fall der untersten Ebene ($WA = rot$, $NT = grün$, $Q = blau$) wird eine Inkonsistenz festgestellt, sobald SA ein Wert zugewiesen werden soll. Die Domäne dieser Variable beinhaltet keinen Wert mehr, der die Bedingungen $WA \neq SA$, $NT \neq SA$ und $Q \neq SA$ erfüllt und demnach ist die Zuordnung global inkonsistent. Der Algorithmus geht einen Knoten zurück und probiert einen anderen Wert für Q aus. Der nachfolgende Pseudo-Code zeigt, wie Backtracking implementiert werden kann.

Algorithmus 3.1.1 Suche mit Backtracking [Russell and Norvig, 2016]

function BTSuche(csp) **return** Lösung/Fehler:
  return $RekursivesBT(\{\}, csp)$

function RekursivesBT(teilZuordnung, csp) **return** Lösung/Fehler:
  if $teilZuordnung$ ist vollständig **then return** $teilZuordnung$
  $var \leftarrow WähleUnbestimmteVariable(Variablen[csp], teilZuordnung, csp)$
  for all $werte$ **do** $SortiereDomänenWerte(var, teilZuordnung, csp)$
    if $wert$ konsistent mit $teilZuordnung$ bzgl. der $Constraints[csp]$ **then**
      füge $\{var = wert\}$ $teilZuordnung$ hinzu
      $ergebnis \leftarrow RekursivesBT(teilZuordnung, csp)$
      if $ergebnis \neq fehler$ **then return** $ergebnis$
      entferne $\{var = wert\}$ aus $teilZuordnung$
  return $fehler$

Bearbeitungsreihenfolge von Variablen und Werten

Die Tiefensuche ist eine uninformierte[18] Suche, weshalb sie sich nicht gut für Probleme mit großen Suchräumen eignet. Normalerweise beschneiden domänen-abhängige Heuristiken den Suchraum, doch die Generizität eines CSP ermöglicht den Einsatz domänen-unabhängiger Heuristiken. Im Folgenden werden drei davon vorgestellt:

Die Erste entscheidet, welche Variable gewählt wird. Sie wählt die mit den wenigsten verbleibenden Werten aus und heißt deshalb *minimum remaining values (MRV)* Heuristik (auch *most constrained value* oder *fail-first*). Da die gewählte Variable am stärksten eingeschränkt ist, führt sie wahrscheinlich schneller zu einem Fehler als andere. Listet eine Variable keinen konsistenten Wert mehr in ihrer Domäne, so wählt MRV diese aus und vermeidet dadurch sinnloses Suchen durch andere Variablen.

Die *degree* Heuristik wählt die Variable, die noch die meisten Constraints mit anderen, bisher unbestimmten Variablen hat. Sie hilft bei der Bestimmung der ersten Variable oder wenn die Wahl durch die MRV Heuristik nicht eindeutig ist. Im „Australian Map"-Problem haben die Variablen folgende Grade: $SA = 5$; NT, Q, $NSW = 3$; WA, $V = 2$; $T = 0$. Im Startzustand wird deshalb SA als erste Variable gewählt.

Sobald eine Variable ausgewählt wurde, gilt es, ihre Werte in einer sinnvollen Reihenfolge abzuarbeiten. Die *least constraining value (LCV)* Heuristik wählt den Wert aus, der für zukünftige Variablenzuweisungen die größte Flexibilität zulässt. Das ist nach der Wahl der beschränktesten Variable zwar kontraintuitiv, allerdings werden so offensichtliche Fehler und damit potenziell überflüssiges *Backtracking* zu Gunsten einer möglichen Lösung hinten angestellt.

[18] Suchalgorithmen, die blind alle Werte absuchen

Constraint Propagation

Bei CSP wird zwischen verschiedenen k-Konsistenzen (1-, 2-, 3-konsistent) unterschieden, die auch Knoten-, Kanten- und Pfadkonsistenzen heißen. Bei der Knotenkonsistenz ist jeder Knoten in sich konsistent, bei der Kantenkonsistenz sind zwei durch Constraints verbundene Variablen zueinander konsistent und bei der Pfadkonsistenz kann jedes Paar benachbarter Variablen noch zu einer k-ten Variable expandiert werden und bleibt konsistent. Ein Graph mit n Knoten und n-Konsistenz ist streng k-konsistent und somit global konsistent. Für den weiteren Verlauf der Arbeit ist nur die lokale Kantenkonsistenz relevant. Eine Kante, beispielseweise von *SA* nach *NSW*, ist konsistent, wenn für jeden Wert x von *SA* mindestens ein y von *NSW* existiert, das mit x konsistent ist. Kantenkonsistenz ist lokal, d.h. auch bei Kantenkonsistenz zweier Variablen a und b besteht die Möglichkeit globaler Inkonsistenz des CSPs. Constraint Propagation prüft diese Konsistenz und entfernt gegebenenfalls inkonsistente Werte aus den Domänen der Variablen. Wenn einer Variable ein Wert entfernt wird, kann dabei wieder eine Inkonsistenz auftreten. Damit dies verhindert wird, muss der Algorithmus wiederholt arbeiten, bis keine Werte mehr entfernt werden müssen.

	WA	NT	Q	NSW	V	SA	T
Startzustand	R G B	R G B	R G B	R G B	R G B	R G B	R G B
WA = rot	R	G B	R G B	R G B	R G B	G B	R G B
Q = grün	R	B	G	R B	R G B	B	R G B
V = blau	R	B	G	R	B	!	R G B

Tabelle 3.1: Schrittweise Teilzuordnung, Fehler in Schritt 3 [Russell and Norvig, 2016]

Im dritten Schritt der obigen Tabelle wird V der Wert *blau* zugewiesen. Diese Zuweisung wird durch die Kanten an die Variablen *NSW* und *SA* weitergeleitet. Die Prüfung der Kantenkonsistenz stellt für $NSW = rot$ fest, dass $V = blau$ konsistent ist, bei $NSW = blau$ hat V keinen konsistenten Wert, deshalb wird der Wert aus der Domäne entfernt. Dann wird *SA* geprüft und der letzte Wert, *blau*, gelöscht, es kommt zu einem Fehler, der durch bspw. Backtracking behoben werden kann. *Arc Consistency 3 (AC-3)* ist ein bekannter, schneller Algorithmus um Kantenkonsistenz zu gewährleisten.

Algorithmus 3.1.2 Kantenkonsistenz-Algorithmus AC-3 [Russell and Norvig, 2016]

function AC-3(CSP) **return** CSP MIT EVENTUELL REDUZIERTEN DOMÄNEN:

while *queue* is not empty **do**
  $(X_i, X_j) \leftarrow NächsteKante(queue)$
  if $EntferneInkonsistenteWerte(X_i, X_j)$ **then**
    for all X_k **do** $Nachbarn[X_i]$
      füge (X_k, X_i) zur *queue* hinzu

3.2 WaveFunctionCollapse-Algorithmus

Im Jahr 2016 entwickelte Maxim Gumin WFC, einen neuen, durch Quantenmechanik inspirierten Algorithmus zur Textursynthese. Er generiert aus einer einzelnen Beispiel-Bitmap eine Vielzahl zum Input lokal ähnlicher Outputs [Gumin, 2016]. Abbildung 3.3 illustriert, wie durch WFC synthetisierte Texturen aussehen können.

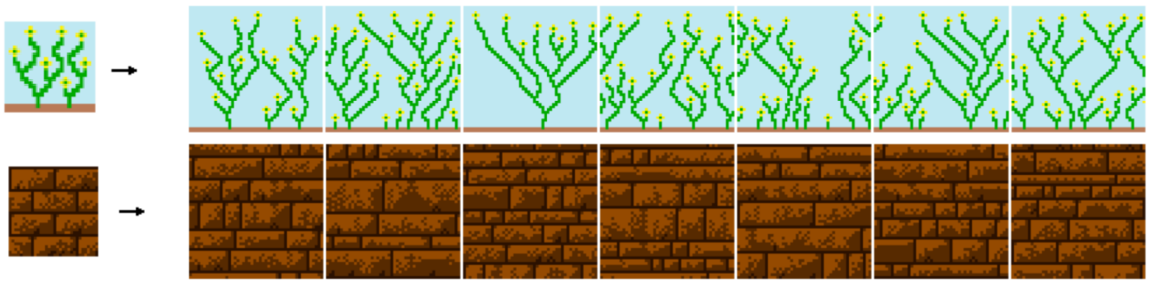


Abbildung 3.3: Zwei Sample-Bitmaps und daraus generierte Texturen [Gumin, 2016]

Gumin definiert lokale Ähnlichkeiten (vgl. Abb. 3.4) durch folgende Bedingungen:

- (C1) Jedes NxN-Muster aus dem Output, taucht mindestens einmal im Input auf.
- (C2) Die Verteilung aller NxN-Muster im Output stimmt, über eine genügend große Anzahl von Outputs, ungefähr mit der im Input überein [Gumin, 2016].

Abbildung 3.4: Visualisierung lokaler Ähnlichkeiten bei $N = 3$ [Gumin, 2016]

In der Quantenmechanik wird ein physikalisches System durch eine Überlagerung (Superposition) mehrerer quantenmechanischer Zustände beschrieben. Der Kollaps der Wellenfunktion beschreibt, dass dieses System bei einer Observation auf einen einzigen Zustand reduziert wird. Diese Reduzierung hat nicht nur Auswirkungen auf das beobachtete System, sondern auch auf andere, die sich an räumlich weit getrennten Orten befinden. In dem bekannten Gedankenexperiment „Schrödingers Katze“ taucht dieses Phänomen auch auf: Wird geprüft, ob die Katze tot oder lebendig ist (also bei einer Messung), kollabiert die Wellenfunktion und die Katze nimmt einen eindeutig definierten Zustand an.

Obwohl sein Algorithmus keine quantenmechanischen Berechnungen durchführt, nutzt *Gumin* dieses Konzept und zieht einige Analogien zur Quantenmechanik. Stark abstrahiert besteht WFC aus der folgenden Prozedur:

Algorithmus 3.2.1 WFC, High-Level Abstraktion [Karth and Smith, 2017]

function WaveFunctionCollapse() **return** Lösung/Fehler:
  $muster \leftarrow LeseInputBitmap()$
  $wellenfunktion \leftarrow InitialisiereWellenfunktion()$
  while $((GesamtEntropie > 0)$ und $!(\exists\ Entropie = undefiniert))$ **do**
    $ObserviereVariable()$
    $PropagiereConstraints()$
  return Falls kein Widerspruch gefunden: $Lösung$, sonst $Fehler$

Im Folgenden wird auf die einzelnen Schritte anhand des „Red Maze“-Beispiels (vgl. Abb. 3.5) aus *Gumins* Implementierung genauer eingegangen.

Analyse der Input-Bitmap

Alle möglichen NxN-Muster auf dem Pixel-Gitter werden abgelaufen und gezählt. Jedes einzigartige Muster wird einem Array an möglichen Werten hinzugefügt. Optional können die Rotationen und Reflexionen aller Muster hinzugefügt werden. Das Array ist äquivalent zur Domäne einer Variable in einem CSP. Für das „Red Maze“-Sample ergeben sich zwölf Muster (vgl. Abb. 3.5). Um die Wahrscheinlichkeiten der Muster zu berechnen, werden ihre Häufigkeiten gezählt [Karth and Smith, 2017].

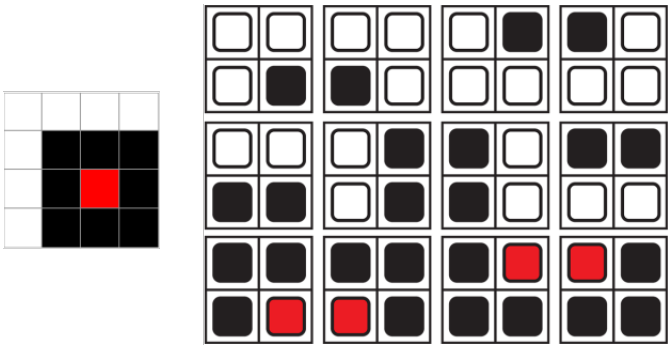


Abbildung 3.5: „Red Maze“-Input-Datei mit allen Muster-Permutationen bei N = 2 und Hinzunahme von Reflexionen und Rotationen [Karth and Smith, 2017]

Initialisierung der Wellenfunktion

Zunächst wird jedem Muster ein Dictionary zugeteilt, das beschreibt, ob die Überlappung eines anderen Musters zu diesem (x,y)-Abstand passt. Die Anzahl der Überlappungsmöglichkeiten berechnet sich mit $(2 * (N - 1) + 1)^2$ und ergibt 9 für $N = 2$. In Abb. 3.6 werden links die Überlappungen und ihr jeweiliger Abstand eines 2x2-Musters und rechts die erlaubten Muster für jeden Abstand zum ersten Muster aus Abb. 3.5 gezeigt. Sie bilden die harten Constraints des CSP [Karth and Smith, 2017].

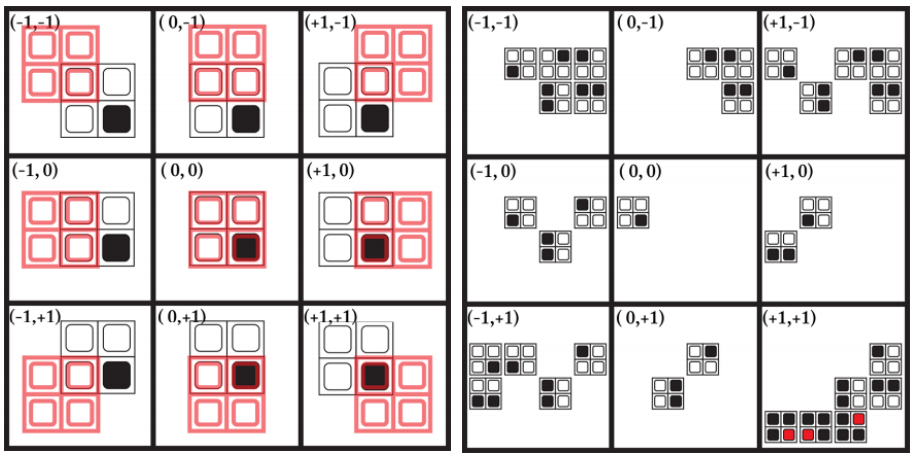


Abbildung 3.6: (a) Neun Überlappungsmöglichkeiten für N = 2, (b) Muster-Constraints des ersten Musters am jeweiligen (x,y)-Abstand [Karth and Smith, 2017]

Ein geschachteltes Array namens „Wave“ speichert den aktuellen Zustand der Wellenfunktion. Jedes äußere Array repräsentiert eine Koordinate der Output-Bitmap und ist eine Variable des CSP. Jedes innere beschreibt den aktuellen Zustand der Domäne dieser Variable und speichert einen boolschen Wert für jedes Muster. Jede Variable steht demnach in Superposition ihrer Domäne, wobei die boolschen Werte der Muster ihre Koeffizienten sind. „True“ heißt, das Muster ist erlaubt und „false“ heißt es ist bereits durch die Einschränkung der Constraints verboten worden. Der Startzustand initialisiert alle Koeffizienten der Wellenfunktion auf „true“, also sind die Domänen aller Variablen zu Beginn uneingeschränkt [Karth and Smith, 2017].

Kollaps der Wellenfunktion

Im Wechsel werden Variablen ausgewählt, zugewiesen und die Änderung durch das Array propagiert. *ObserviereVariable()* wählt in jedem Schritt die Variable mit der geringsten Entropie > 0 aus. Mit $E = -\sum_i P(i) * log_2 P(i)$ wird die Entropie berechnet, wobei $P(i)$ jeweils das Produkt der Wahrscheinlichkeit eines Musters und des boolschen Koeffizienten (0/1) der gewählten Variable ist [Karth and Smith, 2017].

Algorithmus 3.2.2 ObserviereVariable() im Detail [Karth and Smith, 2017]

```
function ObserviereVariable():
    niedrigsteEntropie ← SucheNiedrigsteEntropie()
    if (niedrigsteEntropie = undefiniert) then
        throw Widerspruch
    if (alleVariablenZugewiesen()) then
        Zyklus abbrechen und OutputZeigen()
    else
        Wähle ein Muster gewichtet nach Häufigkeit aus Quelldatei
        Setze alle Koeffizienten dieser Variable false, außer für das gewählte Muster
```

Das Auswahlverfahren ähnelt der MRV Heuristik, allerdings hat die gewählte Variable neben den wenigsten verbleibenden Werten, auch die eindeutigste Wahrscheinlichkeitsverteilung. Das heißt, eine Variable deren Werte eine 10:90% Chance haben, wird einer Variablen deren Werte eine 50:50% Chance haben, vorgezogen. Laut Gumin führe dies zu menschlicheren Ergebnissen [Gumin, 2016]. Um globale Konsistenz zu garantieren, wird die Wahl nach jeder Zuweisung propagiert. WFC nutzt einen AC-3 ähnlichen Algorithmus, der statt Constraints zu lösen, die Wahrscheinlichkeiten aller Werte der Variablen aktualisiert, die mit der zugewiesenen verbunden sind.

Jeder Observationsschritt verringert die Gesamtentropie. Erreicht sie 0, wurde jeder Variablen ein Wert zugewiesen, die Wellenfunktion ist vollständig kollabiert. Die Lösung wird als Textur ausgegeben. Zusätzlich kann nach jedem Schritt eine Teilzuordnung ausgegeben werden, um die Generierung visuell ansprechend zu gestalten. Unbestimmte Variablen werden dann als Durchschnitt ihrer verbleibenden Werte dargestellt (vgl. Abb. 3.7). Ist die geringste Entropie einer Variable undefiniert, wurden alle Muster ausgeschlossen, der Algorithmus ist auf einen Widerspruch getroffen und bricht ab. *Gumin* implementiert für solche Fehler keine Form des Backtracktings.

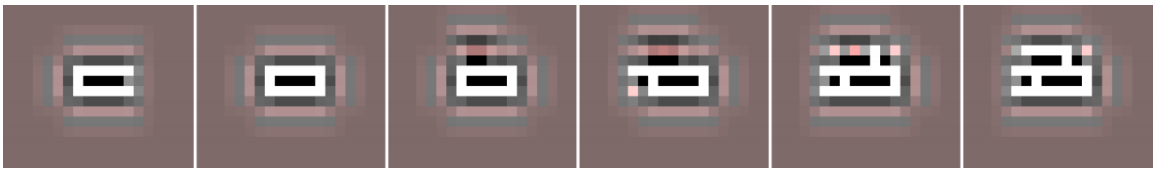


Abbildung 3.7: Sukzessive Auflösung des Problems, Darstellung der Variablen als Durchschnitt ihrer Werte [Karth and Smith, 2017]

Die obige Erläuterung bezieht sich auf das *Overlapping Model* von WFC. Es gibt außerdem noch das *Simple-Tiled Model*, welches eine Art Sonderform des ersten mit $N = 1$ ist. Da es bei $N = 1$ keine Überlappungen der Muster gibt, können die Constraints nicht mehr aus einem Beispiel abgeleitet werden. Würde das *Overlapping Model* mit $N = 1$ arbeiten, so würde es keine Überlappungen finden, demnach gäbe es keine Constraints und jedes Muster wäre neben jedem anderen platzierbar. Stattdessen werden Nachbarschaftsconstraints genutzt, die automatisiert oder per Hand von einem Designer festgelegt werden können [Gumin, 2016].

3.3 WFC als Levelgenerator

Obwohl WFC ursprünglich für die Synthese von Texturen entwickelt wurde, lässt sich der gleiche Algorithmus auch zur Levelgenerierung in Computerspielen verwenden. Wie andere Constraint Solver ist WFC ein inhalts-agnostischer Algorithmus, das heißt, dass er kein spezifisches Wissen über den zu generierenden Inhalt hat. Dank dieser Eigenschaft kann er sehr generisch eingesetzt werden. In *Gumins* Implementierung nimmt WFC Farbwerte einer Bitmap als Input. Wird dieser Input angepasst, so können beispielsweise auch dreidimensionale Objekte als Input dienen.

Damit WFC Level erstellen kann, gilt es zunächst die Levelgenerierung als CSP zu formulieren. Hierfür wird der Level als zwei- bzw. dreidimensionales Gitter betrachtet, dessen Zellen die Variablen des Problems darstellen. Eine Zelle umfasst eine 1x1x1 Einheit des Gitters und hält Platz für ein einzelnes Level-Tile. Die Gesamtheit aller Level-Tiles bildet demnach die Domäne der Variablen. Im Laufe der Zuordnung gilt es, jeder Zelle ein Tile zuzuweisen und somit eine Lösung des CSP zu finden.

Es gibt bereits einige Anwendungen von WFC in Computerspielen. Die bisher wohl Bekannteste ist das Spiel *Bad North*, über dessen Implementierung *Oskar Stålberg* einen Talk auf der *Everything Procedural Conference 2018 (EPC18)* gehalten hat [Stålberg, 2018]. Er implementiert das *Simple-Tiled Model* und bezeichnet die Variablen als Slots und die Werte als Module des CSP. Diese Terminologie wird ab hier übernommen. Neben den bereits erläuterten Grundlagen des Algorithmus, zeigt *Stålberg* einige Erweiterungen dessen auf (vgl. Abschnitt 4.3.3).

Eine weitere nennenswerte Implementierung ist *Marian Kleinebergs* Städte-Generator, der unendliche Städte generiert. Genau wie *Stålberg* verwendet er das *Simple-Tiled Model* mit etwa etwa 100 verschiedenen Modulen, aus denen die Welt generiert wird. Das Besondere an seiner Implementierung ist die Erweiterung von WFC in die Unendlichkeit [Kleineberg, 2019]. Um diese Herausforderung zu lösen, betrachtet der Algorithmus stets nur ein bestimmtes Areal um den Spieler herum, anstatt zu versuchen, alle Slots zu berücksichtigen. Wenn neue Slots geladen werden, werden die Constraints auch dorthin propagiert. Da WFC auf Fehler stoßen kann, implementiert *Kleineberg* außerdem Backtracking, um die Fehler zu korrigieren. So stößt ein Spieler auf seiner Reise nicht auf grafische Artefakte [Kleineberg, 2019].

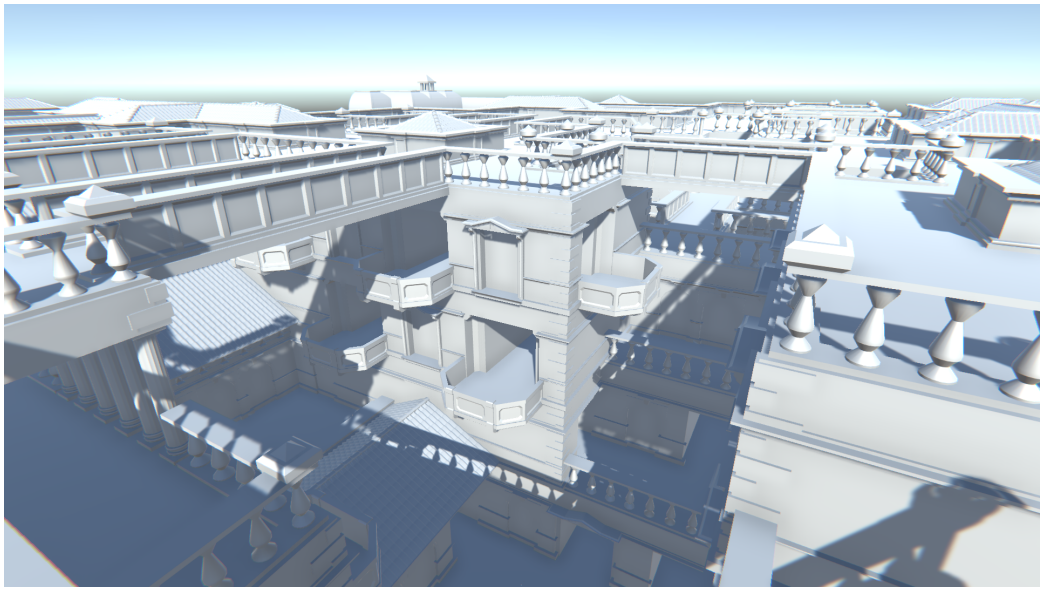


Abbildung 3.8: Screenshot aus *Kleinebergs* Städte-Generator [Kleineberg, 2019]

4 Synthese generierter und handgebauter Level

Um im weiteren Verlauf der Arbeit bewerten zu können, ob dem Generator die Synthese generierter und handgebauter Level gelingt, werden in diesem Kapitel einige Kriterien für gutes Leveldesign genauer betrachtet. Anschließend werden häufige Probleme beim Leveldesign in Spielen mit prozeduralen Leveln herausgestellt. Daraufhin wird ein Überblick über drei Spiele mit angewandter PCG gegeben, wobei jeweils die von ihnen genutzten Techniken, sowie die Adressierung der zuvor genannten Probleme erläutert werden. Diese Erkenntnisse werden genutzt, um die Anforderungen an den in der Thesis entwickelten Level-Generator, *KHollapse*, aufzustellen.

4.1 Prinzipien des Leveldesign

In den meisten Computerspielen steht das Design der Level im Mittelpunkt. Ob Puzzle-, Adventure-, oder Rollenspiel, Level gehören zum Kernpunkt der Interaktion. Sie vereinen die verschiedenen Elemente des Spiels zum großen Ganzen: Dem Spiel selbst. Rückt der Leveldesigner nicht all diese Elemente in ein gutes Licht, so geht der Spielspaß verloren. Einen Level so zu gestalten, dass er den Spielern Freude bereitet, ist keine leichte Aufgabe. Mittlerweile gibt es aber Ansätze vieler bekannter Game- und Leveldesigner, theoretische Grundlagen für gutes Leveldesign zu schaffen.

Eine Definition, was ein Level, Leveldesign und ein Leveldesigner sind, gibt *Tim Ryan* in seinem Gamasutra Artikel *Beginning Level Design*:

> „Level design is the data entry and layout portion of the game development cycle. A level is, for all intents and purposes, the same as a mission, stage, map or other venue of player interaction. As a level designer, you are chiefly responsible for the gameplay.“ [Ryan, 1999a]

Im Folgenden werden, angelehnt an *Taylor's Ten Principles of Good Level Design*, Prinzipien für gutes Design erläutert [Taylor, 2013].

Lösbarkeit

Ein Level muss immer lösbar sein. Ist er es durch grafik- oder gameplay-basierte Bugs nicht, führt dies schnell zur Frustration beim Spieler. Solche Fehler im Leveldesign heißen *Show Stopper*. Durch ausgiebiges Testen können sie gefunden und vermieden werden [Ryan, 1999a].

Immersion

Immersion trägt maßgeblich dazu bei, dass Spieler ihre Zweifel beiseitelegen und gänzlich in die Spielwelt eintauchen. In *Saltzman's Secrets of the Sages: Level Design* sagt *Richard Gray*[19], dass Action, Gameplay und Spaß erst dann Teil des Spiels würden, wenn der Level immersiv genug sei. Zur Authentizität zählt er auch die Kontinuität der Spielwelt. Ein Level sollte ein übergeordnetes, visuelles Thema haben und alle Objekte in ihm sollten diesem folgen. Ein Flickenteppich mehrerer Themen führe zu einem Bruch der Immersion [Saltzman, 1999].

Folgendes Zitat von *Tim Ryan* fasst den Immersionsgedanken zusammen:

> „A player buys a game to escape from his or her reality. Good levels and hence good games will immerse the player and suspend their disbelief. From the moment the title screen comes up, you have their full attention. From that point on, they should see and do nothing that reminds them that they are anywhere but in the world you have them in."
> [Ryan, 1999a]

Landmarks, einzigartige Schauplätze, tragen positiv zur Immersion bei. Sie machen die Welt glaubwürdiger und dienen als Anhaltspunkte zur Orientierung (vgl. Abschnitt Navigation) [Saltzman, 1999]. Wird Architektur außerhalb des spielbaren Bereichs platziert, wirkt die Welt größer und lebendiger für den Spieler [Saltzman, 1999].

Interaktionsmöglichkeiten

> „Videogames are escapism... pure and simple." [Taylor, 2013]

Der von *Taylor* erwähnte Eskapismus knüpft an den Immersionsgedanken an (vgl. Abschnitt Immersion). Im Zuge dessen spricht er auch von Interaktion mit dem Level, die für ihn maßgeblich dazu beiträgt. Laut ihm gehe es um „Bestärkung" des Spielers. Als Beispiele dafür nennt er die Interaktion und/oder Zerstörung von Objekten im Level, sowie folgendes Beispiel aus der eigenen Erfahrung:

> „For Medal of Honor Heroes 2, we wanted to make the secondary objectives more than just a shopping list of hidden Nazi dossiers, so we created side-missions where the player could rescue allied troops, trapped at certain locations hidden throughout the level. These troops, once freed, would fight alongside the player, which made him/her feel that there was a direct consequence, and reward, for his/her actions."
> [Taylor, 2013]

Auch für Game Director *Mark Pacini* (Metroid Prime) ist die Bestärkung ein wichtiges Element. Laut ihm sollte dem Spieler immer erst das Problem und dann die Lösung präsentiert werden, um ihm zu vermitteln, er habe diese selbst gefunden.

[19] In dem erwähnten Artikel taucht Gray namentlich nur unter seinem Pseudonym „Levelord" auf.

Beispielsweise sollte der Schlüssel einer verschlossenen Tür nicht unmittelbar vor ihr, sondern etwa in einem anderen Raum versteckt sein [Crecente, 2008].

Ein guter Level belohnt die interaktive Kreativität des Spielers und hält diverse Möglichkeiten bereit. Bestenfalls sollte ein Level auf die Frage *„Kann ich … tun?“* stets mit *„Klar!“* antworten. Ungewöhnliche Ideen werden so belohnt und steigern den Spielspaß [Ryan, 1999b]. Dies geht oft mit der Identifizierung und Berücksichtigung verschiedener Spielstile während der Design-Phase eines Levels einher. Für das gleiche Szenario könnte ein Spieler eine laute, offensive Taktik wählen, während ein Zweiter sein Ziel bevorzugt unentdeckt erreicht. Das Ziel sollte demnach stets klar sein, der Weg allerdings nicht [Ryan, 1999b; Taylor, 2013]. *Ryan* verdeutlicht außerdem, dass „Schein-Interaktion“ sehr frustrierend sei. Alle Objekte sollten eine Daseinsberechtigung haben. Zerstörung und/oder Bewegung dieser sei dafür ausreichend, solange sie sinnvoll ist. Nichts sei frustrierender für Spieler, als ein vermeintliches Schiebepuzzle lösen zu wollen, welches sich als schlechtes Design herausstellt [Ryan, 1999b].

Pacing

Pacing spielt in Games eine wichtige Rolle und lässt sich auch auf das Leveldesign anwenden, um interessantere Level zu kreieren. Game Designer *John Romero* spricht davon, dass Spieler konstanter Gefahr ausgesetzt sein müssten, während *Cliff Bleszinski* ihm dabei widerspricht: Konstante Gefahr führe zur Abstumpfung der Spieler. Er vergleicht das Pacing in Spielen stattdessen mit dem in Filmen und sagt es ginge darum, den richtigen Moment für eine Überraschung zu finden [Saltzman, 1999].

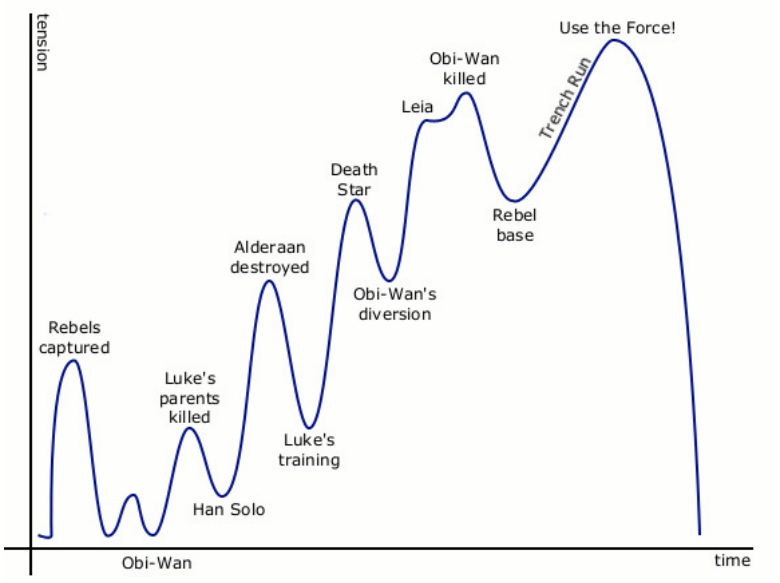


Abbildung 4.1: Pacing-Kurve aus *Star Wars: A new Hope* [Wesołowski, 2009]

Pacing bezieht sich auf viele Elemente des Leveldesigns. Jedes Element folgt entweder seiner eigenen Kurve oder spiegelt die des Levels wieder. Die Platzierung starker und schwacher Gegner in unterschiedlichen Intervallen, das Verteilen von Gegenständen auf der Karte und weitere sind hier zu bedenken [Saltzman, 1999]. Neben grundlegendem Pacing ist das Element der Überraschung sehr entscheidend, um einprägsame Momente in Spielen zu erschaffen [Taylor, 2013]. In *Ten Principles of Good Level Design* beschreibt *Taylor* wie dies für ihn aussehen kann:

> „In terms of level design, surprise could take the form of a unique setting, a moment that teaches the player something new about a mechanic they've already been using for a while, turning the corner to see a beautiful vista, or a radical change in pacing." [Taylor, 2013]

Navigation

Ein Großteil der Spielzeit wird mit der Navigation durch die Level verbracht. Layout, Beleuchtung und weitere visuelle Hinweise sollten einen natürlichen Fluss des Spielers durch den Level gewährleisten [Taylor, 2013]. Um das navigatorische Gameplay interessanter zu gestalten, können einige Areale so versteckt werden, dass sie nur durch Erkundung des Spielers auffindbar sind. Dadurch gewinnt der Level an Tiefe und Replayability. Außerdem sollte mit dem verfügbaren Raum gespielt werden. Enge, sich windende Gassen haben eine komplett andere psychologische Wirkung auf den Spieler als weite, offene Flächen [Taylor, 2013].[20]

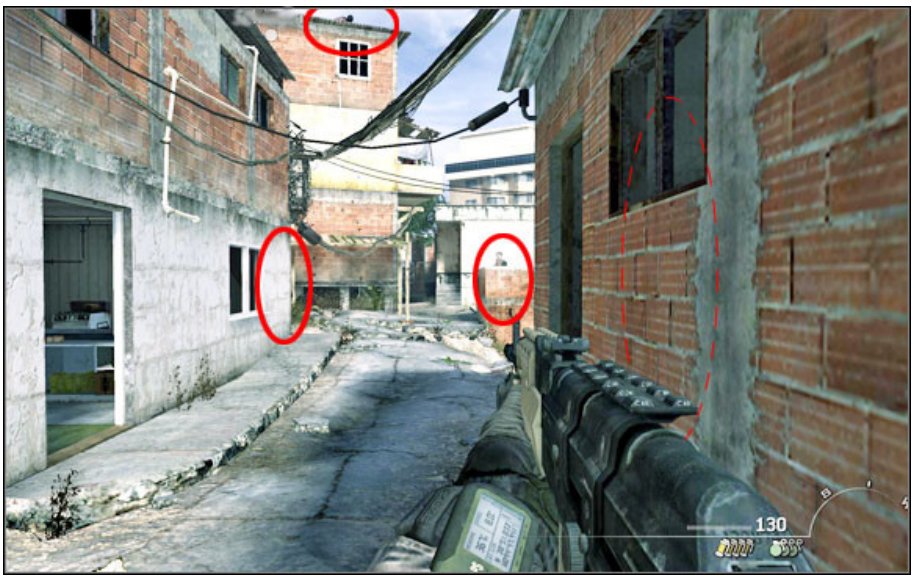


Abbildung 4.2: *CoD Modern Warfare 2 - Favela*. Rote Markierungen zeigen Feinde auf unterschiedlichen Ebenen [Justynski and Lasota, 2015]

[20] Die genaue Erläuterung liegt außerhalb der Reichweite dieser Arbeit, kann aber unter `https://www.gamasutra.com/view/feature/134779/designing_better_levels_through_.php` nachgelesen werden.

Ein gutes Beispiel hierfür ist der Favela-Level aus *Call Of Duty Modern Warfare 2* (vgl. Abb. 4.2). Der Spieler befindet sich in den Slums der brasilianischen Hauptstadt Rio de Janeiro. Während sich Gegner aus allen Gassen und Winkeln auf den Spieler stürzen, intensivieren die engen, schlängelnden Straßen und überladenen Marktplätze die Situation. Weiterhin ist dieser Level ein passendes Beispiel für Vertikalität im Spielraum. Dem Spieler begegnen Gefahren aus unterschiedlich hochgelegenen Punkten und sein Weg führt ihn durch alle Ebenen des Levels, was den Level navigatorisch und spielerisch interessanter macht [Taylor, 2013].

Außerdem ist es nützlich, wenn ein Level ein klares Ziel hat. Der zusätzliche Kontext hilft Spielern, sich besser zu orientieren. Darüber hinaus können neben dem Hauptziel kleinere Ziele existieren, die als Subziele oder Side-Quests[21] dienen [Saltzman, 1999]. Geschickt eingesetzt steigern sie den Erkundungsdrang des Spielers.

Navigatorisches Gameplay hat auch Nachteile, beispielsweise Backtracking. Es beschreibt, dass der Spieler einen Weg wiederholt gehen muss, um zurück auf den Hauptpfad des Levels zu gelangen. Oft entstehen diese Situationen, wenn ein Seitenarm des kritischen Pfades zu einem Schatz führt. Der Spieler folgt ihm, besiegt alle Feinde auf dem Weg, sichert sich den Schatz und läuft zurück [Portnow, 2016]. Wenn der Seitenarm auf den Hauptpfad zurückführt oder für bidirektionales Gameplay sorgt, wird dem entgegengewirkt [Portnow, 2016; Taylor, 2013]. Ein weiteres Problem, vorallem bei offenen Leveln, kann die „Erkundungsangst“ (engl. exploration anxiety) sein [Duvall, 2001]. Sie tritt in zwei Arten auf:

1. Spieler wandern ziellos durch den Level, um etwas (un)bestimmtes zu suchen.
2. Die Angst, noch nicht alles im Level erkundet zu haben und wichtige Gegenstände zu übersehen.

Diese Angst begründet sich mit der Funktionsweise des menschlichen Gehirns, das sich nur etwa sieben Dinge gleichzeitig merken kann [Duvall, 2001; Miller, George Armitage, 1956]. Wenn sich Pfade zu ähnlich sind, sind sie demnach besonders schwer zu merken [Duvall, 2001].

Schwierigkeit

Sollte das erfolgreiche Abschließen eines Levels stets garantiert sein, bleibt irgendwann das Erfolgserlebnis am Ende eines Levels aus. Andersherum führen zu schwierige Level zur Frustration der Spieler [Ryan, 1999a].

> „The trick to good level design is to present challenges that are difficult enough to merit the players' attention and make their heart or mind race, but not so difficult as to always leave them failing and disappointed.“ [Ryan, 1999a]

[21] Eine, meist kleinere, Aufgabe, die parallel zum Verlauf der Hauptaufgabe gelöst werden kann.

Außerdem sollte die Level Progression[22] berücksichtigt werden. In der Regel werden Spieler mit steigender Spielzeit immer mehr Skill und bessere Ausrüstung, Gegenstände etc. haben. Insgesamt sollten spätere Level also schwieriger sein als die Früheren [Ryan, 1999b]. In einigen Situationen ist es allerdings sinnvoll, diese Regel zu brechen, um dem generellen Pacing des Spiels besser zu passen. Beispielsweise könnte nach einem sehr schweren, intensiven Level, erstmal ein ruhigerer folgen, um dem Spieler Zeit zum Durchatmen zu geben [Ryan, 1999b].

Taylor kritisiert das von vielen Spielen verwendete statische Schwierigkeitsstufenmodell und schlägt als Lösung dynamischere Systeme vor, wie sie in Fallout oder Skyrim verwendet werden. Da solche Systeme nicht immer verfügbar seien, ließe ein guter Level den Spieler die Schwierigkeit des Erlebnisses wählen [Taylor, 2013]. Dafür sollte der kritische Pfad auf durchschnittlich geskillte Spieler angepasst werden, während es abseits dessen schwerere und leichtere Pfade gibt, um das Interesse der Spieler anderer Skillstufen wecken [Ryan, 1999b; Taylor, 2013]. Risiko und Entlohnung (vgl. Abschnitt Risk and Reward) sollten dabei stets absehbar sein, damit der Spieler eine informierte Entscheidung treffen kann [Taylor, 2013].

Risk and Reward

Risk and Reward beschreibt Spielsituationen, in denen der Spieler ein erhöhtes Risiko eingehen muss und, falls er diese souverän meistert, eine entsprechende Belohnung erhält. Solche Situationen können von Leveldesignern eingesetzt werden, um Spielern mit mehr Skill interessante Herausforderungen und stärkere Gegenstände zu bieten. Außerdem sorgen sie dafür, dass Spieler wohlüberlegte Entscheidungen treffen müssen. Einen besonders starken Gegner im Kampf begegnen oder einen großen Abgrund überwinden müssen, um an einen Schatz zu kommen, sind typische Beispiele dafür. *Gray* sieht sie als elementare Bestandteile eines guten Levels:

> „A good level should be a series of challenges and rewards. The challenge can come before the reward, or after, but don't just haphazardly strew goodies and bad guys through your level. The players should feel as though they are being run through a gauntlet of contests and prizes."
> [Saltzman, 1999]

Einzigartigkeit

Die Level in einem Spiel sollten zwar alle dem selben, visuellen Thema folgen (vgl. Abschnitt Immersion), sich aber nicht zu sehr ähneln. Zu gleiche Abläufe, Strukturen und Begegnungen machen einen Level langweilig und schaden der Immersion [Ryan, 1999a]. Um ihn einzigartig zu gestalten, hat der Designer die Möglichkeit vorhandene Assets auf unterschiedliche Arten zu kombinieren. Dadurch entstehen neue Möglichkeiten, Abwechslung ins Spiel zu bringen (s.a. Abschnitt Modularität) [Ryan, 1999b].

[22] Beschreibt, wo im Verlauf des Spiels ein Level eingeordnet ist

Modularität

Da die Ressourcen der Entwickler begrenzt sind, entwirft ein guter Designer nicht speziell einen Level, sondern ein modulares Set von durch Spielmechaniken inspirierten Begegnungen. Durch die einzelnen Teile dieser Sets können problemlos eine Vielzahl von Leveln erstellt werden, die die Mechaniken unterschiedlich kombinieren und nutzen [Taylor, 2013]. Durch geringe Eingriffe und Modifikationen können diese abgewandelt und im weiteren Spielverlauf wiederholt eingesetzt werden. Dies gibt Spielern die Chance, bekannte Begegnungen erneut zu erleben und die Spielmechaniken zu meistern, während die Modifikationen Abwechslungen liefern und den Spieler neu fordern [Taylor, 2013]. Eine hohe Modularität sorgt außerdem dafür, dass der Leveldesigner schneller weitere Iterationen seines Levels bauen kann, um herauszufinden, wie er die Elemente am besten in Szene setzen kann [Taylor, 2013].

4.2 Besonderheiten beim prozeduralen Leveldesign

Für viele der diskutierten Prinzipien ist es wichtig, dass der Designer alle Entitäten eines Levels genau kontrollieren kann. Der Einsatz von PCG erschwert die Kontrolle und bringt einige Besonderheiten mit sich.

Die Lösbarkeit aller generierbaren Level zu garantieren, ist bereits eine komplexe Aufgabe. Der Generator darf weder auf dem kritischen Pfad noch auf dessen Seitenarmen unüberwindbare Hindernisse platzieren, die den Spieler am Erhalt einer Belohnung oder gar dem Abschließen des Levels hindern. Auftretende Puzzle, wie z.B. Lock-Key-Puzzle[23], dürfen nicht zu *Show Stoppern* führen. Für jedes Lock muss also gewährleistet werden, dass sein Key erreichbar ist. Hierbei kann es zu erheblicher Programmier- und Testlast kommen (vgl Abschnitt 2.5).

Der algorithmischen Natur prozeduraler Level geschuldet, wirken sie schnell repetitiv. Sie sind oft weniger immersiv, da sie keine bis wenige Landmarks enthalten und sich an vielen Stellen gleich anfühlen (vgl. Abschnitt 2.5, *The Elder Scrolls II: Daggerfall*). Auch *Tanya X. Short* erkennt dieses Problem:

> „One of the dangers of procedural generation is that you end up with a smooth blend of samey content that's essentially predictable in flavor and tone.“ [Short, 2014]

Die Kontrolle des Pacings in Spielen ist ohnehin schwierig, da der Spieler das Geschehen lenkt, wird aber noch komplexer, wenn ein Algorithmus das Leveldesign übernimmt. Da Algorithmen oft anders agieren, als Spieler es erwarten, entstehen jedoch auch häufiger überraschende Situationen. Das Balancing prozeduraler Level ist

[23] Lock-Key-Puzzle beschreiben Rätsel, bei denen Spieler einen Schlüssel zu einer Tür finden muss, um diese zu passieren

ebenfalls komplizierter, da es keine festen Bedingungen gibt. Die Variation des Levels selbst zieht eine Änderung der Schwierigkeit mit sich. Außerdem gibt es zu jedem Zeitpunkt der Level-Progression-Kurve nun nicht mehr eines, sondern (unendlich) viele Level. Demnach muss jeder mögliche Level, der zu einem bestimmten Punkt generiert wird, eine angemessene Schwierigkeit besitzen.

4.3 Syntheseansätze in existierenden Spielen

Die folgenden Abschnitten stellen exemplarisch drei Spiele vor, die handgebaute Elemente mit prozeduralen Techniken vereinen. Es wird erläutert, welche Techniken sie verwenden und warum. Dabei wird gezeigt, wie die Synthese generierter und handgebauter Level hilft, häufige Probleme bei der Levelgenerierung zu vermeiden.

4.3.1 Spelunky

Spelunky ist ein 2D-Platformer mit Inspirationen aus dem *Rogue-like*-Genre. *Derek Yu* hatte die Idee, das klassische Platformer-Gameplay aufzufrischen, indem er Elemente wie Perma-Death und prozedurale Level in sein Spiel einband. Am 21. Dezember 2008 wurde es zum ersten Mal öffentlich verfügbar gemacht [Yu, 2016]. Im gleichnamigen Buch *Spelunky* beschreibt *Yu* detailliert die Levelgenerierung aus dem Spiel.

Jeder Level ist auf einem 4x4-Gitter angeordnet und füllt dieses vollständig aus. In jedem Feld liegt ein 10x8-Tiles großer Raum. Diese Struktur löst automatisch das Problem, dass sich Räume überlappen könnten. Zuerst wird der kritische Pfad auf dem 4x4-Gitter generiert [Yu, 2016]:

1. Wähle einen zufälligen Slot aus der obersten Reihe und markiere ihn als Eingang.
2. Wähle eine zufällige Zahl von 1 bis 5.
   a) Ist die Zahl eine 1 oder 2, bewege dich nach links auf dem Gitter.
   b) Ist sie eine 3 oder 4, bewege dich nach rechts auf dem Gitter.
   c) Ist sie eine 5, bewege dich in die nächste Zeile.
   d) Überschreitest du die Grenzen des Gitters, bewege dich in die nächste Zeile.
3. Wiederhole Schritt 2, bis du in der untersten Zeile angekommen bist.
4. Führe Schritt 2 ein letztes Mal aus, bis Bedingung 2c oder 2d erfüllt sind.
5. Markiere dann den letzten Raum als Ausgang.

Während der Algorithmus sich durch das Gitter bewegt, markiert er die Slots mit Zahlen. „0“ steht für Räume, die der Algorithmus nicht passiert hat, „1“ für Ausgänge auf der linken und rechten Seite. Räume mit einer „2“ haben einen zusätzlichen Ausgang auf der Unterseite, bei einer „3“ ist er auf der Oberseite [Yu, 2016].

Werden zwei „2“-Räume untereinander platziert, bekommt der untere zusätzlich einen Zugang nach oben und ist somit eine Kreuzung. Schritt 2a und 2b markieren den neuen Raum mit einer „1“. Bei den Schritten 2c und 2d wird der vorige Raum mit einer „2“ und der Raum mit einer „2“ oder einer „3“ markiert, um einen Durchgang zu schaffen. So entsteht entlang des kritischen Pfades eine garantiert lösbare Raumabfolge [Yu, 2016].

Abbildung 4.3 zeigt einen fertigen Level, wobei in den Raumecken noch die String-Repräsentation aus Tabelle 4.1 zu sehen ist. Der kritische Pfad ist vom Ein- bis Ausgang rot markiert und ohne den Einsatz von Items passierbar. So kann sich der Spieler seine Bomben etc. aufbewahren, um Seitenräume („0er-Tiles“) zu erkunden und Schätze zu finden.

2 1 1 0
3 1 1 2
0 0 0 2
0 0 0 3

Tabelle 4.1: String-Repräsentation eines generierten *Spelunky*-Levels [Yu, 2016]

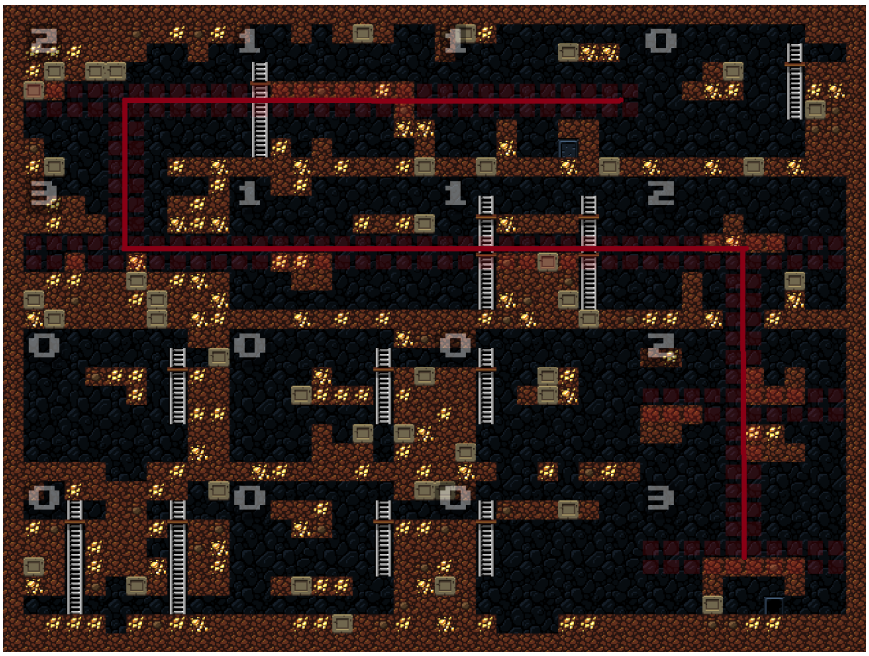


Abbildung 4.3: Beispielhafter Level aus *Spelunky*. [Kazemi, oD]

Anschließend folgt die Generierung der Räume, bei der für jeden Raum, basierend auf den Raumtypen, eines von sechs bis zwölf verschiedenen Templates gewählt wird. Räume sind intern als String repräsentiert. Um sie richtig darzustellen, wird nach jedem zehnten Zeichen ein Zeilenumbruch hinzugefügt. Aus dem String „0000000011006 0000L040000000P110000000L110000000L11000000001100000000111112222111“ lässt sich das in Tabelle 4.2 ersichtliche Template erzeugen.

0	0	0	0	0	0	0	0	1	1
0	0	6	0	0	0	0	L	0	4
0	0	0	0	0	0	0	P	1	1
0	0	0	0	0	0	0	L	1	1
0	0	0	0	0	0	0	L	1	1
0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	1	1
1	1	1	2	2	2	2	1	1	1

Tabelle 4.2: Bsp. Raum-Template [Yu, 2016]

0	Freie Fläche
1	Solider Block
2	je 50% Chance: Solider Block/Freie Fläche
L	Leitertile
P	Leitertile mit Holzfläche zum Stehen
6	5x3 Luft-Chunk
4	Steinblock, der horizontal schiebbar ist

Tabelle 4.3: Character-Tile-Legende [Yu, 2016]

Um den Raum zu erzeugen, lässt sich jedes Zeichen des Templates mit seinem korrespondierenden Tile ersetzen (s. Tabelle 4.3), wobei einige besondere Tiles, sogenannte Chunks, sogar ein 5x3 großes Feld ersetzen. *Yu* nutzt sie, um existierende Templates zu mutieren und mehr Variation in die Welt zu bringen. Es gibt zehn Chunks, aus denen der Algorithmus einen auswählt [Yu, 2016]. Das obige Template stellt einen Raum mit einer Leiter auf der rechten Seite dar. Sie führt zu einem kleinen Durchgang, der von einem schiebbaren Stein blockiert ist. Zusätzlich besteht die Chance, dass der Weg nach unten geöffnet ist und ein Chunk das Template mutiert [Yu, 2016].

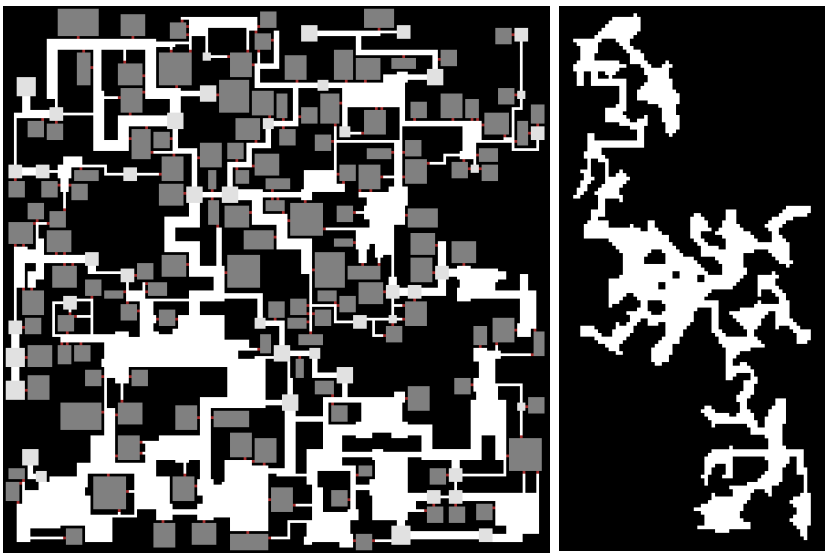
Nach der Erstellung der Räume startet ein finaler Durchlauf, der für jedes „1“er-Tile entschiedet, ob eine Entität darauf oder darum erzeugt wird. Diese Entitäten können Monster, Schätze etc. sein und sind untereinander gewichtet. Sogar umliegende Felder haben Auswirkungen auf ihre Spawnchance. Beispielsweise tauchen Truhen verhäuft in Nischen auf, die nur eine freie Seite haben [Yu, 2016].

Die Probleme prozeduraler Generierung adressiert *Spelunky* geschickt. Durch die Generierung des kritischen Pfades entlang eines Gitters und die sich daran anpassende Raumstruktur treten keine *Show Stopper* auf. Obwohl die Variation möglicher Pfade auf dem Gitter nicht allzu viele sind, sorgen die Raum-Templates und ihre Mutationen für abwechslungsreiche Level. Visuelle Themen (Dschungel, Höhle, ...) erzeugen weitere Abwechslung. Da es immer den ohne weiteres lösbaren kritischen Pfad gibt, ist die Schwierigkeit stets fair. Abseits dessen finden erfahrenere Spieler ihre Herausforderung. Sie können dort beispielsweise nach wertvollen Gegenständen suchen, die unter anderem versteckte Bereiche öffnen. Das Pacing kontrolliert *Yu* über einen Geist, bei dessen Berührung der Spieler sofort stirbt. Er erscheint nachdem der Spieler sich zu lange im gleichen Level befindet. Spieler müssen ihre Zeit also gut nutzen, um den Ausgang und möglichst viele Schätze zu finden, die den Score maximieren. Dies kreiert interessante Risk-Reward-Situationen, in denen Spieler abwägen müssen, ob sie weiter Schätze sammeln oder den nächsten Level betreten.

4.3.2 Cogmind

Cogmind ist ein modernes *Rogue-like*, das ASCII-Grafiken auf Animationen, Partikel- und Soundeffekte treffen lässt. Der Spieler schlüpft in die Rolle des „Cogmind“, einen Roboter, der auf seinem Weg durch prozedural generierte Höhlen und Dungeons in einer Science-Fiction-Welt auf andere, feindliche Roboter trifft. Für die Generierung der Level nutzt Entwickler *Josh Ge* eine Kombination aus *Drunkard's Walk*-Algorithmen und ZA, sowie einiger händischer Eingriffe [Ge, 2014].

Dungeons werden mit einer Abwandlung des *Drunkard's Walks* generiert (vgl. Abschnitt 2.4.1). Die Walker ändern ihre Parameter mit der Zeit, wodurch organischere Dungeons entstehen. Die Resultate sind weniger repetitiv und eignen sich besser für größere Level [Ge, 2014]. Höhlen werden mit einer ZA Variante generiert, die zufällig gewählte Zellen besucht und basierend auf ihrer Nachbarschaft „tötet“ oder „belebt“. Dies erhöht zwar die Variation, steigert aber auch die Wahrscheinlichkeit unverbundener Höhlenabschnitte. Als Lösung führt *Ge* „Grabungs-Phasen“ ein, in denen schrittweise Tunnel zwischen unverbundenen Teilen gegraben werden. Um den Level-Flow und die Backtracking-Problematik besser zu kontrollieren, verwendet er „geführte zelluläre Automaten (ZA)“, die bei der Generierung einer grundlegenden Form folgen (vgl. Abb. 4.3.2) [Ge, 2014].



(a) *(b)*

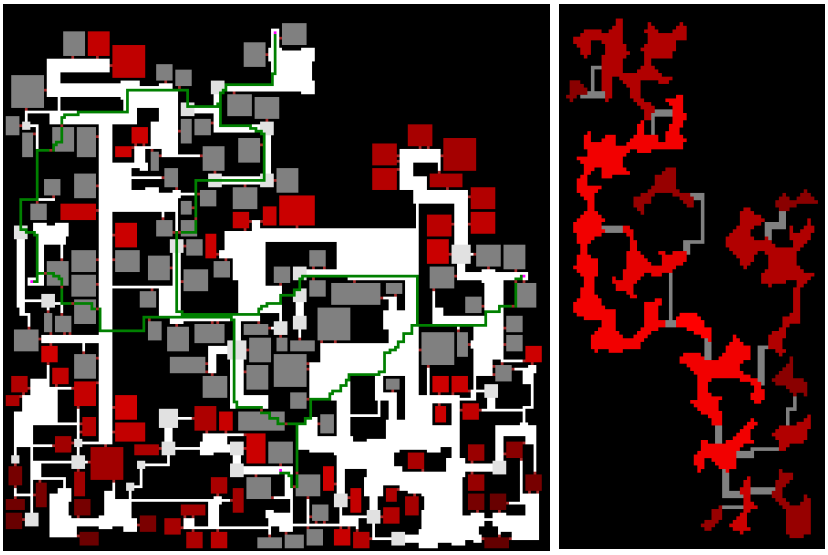
Abbildung 4.4: (a) Dungeon und (b) korridorförmige Höhle aus *Cogmind* [Ge, 2014]

Cogmind nutzt eigene Dungeon-Metriken, um bewerten zu können, ob ein generierter Level brauchbar ist. Faktoren wie die Offenheit der Welt und die Anzahl kleiner/mittlerer/großer Räume werden genutzt, um den Spielraum zu bewerten. Ein A*-Pathfinding-Algorithmus prüft, ob die von Walkern kreierten Räume miteinander verbunden sind. Gleichzeitig sorgt er für ausreichend Distanz zwischen Ein- und Ausgang, da zu kleine Abstände den Spielspaß senken würden.

Diese Level-Daten werden genutzt, um Objekte wie Terminals, Schätze und Gegner zu platzieren. Terminals tauchen oft dort auf, wo kleine Räume Korridore verbinden, größere Maschinen eher in Hallen und Patrouillen auf Wegkreuzungen [Ge, 2014].

Der nächste Schritt unterscheidet sich je nach Areal-Typ:

- Bei Dungeons wird die Abgeschiedenheit vom schnellsten Weg für den Spieler gemessen: Je weiter ein Raum von diesem entfernt ist, desto eher enthält er wertvolle Maschinen und Items (vgl. Abb. 4.5a).
- Bei Höhlen wird die Anbindung der Räume gemessen: Schlechter angebundene Räume werden für Ein- bzw. Ausgänge genutzt, sehr gut angebundene beheimaten herausfordernde Kämpfe (vgl. Abb. 4.5b) [Ge, 2014].



(a) Für Dungeons *(b) Für Höhlen*

Abbildung 4.5: Rötung zur Visualisierung der Anbindung/Abgeschiedenheit [Ge, 2014]

Ge bindet „Prefabs“, von Hand gebaute Räume, in seine Level ein. Diese bleiben, ähnlich wie in *Spelunky*, bei jedem Durchlauf bis auf einige Details gleich und dienen als Landmarks [Ge, 2014]. Sie werden vor der Levelgenerierung auf zufälligen Koordinaten im Level platziert und ihre Wände als permanent markiert, damit Walker nicht zufällig die Struktur des Raumes abändern. An den Ausgängen befindet sich jeweils ein Walker, der zu Beginn der Generierung losgräbt, um auf den restlichen Level zu stoßen und ihn mit dem Prefab zu verbinden [Ge, 2014]. Es gibt zwei weitere Prefab-Arten, die allerdings erst nach der Generierung platziert werden: „Zentrierte“ und „Eingebundene“. Die „Zentrierten“ werden dabei in das geometrische Raumzentrum platziert und überschreiben bestehende Geometrie. „Eingebundene“ lassen sich in Lücken bestehender Räume und Korridore integrieren, ohne dabei die generierte Welt zu verändern [Ge, 2014].

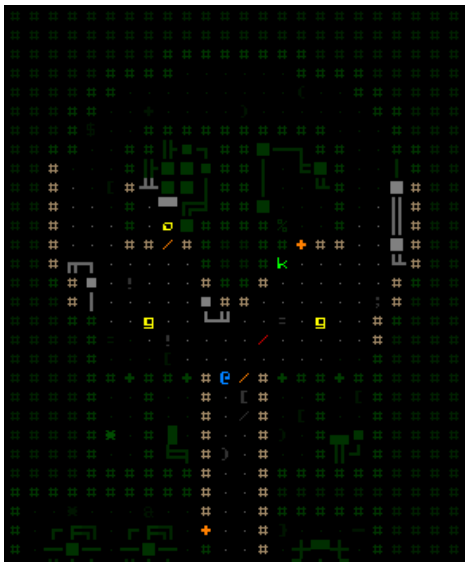


Abbildung 4.6: Prefab eines Raumes in Totenkopf-Form. [Ge, 2014]

Cogmind gelingt die Synthese handgebauter und generierter Welten durch die Verwendung und Abwandlung klassischer Algorithmen, sowie den Einsatz von Prefabs, die in den Generierungsprozess eingebunden werden. Die Variationen unter ihnen sorgen, wie in *Spelunky*, selbst in statischen Räumen für Abwechslung und Überraschungen für den Spieler. Zusätzlich schafft der modernisierte ASCII-Stil (Animationen, Partikeleffekte etc.) und das dazu passende Sci-Fi-Setting für einen hohen Immersionsgrad. Durch die Dungeon-Metriken werden das Pacing und die Schwierigkeit kontrolliert.

4.3.3 Bad North

Bad North ist ein *Rogue-lite* Echtzeit-Strategie Spiel vom schwedischen Entwickler *Plausible Concept*. Der Spieler kontrolliert eine Armee kleiner Soldaten auf einer Insel, mit dem Ziel, diese vor angreifenden Wikingern zu schützen. Für die Levelgenerierung imlementiert *Bad North* den WFC-Algorithmus (vgl. Abschnitt 3.2).

Insgesamt nutzt das Spiel etwa 300 verschiedene Module, um Inseln zu generieren. Die Generierung findet vollständig im dreidimensionalen Raum statt und erweitert den Algorithmus um einige Details. Die erste und wichtigste Erweiterung ist die Einführung einer Navigierbarkeits-Heuristik die misst, wie gut ein Level begehbar ist. Dabei versucht der Algorithmus mit jedem observierten Slot den begehbaren Bereich zu vergrößern. Da ein Haus stets begehbar ist, wird immer damit begonnen und von dort aus der navigierbare Bereich expandiert [Stålberg, 2018]. Weiterhin hat *Stålberg* WFC um eigene Constraints erweitert, die beispielsweise die Anforderungen von Wasser auf dem Boden und Himmel an den Außenseiten definieren. Da diese Constraints sich eher auf den Rand des Spielfeldes beziehen, werden sie zuerst aufgelöst

und die resultierende Teilzuordnung gespeichert. Der übliche WFC-Ablauf generiert danach den restlichen Level. Bei einem Widerspruch, lädt die gespeicherte Teilzuordnung und ein neuer Durchlauf mit anderem Seed startet. Laut *Stålberg* sei dieser Ansatz effektiver als Backtracking, weshalb er darauf verzichtet [Stålberg, 2018]. Das Laufzeit-Problem von WFC adressiert *Bad North* mit einer relativ kleinen Maximalgröße für die Inseln [Stålberg, 2018].

Abbildung 4.7: Generierte Insel aus *Bad North* [Plausible Concept, 2017]

Durch dynamische Inselgrößen, Themen, Wetterstimmungen und Details in den 3D-Assets bieten die generierten Level viel Abwechslung. *Stålberg* hat außerdem flexible Tile-Größen in seinen Algorithmus implementiert, um mehr artistische Freiheit zu erhalten und das offensichtliche Tiling-Pattern zu brechen.

Die Synthese basiert auf der algorithmischen Zusammensetzung handgebauter Tiles und lässt durch die erwähnten Handgriffe einigen artistischen Spielraum. Die generierten Level weisen keine hohe Komplexität auf, passen jedoch in das Game Design des Spiels. Sowohl WFC an sich als auch die erwähnten Erweiterungen sorgen dafür, dass der Designer bedeutsame Eingriffe in den Generierungsprozess tätigen kann.

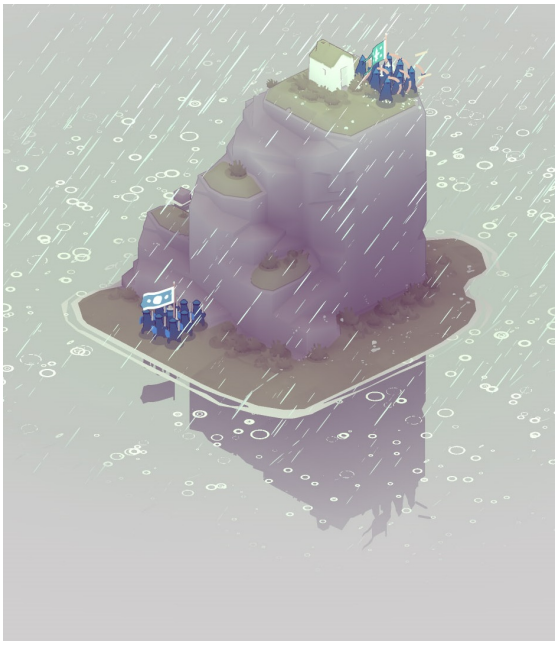


Abbildung 4.8: Insel aus *Bad North* bei Regen [Plausible Concept, 2017]

4.4 Kriterien einer erfolgreichen Synthese

Um im nächsten Kapitel ein Konzept für den Generator entwickeln zu können, werden in diesem Abschnitt die Erkenntnisse dieses Kapitels in einer Anforderungsanalyse reflektiert.

4.4.1 Beschreibung der Hauptkriterien

Die hier beschriebenen Kriterien sind alle gleichermaßen wichtig für die erfolgreiche Synthese. Sollte eines oder mehrere nicht eingehalten werden können, gilt die Synthese nicht automatisch als erfolglos, ist aber von geringerer Qualität. Der Generator sollte also darauf ausgelegt werden, ihre Herausforderungen und Probleme zu lösen.

1. Lösbarkeit

Wie bereits in den Abschnitten 4.1 und 4.2 diskutiert, ist die stetige Lösbarkeit eines Levels essenziell. Bereits ein geringer Prozentsatz unlösbarer Level führt zu hohen Verlusten bezüglich der Spieleranzahl. Der Generator muss gewährleisten, dass bei der Synthese gar nicht erst unlösbare Level entstehen. Sollte es dennoch einen unlösbaren Level geben, muss dieser identifiziert und neu erstellt werden.

2. Immersion & Einzigartigkeit

Einer der größten Vorteile von PCG ist die schier unendliche Vielfalt generierter Elemente. Um diesen Vorteil nicht zu verlieren, muss darauf geachtet werden, dass die generierten Elemente bedeutsame Unterschiede voneinander haben. Wenn jeder Level den anderen zu sehr ähnelt und keine einzigartigen Spielerlebnisse bietet, langweilen sich die Spieler und erleben keine neuen Herausforderungen. Die Nutzung variabler Level-Chunks in Spielen wie *Spelunky* oder *Cogmind* ist hierfür ein guter Lösungsansatz. Auch die Abwechslung durch verschiedene, visuelle Themen hilft dabei (s. a. Abschnitt 4.3.3). Die Nutzung und Platzierung von Landmarks ist ebenfalls von entscheidender Bedeutsamkeit. Das oberste Ziel dieser Anforderung ist es, dem Spieler interessante Level zu liefern.

3. Schwierigkeit

Der Generator sollte Möglichkeiten liefern, den Schwierigkeitsgrad im Level zu integrieren. Neben dem kritischen Pfad sollte es also weitere Seitenarme geben, auf denen zusätzliche Herausforderungen oder alternative Wege geboten werden. In einer vollwertigen Produktion könnten weitere Systeme des Spiels diese Infrastruktur dann nutzen, um die Schwierigkeit auszugestalten.

4. Risk and Reward & Interaktion

Durch die Bereitstellung mehrerer Pfade werden unterschiedliche Spielstile berücksichtigt. Weiterhin sollten alle Elemente im Level sinnvoll platziert und Möglichkeiten der Schein-Interaktion vermieden werden. Gegner, Gegenstände und weitere Entitäten werden so platziert, dass interessante, abwägbare Risk-Reward-Situationen entstehen. Auf Seitenarmen und in abgelegeneren Arealen befinden sich wertvollere Gegenstände, aber auch herausforderndere Begegnungen.

5. Navigation

Der Generator nutzt den Spielraum voll aus und sorgt für Vertikalität im Level, um ihn interessanter zu gestalten. Durch einfache, abstrakte Ziele wird in Ansätzen der navigatorische Fluss des Levels gewährleistet. Der Kontext eines vollwertigen Spiels wird ihnen jedoch fehlen. Außerdem ist eine gewisse Ziellosigkeit im Genre der *Rogue-likes* inhärent, da das Finden des Ziels Teil des Core-Loops ist. Nachteile wie *Backtracking* und *exploration anxiety* werden weitestgehend vermieden.

4.4.2 Beschreibung zweitrangiger Kriterien

Die nachfolgenden Kriterien sind weniger relevant als die vorherigen. Ihre Erfüllung kann als „nice to have“ angesehen werden. Sie würden die Qualität der Software zwar steigern, sind aber in gewissermaßen außerhalb der Reichweite dieser Arbeit.

Performance

Der Synthesevorgang sollte innerhalb einer tolerierbaren Zeit stattfinden. Lange Ladezeiten während des Spielens stören den Spielfluss, was zur Frustration bei Spielern führen kann. Eine akzeptable Zeitspanne liegt also im Bereich weniger Sekunden. Weiterhin ist wichtig, dass der Generierungsprozess performant genug ist, um parallel zu anderen Systemen eines Spiels stattzufinden. Blockiert allein die Levelgenerierung die volle Rechenleistung, ist sie für eine reale Produktion nicht anwendbar.

Dreidimensionalität

Für Level im zweidimensionalen Raum gibt es bereits eine Vielzahl interessanter und gut funktionierender Levelgeneratoren. Der dreidimensionale Raum bietet allerdings deutlich weniger interessante Systeme. Da gerade WFC eine gute Möglichkeit dafür sein könnte, ist eines der Ziele dieser Arbeit die Generierung in den dreidimensionalen Raum zu bringen. Sollte dies nicht gelingen, gilt auch die Umsetzung in 2D als erfolgreich, da sie die Hauptthese der Arbeit verifizieren bzw. falsifizieren kann.

4.4.3 Was der Generator nicht leistet

Neben all diesen Anforderungen folgt nun, was der Generator nicht leisten soll bzw. kann:

Da es sich bei der entwickelten Software zu dieser Thesis nicht um ein komplettes Spiel handelt, ist eine eingehende Bewertung der Schwierigkeit eines Levels nicht möglich. Demnach können Level auch nicht sinnvoll in die Level Progression eingeordnet werden. Daraus folgt auch, dass das Pacing nur innerhalb eines Levels, auf struktureller Ebene beachtet werden kann. Für eine sinnvolle, Level-übergreifende Implementierung fehlt auch hier der Kontext eines Spiels. Zerstörbare Objekte, Umgebungen etc. sind nicht Teil dieser Arbeit und deshalb nicht Teil der Interaktion zwischen Spieler und Level. Ferner ist es nicht möglich, das navigatorische Gameplay in voller Tiefe auszuschöpfen. Dinge wie (sekundäre) Missionsziele und weitere spielspezifische Aufgaben können nicht berücksichtigt werden.

Zu guter Letzt sei gesagt, dass die generierten Level keine visuelle Vielfalt im Ausmaße der hier vorgestellten Spiele haben werden, da der Fokus nicht auf der Erstellung ansprechender 3D-Modelle liegt.

4.4.4 Übersicht der aufgestellten Kriterien

Zusammenfassend finden sich hier die erläuterten Kriterien auf einem Blick:

1. Ein synthetisierter Level enthält keine *Show Stopper* (Lösbarkeit).
2. Synthetisierte Inhalte bieten ausreichend Abwechslung ohne die Immersion zu vernachlässigen (Einzigartigkeit & Immersion).
3. Integration des Schwierigkeitsgrades (Schwierigkeit).
4. Sinnvolle Platzierung von Elementen im Level (Risk and Reward & Interaktion).
5. Interessantes navigatorisches Gameplay (Navigation).

Neben diesen Hauptkriterien gelten folgende niedriger priorisierten Anforderungen:

- Die Synthese findet innerhalb einer angemessenen Zeit statt (Performance).
- Anwendung auf den dreidimensionalen Raum (Dreidimensionalität).

5 Konzeption

Nachdem nun alle Grundlagen bekannt sind und Kriterien für eine erfolgreiche Synthese aufgestellt wurden, wird in diesem Kapitel das Konzept zu dem in der Thesis entwickelten Levelgenerator vorgestellt.

5.1 Entwicklungsumgebung

Da sich die Arbeit auf die Entwicklung eines Levelgenerators fokussiert, wird dieser als Plugin einer bestehenden Game-Engine entwickelt. Eine Game-Engine kümmert sich um grundlegende Probleme wie Rendering- oder Physics-Systeme und erlaubt Entwicklern sich auf ihr Spiel zu konzentrieren. Für die Implementierung von *KHollapse* wurde die *Unity-Engine* gewählt. Sie basiert auf der Programmiersprache *C#*, demnach wird auch *KHollapse* in *C#* programmiert. Prinzipiell ist die Wahl der Game-Engine arbiträr - das Konzept des Generators lässt sich auch auf andere Programmiersprachen und Engines übertragen. Die gewählte Engine sollte jedoch 3D-fähig sein, da der Generator dreidimensionale Level erstellt. Für den weiteren Verlauf der Arbeit werden Grundkenntnisse der Engine vorausgesetzt. Der Name „*KHollapse*“ ist angelehnt an den Namen des Algorithmus *WaveFunctionCollapse* und setzt sich aus dem englischen Wort „collapse“ und den Initialen des Autors zusammen.

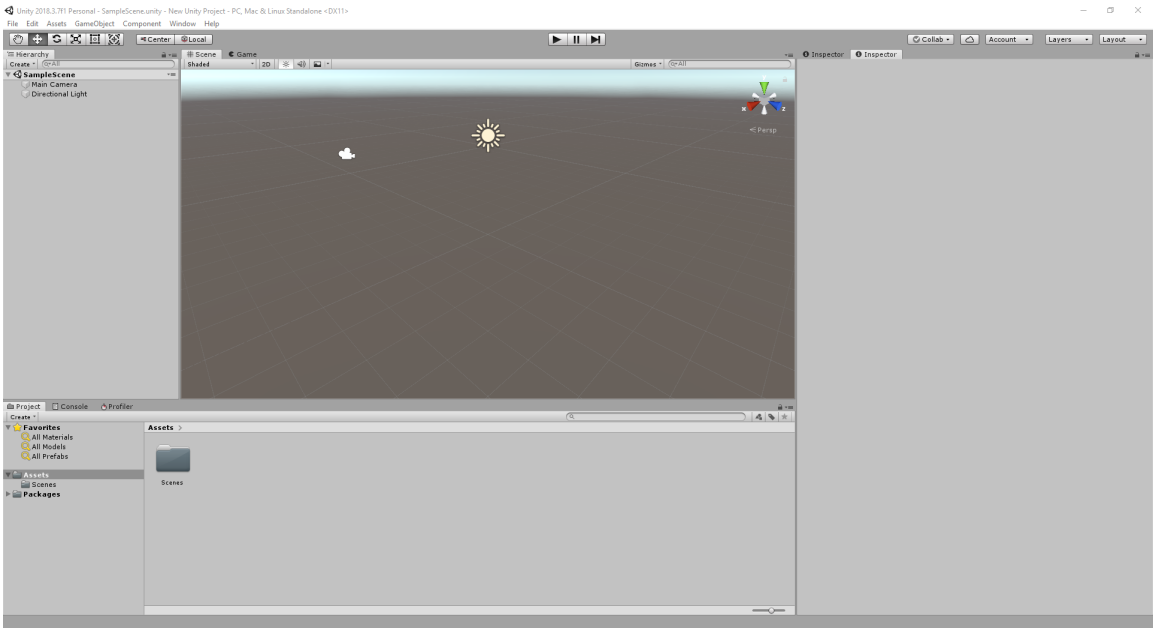


Abbildung 5.1: Grafische Oberfläche von *Unity* in leerem Projekt, eigene Grafik

5.2 KHollapse

Dieser Abschnitt legt das Konzept des Generators dar. Dazu werden zuerst die zu treffenden Anpassungen am WFC-Algorithmus beschrieben und begründet. Daraufhin folgt eine Erläuterung geplanter Erweiterungen, die die generierten Ergebnisse verbessern sollen. Der nächste Abschnitt beschäftigt sich mit den relevanten Workflows für das Arbeiten mit dem Levelgenerator. Eine Übersicht der geplanten Software-Architektur fasst das Konzept zusammen und liefert den Abschluss des Kapitels.

5.2.1 Anpassungen des Algorithmus

Die ersten beiden Änderungen betreffen die In- bzw. Outputs des Algorithmus. Zum einen gilt es, das Einlesen dieser anzupassen, zum anderen bedarf es einer Anpassung des Input-Datentyps.

Gumin speichert die Daten seiner Samples in XML-Dateien. Die Datei „samples.xml“ speichert mehrere Einträge von Sample-Konfigurationen, bei denen jede Konfiguration einen Output erzeugt. Für die gleichen Samples sind unterschiedliche Einstellungen nutzbar, um verschiedene Outputs zu erzeugen. Eine Konfiguration definiert, mit welchen Werten ein Model startet und welche Maße der Output besitzt.

samples.xml

```
<samples>
    <overlapping name="Chess" N="2" periodic="True"/>
    <overlapping name="Chess" N="2" width="47" height="47" periodic="True"/>
    <overlapping name="Skyline" N="3" symmetry="2" ground="-1" periodic="True"/>
    <overlapping name="Platformer" N="3" symmetry="2" ground="-1" periodic="True"/>
    <overlapping name="Red Maze" N="2" periodic="True"/>
    <simpletiled name="Summer" width="15" height="15"/>
    <simpletiled name="Castle" width="20" height="20"/>
    <simpletiled name="Circuit" subset="Turnless" width="34" height="34" periodic="True"/>
    <simpletiled name="Knots" subset="Standard" width="24" height="24" periodic="True"/>
    ...
</samples>
```

Abbildung 5.2: Auszug aus der Datei samples.xml, eigene Grafik

Da der Input beim *Simple-Tiled Model* aus mehreren, jeweils durch eine eigene Bitmap repräsentierten Modulen besteht, wird in einer weiteren XML-Datei definiert, welche Module zu einem Simple-Tiled Input gehören. Außerdem definiert *Gumin* darin die Nachbarschaftsregeln des Inputs (vgl. Abb. 5.3).

```
<set size="10">
    <tiles>
        <tile name="corner" symmetry="L"/>
        <tile name="cross" symmetry="I"/>
        <tile name="empty" symmetry="X"/>
        ...
    </tiles>
    <neighbors>
        <neighbor left="corner 1" right="empty"/>
        <neighbor left="cross" right="cross"/>
        <neighbor left="cross" right="cross 1"/>
        ...
    </neighbors>
</set>
```

Abbildung 5.3: Datei data.xml aus dem *Knots*-Tileset, eigene Grafik

Für eine *Unity*-Implementierung ist die Nutzung einer XML-Datei allerdings eher ungeeignet. Die Engine liefert mit ihrem Inspektor bereits Möglichkeiten, um Daten von Objekten und Datencontainern zur Editor-Zeit zu bearbeiten, weshalb es wenig sinnvoll wäre, einem Leveldesigner ein neues, zusätzliches Tool in Form von XML-Dateien an die Hand zu geben, das für ihn mehr Aufwand bedeutet. Die Daten müssen deshalb in eine für die Engine lesbare, bearbeitbare und schreibbare Datenstruktur gebracht werden.

In *Gumins* WFC sind die Module entweder einzelne Bitmaps (*Simple-Tiled Model*) oder NxN-Muster (*Overlapping Model*). Als Eingabe-Parameter des Konstruktors wird der Name des Sets genutzt, aus dem die Bitmaps eingelesen werden. Bei der Erstellung dreidimensionaler Level in *Unity* werden die Module 3D-Modelle sein, die es in der Szene zu platzieren gilt. Jedes Objekt in einer Szene ist ein *GameObject*.[24] Der Algorithmus muss also mit dieser *Unity*-internen Datenstruktur arbeiten. Dafür erhält jedes Modul eine einzigartige, zahlenbasierte ID, um es eindeutig zu identifizieren. Für jedes Sample lassen sich die verwendeten IDs angeben, die als Input für das Model genutzt werden. Das Model selbst nutzt keine Bitmaps mehr, sondern Ganzzahlen. Das erlaubt dem Generator mit diversen Inputs zu arbeiten, solange diese vorher in Zahlen konvertiert werden.

Ursprünglich besteht WFC aus drei Klassen: Einer abstrakten Basisklasse *Model*, sowie dessen Implementierungen *SimpleTiledModel* und *OverlappingModel*. Die Basisklasse kümmert sich um grundlegendes Verhalten wie das Initialisieren und Aufräumen der Wellenfunktion oder den Main-Loop des Programmes (vgl. Algorithmus 3.2.1). Die Subklassen implementieren einen modelabhängigen Konstruktor, in dem jeweils die Analyse der Input-Werte stattfindet (vgl. Abschnitt 3.2 u. Abb. 5.4).

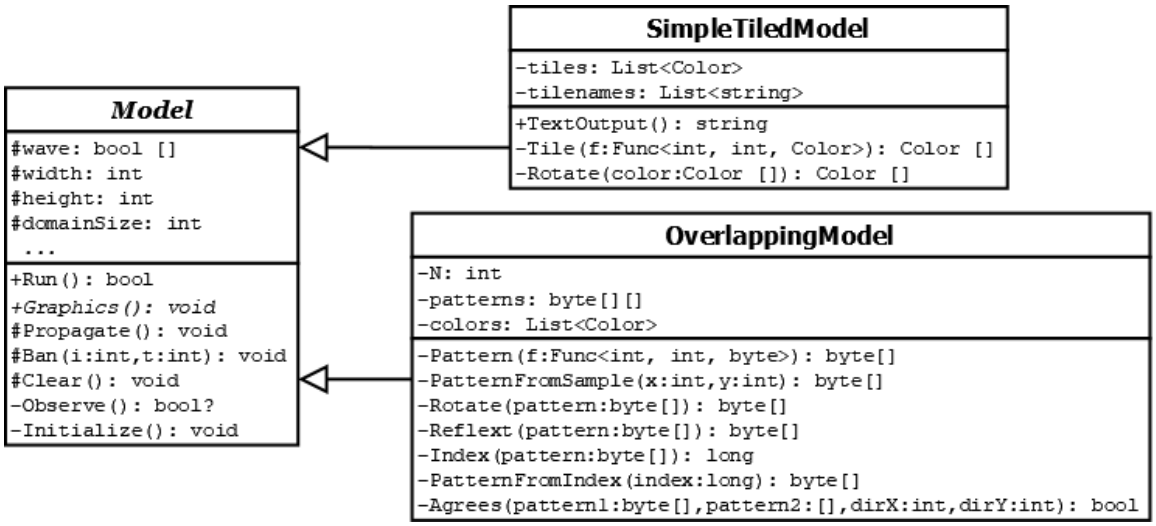


Abbildung 5.4: UML-Klassendiagramm von *Gumins* Implementation, eigene Grafik

[24]Bei Bedarf, siehe `https://docs.unity3d.com/Manual/GameObjects.html`

Gumins Architektur eignet sich hervorragend, um schnell einen Überblick über die Funktionsweise des Algorithmus zu erlangen, ist in einigen Fällen jedoch nur schwer erweiterbar. Deshalb wird in *KHollapse* die Funktionalität in kleinere Module mit eindeutigeren Verantwortlichkeiten aufgeteilt. Die Unterteilung des Models in eine abstrakte Basisklasse, sowie ihrer beiden Implementierungen bleibt erhalten. Darüber hinaus entstehen die Klassen *Slot*, *AbstractModule*, *Pattern* und *Module*. In *Slot* wird das Verhalten der Variablen gekapselt und *Pattern* bzw. *Module* sind die Implementierungen der gemeinsamen Basisklasse *AbstractModule*, die das Verhalten der Werte kapselt. Das UML-Diagramm in Abschnitt 5.2.4 liefert Details dazu.

Die letzte Änderung umfasst die Erweiterung der Dimensionalität von WFC in den dreidimensionalen Bereich, damit echt-dreidimensionale Level generiert werden.

5.2.2 Erweiterungen des Algorithmus

Neben den oben erläuterten Änderungen sind einige Erweiterungen des Algorithmus geplant, dessen Aufgabe es ist, die Resultate des Generators zu verbessern.

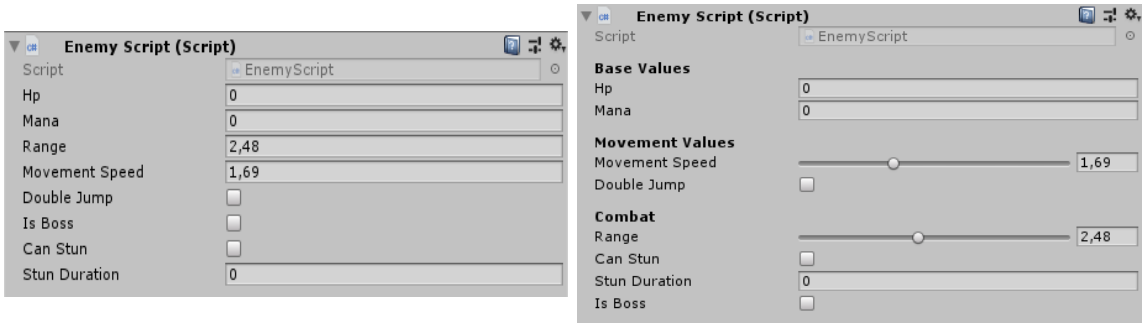
Als erste Erweiterung wird die von *Stålberg* eingeführte Navigierbarkeits-Heuristik implementiert. Wie in Abschnitt 4.3.3 beschrieben, gewichtet sie Module, die die Navigierbarkeit erhöhen, stärker als andere und expandiert somit den begehbaren Bereich eines Levels. Durch die Expansion werden außerdem verbundene Pfade generiert, entlang derer sich der Spieler bewegen kann.

Ferner ist geplant, ein an WFC angepasstes Constraint-Framework zu entwickeln, welches das Hinzufügen benutzerdefinierter Constraints ermöglicht. Der Gedanke ist angeknüpft an die Constraints aus *Bad North*, die die Domäne für den Levelrand auf bestimmte Module einschränken. Durch die Constraints soll der Generator eine erhöhte Flexibilität und Erweiterbarkeit erlangen und Designern bzw. Programmierern durch minimalen Aufwand erlauben, die Resultate wirkungsvoll zu beeinflussen.

Als dritte und letzte Erweiterung soll der Levelgenerator wie in *Spelunky* erst eine Struktur verbundener Level-Chunks auf einem Gitter und dann jeden Chunk mit einer eigenen WFC-Instanz generieren. Hierbei muss dafür gesorgt werden, dass die Übergänge zwischen Chunks nicht visuell fehlerhaft sind und der Pfad vom Anfang bis zum Ende navigierbar bleibt. Beide Problematiken sollen durch eine Kombination der beiden zuvor entwickelten Systeme adressiert werden: Damit die Übergänge stimmig sind, werden in einem ersten Durchlauf an den Verbindungen der Chunks die Outputs der Übergänge generiert, die in einem zweiten Durchlauf als Constraints bei der Generierung der Chunks dienen. Die Navigierbarkeit wird gewährleistet, indem der Algorithmus sich den Pfad vom Start- bis zum Endchunk merkt und einige Punkte entlang dessen abspeichert. Sie werden als Constraints für die Navigierbarkeits-Heuristik genutzt, die bedient werden müssen, damit der Level valide ist.

5.2.3 Workflow

Beim Einsatz von PCG-Techniken spielt der Workflow der Game- und Leveldesigner eine wichtige Rolle. Der Generator und seine Bestandteile müssen für sie schnell und einfach verständlich sein, um effizientes Arbeiten zu gewährleisten. Deshalb ist es wichtig, dass *KHollapse* auch als Tool für Nicht-Programmierer verständlich ist. *Unity* bietet Entwicklern viele Möglichkeiten ihre Intentionen zu kommunizieren. Durch das Hinzufügen von Attributen im Code lassen sich Überschriften, Tooltips und Struktur in den Inspektor[25] der Engine bringen. Ein solcher Inspektor ist sehr viel übersichtlicher und verständlicher für Außenstehende. Weiterhin lassen sich durch Gizmos[26] abstrakte Elemente gut visualisieren. Zusätzlich kann der Standard-Inspektor entweder erweitert oder komplett überschrieben werden, um die GUI des eigenen Tools umzusetzen. Die Skripte aus *KHollapse* bedienen sich daher *Unitys* Möglichkeiten zur Erweiterung der Inspektoren. Der Input falscher Werte (beispielsweise negative Leveldimensionen) wird abgefangen und verhindert unnötiges Fehlerpotenzial.



(a) *(b)*

Abbildung 5.5: (a) Standard-Inspektor und (b) Inspektor mit Attributen, eigene Grafik

Ein weiterer wichtiger Bestandteil von *KHollapse* ist die Automatisierung sich wiederholender Prozesse. Bereits das Hinzufügen neuer Module zum Algorithmus stellt eine sehr repetitive und zeitintensive Aufgabe dar. Im manuellen Prozess muss hierfür ein 3D-Objekt mit den richtigen Einstellungen in das Projekt importiert werden, anschließend eine Instanz dessen in einer *Unity*-Szene mit dem entsprechenden Modul-Skript und passendem Collider versehen und als *Prefab*[27] abgespeichert werden. Diesem muss letztlich noch eine einzigartige ID zugewiesen werden. Das Speichern der *Prefabs* muss sogar für jedes einzeln durchgeführt werden. All diese Schritte weisen hohes Fehlerpotenzial auf, kosten viel Zeit und können automatisiert werden. *Unity* bietet die Möglichkeit, die Asset-Import-Pipeline anzupassen. Dafür wird von der Klasse *AssetPostProcessor* geerbt, die einige Callbacks bereitstellt, um importierte Texturen, Sounds, Modells etc. eigenhändig zu verarbeiten. *KHollapse* automatisiert den

[25] Eine graphische Benutzeroberfläche für die In-Editor Bearbeitung von *Unity*-Skripten
[26] Visuelle Unterstützungen in der *Unity-Engine*, z.B. das Licht- und Kamerasymbol in Abb. 5.1.
[27] *Prefabs* sind in *Unity* Templates von *GameObjects*, die zur Laufzeit instanziiert werden können

Import-Prozess soweit, dass aus neuen Modellen, die in ein bestimmtes Verzeichnis importiert werden, sofort *Prefabs* mit der richtigen Skript-Komponente und Collidern erstellt werden. Außerdem werden sie einer globalen Modul-Liste hinzugefügt, über die sie eine einzigartige ID zugewiesen bekommen. Die restlichen Einstellungen eines Moduls wie mögliche Nachbarn oder Häufigkeit, werden nicht automatisiert und müssen weiterhin manuell gepflegt werden.

5.2.4 Architektur

Der Generator besteht im Kern aus dem WFC-Algorithmus, der wie in Abschnitt 5.2.1 diskutiert implementiert wird. Neben der Kern-Bibliothek des Algorithmus gibt es noch weitere Skripte, die als Schnittstelle zwischen WFC und *Unity* dienen. Sie sind alle entweder *MonoBehaviours*[28] oder *ScriptableObjects*[29]. Da die Inputs für WFC *Prefabs* sind, bedarf es einer Möglichkeit diese zu konfigurieren. Das geschieht über eine *AbstractTemplate*-Klasse, von der jeweils *OverlappingTemplate* und *ModuleTemplate* erben. Diese Struktur ist parallel zu der vom Kern verwendeten *AbstractModule*-Architektur (vgl. Abschnitt 5.2.1). Dadurch sind die Module bequem per Inspektor in der Szene oder im Prefab-Edit-Mode zu bearbeiten.

Die Klassen *ModuleSetup* und *Theme* sind ScriptableObjects. In *ModuleSetup* werden alle Module gesammelt, um ihnen ihre IDs zuzuweisen und in einem *Theme* können *AbstractTemplates* festgelegt werden, die als Inputs für dieses *Theme* dienen. Diese Struktur ist ähnlich zu *Gumins* verwendeter „samples.xml“. Letztlich gibt es noch die Klasse *LevelGenerator*, die ein Theme entgegennimmt und seine Inputs an die WFC-Bibliothek weiterleitet, um aus ihnen das Model zu generieren. Sobald das Modell durchgelaufen ist, instanziiert der Generator die Module in der Szene. Zu guter Letzt gibt es noch das *ScriptableObject WFCToolSettings*, in dem toolbasierte Einstellungen wie z.B. die Farben der zu zeichnenden Gizmos, einstellbar sind.

In Abbildung 5.6 ist das resultierende UML-Klassendiagramm ersichtlich. Es zeigt die wichtigsten Vererbungshierarchien und Assoziationen zwischen Objekten. Der Übersicht halber sind nicht alle Informationen ganzheitlich dargestellt.

[28]Bei Bedarf, siehe `https://docs.unity3d.com/ScriptReference/MonoBehaviour.html`
[29]Bei Bedarf, siehe `https://docs.unity3d.com/Manual/class-ScriptableObject.html`

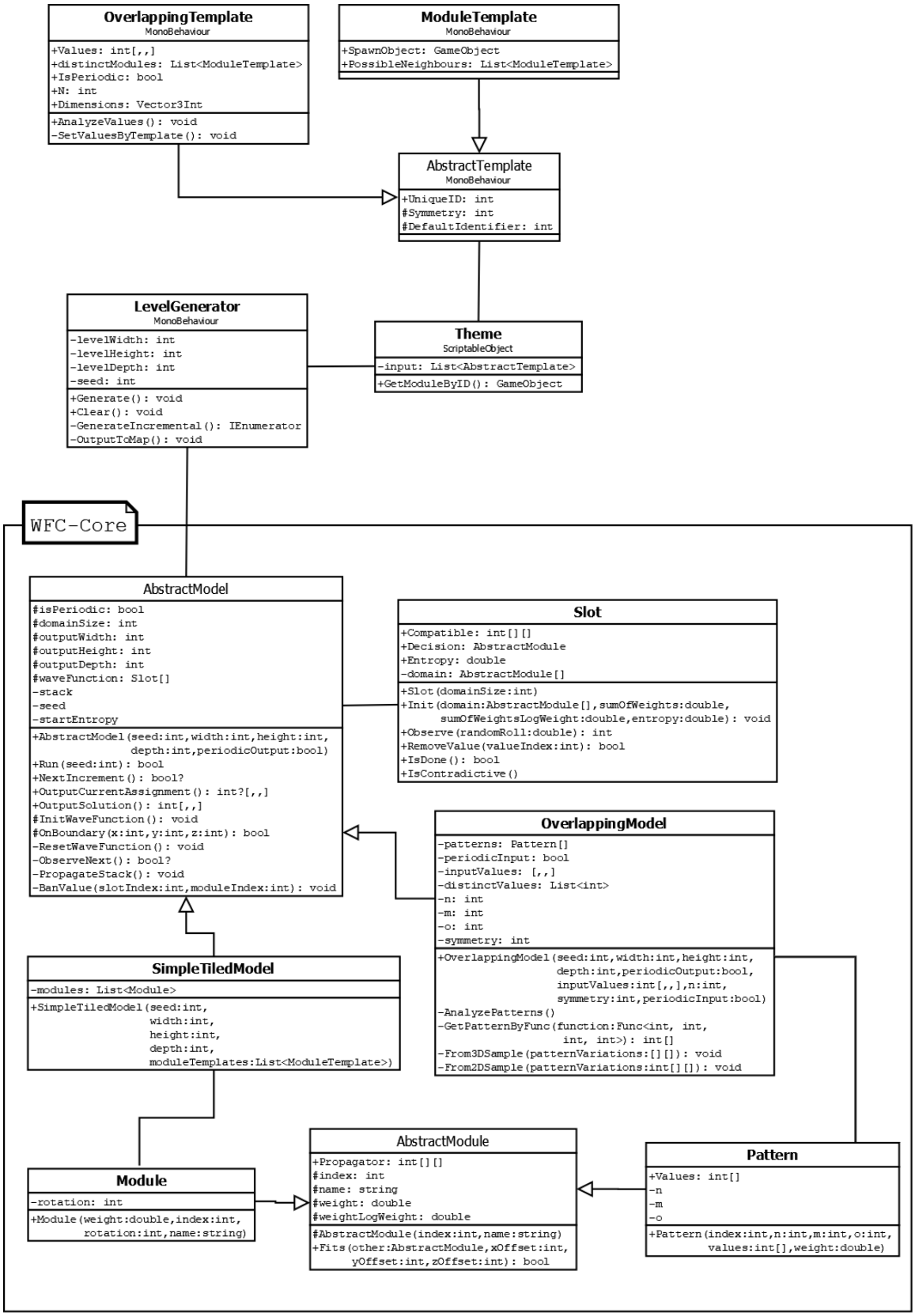


Abbildung 5.6: Geplantes UML-Diagramm von *KHollapse*, eigene Grafik

6 KHollapse - Implementierung

In diesem Kapitel wird die konkrete Implementierung der Software *KHollapse* dargelegt. Zunächst werden dafür die einzelnen Software-Module vorgestellt, erläutert und gegebenenfalls Unterschiede zur geplanten Software-Architektur begründet. Die Vorstellung beginnt mit der Kern-Bibliothek des Algorithmus und widmet sich danach dem Generator. Daraufhin folgt eine begründete Erläuterung zu konzeptuellen Abweichungen. Abschließend werden einige generierte Resultate gezeigt.

6.1 WaveFunctionCollapse-Core

Der Kern des Generators ist eine eigene WFC-*Unity*-Portierung, in dessen Mitte das *AbstractModel*, sowie die Implementierungen *SimpleTiledModel* und *OverlappingModel* stehen. Die Variablen des CSPs werden durch die Klasse *Slot* repräsentiert, um die Terminologie *Stålbergs* aufzugreifen. *AbstractModule*, *Module* und *Pattern* dienen zur Beschreibung der Module. Eine Abweichung zum konzipierten UML-Diagramm besteht in der Einführung der Hilfsklassen *SymmetrySystem* und *Orientations*.

6.1.1 Hilfsklassen

Um die Aufgaben der einzelnen Model-Klassen noch weiter zu trennen, wurden die beiden Klassen *SymmetrySystem* und *Orientations* eingeführt, die sowohl von der Kern-Bibliothek als auch von anderen Klassen in *KHollapse* genutzt werden.

Orientations

Orientations ist eine statische Klasse, die eine global einheitliche Beschreibungsform für mit Zahlen codierte Richtungen liefert. Alle Richtungsangaben in *KHollapse* werden durch die Zahlen 0-5 beschrieben, die durch ein Enum repräsentiert werden. Die Klassenvariablen und Klassenmethoden aus *Orientations* bieten bspw. Möglichkeiten, um zu prüfen, ob eine Richtung vertikal bzw. horizontal ist oder ihre entgegengesetzte Richtung zu ermitteln. Abbildung 6.1 zeigt das entsprechende UML-Diagramm.

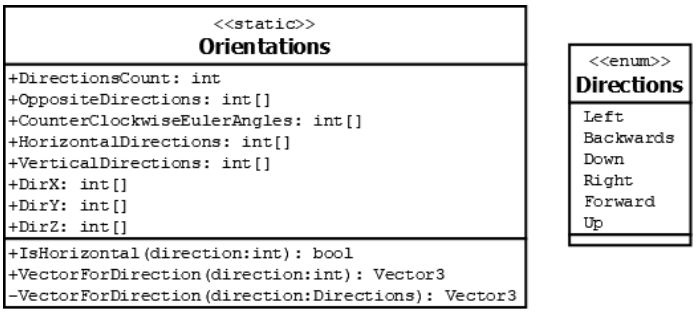


Abbildung 6.1: UML-Diagramm der Klasse *Orientations*, eigene Grafik

SymmetrySystem

Das Symmetrie-System beschreibt, welche und wieviele eindeutige Rotationen bzw. Reflexionen ein Modul hat. Im zweidimensionalen Raum hat ein gegebenes Modul jeweils bis zu vier verschiedene Rotationen und Reflexionen, was zu einer maximalen Kardinalität von acht führt. Die acht verschiedenen Kardinalitäten lassen sich herbeiführen, indem ein Modul zuerst schrittweise um 90° gegen den Uhrzeigersinn rotiert und danach jede Rotation entlang der y-Achse gespiegelt wird. Abbildung 6.2 verdeutlicht das System anhand eines Moduls.

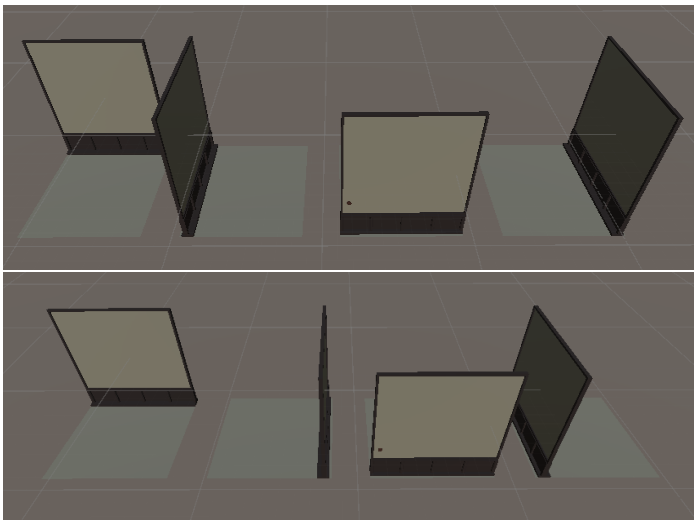


Abbildung 6.2: Rotationen und Reflexionen eines Moduls, eigene Grafik

Um die Symmetrie-Stufe eines Moduls eindeutig zu beschreiben werden die unterschiedlichen Symmetrien durch die Zeichen L, T, I, / und X dargestellt. In Abbildung 6.3 ist illustriert, wie das System für Buchstaben funktioniert. Die Beschreibungsform durch Zeichen verwendet Herr *Gumin* bereits in seiner Original-Implementierung. Sie wurde auch in *KHollapse* adaptiert.

Buchstabe	1	2 (90°)	3	4	5	6 (Flip)	7	8	Kardinalität
T	T	⊢	⊥	⊣	T	⊣	⊥	⊢	4
L	L	⌟	⅂	⌜	⅂	⌜	L	⌟	4
I	I	⊢	I	⊢	I	⊢	I	⊢	2
/	/	\	/	\	/	\	/	\	2
X	X	X	X	X	X	X	X	X	1
P	P	ɹ	d	ɔ	ꟼ	ɟ	b	ɔ	8

Abbildung 6.3: Visualisierung des Symmetrie-Systems, eigene Grafik

In der obigen Abbildung taucht zusätzlich das Zeichen P auf, durch welches ein Modul mit der Kardinalität acht beschreibbar ist. In *KHollapse* ist der Einsatz dessen bisher jedoch nicht nötig, weshalb es nicht im System auftaucht.

Obwohl die Module im Levelgenerator dreidimensional sind, reicht das zweidimensionale Symmetrie-System aus. Im dreidimensionalen Raum besitzt ein Objekt zusätzliche Symmetrie-Ebenen, die durch Spiegeln und Rotieren entlang anderer Achsen noch weitere Permutationen des Moduls erzeugen. Für *KHollapse* sind diese allerdings nicht sinnvoll, da sie z.B. Module um 180° entlang der z-Achse rotieren (vgl. Abbildung 6.4).

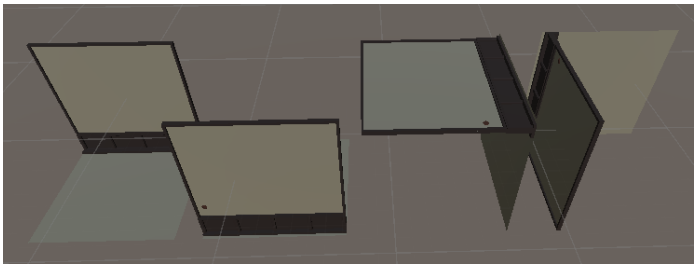


Abbildung 6.4: Ungewünschte Permutationen durch zusätzliche Achsen im dreidimensionalen Raum, eigene Grafik

Gumin berechnet die Symmetrien der Module für jedes Modul erneut. Die Ergebnisse seiner Formeln liefern jedoch für die gleiche Symmetrie jedes Mal die gleichen Ergebnisse im Wertebereich [0, 3]. Die Werte beschreiben, welche Permutation eines Moduls nach der jeweiligen Rotation bzw. Reflexion resultiert. Da in dem verwendeten Symmetrie-System nur Module mit einer maximalen Kardinalität von vier auftauchen, gibt es auch maximal vier Permutationen eines Moduls. Die Permutation drei ist stellvertretend für die 270° gegen den Uhrzeigersinn rotierte Variante eines Moduls. Die Klasse *SymmetrySystem* berechnet einmalig Symmetrie-Maps für die unterschiedlichen Symmetrien und speichert sie. Eine Map besteht aus acht Werten, wovon die ersten vier aus den Rotationen und die letzten aus den Reflexionen resultieren.

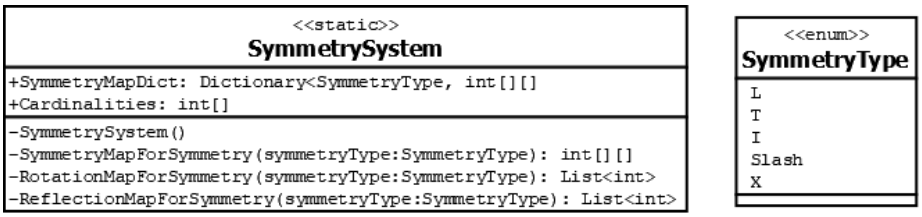


Abbildung 6.5: UML-Diagramm der Klasse *SymmetrySystem*, eigene Grafik

6.1.2 Slot und Modul

Slots und *Modules* sind die Variablen bzw. Werte des CSPs. Anders als in der ursprünglichen Variante von WFC wurden für *KHollapse* einzelne Klassen für sie entworfen. Die Implementierung dieser folgt dem in Abbildung 5.6 gezeigten UML-Diagramm.

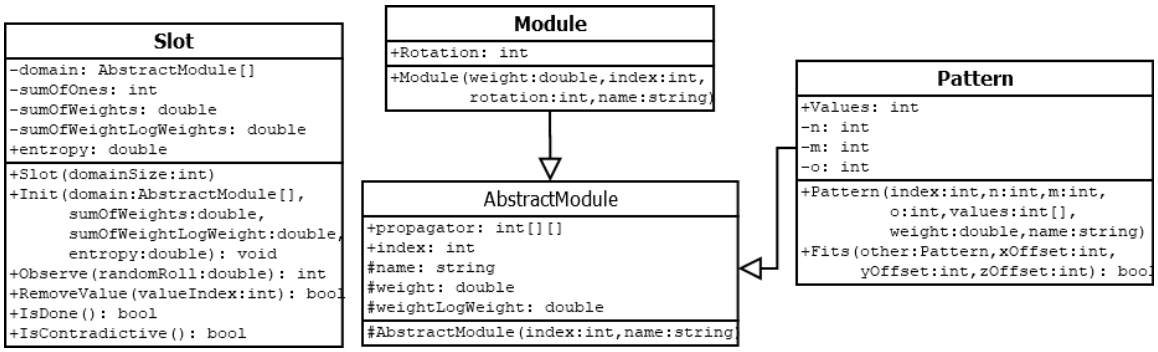


Abbildung 6.6: UML-Diagramm der *Slots* und *Modules*, eigene Grafik

Slot

Die Variablen der Wellenfunktion werden durch *Slots* repräsentiert. Sie speichern die verbleibende Entropie, den aktuellen Zustand ihrer Domäne und das kollabierte Modul. Der Zustand der Domäne wird durch das verzweigte Integer-Array *Compatible* dargestellt. Es gibt an, wie viele verbleibende Nachbarn der Slot für ein gegebenes Modul in eine gegebene Richtung noch übrig hat.

Die Methoden *IsDone()* und *IsContradictive()* geben Auskunft über den Zustand des Slots. *Observe()* summiert die Gewichte der verbleibenden möglichen Module und wählt zufällig eines davon aus, um den Slot zu kollabieren. Der gewählte Wert wird gespeichert, alle anderen werden später vom Model durch Aufrufen der Methode *RemoveValue()* entfernt.

AbstractModule

Das *AbstractModule* ist die abstrakte Basis-Klasse für die Modul-Implementierung des jeweiligen Models. Es kapselt die gemeinsamen Werte in einer Klasse und ermöglicht durch den Einsatz von Polymorphie im *AbstractModel*, dass keine Type-Casts in ein bestimmtes Model nötig sind.

Der Konstruktor weist jedem Modul einen Index und einen Namen zu, damit es einfacher zu identifizieren ist. Das verzweigte Integer-Array *Propagator* ist eine Index-Struktur, die die kompatiblen Module zur jeweiligen Modul-Instanz in eine bestimmte Richtung speichert.

Pattern

Pattern ist die Implementierung des *AbstractModules* für das *OverlappingModel*. Das Integer-Array *Values* speichert die im Muster auftretenden Module. Die Methode *Fits()* prüft, ob ein anderes Muster in Richtung des jeweiligen Abstandes passend überlappt. Diese Methode wird im Konstruktor des *OverlappingModels* aufgerufen, um den Propagator für die analysierten *Patterns* zu erstellen.

Module

Die *Module*-Klasse wird im *SimpleTiledModel* verwendet. Bei einem *Module* ist der Index die einzigartige ID des zugehörigen *ModuleTemplates*. Weiterhin überschreibt die Klasse den Konstruktor und nimmt zusätzlich einen Integer-Wert für die Rotation entgegen, der beschreibt, um welche Symmetrie des Moduls es sich handelt.

6.1.3 Model

AbstractModel und seine beiden Implementierungen, *SimpleTiledModel* und *OverlappingModel*, wurden wie geplant umgesetzt. Der einzige Unterschied ist die Änderung des zurückgegebenen Datentyps für die Output-Methoden. Anstelle eines dreidimensionalen Integer-Arrays wird ein dreidimensionales 2-Tupel, dessen beiden Werte Integer sind, genutzt. Das erste Objekt im Tupel ist die ID des Output-Moduls und das Zweite die Symmetrie-Permutation. Dies hat den Vorteil, dass die entwickelte Anwendung sich nicht um das Speichern aller Permutationen jedes Moduls kümmern muss, sondern diese nur lokal im Model verwendet werden.

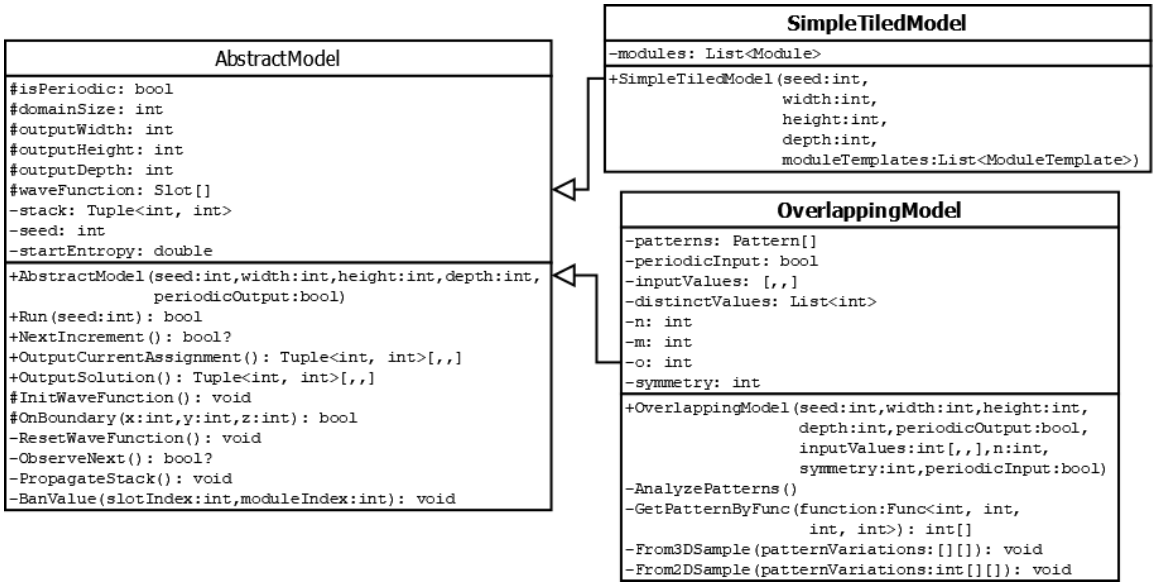


Abbildung 6.7: Aufbau des Models als UML-Klassendiagramm, eigene Grafik

AbstractModel

Die abstrakte Basis-Klasse kapselt das modelunabhängige Verhalten von WFC, also die Observierung und Propagierung. Weiterhin stellt es abstrakte Methoden zur Erzeugung des Outputs bereit. In *Run()* wird zunächst die Wellenfunktion erzeugt und initialisiert und danach der Main-Loop des Programmes, bestehend aus *ObserveNext()* und *PropagateStack()*, durchgeführt, bis entweder ein Widerspruch oder eine Lösung auftaucht. *NextIncrement()* ist der gleiche Main-Loop, allerdings inkrementell. Diese Funktion dient vor allem zur Visualisierung des Algorithmus.

ObserveNext() sucht den Slot mit der geringsten Entropie und prüft dabei gleichzeitig, ob einer der Slots einen Widerspruch gefunden hat. An dem Slot mit der geringsten Entropie wird *Observe()* aufgerufen, um den Slot auf einen zufälligen Wert zu kollabieren. Im Anschluss werden alle verbleibenden Werte des Slots aus seiner Domäne gebannt und dem Propagierungs-Stack hinzugefügt. Nach der Observierung folgt die Propagierung, in der die Implikationen der entfernten Werte propagiert werden.

OverlappingModel

Das *OverlappingModel* überschreibt den Konstruktor der Basisklasse und nimmt als zusätzliche Parameter ein zweidimensionales Array ganzzahliger Inputwerte, die Mustergröße, die Anzahl der zu berücksichtigenden Symmetrien und einen boolschen Wert, der beschreibt, ob es sich um einen periodischen Input handelt. Zu Beginn werden die Inputwerte in eine null-basierte Index-Struktur umgewandelt, so dass der erste Wert durch null statt seines ursprünglichen Wertes repräsentiert wird. Danach folgt die Analyse der Inputwerte. Schrittweise wird jedes NxN-Muster geprüft, wobei für jedes neue der Domäne ein *Pattern* hinzugefügt und für jedes bereits gefundene seine Wahrscheinlichkeit erhöht wird. Neben N wurden die Werte M und O eingeführt, wobei i.d.R. $N = M = O$ gilt. Die Einführung dieser zusätzlichen Werte ermöglicht den Einsatz des gleichen Models für den drei- und zweidimensionalen Raum. Ist einer der drei Werte gleich 1, handelt es sich um ein zweidimensionales Model.

Bei der Analyse der Muster werden, basierend auf der angegebenen Symmetrie-Stufe, zusätzlich die Rotationen und Reflexionen des Musters berücksichtigt. Zweidimensionale Muster haben bis zu acht verschiedene Permutationen, dreidimensionale bis zu 20. In der Praxis hat sich gezeigt, dass für dreidimensionale Muster oftmals die Verwendung der ersten acht ausreichend ist, da sie ansonsten die gleichen unerwünschten Rotationen und Reflexionen herbeiführen, die in Abbildung 6.4 gezeigt wurden. Nach der Analyse der Muster werden diese der Domäne der Slots hinzugefügt und die Initialisierung durch den Konstruktor ist beendet. In der überschriebenen *OutputCurrentAssignment()*-Methode wird der Index des zugewiesenen *Patterns* wieder in die ID des *ModuleTemplates* umgerechnet.

SimpleTiledModel

Das *SimpleTiledModel* ist die zweite Implementierung des *AbstractModels*. Es nimmt ein *TilingTheme* entgegen und erstellt für jedes *ModuleTemplate* des *TilingThemes* ein *Module* und fügt dieses einer Liste hinzu. Da ein Theme nur die nullte Symmetrie eines Moduls beinhaltet, werden für alle *ModuleTemplates* basierend auf ihrer Kardinalität entsprechend viele Symmetrien erzeugt. Danach werden alle Adjazenz-Regeln des Themes in den internen Propagator umgerechnet. Die Initialisierung ist damit abgeschlossen. In *OutputCurrentAssignment()* weist das *SimpleTiledModel* jedem für jedes Element der Wellenfunktion dem Ergebnis-Tupel die ID des kollabierten Moduls und seine Rotation zu.

6.2 Der Levelgenerator

Die Umsetzung der WFC-Bibliothek reicht nicht, um damit in *Unity* Level zu generieren. In diesem Abschnitt werden der *Unity*-Levelgenerator und die von ihm verwendeten Klassen erläutert. Die ursprünglich geplante Umsetzung stellte sich an einigen Stellen als problematisch heraus, weshalb von ihr abgewichen wurde. An den entsprechenden Stellen wird dies erläutert und begründet.

6.2.1 Inputs

Bei den Inputs lässt sich bereits die erste Abweichung finden. Geplant war eine zur WFC-Bibliothek parallele Architektur. Während der Entwicklung dieser hat sich herausgestellt, dass die beiden konkreten Implementierungen weniger gemeinsam hatten als gedacht. Deshalb wurde die Abstraktion durch die Klasse *AbstractTemplate* entfernt und stattdessen gibt es nun die beiden Klassen *Sample*, der Input des *Overlapping Models*, und *ModuleTemplate*, der Input des *SimpleTiled Models*.

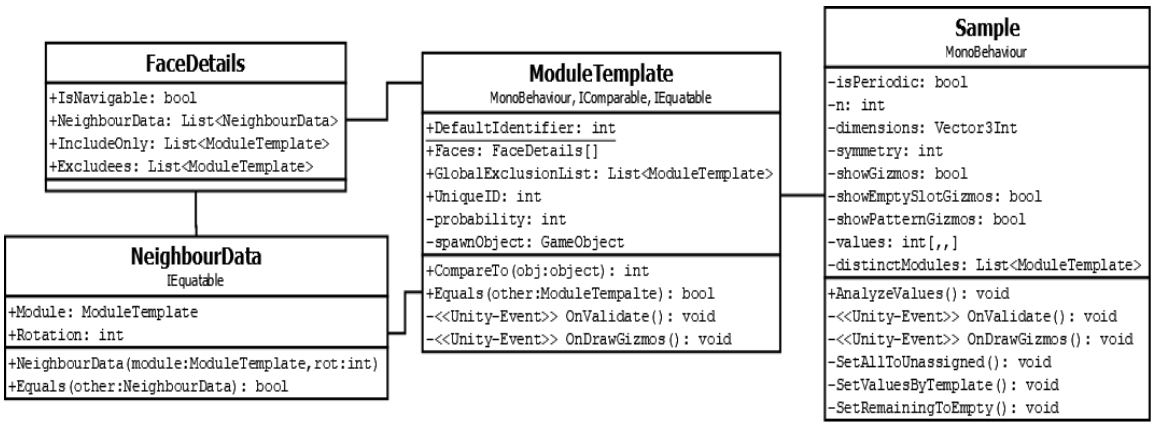


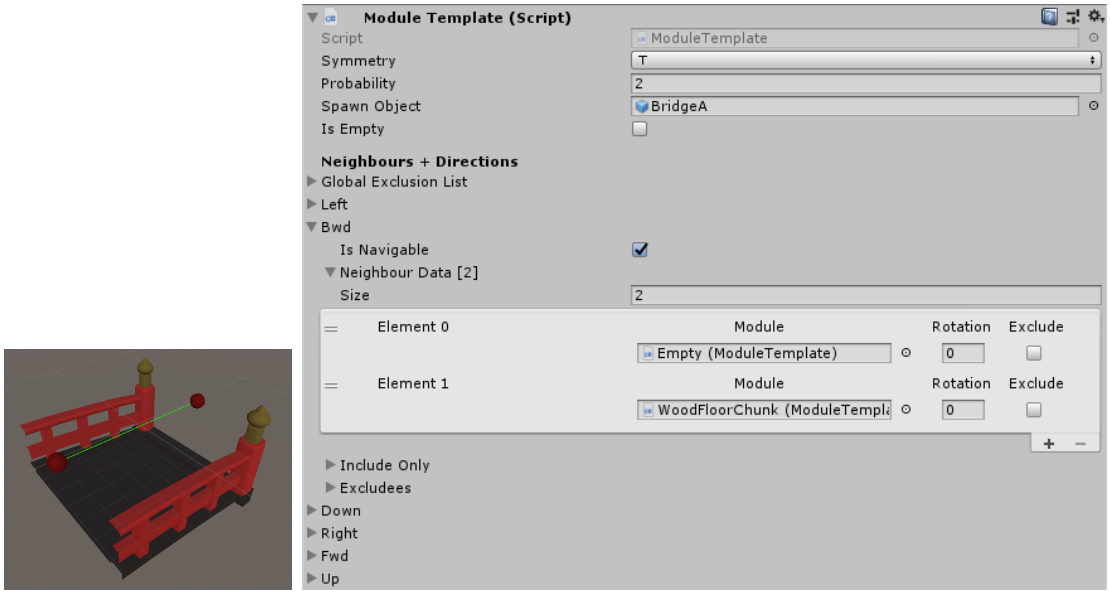
Abbildung 6.8: UML-Klassendiagramm der Inputs von *KHollapse*, eigene Grafik

FaceDetails

Diese Klasse beschreibt eine Seite eines Moduls. Sie speichert, ob ein Modul in die Richtung dieser Seite navigierbar ist oder nicht und welche Module mögliche Nachbarn für den an diese Seite angrenzenden Slot wären. Um einige *ModuleTemplates* auszuschließen, können sie der Liste *Excludees* hinzugefügt werden. Wenn nur wenige bestimmte Module passen, können diese stattdessen der Liste *IncludeOnly* hinzugefügt werden. Der Designer hat so die Möglichkeit, logisch mögliche, aber ästhetisch unerwünschte Nachbarn auszuschließen. Die Nachbarschafts-Daten sind in einer Liste des Typs *NeighbourData* gespeichert, einer Datenklasse die jeweils ein Modul und eine Rotation beinhaltet. Hätte eine *FaceDetails*-Instanz ein *NeighbourData*-Objekt mit den Werten Modul A, Rotation 2, hieße dies, dass das Modul A um 180° gedreht, ein möglicher Nachbar sei.

ModuleTemplate

Ein *ModuleTemplate* ist ein *MonoBehaviour*, welches einem Modul hinzugefügt wird, um es als solches zu markieren. Es speichert die einzigartige ID, die Wahrscheinlichkeit, die Symmetrie und das 3D-Model des Moduls. Weiterhin speichert ein *ModuleTemplate* für jede der sechs Richtungen eine *FaceDetails*-Instanz. Die ID des Objektes wird automatisch vom *ModuleSetup* gesetzt (vgl. Abschnitt 6.2.4). Standardmäßig ist das 3D Model das des Prefabs, kann aber bei Bedarf vom Designer geändert werden.

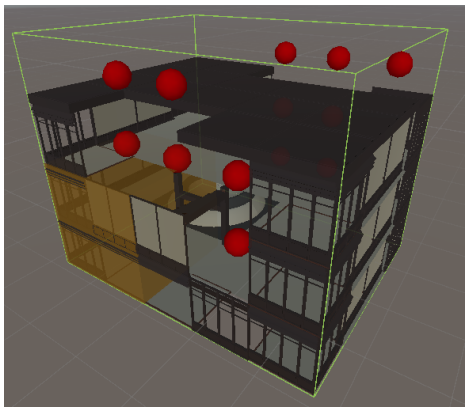


(a) *(b)*

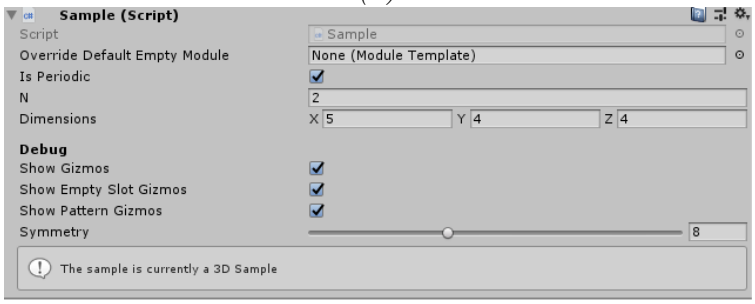
Abbildung 6.9: (a) Modul mit Navigierbarkeit (b) *ModuleTemplate*-Inspektor, eigene Grafik

Sample

Ein *Sample* besteht aus mehreren, manuell zusammengesetzten *ModuleTemplates*. Um aus einem *Sample* Input für das *OverlappingModel* zu generieren, muss jeder Slot im *Sample* mit einem *ModuleTemplate* belegt sein. Ihre IDs dienen als Inputwerte des *Samples*. Im Inspektor werden die Größe, der gewünschte Wert für N, die zu berücksichtigenden Symmetrien und ob es sich um einen periodischen Input handelt, eingestellt. Zur Visualisierung lassen sich außerdem einige Gizmos an- bzw. ausschalten. Sie zeigen die Grenzen des *Samples*, die Größe der zu analysierenden Patterns und wo sich noch freie Slots im *Sample* befinden. Zur Laufzeit wird automatisch allen leeren Slots das *EmptyModule* zugewiesen, damit ein Leveldesigner an gewollt leeren Stellen (bspw. Luft) keine leeren Module platzieren muss. Standardmäßig wird das in *WFCToolSettings* gesetzte Modul gewählt, kann aber mit dem Feld *OverrideDefaultEmptyModule* für bestimmte *Samples* überschrieben werden. Abbildung 6.10a zeigt ein *Sample* und seine Gizmos. Leere Plätze werden als rote Sphäre, die Mustergröße als gelbe Box und die *Sample*-Größe als grüner Kasten angezeigt.



(a)



(b)

Abbildung 6.10: (a) Beispiel eines *Samples* in *KHollapse* und (b) der dazugehörige Inspektor, eigene Grafik

6.2.2 Themes

Da sich die Architektur der Inputs geändert hat, folgte auch eine Anpassung in der Struktur der *Theme*-Klasse. Aufgrund der Trennung zwischen *ModuleTemplate* und *Sample* braucht es einen anderen Weg, dem Generator zu ermöglichen beide Inputs zu nutzen. Hierfür wurden die Themes in eine Basisklasse *Theme* und die Subklassen *TilingTheme* und *OverlappingTheme* aufgebrochen. Alle drei sind *ScriptableObjects* und ermöglichen somit das Erstellen einer Asset-Datei. Verschiedene Themes können dem Projekt einfach durch neue Assets des jeweiligen Themes hinzugefügt werden.

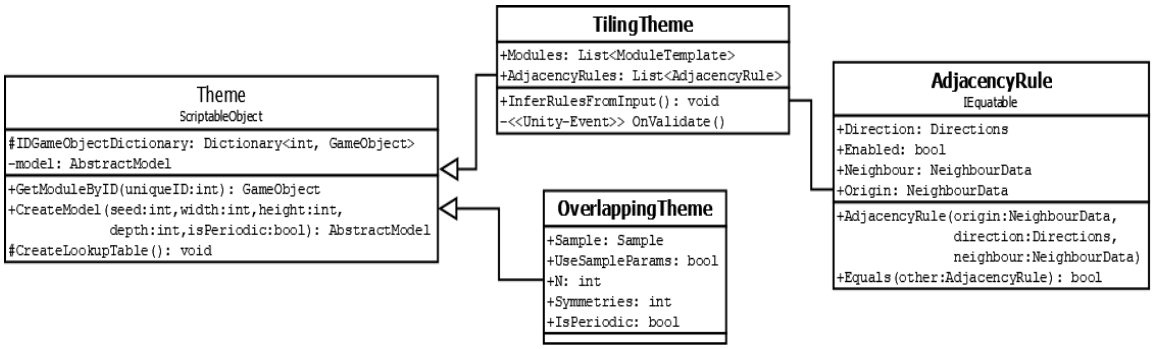


Abbildung 6.11: Neue Struktur der *Themes* als UML-Diagramm, eigene Grafik

Theme

Die abstrakte Klasse *Theme* bildet die Basis der *Theme*-Implementierung. Sie stellt ein Mapping zwischen den IDs der verwendeten Module und ihren GameObjects her. *GetModuleByID()* gibt das 3D-Modell zu einer ID zurück und wird vom Generator genutzt, um den Output des Models in Geometrie zu übersetzen. Die abstrakten Methoden *CreateModel()* und *CreateLookUpTable()* werden von den Subklassen implementiert. Erstere ruft den Konstruktor des jeweiligen Models auf und übergibt seine Werte daran, letztere erstellt das Mapping.

TilingTheme

Das *TilingTheme* ist die Implementierung für das *SimpleTiledModel*. Es besteht aus einer Liste der verwendeten *ModuleTemplates* und einer Liste der *AdjacencyRules*. Designer können die aufgestellten Regeln hier verändern oder neue hinzufügen.

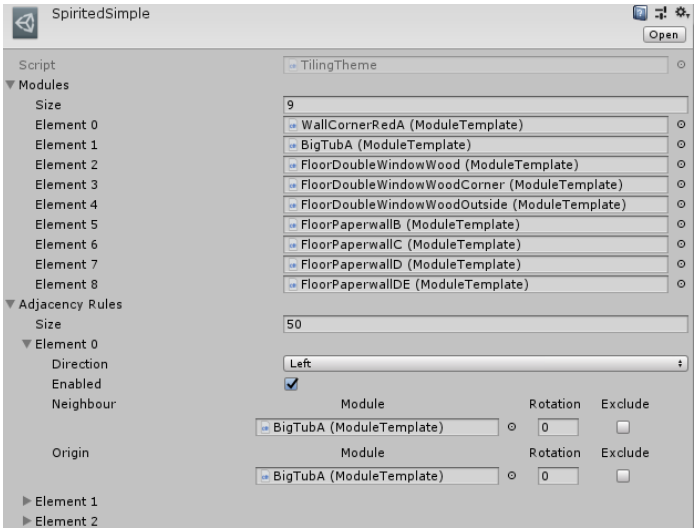


Abbildung 6.12: Inspektor eines *TilingThemes*, eigene Grafik

AdjacencyRule

Eine *AdjacencyRule* setzt zwei *ModuleTemplates* miteinander in Relation und beschreibt, dass eine Nachbarschaft zwischen ihnen in die angegebene Richtung möglich ist. Eine *AdjacencyRule* ist ähnlich zur *NeighbourData* und wird deshalb aus ihr abgeleitet. Aus einer aufgestellten *NeighbourData* lassen sich durch das verwendete Symmetrie-System sieben weitere Regeln ableiten.

OverlappingTheme

Jedem *OverlappingTheme*-Asset muss ein *Sample* zugewiesen werden, da aus diesem der Input und das Model für den Generator zusammengestellt werden. Die Parameter des *Samples* sind überschreibbar, damit das gleiche *Sample*-Prefab für verschiedene Outputs genutzt werden kann. Wird im Inspektor der Wert *UseSampleParams* nicht gesetzt, werden die im *OverlappingTheme* festgelegten Parameter für das *OverlappingModel* genutzt, ansonsten die direkt auf dem *Sample* definierten Werte.

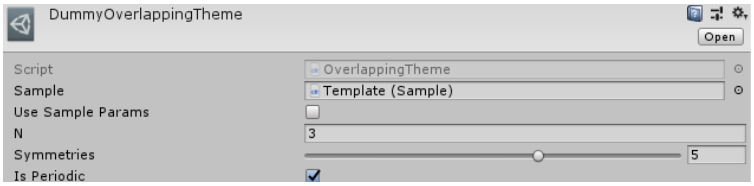


Abbildung 6.13: *OverlappingTheme* im Inspektor, eigene Grafik

6.2.3 Der Generator

Damit aus all diesen Komponenten auch wirklich ein Level erstellt werden kann, implementiert der Generator die Logik, um das *Model* der *Themes* zu berechnen und den erhaltenen Output in Levelgeometrie umzuwandeln.

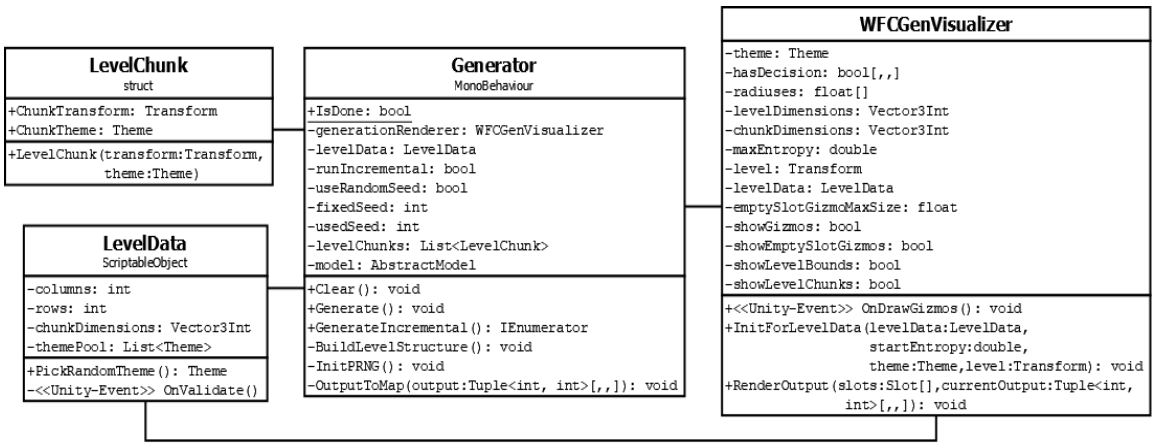


Abbildung 6.14: *Generator* und seine Komponenten als UML-Diagramm, eigene Grafik

LevelData

Um das Erstellen neuer Level so unkompliziert wie möglich zu gestalten, wurde das *ScriptableObject LevelData* eingeführt. Ein *LevelData*-Asset liefert dem Generator Informationen über die Maße des Levels und bietet einen Pool verwendbarer *Themes* für dieses Level. Dabei implementiert *LevelData* bereits die Möglichkeit, die Anzahl der Zeilen und Spalten eines Levels anzugeben, wobei in jede entstehende Zelle mit einem Level-Chunk gefüllt wird. Die Größe der Chunks kann mit *ChunkDimensions* eingestellt werden. Für jeden Chunk wird ein *Theme* aus dem Pool gewählt. Der *Theme*-Pool gestaltet die generierten Level abwechslungsreicher.

Bei der Implementierung hat sich die Abspaltung dieser Klasse aus der Generator-Logik als sinnvoll erwiesen, da über Assets einfacher neue Level mit anderen Parametern erstellt werden können. Der Generator braucht sich somit nicht mehr um seine Daten, sondern nur noch die Logik kümmern. Außerdem erlaubt die Verwendung der abstrakten Basisklasse für den *Theme*-Pool, dass der Generator Level sowohl aus *TilingThemes* als auch aus *OverlappingThemes* erstellen kann.

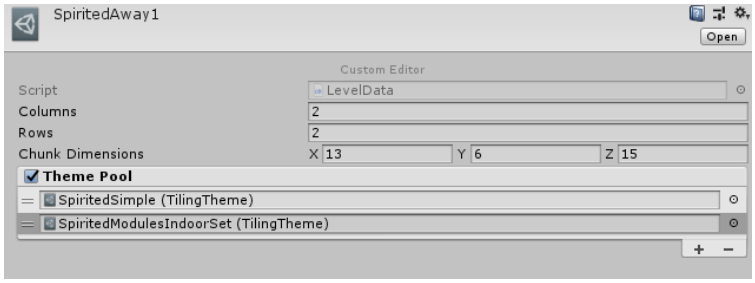


Abbildung 6.15: *LevelData*-Inspektor, eigene Grafik

Generator

Die Klasse *Generator* regelt den Ablauf der Generierung. Sie nimmt ein *LevelData*-Asset und baut basierend auf dessen Parametern zuerst die grundlegende Struktur aus Chunks und weist jedem Chunk ein Theme aus dem Pool zu. Daraufhin werden die Models der Themes initialisiert, berechnet und anschließend durch die Methode *OutputToMap()* in Levelgeometrie umgewandelt. Zur Abgabe dieser Thesis funktioniert der Generator zwar nur mit einem einzigen Level-Chunk, wurde der Erweiterbarkeit halber jedoch so entwickelt, dass er theoretisch mehrere unterstützt.

Um den Algorithmus bei der inkrementellen Generierung zu visualisieren, gibt der Generator nach jedem Schritt den aktuellen Output an den *WFCGenVisualizer*.

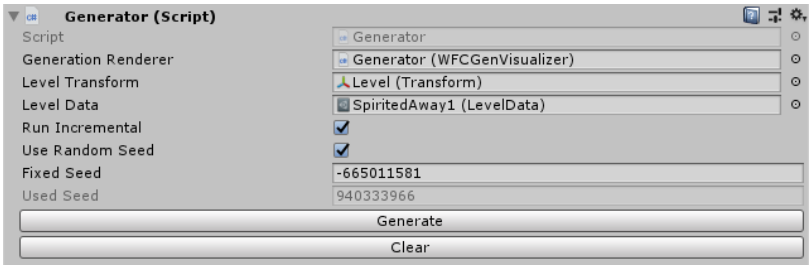


Abbildung 6.16: *Generator* in der Inspektor-Ansicht, eigene Grafik

WFCGenVisualizer

Der *WFCGenVisualizer* kümmert sich zur Laufzeit um die Visualisierung des Algorithmus und zur Editor-Zeit um die Visualisierung des Levels und seiner Chunks. Bei der Generierung werden die unbestimmten Slots mit einer Sphere visualisiert, dessen Größe die verbleibende Entropie des Slots widerspiegelt. Alle Gizmos können über den Inspektor an- bzw. ausgeschaltet werden (vgl. Abb. 6.17).

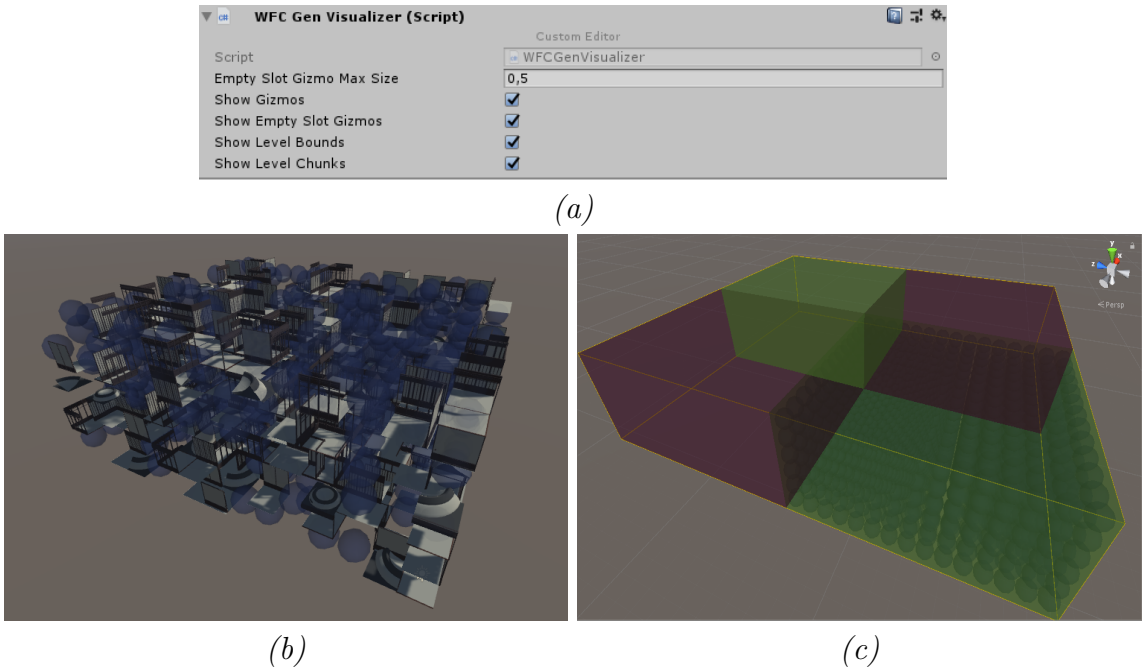


Abbildung 6.17: (a) *WFCGenVisualizer*-Inspektor, (b) Visualisierung d. Generierung und (c) Darstellung eines Levels mit Chunks, Größen und leeren Slots, eigene Grafik

6.2.4 Automatisierung und Tools

Der letzte Teil der Implementierung besteht aus der Erläuterung für *KHollapse* entwickelter Prozess-Automatisierungen und Tools.

ModuleSceneManager

Um das Hinzufügen neuer Themes so simpel wie möglich zu gestalten, wurde die Klasse *ModuleSceneManager* implementiert. Dieses Skript wird einem leeren *GameObject* in einer neuen *Unity*-Szene hinzugefügt. Anschließend werden diesem Objekt dann alle gewünschten Module des zu erstellenden *Themes* hinzugefügt. In der Szene sind die Module problemlos zu bearbeiten. Designer haben hier die Möglichkeit, die Navigierbarkeit, Symmetrie, Wahrscheinlichkeit etc. der Module für ihr Set zu konfigurieren. Befinden sich alle Module in der Szene, können die Nachbarschaftsregeln zwischen ihnen per Klick auf den Button „Set Neighbour Data“ berechnet werden. Die Berechnung geschieht auf Basis der Symmetrien der Module und ihrer Navigierbarkeit. Nachdem die Szene fertig bearbeitet wurde, kann durch ein Kontextmenü des Inspektors automatisiert ein neues *TilingTheme*-Asset mit den Modulen der Szene erstellt werden (vgl. Abb. 6.18). Der Name richtet sich nach der Szene, weshalb auch diese entsprechend benannt werden sollte.

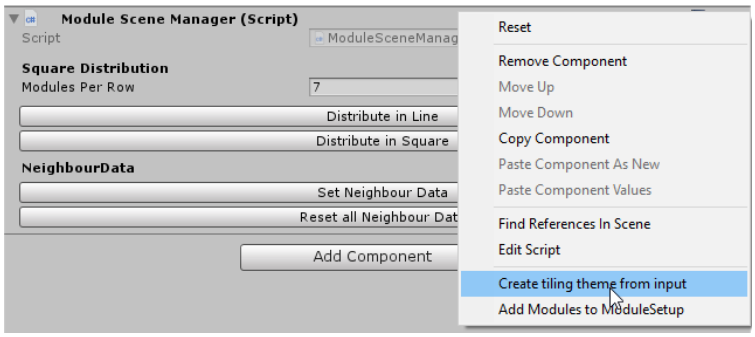


Abbildung 6.18: *ModuleSceneManager*-Inspektor, eigene Grafik

WFCToolSettings

Die *WFCToolSettings* bieten einen zentralen Ort zur Konfiguration aller für *KHollapse* relevanter Tool-Einstellungen. Hier sind die Farben, Radien etc. der unterschiedlichen Gizmos einstellbar (vgl. Abb. 6.19).

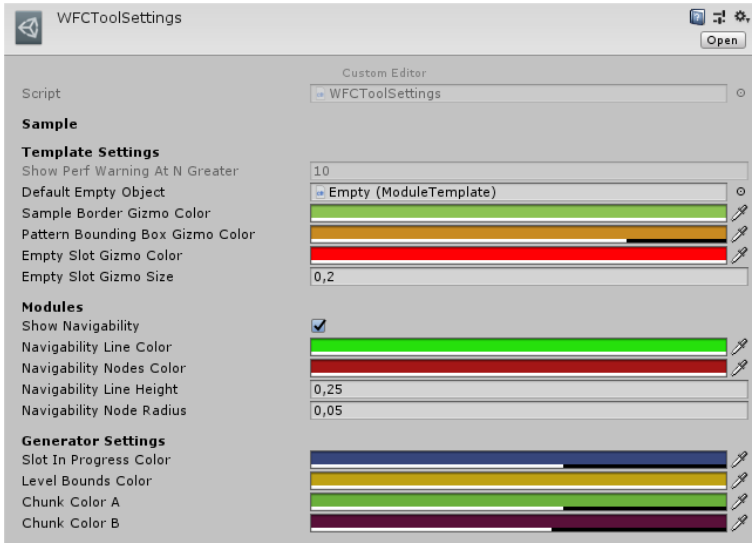


Abbildung 6.19: Inspektor der *WFCToolSettings*, eigene Grafik

ModuleSetup

Die zentrale Sammelstelle aller sich im Projekt befindenden Module bildet das Skript *ModuleSetup*. Seine Aufgabe ist es, allen Module ihre einzigartige ID zu geben. Diese Klasse arbeitet mit der Import-Pipeline zusammen, um direkt nach dem Import die neuen Module der Liste hinzuzufügen und ihre IDs zu setzen. Für den Fall, dass im Laufe des Projektes Fehler bzgl. der IDs auftreten, können diese jederzeit per Knopfdruck zurückgesetzt und neu verteilt werden (vgl. Abb. 6.20).

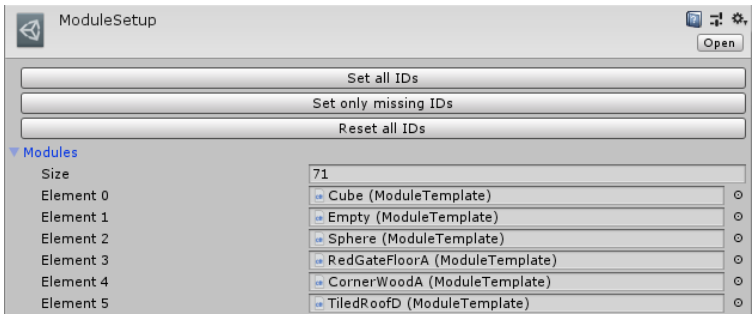


Abbildung 6.20: *ModuleSetup*-Inspektor, eigene Grafik

Import-Pipeline

Die Import-Pipeline wurde nach dem erläuterten Konzept umgesetzt. Das Hinzufügen neuer Module erfolgt durch das Importieren der entsprechenden Datei in den Ordner „Meshes“. Um Ordnung und Struktur im Projekt zu halten, werden mehrere Module in einer Datei gebündelt. So befinden sich z.B. alle zu einer Brücke gehörigen Module in der 3D-Datei „Modules_Bridges“. Beim Import wird ein Ordner mit dem Namen der Datei angelegt (z.B. „Modules_Bridges“) und die Module der 3D-Datei als Prefabs in diesem gespeichert. Dabei ist der Präfix „Modules“ eine allgemeine Namenskonvention, die beschreibt, dass es sich bei dem importierten Objekt um Levelgeometrie handelt. Für alle so importierten Objekte wird ein Check ausgeführt, ob die Namenskonvention für einen Collider erfüllt sei und gegebenenfalls dieser dem Prefab hinzugefügt. Sollte der Zielordner des Imports ein Unterordner von „Meshes/WFC“ sein, erhält das Prefab *ModuleTemplate*-Skript und wird dem *ModuleSetup* hinzugefügt. Letzteres weist nach dem Import aller Objekte automatisch den neuen Modulen ihre ID zu.

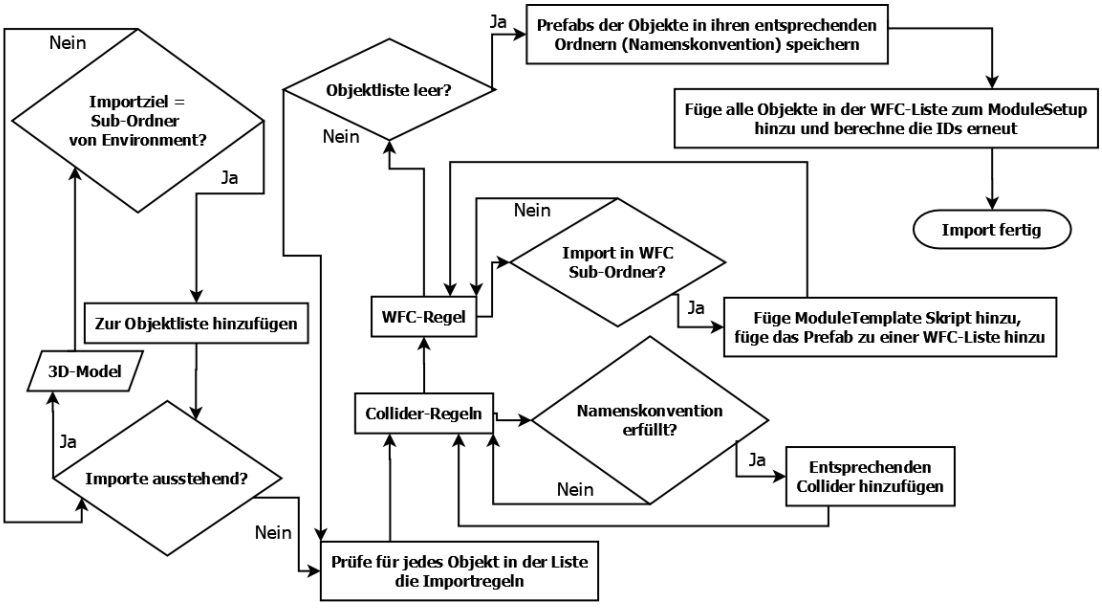


Abbildung 6.21: Flussdiagramm der Import-Pipeline aus *KHollapse*, eigene Grafik

6.3 Unterschiede zum Konzept

Das in Kapitel 5 vorgestellte Konzept wurde zum Großteil wie geplant implementiert. Die Abweichungen von der Software-Architektur wurden bereits erläutert. Abbildung 6.22 zeigt in einem gemeinsamen UML-Diagramm das finale Zusammenspiel der wichtigsten Klassen aus *KHollapse*. Das Diagramm ist nicht vollständig und zeigt bewusst nur Klassen ohne Details, da ein vollständiges UML-Diagramm sehr unübersichtlich und zu groß zur vernünftigen Darstellung wäre.

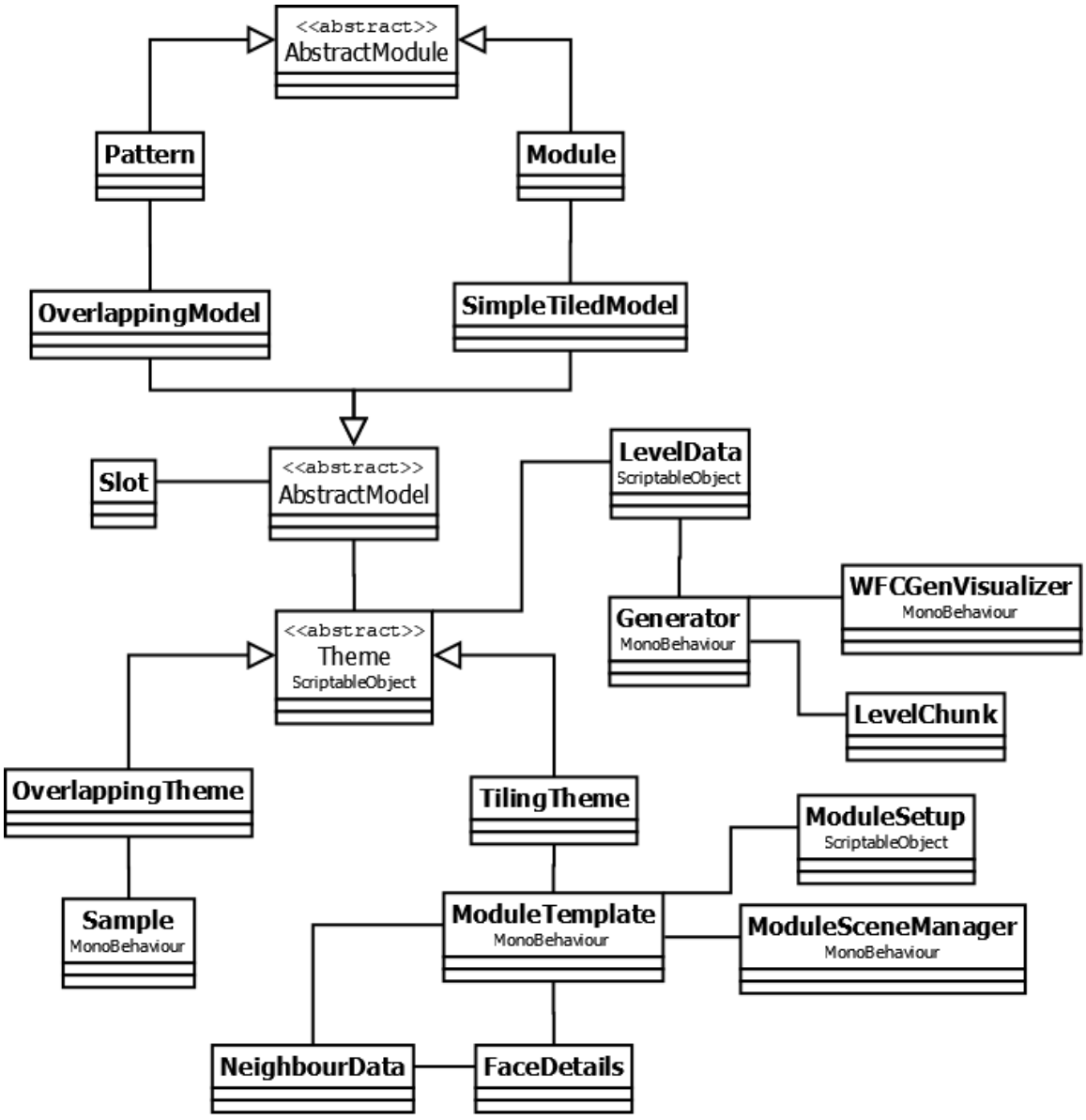


Abbildung 6.22: Zusammenspiel der relevantesten Klassen aus *KHollapse*, eigene Grafik

Neben den Abweichungen in der Architektur wurden zwei der drei geplanten Erweiterungen nicht implementiert, weshalb hier eine kurze Erläuterung dazu folgt.

Die Erste ist das geplante Constraint-Framework, welches dem Generator mehr Flexibilität geben sollte. Während der Entwicklung des Generators hat sich herausgestellt, dass dieses Feature weit umfangreicher sein würde, als geplant. In der Planung wurde angenommen, dass der Umbau der Architektur reicht, um den Generator offen und erweiterbar genug für das Framework zu gestalten. Eine saubere Implementierung dessen erfordere jedoch viel Zeit und wäre wahrscheinlich schon ein eigenständiges Projekt. Da die verschiedenen Ziele des Generators streng priorisiert werden mussten, wurde das Constraint-Framework als Erweiterung aus dem Generator gestrichen. Für die Beantwortung der Leitfrage ist es nicht unbedingt notwendig, auch wenn es sicherlich die Ergebnisse verbessert hätte.

Die zweite nicht implementierte Erweiterung ist Generierung des Levels in Chunks mit einem lösbaren Pfad. Da die Probleme und Herausforderungen dieser Erweiterung mit Hilfe des Constraint-Frameworks gelöst werden sollten, folgt aus der fehlenden Implementierung dessen, dass auch diese Erweiterung nicht umgesetzt werden konnte.

6.4 Resultate

Die Ergebnisse des Generators, der Quelltext und das Projekt sind im zur Thesis gehörigen Git-Repository zu finden. Die Abgabeversion hat das Tag „v1.0“.[30] Um das Projekt öffnen zu können wird mindestens die *Unity*-Version 2018.3.0f1 benötigt. Alle in der Arbeit verwendeten 3D-Modelle hat Jason Coffi entworfen.[31] Im Projekt sind sie im Order „*Assets\Meshes\WFC\SpiritedAway-Modules*“ zu finden.

Die im nachfolgenden präsentierten Ergebnisse zeigen den Stand des Generators zur Zeit der Abgabe der Thesis. Es ist auffällig, dass die generierten Level keine bis wenige sinnvolle Strukturen aufweisen. Das liegt daran, dass der Kern des WFC-Algorithmus zwar richtig implementiert ist, jedoch bei der Ableitung der Regeln in den *Themes* ein Software-Fehler existiert. Dieser Fehler sorgt dafür, dass mehr und falsche Regeln erkannt werden, wodurch die Nachbarschaft der Module teilweise untauglich wird. Aufgrund zeitlicher Begrenzungen konnte dieser Fehler für die Abgabe nicht mehr behoben werden. Nichtsdestotrotz zeigt die Implementierung, dass und wie mittels WFC Level generierbar sind. Hierbei hat sich herausgestellt, dass die Generierung durch das *SimpleTiledModel* besser performt. Das *OverlappingModel* ist schwierig an wichtige Erweiterungen wie die Navigierbarkeits-Heuristik anzupassen. Weiterhin ist es ein komplexer Prozess, die analysierten Muster eines *Samples* und seine Beziehungen zu anderen Mustern zu verstehen, wodurch es schwer kontrollierbar ist.

[30] Direktlink: `https://github.com/KevinHagen/KHollapse/releases/tag/v1.0`

[31] Bei Interesse finden sich weitere Arbeiten von Herrn Coffi unter `https://www.artstation.com/ganishka`

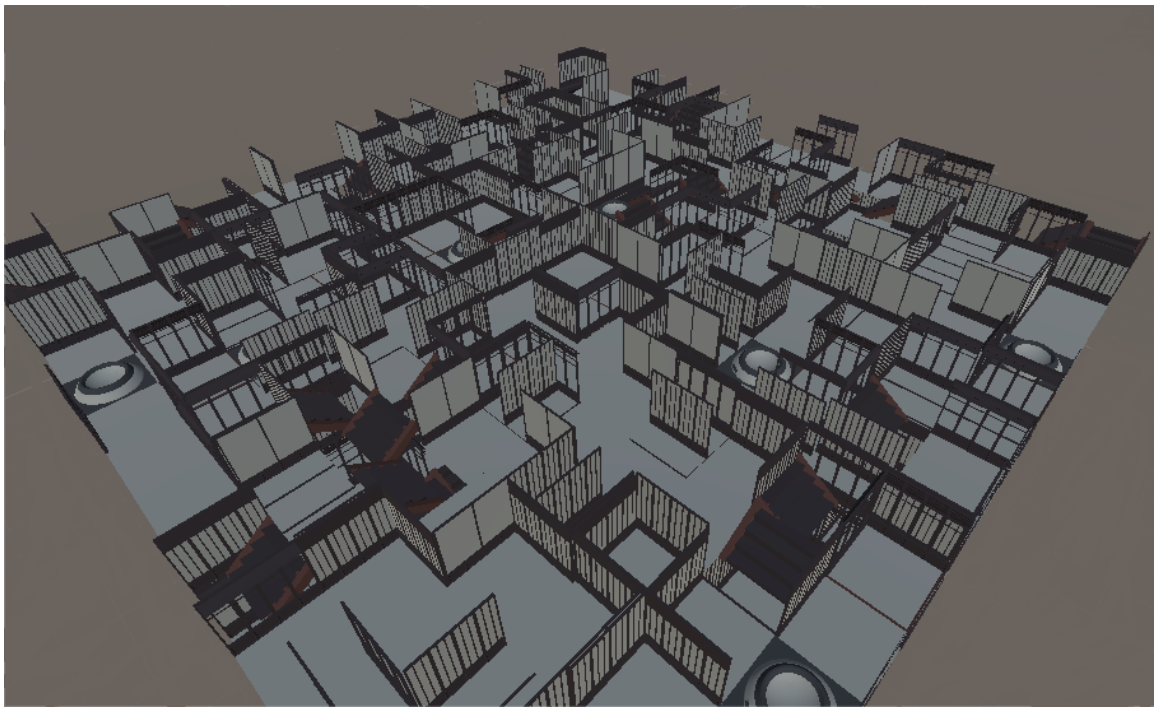


Abbildung 6.23: Ergebnis des Generators mit dem *TilingTheme* „SpiritedModulesIndoorSet“, eigene Grafik

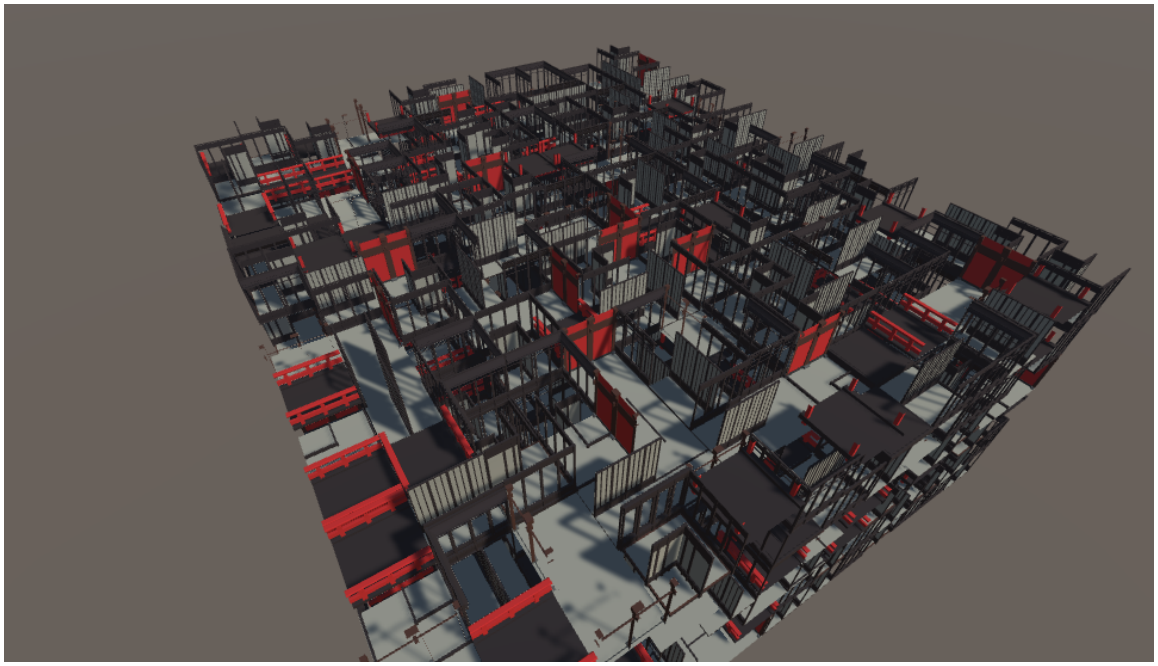


Abbildung 6.24: Ergebnis des Generators mit dem *TilingTheme* „SpiritedAll“, eigene Grafik

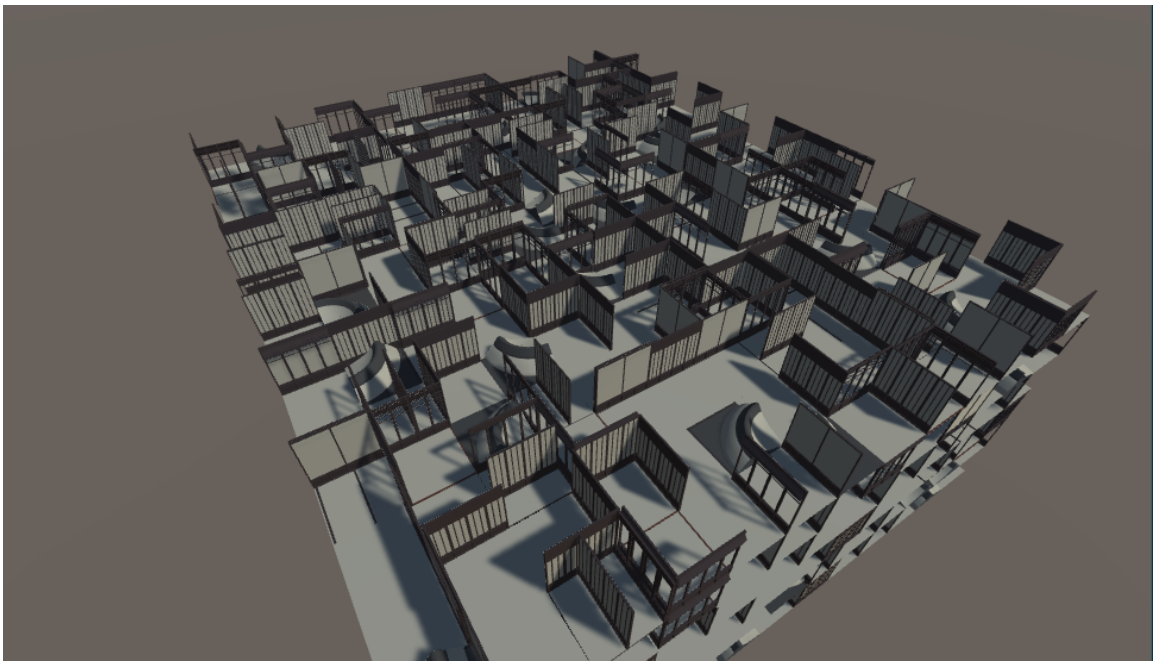
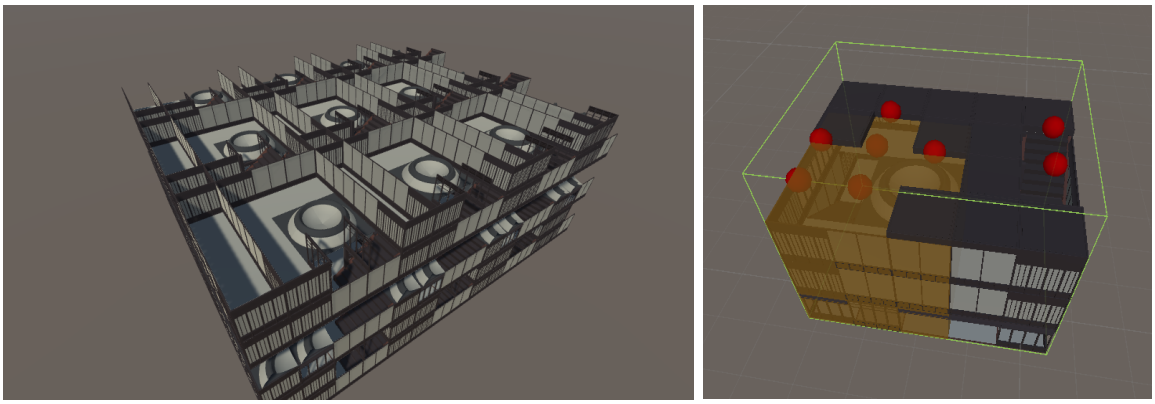


Abbildung 6.25: Ergebnis des Generators mit dem *TilingTheme* „SpiritedSimple“, eigene Grafik



(a) *(b)*

Abbildung 6.26: (a) Ergebnis des Generators mit dem *OverlappingTheme* „SpiritedOverlappingExample“ und (b) verwendetes Sample, eigene Grafik

7 Resümee

Dieses Kapitel bildet den Abschluss der Thesis. Zu Beginn erfolgt eine Evaluation des Algorithmus anhand eigener und fremder Ergebnisse, um zu bewerten, ob die Synthese generierter und handgebauter Welten mittels WFC als erfolgreich gilt und welche Schlüsse daraus zu ziehen sind. Daran anschließend werden Zukunftsaussichten für die mögliche Weiterentwicklung des Generators diskutiert. In einem Fazit werden abschließend die Arbeit und ihre Ergebnisse reflektiert.

7.1 Evaluation der Ergebnisse

In diesem Abschnitt werden die Software *KHollapse* und ihre produzierten Ergebnisse evaluiert. Anschließend werden die durch die Entwicklung gewonnenen Erfahrungen genutzt, um anhand der in Abschnitt 4.4 aufgestellten Kriterien zu bewerten, inwiefern sich der Algorithmus als sinnvolles Tool zur Synthese generierter und handgebauter Welten eignet.

7.1.1 KHollapse

Die Software *KHollapse* wurde bis auf einige kleinere Abweichungen sehr ähnlich wie im dargelegten Konzept implementiert. Die Resultate aus dem obigen Abschnitt zeigen, das mit WFC definitiv Level generiert werden können. Aufgrund eines Fehlers im Automatisierungsprozess der Nachbarschaftsregeln liefert der Generator zum aktuellen Stand jedoch keine oder nur sehr selten sinnvolle Ergebnisse. Die implementierten Tools und Workflows in *KHollapse* sind allerdings sehr hilfreich und erlauben auch Designern, sich effizient mit der Software auseinanderzusetzen. Da die Beurteilung dieser Ergebnisse die Beantwortung der Leitfrage erschwert, werden die Ergebnisse *Stålbergs* und *Kleinebergs* ebenfalls in Betracht gezogen.

7.1.2 Bewertung der Synthese

Obwohl *KHollapse* aufgrund eines Fehlers zum Stand der Abgabe keine sinnvollen Level generiert, können aus den bisherigen Zwischenresultaten bereits wertvolle Schlüsse gezogen werden. Weiterhin liefern die Ergebnisse anderer, im Laufe der Arbeit bereits diskutierter, Implementierungen ebenfalls hilfreiche Ergebnisse.

Show Stopper

Die erste Anforderung war, dass der Generator keine Level erstellt die *Show Stopper* beinhalten. Durch WFC ist dieses Problem bedingt lösbar. Die Zwischenresultate aus *KHollapse* weisen eine Menge *Show Stopper* auf, da die dort bestehenden Nachbarschaftsregeln häufig isolierte Räume unterschiedlichster Größen erzeugen. Diese Räume könnten dafür sorgen, dass der Spieler darin gefangen ist und das Ziel so nicht erreichen kann. Durch einen weiteren Algorithmus, der den Level auf diese Räume hin analysiert, um dann Start- und Zielpunkte entsprechend in einem zusammenhängenden, großen Raum zu platzieren, ließe sich das Problem beheben. Zusammen mit einer zerstörbaren Umgebung könnten die kleineren Räume für wertvolle Schätze oder starke Gegner genutzt werden. *Stålberg* erwähnt dieses Problem ebenfalls in seinem Talk über WFC. *Show Stopper* für *Bad North* wären bspw. Inseln, die nicht ausreichend navigierbar sind, keine Häuser oder zu wenig Strand enthalten. Seine Lösung ist ebenfalls die Verwendung eines weiteren Algorithmus der nach der Generierung prüft, ob das Level seinen Anforderungen entspricht. Ein weiterer Lösungsansatz könnte die Generierung eines garantiert lösbaren Pfades durch den Level sein, sofern dessen Wegpunkte erfolgreich als Constraints für WFC einsetzbar sind.

Einzigartigkeit und Immersion

Bereits das Experimentieren mit wenigen Modulen in *KHollapse* hat gezeigt, dass WFC in der Lage ist, mit einer geringen Auswahl an Inputs sehr diverse Outputs zu erzeugen. Da die Level auf lokalen Ähnlichkeiten eines Inputs basieren, ist ein gewisses Maß an Wiederholung jedoch inhärent. Die Einführung von Themes steigert die Abwechslung der erzeugten Geometrie stark. Sie bieten eine gute Möglichkeit, mehr Vielfalt in den Generator zu bringen, indem bestehende Module in verschiedenen Themes unterschiedlich kombiniert werden oder indem Sub-Themes aus bestehenden Themes gebildet werden. Generell liefern visuell unterschiedliche Themes logischerweise unterschiedliche Level, was die Einzigartigkeit dieser wiederum steigert. Sehr gute Beispiele hierfür sind *Kleinebergs* Städte-Generator und *Bad North*, die beide aus einem relativ kleinen Input-Set sehr unterschiedliche Levelgeometrie generieren.

Schwierigkeit

Inwiefern der Schwierigkeitsgrad mittels WFC in den Level integrierbar sei, bleibt eine ausstehende Frage. Die aktuelle Implementierung in *KHollapse* liefert hierfür zwar keinen Ansatz, die letzte geplante Erweiterung des Konzepts möglicherweise jedoch schon. Teil der Erweiterung war, dass der generierte Pfad durch die Level-Chunks als Constraint für WFC genutzt wird. Sollte es sich als möglich herausstellen im Vorfeld einen garantiert navigierbaren Pfad zu generieren, spricht in der Theorie nichts dagegen, weitere Pfade zu generieren, die als schwierigere bzw. leichtere Seitenarme des kritischen Pfades fungieren.

Risk and Reward und Interaktion

Die Experimente mit WFC haben deutlich gezeigt, dass der Algorithmus sich nicht eignet, um einzelne Elemente im Level zu verteilen. Die Stärken des Algorithmus liegen in der Generierung der Levelgeometrie und nicht darin Elemente in dieser zu verteilen. Für einige Ausnahmen mag der Algorithmus durchaus sinnvolle Ergebnisse bei der Platzierung von Risk and Reward Situationen oder anderer Elemente im Level liefern, im Normalfall ist dies jedoch nicht so. Hierum sollte sich nach der Generierung der Geometrie ein weiterer Algorithmus kümmern, wie es z.B. auch in *Cogmind* der Fall ist (vgl. Abschnitt 4.3.2).

Navigation

Zur Navigation durch die generierten Level lässt sich bisher nicht sonderlich viel sagen. WFC erstellt zuversichtlich Level, die ihre Vertikalität voll ausschöpfen. Jedoch ist schwierig zu beurteilen, ob die generierten Ergebnisse nicht zu verworrene, ähnliche Pfade beinhalten. *KHollapse* liefert hierfür keine interessanten Ergebnisse. In *Bad North* wird diese Frage ebenfalls nicht direkt beantwortet, da das Gameplay dies nicht verlangt. *Kleinebergs* Städte-Generator generiert häufig strukturell ähnliche Gänge und Passagen, wodurch die Orientierung zeitweise schwer fällt. Da sein Generator jedoch unendliche Welten generiert und die Anzahl der verwendeten Module nicht allzu hoch war, lässt sich auf Basis dieser Erkenntnis auch keine Beurteilung treffen.

Performance

WFC generiert die Level in angemessener Zeit. Die Zwischenresultate aus *KHollapse* wurden alle innerhalb weniger Sekunden erzeugt. Neben dem eigenen Generator beweist auch die Implementierung in *Bad North*, dass WFC performant genug ist, um mit anderen Systemen eines Spiels koexistieren zu können. *Kleinebergs* Städte-Generator zeigt außerdem eindrucksvoll, dass durch WFC nicht nur begrenzt große Level, sondern sogar unendliche Welten zur Laufzeit generierbar sind.

Dreidimensionalität

Die Erstellung dreidimensionaler Level ist für WFC ohne Probleme möglich. Den ursprünglichen Algorithmus um eine dritte Dimension zu erweitern, erfordert nur einige wenige Anpassungen und wurde auch mittlerweile von mehreren Entwicklern implementiert.

7.2 Weitere Aussichten

Zunächst sollte der in Abschnitt 6.4 erwähnte Fehler des Generators behoben werden. Danach bietet es sich an, die bisher fehlenden Erweiterungen des Algorithmus' zu implementieren und ihre Auswirkungen zu untersuchen (vgl. Abschnitt 6.3). Während der Entwicklung von *KHollapse* wurde festgestellt, dass sich WFC generell sehr gut eignet, um Level-Geometrie zu generieren. Die Generierung bestimmter Level-Strukturen ist bisher nur bedingt umsetzbar. Eine spannende Erweiterung wäre daher, mit einem anderen Algorithmus wie *Drunkard's Walk* oder ZA Strukturen zu erstellen, die WFC als Constraints bei der Generierung nutzt. Allerdings baut auch diese Erweiterung auf den vorigen geplanten auf.

Eine weitere ausstehende Frage ist, wie die Bewertung generierter Level zur Laufzeit geschehen kann. Convolutional Neural Networks werden eingesetzt, um Bild- und Audiodaten zu verarbeiten. Möglicherweise liefern sie brauchbare Ergebnisse zur Bewertung der Levelqualität, wenn sie generierte Level als Input nutzen.

Im Laufe der Arbeit wurde immer wieder die Generizität von CSPs und WFC betont. Mit dieser Arbeit wurde jedoch nur das Thema der Levelgenerierung durch WFC abgedeckt. Die Erforschung weiterer auf WFC basierender Tools bietet sicherlich noch viele Möglichkeiten. *Gumin* fügt laufend neue, auf seinen Algorithmus basierende Anwendungen zu seinem Repository hinzu. Die dort gelisteten Projekte liefern eine Menge Inspiration für weitere Forschungsansätze.

7.3 Fazit

Module, Themes und die Konfiguration ihrer Regeln eignen sich hervorragend, um den zu generierenden Inhalt in eine bestimmte Richtung zu lenken und die eigenen Visionen umzusetzen. Gerade die Verwendung von Themes sorgt für visuelle Vielfalt bei der Generierung. Die Flexibilität des Algorithmus bietet viel Potenzial diesen zu erweitern und ihn sinnvoll an das eigene Spiel anzupassen. Wie viele andere Algorithmen auch, ist WFC nicht für den alleinigen Einsatz zur PCG in einem Spiel geeignet. Die Stärke des Algorithmus liegt in der Generierung der Levelgeometrie und dafür sollte er auch genutzt werden. Weiterhin hat die Thesis gezeigt, dass das *SimpleTiledModel* besser für den Kontext der Levelgenerierung geeignet ist, als das *OverlappingModel*. Letzteres ist schlichtweg zu schwierig an sinnvolle und nötige Erweiterungen wie die Navigierbarkeits-Heuristik anzupassen. *KHollapse* ist trotz seiner Fehler eine nützliche Software geworden, die hilft den WFC-Algorithmus zu verstehen und seine Komponenten zu visualisieren. Die Software eignet sich gut als Tool zur Levelgenerierung und zeigt viel Potenzial, vor allem mit den noch bevorstehenden Erweiterungen. Abschließend lässt sich sagen, dass die Synthese generierter und handgebauter Welten mittels *WaveFunctionCollapse* aufgrund der Ergebnisse dieser Thesis als erfolgreich betrachtet werden kann.

8 Glossar

PCG prozedurale Contentgenerierung

WFC WaveFunctionCollapse

PRNG Pseudorandom number generator

ZA zelluläre Automaten

EPC18 Everything Procedural Conference 2018

CSP Constraint Satisfaction Problem

FCSP finite Constraint Satisfaction Probleme

MRV minimum remaining values

LCV least constraining value

AC-3 Arc Consistency 3

9 Abbildungsverzeichnis

10 Tabellenverzeichnis

11 Code-Snippetverzeichnis

12 Literaturverzeichnis

Alexander, M. (2015). Procedural Music – A Viable Alternative? | Blog. https://www.yoyogames.com/blog/119/procedural-music-a-viable-alternative (Zugriff: 17.01.2019).

Barton, M. and Loguidice, B. (2009). The History of Rogue: Have @ You, You Deadly Zs. https://www.gamasutra.com/view/feature/4013/the_history_of_rogue_have__you_.php?print=1 (Zugriff: 17.01.2019).

Beca, S. (2017). Procedural generation: a primer for game devs. https://www.gamasutra.com/blogs/ScottBeca/20170223/292255/Procedural_generation_a_primer_for_game_devs.php (Zugriff: 21.01.2019).

CGW Magazine (2014). SpeedTree Brings Photoreal Vegetation to The Wolf of Wall Street. http://www.cgw.com/Press-Center/Web-Exclusives/2014/SpeedTree-Brings-Photoreal-Vegetation-to-The-Wol.aspx (Zugriff: 19.01.2019).

Crecente, B. (2008). Three Developers Explain LittleBigPlanet Level Design to a 7-Year-Old. https://kotaku.com/three-developers-explain-littlebigplanet-level-design-t-5057652 (Zugriff: 21.02.2019).

Doull, A. (2014). What PCG Is. http://pcg.wikidot.com/what-pcg-is (Zugriff: 18.01.2019).

Duden Online (2018a). Content, der. https://www.duden.de/rechtschreibung/Content (Zugriff: 23.01.2019).

Duden Online (2018b). prozedural. https://www.duden.de/rechtschreibung/prozedural (Zugriff: 23.01.2019).

Duden Online (2018c). Synthese, die. https://www.duden.de/rechtschreibung/Synthese (Zugriff: 23.01.2019).

Duvall, H. (2001). It's All in Your Mind: Visual Psychology and Perception in Game Design. https://www.gamasutra.com/view/feature/131506/its_all_in_your_mind_visual_.php (Zugriff: 12.02.2019).

EPYX Inc. (1985). Rogue Instruction Manual © 1985 EPYX, Inc. http://www.roguelikedevelopment.org/archive/files/misc/EpyxRogueDOSManual/manual.htm (Zugriff: 17.01.2019).

Ge, J. (2014). Procedural Map Generation - Cogmind / Grid Sage Games. https://www.gridsagegames.com/blog/2014/06/procedural-map-generation/ (Zugriff: 23.01.2019).

Gumin, M. (2016). mxgmn/WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse (Zugriff: 18.01.2019).

Gustavson, S. (2005). Simplex noise demystified. http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf (Zugriff: 29.01.2019).

Hill, J. (2008). Your Turn: Sticking to procedure. http://blogs.theage.com.au/screenplay/archives//009344.html (Zugriff: 09.02.2019).

Interactive Data Visualization (2014a). Speedtree. http://www.speedtree.com (Zugriff: 23.01.2019).

Interactive Data Visualization (2014b). Speedtree - mesh fonds. https://docs.speedtree.com/doku.php?id=mesh_fronds (Zugriff: 21.03.2019).

Introversion Software (2007). Procedural Content Generation. http://www.gamecareerguide.com/features/336/procedural_content_.php?page=1 (Zugriff: 09.02.2019).

Justynski, A. and Lasota, J. (2015). Call of Duty: Modern Warfare 2 Game Guide & Walkthrough. http://www.gameguides.cc/Article/gameguides/C/201512/8032.html (Zugriff: 31.01.2019).

Karth, I. and Smith, A. M. (2017). Wavefunctioncollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 68:1–68:10, New York, NY, USA. ACM.

Kazemi, D. (o.D.). Spelunky Generator Lessons. http://tinysubversions.com/spelunkyGen/ (Zugriff: 09.02.2019).

Kleineberg, M. (2019). Infinite procedurally generated city with the wave function collapse algorithm. https://marian42.de/article/wfc/ (Zugriff: 14.03.2019).

Lague, S. (2016). Procedural Landmass Generation (E01: Introduction). https://www.youtube.com/watch?v=wbpMiKiSKm8&index=1&list=PLFt_AvWsXl0eBW2EiBtl_sxmDtSgZBxB3 (Zugriff: 29.01.2019).

Lait, J. (2008). Berlin Interpretation. http://www.roguebasin.com/index.php?title=Berlin_Interpretation (Zugriff: 24.01.2019).

Lambe, I. (2012). Procedural Content Generation: Thinking With Modules. https://www.gamasutra.com/view/feature/174311/procedural_content_generation_.php (Zugriff: 29.01.2019).

Lee, J. (2014). How Procedural Generation Took Over The Gaming Industry. https://www.makeuseof.com/tag/procedural-generation-took-gaming-industry/ (Zugriff: 20.01.2019).

Miller, George Armitage (1956). *The Magical Number Seven, Plus Or Minus Two - Some Limits on Our Capacity for Processing Information*. Freuburg i.B.

Nyblom, P. (2012). Procedural Music Editor. https://pernyblom.github.io/abundant-music/index.html (Zugriff: 25.01.2019).

Perlin, K. (1999). Making noise. https://web.archive.org/web/20160308070426/http://noisemachine.com:80/talk1/ (Zugriff: 29.01.2019).

Perlin, K. (2002). Improving noise. *ACM Trans. Graph.*, 21(3):681–682.

Plausible Concept (2017). Home. https://www.badnorth.com/ (Zugriff: 14.02.2019).

Portnow, J. (2015). Procedural Generation - How Games Create Infinite Worlds - Extra Credits [YouTube]. https://www.youtube.com/watch?v=TgbuWfGeG2o (Zugriff: 29.01.2019).

Portnow, J. (2016). Backtracking and Level Design - Making a Way Out - Extra Credits [YouTube]. https://www.youtube.com/watch?v=-H97gCCJFXA (Zugriff: 17.02.2019).

Read, M. (2014). Random Walk Cave Generation. http://www.roguebasin.com/index.php?title=Random_Walk_Cave_Generation (Zugriff: 01.02.2019).

Remo, C. (2008). MIGS: Far Cry 2 's Guay On The Importance Of Procedural Content. http://www.gamasutra.com/php-bin/news_index.php?story=21165 (Zugriff: 22.01.2019).

Russell, S. and Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Always learning. Pearson.

Ryan, T. (1999a). Beginning Level Design, Part 1. https://www.gamasutra.com/view/feature/131736/beginning_level_design_part_1.php?page=1 (Zugriff: 11.02.2019).

Ryan, T. (1999b). Beginning Level Design Part 2: Rules to Design By and Parting Advice. http://www.gamasutra.com/view/feature/131739/beginning_level_design_part_2_.php (Zugriff: 11.02.2019).

Saltzman, M. (1999). Secrets of the Sages: Level Design. https://www.gamasutra.com/view/feature/131767/secrets_of_the_sages_level_design.php (Zugriff: 11.02.2019).

Seppala, T. J. (2016). Crafting the algorithmic soundtrack of "no man's sky". https://www.engadget.com/2016/08/11/no-mans-sky-soundtrack-65daysofstatic-interview/ (Zugriff: 27.01.2019).

Shaker, N., Togelius, J., and Nelson, M. J. (2016). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Shiffman, D. (2012). *The Nature of Code*. Shiffman, Daniel.

Short, T. (2014). Level Design in Procedural Generation. http://www.gamasutra.com/blogs/TanyaXShort/20140204/209176/Level_Design_in_Procedural_Generation.php (Zugriff: 12.02.2019).

Stålberg, O. (2018). EPC2018 - Oskar Stålberg - Wave Function Collapse in Bad North [YouTube]. https://www.youtube.com/watch?v=0bcZb-SsnrA&t=2033s (Zugriff: 18.02.2019).

StrategyWiki (2016). .kkrieger. https://strategywiki.org/wiki/.kkrieger (Zugriff: 01.02.2019).

Taylor, D. (2013). Ten Principles of Good Level Design (Part 1). http://www.gamasutra.com/blogs/DanTaylor/20130929/196791/Ten_Principles_of_Good_Level_Design_Part_1.php (Zugriff: 11.02.2019).

TheSheep (2016). Cellular Automata Method for Generating Random Cave-Like Levels. http://roguebasin.roguelikedevelopment.org/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels (Zugriff: 29.01.2019).

Victusmetuo (2012). Legendary Improvements Preview. https://www.diabloii.net/blog/comments/legendary-improvements-blog (Zugriff: 23.01.2019).

WarBlade (2009). Submachine Gun. https://borderlands.fandom.com/wiki/Submachine_Gun (Zugriff: 23.01.2019).

Wesołowski, J. (2009). Beyond pacing: Games aren't hollywood. https://www.gamasutra.com/view/feature/132423/beyond_pacing_games_arent_.php (Zugriff: 31.01.2019).

Wichmann, G. (1997). A Brief History of "Rogue". http://www.digital-eel.com/deep/A_Brief_History_of_Rogue.htm (Zugriff: 19.01.2019).

Wikipedia (2018). Random Walk — Wikipedia, Die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=Random_Walk&oldid=182847144 (Zugriff: 28.01.2019).

Williams, J. (2015). Procedural Generation As The Future. https://wccftech.com/article/procedural-generation-future/ (Zugriff: 01.02.2019).

Yu, D. (2016). *Spelunky*. Boss Fight Books. Boss Fight Books.

Zucker, M. (2001). The perlin noise math faq. https://mzucker.github.io/html/perlin-noise-math-faq.html (Zugriff: 29.01.2019).

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Hamburg, 02.04.2019 Kevin Hagen