

Hochschule für Technik und Wirtschaft Dresden

Fakultät Informatik/Mathematik

Bachelorarbeit

im Studiengang Medieninformatik

Titel: Semi-automatisierte Generierung von Dungeon-Strukturen in Videospielen

eingereicht von: Niclas Mart

eingereicht am: 13.09.2021

Betreuer: Prof. Dr. rer. nat. Marco Block-Berlitz,

Prof. Dr. rer. nat. habil. Marco Hamann

Inhaltsverzeichnis

1 Motivation und Einführung	1
2 Theorie und verwandte Arbeiten	4
2.1 Einführung in die Notation	5
2.2 Verwendete Algorithmen	6
2.3 Verwendete Taxonomie	7
2.3.1 Online versus offline Generation	8
2.3.2 Verpflichtender versus optionaler Inhalt	8
2.3.3 Zufälliger Seed versus Parameter	9
2.3.4 Generische versus adaptive Ansatz	9
2.3.5 Stochastisches versus deterministisches Ergebnis	9
2.3.6 Konstruktiver versus such-basierter Ansatz	9
2.3.7 Automatische versus semi-automatisierte Generation	10
2.4 Vorstellung prozeduraler Generationsmethoden	10
2.4.1 Genetischer Ansatz	10
2.4.2 Zellulärer Automat	12
3 Architektur des entwickelten Generationsverfahrens	14
3.1 Anforderungsanalyse	14
3.2 Vorstellung der verfügbaren Parameter	15
3.3 Grundlegende Arbeitsweise des Algorithmus	18
3.3.1 Generation der Dungeonstruktur	19
3.3.2 Setzen von Start- bzw. Endraum	23
3.3.3 Platzierung der Asset-Tiles	26
3.4 Problemfelder und deren Lösungsansätze	26
3.5 Mechanismen zur Konfiguration	29
3.5.1 Nutzung von Blueprints als Vorlage	29
3.5.2 Tile-Sets zur Konfiguration der verwendeten Assets	32
4 Experimente und Auswertung	34
4.1 Praktische Laufzeitanalyse	34
4.2 Levelanalyse	36
4.3 Einordnung in die Taxonomie	38
5 Zusammenfassung und Ausblick	41

1 Motivation und Einführung

Procedural-Content-Generation (PCG) ist ein Feld der Informatik, das sich mit der algorithmischen Erzeugung von digitalen Inhalten befasst. Bei diesen kann es sich z. B. um Texte bzw. Namen, 3D-Modelle oder sogar Musik handeln, welche automatisch durch einen Algorithmus erzeugt werden können [7, 11, 9]. Die Eigenschaften und somit das genaue Erscheinungsbild des erzeugten Inhalts werden durch den verwendeten Algorithmus und die übergebenen Parameter bestimmt. Dabei können Zufallsprozesse eine wichtige Rolle spielen oder aber auch adaptive Systeme, die durch User-Input das Ergebnis des Algorithmus beeinflussen können [12].

Besonders im Bereich der Videospiele besitzen Verfahren zur prozeduralen Generation von Inhalten einen hohen Stellenwert. So lassen sich in Spielen z. B. Texturen, Items, Quests, Dialoge, Charaktere, Landschaften oder komplette Level durch PCG erzeugen [10]. Ein großer Vorteil der algorithmischen Generation von Inhalten, im Kontrast zur manuellen Erstellung, ist die Vielfalt der erzeugbaren Inhalte und die damit einhergehende Einzigartigkeit dieser. Somit kann nicht nur der Wiederspielwert enorm erhöht werden, sondern auch die Spielerfahrung entfaltet sich für jeden Spieler anders. PCG ermöglicht es besonders kleineren Indie-Studios interessante und komplexe Spielwelten zu gestalten, ohne dabei hohe Produktionskosten für die manuelle Erstellung dieser aufwenden zu müssen. Aber auch große Studios nutzen in ihren Spielen verschiedene PCG-Verfahren. Returnal¹ (Sony Interactive Entertainment, 2021), Borderlands² (2K Games, 2009, siehe Abb. 1.1 rechts), Minecraft³ (Mojang Studios, 2009, siehe Abb. 1.1 links), Diablo 3⁴ (Blizzard Entertainment, 2012) und Rogue (Troy and Wichman, 1980) bilden nennenswerte Beispiele.

Zu beachten ist, dass in den Generationsprozess neben den gegebenen technischen Beschränkungen ebenfalls Game-Design Entscheidungen und Konzepte mit einfließen sollten. Dies ist von großer Bedeutung, da durch die automatische Generation von Inhalten der genaue Einfluss auf das Resultat verloren geht. Normalerweise besitzt jede Entscheidung bezüglich der Gestaltung einer Landschaft bzw. einer Umgebung in einem Videospiel ihr Fundament im Game Design. Jedes Element wird so platziert, dass es einen gewissen Zweck erfüllt und in das Gesamtbild der Spielwelt passt. Diese Kontrolle geht bei durch PCG erzeugten Inhalten verloren, weswegen die entsprechenden Algorithmen so konstruiert werden müssen, dass sie zuvor festgelegte Regeln einhalten und durch Game-Designer erstellte Konzepte und Zielstellungen bei der Generation verfolgen. Die Regeln des Game-Designs müssen sich also trotz automatischer Generation im Ergebnis widerspiegeln.

¹<https://www.playstation.com/de-de/games/returnal/>

²<https://borderlands.com/de-DE/>

³<https://www.minecraft.net/de-de>

⁴<https://eu.diablo3.com/de-de/>

1 Motivation und Einführung



Abbildung 1.1: Zwei sehr bekannte Beispiele für die Nutzung von PCG in Videospiele sind Minecraft (Bild links) und die Borderlands-Reihe (Bild rechts, Borderlands 3). In Minecraft wird die gesamte Spielwelt prozedural aus Blöcken generiert, sodass mit jedem neuen Spielstand eine andere Welt entsteht. Die Borderlands-Reihe ist für ihre extrem große Waffenvielfalt bekannt, die nur durch prozedurale Generierung gewährleistet werden kann.

Da im Vergleich zur manuellen Erstellung die Kontrolle über das Ergebnis trotz Steuerungsparametern verhältnismäßig gering ausfällt, werden PCG-Verfahren eher selten für die Generation ganzer Level verwendet. Diese fordern in ihrer Gesamtheit meist ein komplexes Zusammenspiel von verschiedenen Aspekten des Game-Designs. Eine Ausnahme bilden hier Dungeon-Level, die in ihrer Struktur einem klaren Schema folgen und dem Spieler in einem stark begrenztem Raum freie Möglichkeiten zur Erkundung bieten. Durch die regelbasierte Struktur, eignen sich Dungeonkomplexe sehr gut für die Generation mittels PCG [16].

Shaker definiert ein Dungeon als “(...) a labyrinthine environment where adventurers enter at one point, collect treasures, evade or slay monsters, rescue noble people, fall into traps and ultimately exit at another point.“ [10, S. 32]. Diese Definition lässt sich gut auf die wichtigsten Kriterien eines Dungeons herunterbrechen:

1. Es handelt sich um einen abgeschlossenen Bereich, der sich aus labyrinthisch angeordneten Gängen und Räumen zusammensetzt.
2. Es existiert mindestens ein Eingang und ein Ausgang, die über einen oder mehrere Wege miteinander verbunden sein müssen.
3. Innerhalb des Dungeons existieren verschiedene Aufgaben, Hindernisse und/oder Belohnungen.

Diese drei grundlegenden Kriterien sollen dabei keinesfalls eine vollständige Definition eines Dungeons darstellen. Sie haben eher das Ziel, einen Überblick über die geforderten Schwerpunkte zu geben und als Grundlage für die weiteren Betrachtungen dienen. Zudem kann somit eine Abgrenzung zu anderen Umgebungen und Strukturen geschaffen werden.

Im Vergleich zu natürlichen Landschaften bieten Dungeons den Vorteil, dass sie klaren Strukturen und Regeln folgen und trotz ihres scheinbar chaotischen Aufbaus ein klares Schema aufweisen. Ihre grundlegende Struktur lässt sich gut mithilfe von Graphen und Bäumen widerspiegeln. Dadurch können Algorithmen aus der Graphentheorie eingesetzt werden, um zum Beispiel einen Weg von einem Raum zu einem anderen zu finden oder ähnliche Probleme zu lösen. Diese Eigenschaften machen sie so interessant für die automatisierte Generation, sodass sich das eigene Forschungsfeld der Procedural-Dungeon-Generation (PDG) ergibt [17].

I Motivation und Einführung

Im Bereich der PDG existieren bereits viele verschiedene Verfahren, auf die im nächsten Abschnitt genauer Bezug genommen werden soll. Ziel der Arbeit ist es, sich diesen Verfahren zu nähern und einen eigenen Algorithmus zu entwerfen, der als Generator für eine Vielzahl von verschiedenen Dungeon-Typen geeignet sein soll. Dabei steht vor allem die Kontrolle über die Struktur und das Aussehen des Dungeons im Vordergrund. Das Verfahren hat den Anspruch dem Game-Designer möglichst präzisen Werkzeuge für die Konfiguration des Ergebnisses an die Hand zu geben, ohne dabei die Vorteile der PCG zu verlieren. Die Generation soll dabei auf Grundlage eines tile-basierten Systems erfolgen, das vorgefertigte Elemente, sogenannte Assets, für die Generation verwendet. Diese Assets müssen im Voraus vom Game-Designer erstellt werden und können anschließend an den Algorithmus übergeben werden. Die genauen Kriterien und Anforderungen werden noch einmal unter Punkt 3.1 genauer herausgestellt und diskutiert.

2 Theorie und verwandte Arbeiten

PDG weist eine große Vielfalt an verschiedenen Ansätzen und Verfahren auf, von denen im Folgenden einige vorgestellt werden sollen. Grob können diese in zwei unterschiedliche Kategorien eingeteilt werden:

Such-basierte Verfahren Im Bereich der PDG werden such-basierte Algorithmen dazu eingesetzt, um ein möglichst optimales Ergebnis zu erzielen. Meist basieren diese auf metaheuristischen Ansätzen und suchen innerhalb eines gegebenen Suchbereichs nach einer optimalen Lösung. Der Inhalt wird hier generiert, ausgewertet und die daraus gewonnenen Informationen für den nächsten Generationsprozess verwendet. Die Generation erfolgt also in mehreren Schritten, wobei sich das Ergebnis mit jedem Generationsschritt verbessert und einem Optimum innerhalb des Suchbereichs annähert. Ein sehr häufig genutzter Ansatz stellen dabei genetische Algorithmen dar, welche nach dem grundlegenden Prinzip der Evolutionstheorie arbeiten. Die Auswertung des generierten Inhalts erfolgt anhand einer Fitnessfunktion. Diese wird zuvor festgelegt und definiert die Eigenschaften, die an den Inhalt gestellt werden. Durch die Generation in mehreren Stufen wird versucht, die vorgegebenen Anforderungen bestmöglich zu erreichen. Je nach Komplexität und Größe des zu generierenden Inhalts kann dies jedoch etliche von Iterationen in Anspruch nehmen [17].

Konstruktive Verfahren Zu diesem Gebiet zählen Algorithmen, die regel-basiert arbeiten und den Inhalt, meist auf zuvor festgelegten Parametern erzeugen. Oft werden Zufallsprozesse ausgenutzt, um interessante und einzigartige Strukturen zu generieren. Die Generation erfolgt dabei im Gegensatz zu such-basierten Verfahren in einem Durchlauf. Aus diesem Grund muss bereits im ersten Generationsschritt ein möglichst gutes Ergebnis erzielt werden. Die zum Einsatz kommenden Algorithmen unterscheiden sich meist in ihrer Funktionsweise sehr stark voneinander. Beispiele für genutzte Algorithmen sind Zelluläre Automaten und Generative Grammatiken [17].

Während in der Forschung der Fokus zum Großteil auf such-basierten Verfahren liegt, werden in der Industrie meist konstruktive Ansätze verwendet. Dies begründet sich aus der Laufzeit, die bei such-basierten Verfahren durch die Generation in mehreren Iterationen deutlich höher ausfällt [5]. Mithilfe der mehrmaligen Generation können zwar meist sehr gute Ergebnisse innerhalb der gegebenen Grenzen gefunden werden, jedoch eignen sie sich meist nicht für die Generation von Strukturen zur Laufzeit des Spiels.

Wichtig dabei ist zu verstehen, dass es sehr unterschiedliche Anforderungen an die Struktur und den Aufbau eines Dungeon geben kann, weshalb es nahezu unmöglich ist, mit einem Algorithmus alle Anforderungen abzudecken. Somit existiert nicht das eine optimale Verfahren, da jedes gewisse Vor- und Nachteile birgt und den Schwerpunkt auf einen anderen Aspekt der Generation

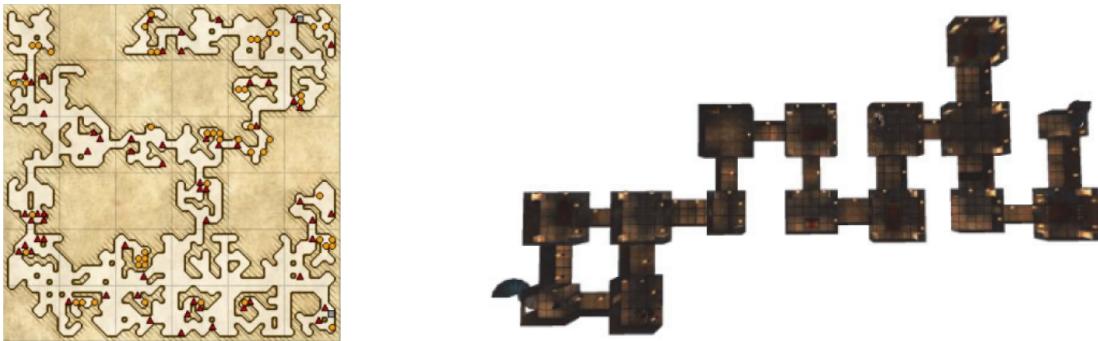


Abbildung 2.1: Links: Das Bild zeigt die schematische Übersicht eines höhlenartige Dungeons mit ungleichmäßiger Struktur, welches durch A. Liapis generiert wurde [8]; Rechts: Im Gegensatz dazu weist das auf diesem Bild gezeigte Dungeon eine völlig andere Struktur auf. Diese ist eher kerkerartig, bestehend aus rechteckigen Räumen mit einer gleichmäßigen, geometrischen Struktur. Generiert wurde das Dungeon durch van der Linden, Lopes und Bidarra [15]

legt. Beispiele für sehr unterschiedliche Dungeons sind höhlenartige Strukturen wie in Abbildung 2.1 links und kerkerartig strukturierte Dungeons wie in Abbildung 2.1 rechts. Die starken strukturellen Unterschiede führen zu sehr verschiedenen Anforderungen an das Generationsverfahren, weswegen meist unterschiedliche Algorithmen eingesetzt werden müssen.

2.1 Einführung in die Notation

In diesem Abschnitt sollen die in der Arbeit verwendeten Fachbegriffe und spezielle Notationen eingeführt und genauer definiert werden.

Grid bzw. Raster Beschreibt eine zweidimensionale Fläche, die sowohl horizontal als auch vertikal in quadratische, gleichgroße Teilflächen untergliedert ist (siehe Abb. 2.2 links). Die Größe dieser Fläche wird dabei durch die Anzahl der Teilflächen in x- bzw. y-Richtung definiert.

Asset Begriff für digitale Inhalte, die zur Erstellung eines Videospiels verwendet werden. Dabei kann es sich zum Beispiel um Texturen, Musik, 3D-Modelle, Animationen oder auch Kombinationen aus Inhalten handeln¹. In dieser Arbeit bezieht sich der Begriff hauptsächlich auf zuvor angelegte 3D-Modelle von Böden und Wänden, die bereits in texturierter Form vorliegen und zum Aufbau des Dungeons genutzt werden.

Tile Bezeichnet ein Asset, welches genau die Größe einer Teilfläche eines Grids aufweist. Diese können somit präzise auf dem Grid platziert werden, um nahtlose Strukturen zu erzeugen (siehe Abb. 2.2 rechts).

Map Bezeichnet im Zusammenhang mit Game-Design einen einzelnen Level-Komplex, der eine eigene in sich geschlossene Struktur aufweist. Im folgenden wird der Begriff auch synonym für Dungeons verwendet, da ein Dungeon-Komplex ein in sich geschlossenes Level und somit eine Map darstellt.

¹<https://unity.com/how-to/beginner/game-development-terms>

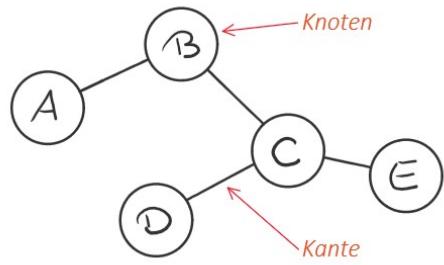
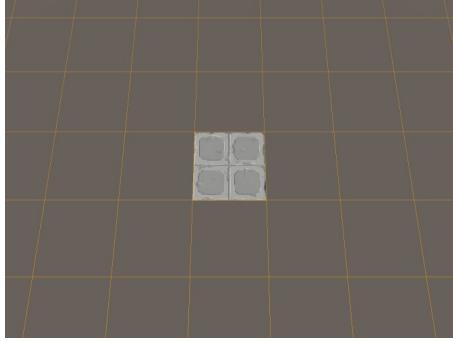


Abbildung 2.2: Links: Die Teilflächen des dargestellten Grids (orangene Linien) besitzen eine Größe von 5 Unity-Metern. Im Grid befindet sich ein Boden-Tile, das genau eine Teilfläche vollständig ausfüllt. Rechts: Dargestellt wird ein ungerichteter, kreisfreier Graph G mit $V = 5$ und $E = 4$.

Grundlagen der Graphentheorie Ein Graph G ist ein abstraktes Modell in der Mathematik, um eine Menge von Objekten und deren Beziehungen untereinander darzustellen. Die Objekte werden dabei als Knoten (Ecken, Vertices) und die Verbindungen zwischen diesen als Kanten (Edges) bezeichnet. G besteht somit aus einer Knotenmenge $V(G)$ und einer Kantenmenge $E(G)$ und kann damit als $G = (V, E)$ definiert werden (siehe Abb. 2.2 rechts). Die Mächtigkeit der Knotenmenge $V(G)$ wird auch als die Ordnung des Graphen bezeichnet und kann als $|G|$ geschrieben werden. Zwei Knoten x, y innerhalb der Menge V können über eine Kante e miteinander verbunden sein. Diese zwei Knoten bilden die Endpunkte der Kante e und werden im folgenden als benachbart bezeichnet. Die beschriebene Kante e ist dabei definiert durch $\{x, y\}$ und kann als xy oder yx bezeichnet werden. Die Anzahl der Nachbarn eines Knotens wird auch als dessen Grad $d(v)$ bezeichnet. Der durchschnittliche Grad des gesamten Graphen G kann dabei durch die Formel

$$d(G) := \frac{1}{|V|} \sum_{v \in V} d(v) \quad (2.1)$$

berechnet werden. Ein Weg innerhalb eines Graphen stellt dabei einen Teilgraphen P von G dar, wobei $P \subseteq G$ gilt. Damit der Teilgraph als Weg bezeichnet werden kann muss zusätzlich gelten:

$$V_P = \{x_0, x_1, \dots, x_k\} \quad E_P = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k\}$$

Alle Knoten x_i von P müssen voneinander verschieden sein. Die Anzahl der Kanten von P wird auch als die *Länge* des Weges bezeichnet. Im weiteren Verlauf werden vor allem ungerichtete Graphen betrachtet, bei denen eine Kante zwischen zwei Knoten in beide Richtungen verwendet werden kann. [3, S. 2-8]

2.2 Verwendete Algorithmen

In diesem Abschnitt sollen die zum Einsatz kommenden Algorithmen kurz vorgestellt werden. Bei diesen handelt es sich meist um in der Informatik bereits gut etablierte Verfahren. Aus die-

sem Grund soll nur die grobe Funktionsweise kurz vorgestellt werden, ohne jedoch genauer auf Details der Implementierung einzugehen.

Dijkstra-Algorithmus Ist ein Algorithmus, der vor allem in der Graphentheorie zum Einsatz kommt. Das Ziel besteht darin, den kürzesten Weg zwischen zwei gegebenen Knoten, innerhalb eines gerichteten Graphen mit nicht negativen Kantengewichten, zu finden (siehe Abb. 2.3 links). Jeder Knoten im Graph erhält dazu die Eigenschaften *Distanz* und *Vorgänger*. Dabei wird die Distanz im Startknoten initial mit 0 und in allen anderen Knoten mit ∞ angegeben. Zudem wird eine Datenstruktur Q benötigt, in welcher die Knoten, zu denen noch kein kürzester Weg gefunden wurde, gespeichert werden. Diese wird zu Beginn nur mit allen Knoten des Graphen gefüllt. Der eigentliche Algorithmus iteriert nun über Q und findet den Knoten mit der geringsten *Distanz*. In der ersten Iteration handelt es sich bei diesem immer um den Startknoten. Anschließend wird dieser aus Q entfernt und durch aufsummieren die *Distanzen* aller Nachbarknoten berechnet. Nun kann erneut über Q iteriert werden, um den Knoten mit der geringsten *Distanz* zu finden. Dieser Prozess wird so lange wiederholt, bis alle Knoten im Graphen ausgewertet wurden, Q also keine weiteren Knoten enthält [2, S. 658].

Gift-Wrapping-Algorithmus Dieser Algorithmus ermöglicht es die konvexe Hülle aus einer gegebenen Punktmenge S zu berechnen. Die konvexe Hülle ist durch das kleinste konvexe Polygon gegeben, welches durch das Verbinden aller Punkte der Menge S entsteht und diese komplett enthält. Vereinfacht kann die konvexe Hülle als ein Gummiband verstanden werden, welches sich um die äußersten Punkte der Menge spannt (siehe Abb. 2.3 rechts). Der Algorithmus findet alle Punkte aus S , die zur konvexen Hülle gehören. Dazu wird ein Startpunkt P_0 aus S gewählt, welcher Teil der konvexen Hülle sein muss. Von diesem wird eine Verbindung zu einem beliebigen weiteren Punkt P_1 aus S gezogen. Anschließend wird für jeden Punkt aus S überprüft, ob dieser auf der linken Seite der gezogenen Verbindung liegt. Ist dies der Fall, wird der entsprechende Punkt anstelle von P_1 mit P_0 verbunden. Dieser Prozess wird so lange wiederholt, bis alle Punkte aus S auf der rechten Seite der Verbindung liegen. Die zwei verbundenen Punkte gehören somit zur konvexen Hülle und bilden die erste Seite des Polygons. Anschließend wird der ganze Prozess mit dem neu gefundenen Punkt ausgeführt. Am Ende bilden alle verbundenen Punkte die konvexe Hülle der Punktmenge S [2, S. 1037].

2.3 Verwendete Taxonomie

Die durch Togelius vorgestellte Taxonomie schafft eine einheitliche Vergleichsbasis für verschiedene PCG-Verfahren und ermöglicht die strukturelle Gliederung dieser [13]. Die hier verwendete Taxonomie basiert auf Togelius, wurde jedoch durch Shaker um weitere Aspekte erweitert [10]. Der Vergleich erfolgt nach sieben Gesichtspunkten, wobei diese nicht als binär angesehen werden können. Vielmehr handelt es sich hier um Spektren, auch wenn die verglichenen Verfahren tendenziell einem Extrem stärker zugeordnet werden könne.

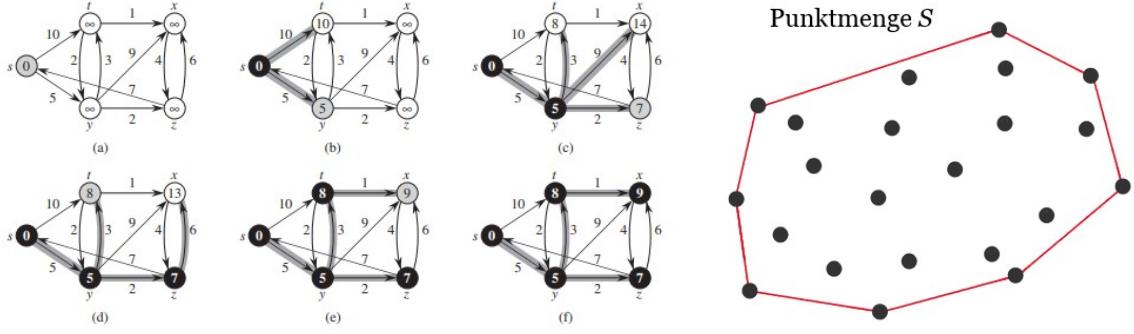


Abbildung 2.3: Links: Die Abbildung zeigt das Vorgehen des Dijkstra-Algorithmus. Im Initialzustand ist der Startknoten s mit einer Distanz von 0 initialisiert, alle anderen Knoten besitzen eine Distanz von ∞ . Danach wird der Graph iterativ durchlaufen und immer der Knoten mit der kürzesten Distanz zum Startknoten ausgewertet. Somit kann sichergestellt werden, dass am Ende für jeden Knoten im Graphen die kürzeste Distanz zum Startknoten bekannt ist [2, S. 659]. Rechts: Zu sehen ist eine Punktmenge S und deren konvexe Hülle, welche durch die Punkte gegeben ist, die mit einer roten Linie verbunden sind.

2.3.1 Online versus offline Generation

Eines der maßgebendsten Kriterien für die Differenzierung ist, inwiefern ein Verfahren die Generation während der Laufzeit des Spiels zulässt oder nicht. Ein Verfahren das online, also zur Laufzeit arbeitet, sollte dabei sehr geringe Berechnungszeiten aufweisen, sodass Inhalte entweder parallel zum Spiel oder mit kurzen Ladezeiten erzeugt werden können. Zudem können prozedurale Verfahren auch offline, zum Beispiel zur Generation von Inhalten während der Entwicklung des Spiels genutzt werden, wobei der erzeugte Inhalt anschließend als fester Bestandteil in das Spiels übernommen wird. Unter diesen Bedingungen kann die Laufzeit des Algorithmus deutlich höher sein, da das Ergebnis nicht innerhalb eines kritischen Zeitfensters generiert werden muss [10, 13].

2.3.2 Verpflichtender versus optionaler Inhalt

Dieser Punkt bewertet den Zweck des Inhalts, der durch das betrachtete Verfahren generiert wird. Dabei wird unterschieden, ob die zu generierenden Inhalte essenziell für das Fortschreiten im Spiel notwendig sind oder nur optional zur Verfügung stehen und somit auch übersprungen werden können. Diese Unterscheidung ermöglicht es, das Verfahren hinsichtlich seiner Zuverlässigkeit zu beurteilen. So sollten verpflichtende Inhalte stets richtig generiert werden und keine Fehler aufweisen. Im Gegensatz dazu ist die Toleranz gegenüber Fehlern bei optionalen Inhalten deutlich höher. Dabei bezieht sich dieser Punkt jedoch nicht auf Fehler wie Bugs, sondern vielmehr auf problematische Kombinationen von Eigenschaften generierter Inhalte, wodurch diese nicht in angedachter Weise genutzt werden können. Ein Beispiel wäre hier ein Waffengenerator, der unterschiedliche Gegenstände mit zufälligen Kombinationen an Eigenschaften generiert. Dieser erzeugt nun eine Waffe, deren Eigenschaften sich gegenseitig behindern, wodurch sie zum Beispiel keinen Schaden verursacht. Sollte es sich dabei um die einzige Waffe handeln, die dem Spieler zu diesem Zeitpunkt zur Verfügung steht und diese zwingend für den Fortschritt benötigt werden, so wäre dies gravierend für den Spielfluss und das Spiel könnte möglicherweise

nicht fortgesetzt werden. Sollten dem Spieler jedoch verschiedene Waffen zur Verfügung stehen, sodass dieser sich entscheiden kann eine andere Waffe zu benutzen, führt der Fehler zu keiner gravierenden Einschränkung [10, 13].

2.3.3 Zufälliger Seed versus Parameter

Dieses Kriterium ermöglicht eine Unterscheidung der Verfahren hinsichtlich ihrer Parametrisierung und der Kontrolle über das Verfahren. Eine Möglichkeit die Generation zu kontrollieren, besteht in der Nutzung eines sogenannten Seeds. Dabei handelt es sich um eine mehrstellige Kombination aus Zahlen, die als Initialwert für die zufallsgesteuerten Prozesse innerhalb des Algorithmus genutzt wird. Zwar lassen sich durch einen Seed nicht die Eigenschaften des generierten Inhalts gezielt bestimmen, jedoch entsteht bei der Nutzung des gleichen Seeds immer genau dasselbe Ergebnis bei der Generation. Im Kontrast dazu kann eine Reihe von Parametern genutzt werden, um das Resultat zu kontrollieren. Diese ermöglichen ein gezieltes Eingreifen in den Generationsprozess und damit auf das Ergebnis [10, 13].

2.3.4 Generische versus adaptive Ansatz

Dieses Kriterium unterscheidet Verfahren in Bezug auf die Nutzung von Laufzeitinformationen zur Generation. Generische Verfahren sind dabei statisch und verwenden nur Parameter bzw. Informationen, die zufällig gewählt oder im Entwicklungsprozess des Spiels festgelegt werden. Die meisten PCG-Verfahren arbeiten nach diesem Prinzip, jedoch existieren auch einige Verfahren, die den adaptiven Ansatz verfolgen. Dabei werden Informationen über das Verhalten des Spielers als zusätzlicher Einfluss für die Generation genutzt, um das Ergebnis besser auf den jeweiligen Spieler und dessen Spielweise anzupassen [10].

2.3.5 Stochastisches versus deterministisches Ergebnis

Die Unterscheidung in stochastische und deterministische Verfahren wird auf Grundlage der Reproduzierbarkeit des Ergebnisses getroffen. Auch wenn dieses Kriterium augenscheinlich eine Verbindung zu Kriterium 3 - „Zufälliger Seed versus Parameter“ aufweist, betrachtet dieser Punkt nur das Ergebnis, wohingegen Punkt 3 die initialen Parameter des Verfahrens bewertet. Deterministische Verfahren ermöglichen es, durch Nutzung der gleichen Eingabewerte, immer dasselbe Ergebnis zu erzielen. Stochastische Verfahren ermöglichen dies nicht und jeder Generationsprozess führt mit sehr hoher Wahrscheinlichkeit zu einem anderen Ergebnis [10, 13].

2.3.6 Konstruktiver versus such-basierter Ansatz

Wie bereits herausgestellt bezieht sich die Unterscheidung in konstruktive und such-basierte Verfahren auf das zur Generation genutzte Verfahren. Während konstruktive Verfahren den Inhalt einmal generieren, wird bei such-basierten Verfahren der Prozess aus Generation und Auswertung etliche Male durchlaufen, um ein bestmögliches Ergebnis zu erzielen [10, 13].

2.3.7 Automatische versus semi-automatisierte Generation

Dieses Kriterium bewertet, wie autonom das Verfahren arbeitet und somit wie viel Einfluss auf das Resultat von außen genommen werden kann. Der Einfluss auf semi-automatisierte Verfahren kann dabei durch den Game-Designer, bei adaptiven Verfahren aber auch durch den Spieler gegeben sein. Während bei autonomen Verfahren, bis auf einige Parameter, kaum Möglichkeiten zur Kontrolle gegeben sind, bieten semi-automatisierte Verfahren neben einer Vielzahl von verschiedenen Parametern oft weitere Möglichkeiten, Einfluss auf das Ergebnis zu nehmen. Dadurch kann das Game-Design stärker in den Generationsprozess eingebunden werden [10].

2.4 Vorstellung prozeduraler Generationsmethoden

Wie bereits angemerkt, existiert eine große Anzahl verschiedener Algorithmen zur Generation von Dungeons. Die beiden in diesem Abschnitt vorgestellten Verfahren sollen stellvertretend einen Überblick über die Unterschiede zwischen dem such-basierten und dem konstruktiven Ansatz geben. Während das erste Verfahren gute Möglichkeiten zur Konfiguration bereitstellt und dem Game-Designer somit viel Einflussnahme auf das Ergebnis ermöglicht, verfolgt das zweite Verfahren einen schnellen und effizienten Ansatz mit sehr geringer Laufzeit. Beide Aspekte stellen dabei wichtige Anforderungen an das in dieser Arbeit vorgestellte Verfahren dar, weswegen sie gut zum Vergleich herangezogen werden können.

2.4.1 Genetischer Ansatz

Das durch Valtchanov und Brown vorgestellte Verfahren verfolgt den Ansatz der genetischen Algorithmen und gehört somit zu den such-basierten Verfahren. Genetische Algorithmen nutzen verschiedene Konzepte der darwinschen Evolutionstheorie wie das Überleben des Stärkeren (*Survival of the fittest*) und Genetik, um ein gegebenes Optimierungsproblem zu lösen. Die Generation erfolgt dabei in mehreren Schritten, wobei in jedem eine Menge an Inhalten, auch Population genannt, generiert wird. Jedes Individuum dieser Population besitzt bestimmte Eigenschaften, für deren Darstellung und Speicherung eine geeignete Repräsentation gefunden werden muss. Diese wird auch als Chromosom bezeichnet. Somit besitzt jedes Individuum ein eigenes Chromosom, in dem seine Eigenschaften gespeichert vorliegen. Nach jedem Generationsschritt werden die unterschiedlichen Chromosomen mithilfe einer Fitnessfunktion bewertet. Diese berechnet anhand der im Chromosom codierten Eigenschaften für jedes Individuum einen Fitnessscore. Umso höher dieser ist, desto besser löst das Individuum das gestellte Problem. Die Chromosomen der besten Individuen werden im nächsten Iterationsschritt genutzt, um eine neue Population zu generieren. Dazu werden meist mehrere Chromosomen durch Crossover miteinander kombiniert und zufällige Mutationen angewendet, um die Vielfalt der neuen Generation zu erhöhen. Mit jeder Generationsstufe werden somit nur die besten Eigenschaften an die nächste Generation weiter vererbt. Der Vorgang wird meist durch einen zeitlichen Rahmen begrenzt oder das Erreichen eines bestimmten Fitnessscore führt zur Terminierung des Prozesses [1, 14].

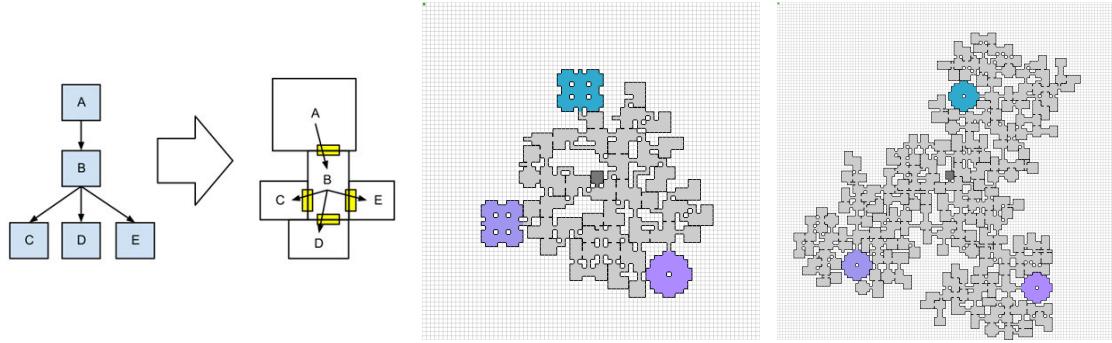


Abbildung 2.4: Links: Das Bild zeigt die theoretische Überführung eines Chromosoms in eine Dungeonstruktur. Jeder Knoten in der Baumstruktur des Chromosoms repräsentiert dabei einen Raum und jede Kante einen Verbindungsweg zwischen den Räumen [14]. Mitte: Hier ist die Struktur eines durch den Algorithmus erzeugten Dungeons nach 500 Generationen zu sehen. Die zur Generation benötigte Zeit beträgt dabei in etwa 30 Sekunden [14]. Rechts: Dieses Dungeon wurde durch 2000 Generationszyklen erzeugt [14].

Im Falle von Valtchanov und Braun wird als Chromosom eine Baumstruktur verwendet, deren einzelne Knoten die Räume des Dungeons repräsentieren (siehe Abb. 2.4 links). Der Prozess arbeitet dabei mit vordefinierten Raumstrukturen, die auf einem Raster aus Tiles, auch Map genannt, platziert werden. Innerhalb einer Population werden verschiedene solcher Baumstrukturen erzeugt, die mithilfe der zuvor definierten Fitnessfunktion auf ihre Eigenschaften überprüft werden. Dazu muss jedes Chromosom in sein äquivalentes Dungeon überführt werden. Nach dem Prinzip der Breitensuche wird im Baum über jeden Knoten iteriert, die alle nötigen Informationen über den zu platzierenden Raum enthalten. Das Chromosom wird somit stückweise in eine Dungeonstruktur überführt. Sollte es nicht möglich sein einen Raum mit den gegebenen Eigenschaften zu platzieren, da zum Beispiel die Position auf der Map bereits belegt ist, wird der Knoten und all seine Kinder aus dem Baum entfernt. Das entstandene Dungeon kann dann hinsichtlich seines Aufbaus ausgewertet und ein Fitnessscore berechnet werden. Im nächsten Generationszyklus werden die besten Chromosomen als genetisches Material für die neue Population genutzt. Aus den alten Chromosomen werden also durch verschiedene Operationen neue Chromosomen gebildet, wobei diese die besten Eigenschaften der alten Generation erben. Somit verbessert sich die Struktur mit jedem Zyklus, sodass am Ende ein Dungeon vorliegt, das den in der Fitnessfunktion geforderte Kriterien bestmöglich gerecht wird [14].

Durch das Verfahren lässt sich eine Vielzahl komplexer Strukturen erzeugen. Die Fitnessfunktion ermöglicht es dabei, leicht auf die Arbeitsweise des Verfahrens Einfluss zu nehmen. Durch einfache Änderungen an dieser kann das Aussehen des erzeugten Dungeons grundlegend verändert werden. Somit lassen sich mit ein und demselben Algorithmus sehr unterschiedliche Arten von Dungeons erzeugen, ohne dass das Verfahren an sich angepasst werden muss. Durch die such-basierte Arbeitsweise besitzt der Algorithmus jedoch eine relative hohe Laufzeit, weswegen er nicht gut für den online Einsatz geeignet ist. Zwar lassen sich wie in Abbildung 2.4 mittig auch kleinere Dungeons erzeugen, die durch ihre geringe Anzahl an Generationsschritten eine verhältnismäßig geringe Generationszeit von ca. 30 Sekunden aufweisen. Größere Dungeons mit einer komplexeren Struktur wie in Abbildung 2.4 rechts benötigen jedoch deutlich mehr Generationszyklen, um ein zufriedenstellendes Ergebnis zu erreichen, was die Laufzeit enorm erhöht. Somit ist das Verfahren eher für die offline Generation geeignet.

Auch wenn das Verfahren an sich nicht mittels eines Seeds gesteuert wird, kann das Chromosomen des finalen Dungeons als ein solcher genutzt werden. Da das Chromosom die Eigenschaften und Struktur des Dungeons kapselt, lässt sich mit diesem ein und dasselbe Dungeon erneut generieren.

2.4.2 Zellulärer Automat

Ein zellulärer Automat besteht grundlegend aus einem n-dimensionalen Grid, auch *Zellularraum* genannt, einer Menge an Zuständen Q und einer Menge an Übergangsfunktionen. Jede Zelle innerhalb des Zellularraums kann zu einem Zeitpunkt t nur genau einem Zustand aus Q entsprechen. Im einfachsten Fall gibt es nur zwei Zustände, die eine Zelle annehmen kann: *an* oder *aus*. Die Überführungsfunktionen legen dabei fest, welchen Zustand eine Zelle zum Zeitpunkt $t + 1$ annimmt. Dazu wird der eigene Zustand der Zelle und die Zustände der Zellen in der Nachbarschaft zum Zeitpunkt t betrachtet. Zum Zeitpunkt $t = 0$, liegt der gesamte Zellularraum in einem initialen Zustand vor. Anschließend wird über jede Zelle iteriert und ein neuer Zustand für den Zeitpunkt $t + 1$ anhand der Überführungsfunktionen berechnet. Wurde der gesamte Zellularraum ausgewertet, wird dieser als neuer Initialzustand verwendet und erneut über diesen iteriert [10].

In dem durch Johnson vorgestellten Verfahren, wird ein zellulärer Automat zur Generation von höhlenartigen Dungeons verwendet. Die Funktionsweise unterscheidet sich dabei nicht sonderlich von der grundlegenden Arbeitsweise eines zellulären Automaten. Das System arbeitet mit zwei Zuständen (*Stein* und *Boden*) auf einem 2-Dimensionalem Grid. Der initiale Zustand wird erzeugt, indem allen Zellen der Zustand Boden zugeordnet wird und anschließend $r\%$ zufällig in den Zustand Stein überführt werden (siehe Abb. 2.5 links). Anschließend beginnt der Iterationsprozess, wobei auf jede Zelle die Übergangsfunktion angewendet wird. Diese besagt, dass jede Zelle, die T oder mehr benachbarte Stein-Zellen besitzt, ebenfalls in den Zustand Stein überführt wird. Andernfalls bekommt sie den Zustand Boden zugeordnet. r und T sind dabei frei wählbare Parameter. Der Algorithmus läuft für mehrere Zyklen bis er nach einer definierten Anzahl terminiert oder keine Veränderungen von Zellen mehr stattfinden. Anschließend wird noch ein weiteres Mal über den Zellularraum iteriert und alle Stein-Zellen, die über mindestens eine angrenzende Boden-Zelle verfügen, in Wand-Zellen umgewandelt. Dieser dritte Zustand wird also nicht für den eigentlichen Generationsprozess verwendet, sondern nur für den strukturellen Aufbau des Dungeons. Beispiele für die dadurch entstehenden Strukturen sind in Abbildung 2.5 mittig und rechts aufgeführt [6].

Der Vorteil von zellulären Automaten ist ihr einfacher Aufbau. Wie im beschriebenen Beispiel reichen wenige, sehr einfache Regeln aus, um eine komplexe Struktur zu erzeugen. Durch das Auswechseln der Übergangsfunktion oder der Nachbarschaftsgröße lassen sich leicht völlig andere Strukturen erzeugen. Auch arbeiten zelluläre Automaten sehr Effizienz und können somit gut für echtzeit-basierte Generation genutzt werden. Jedoch kann der Game-Designer nur sehr wenig Einfluss auf das Resultat nehmen. Zwar können Parameter genutzt werden, um das Ergebnis zu beeinflussen, jedoch sind diese sehr abstrakt und geben keinen Aufschluss darüber, wie genau sie das Aussehen der Struktur verändern. Ein weiter Nachteil besteht in der Zusicherung von Anforderungen an einen Dungeon. So kann durch die reine Generation mittels zellulären

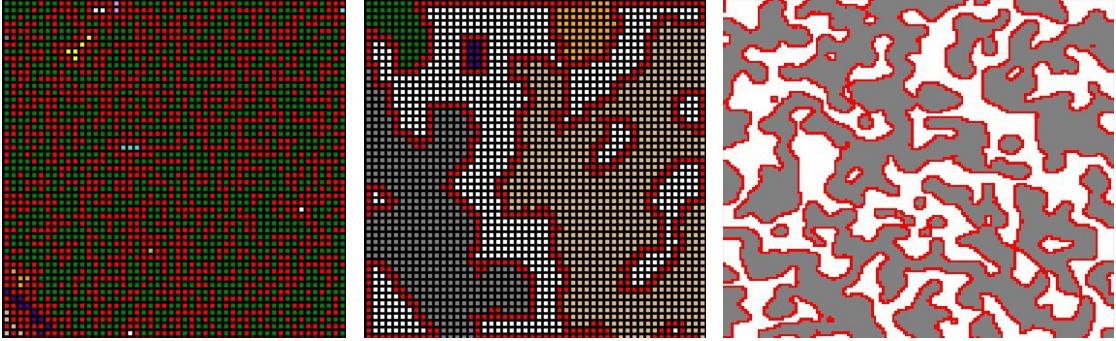


Abbildung 2.5: Links: Zu sehen ist der Zellulärraum in seinem initialen Zustand mit $r = 50$. Die rote Farbe markiert Stein-Zellen und alle bei allen anderen Feldern handelt es sich um Boden-Zellen [6]. Mitte und Rechts: Die durch den Algorithmus entstehenden Strukturen weisen eine höhlenartige und organische Struktur auf. Durch die Wahl von unterschiedlichen Parametern können dabei sehr unterschiedliche Strukturen erzeugt werden. Die rote Farbe steht für Wand-Zellen und die weiße für Stein-Zellen. Bei den restlichen Zellen handelt es sich um begehbarer Boden-Zellen. Die unterschiedliche Farbgebung für Boden-Zellen im mittleren Bild markiert zusammenhängende Bereiche [6].

Automaten nicht sichergestellt werden, dass ein Weg zwischen allen Höhlen-Komplexen existiert (siehe Abb. 2.5 mittig). Es muss also entweder eine erweiterte Funktionalität implementiert werden, die zusätzliche Gänge einfügt oder Stein- und Wand-Tiles müssen sich im Spiel wie in Minecraft abtragen lassen, sodass sich der Spieler selbst seinen Weg suchen kann [6, 10].

3 Architektur des entwickelten Generationsverfahrens

Im folgenden Abschnitt soll das entwickelte Verfahren in Detail vorgestellt und analysiert werden. Dabei soll auf dessen Arbeitsweise und ausgewählte Probleme und deren Lösungen eingegangen werden. Das Projekt und der gesamte Quellcode sind auf GitHub unter folgendem Link einzusehen https://github.com/NiclasMart/BA_Project_PDG.

3.1 Anforderungsanalyse

Um das Verfahren im Anschluss besser bewerten zu können, sollen zuerst die an das Verfahren gestellten Ansprüche und Zielstellungen genauer charakterisiert werden. Dabei sollen neben den geforderten Funktionskriterien ebenso Abgrenzungskriterien aufgezeigt werden, die definieren, was das Verfahren nicht leisten soll. Eines der wichtigsten Ziele besteht darin, dem Designer den größtmöglichen Einfluss auf das Resultat zu geben und eine Vielfalt an verschiedenen Parametern und Möglichkeiten an die Hand zu geben, um das Ergebnis zu beeinflussen. Es muss also ein Mittelweg zwischen automatisierter Generation und manuellen Entscheidungen gefunden werden.

Das Verfahren soll

- ein kerkerartiges Dungeon, bestehend aus Räumen und Gängen erzeugen, das den ersten beiden in der Einleitung aufgeführten Definitionskriterien eines Dungeons gerecht wird.
- raster-basiert auf einem Grid arbeiten.
- vorgefertigte Assets als Tiles für den Aufbau verwenden, welche durch den Designer bereitgestellt werden.
- eine geringe Laufzeit aufweisen, um ohne große Ladezeiten im Spiel neue Strukturen erzeugen zu können (maximal 5 s).
- sich einfach über Parameter genau konfigurieren lassen und dem Game-Designer größtmögliche Kontrolle geben.

Des Weiteren gilt, dass sich die erzeugte Dungeonstruktur im zweidimensionalen Raum erstrecken und als eine Art Bauplan für die übergebenen 3D-Asset dienen soll. Somit sollen vor allem Dungeons für top-down-basierte Spiele wie Diablo¹ generiert werden können. Der Schwerpunkt

¹<https://eu.diablo3.com/de-de/>

des Algorithmus soll dabei auf der Struktur des Dungeonkomplexes liegen und nicht auf dem Aussehen der einzelnen Räume, weswegen nur einfache, rechteckige Räume zur Generation verwendet werden. Jedoch sollen durch den Designer Blueprints (siehe Unterabschnitt 3.5.1) übergeben werden können, welche eine Raumstruktur definieren. Diese sollen mit zur Generation der Räume verwendet werden und dem Designer somit eine gute Kontrollmöglichkeit geben, welche sonst über einfache Parameter nur schwer zu erreichen wäre.

Das Verfahren soll nicht

- für das Befüllen des Dungeons mit Inhalten wie Gegnern oder Belohnungen verantwortlich sein, sondern sich nur um dessen Erstellung kümmern.
- mit komplett vorgefertigten Raum-Assets arbeiten, sondern diese selbst zur Laufzeit generieren.
- dafür verantwortlich sein, den generierten Inhalt auf Sinnhaftigkeit zu überprüfen oder für jede Parameterkombination ein optimales Ergebnis liefern.

Durch die aufgestellten Abgrenzungskriterien wird das dritte, unter Kapitel 1 vorgestellte Definitions kriterium ausgeschlossen, da der Fokus des Verfahrens auf der Struktur und der Generation des eigentlichen Dungeons liegen soll. Das Füllen dieses mit zusätzlichen Inhalten soll dabei keine Rolle spielen. Die aufgeführten Kriterien können in ihrer Gesamtheit von keinem der unter Kapitel 2 vorgestellten Algorithmen erfüllt werden, jedoch sollen einige grundlegende Ansätze und Konzepte als Anregung für das neu entwickelte Verfahren dienen. Dabei soll vor allem die Schnelligkeit und die Konfigurationsmöglichkeiten im Vordergrund stehen.

3.2 Vorstellung der verfügbaren Parameter

Im Folgenden soll auf die Parameter, die für die Kontrolle des Verfahrens zur Verfügung stehen, eingegangen und diese kurz vorgestellt und erläutert werden. Unter Abschnitt 3.3 wird dann der technische Einsatz dieser innerhalb des Algorithmus eingehender diskutiert. Die Parameter lassen sich in verschiedene Kategorien einteilen, je nachdem auf welchen Aspekt der Generation sie sich auswirken. Besonders wichtig ist dabei der Parameter `tileSize`, der im Unity Inspektor ganz oben zu finden ist und keiner Kategorie zugeordnet wird. Mit diesem lässt sich der Algorithmus auf die Größe der verwendeten Asset-Tiles konfigurieren. Die Einheit entspricht dabei einem Meter in Unity. Alle darauffolgenden Parameter beziehen sich auf diese Größenangabe.

Dungeon-Parameter

Die folgenden Parameter beeinflussen die generelle Struktur und den Aufbau des Dungeons. Somit kann Einfluss auf dessen Größe, aber auch auf den generellen strukturellen Aufbau genommen werden.

`dungeonSize` Bestimmt die maximale, flächenmäßige Größe des Dungeons.

`roomCount` Legt die maximale Anzahl der Räume fest, über die das Dungeon verfügen darf.

compressFactor Dieser Parameter bestimmt, wie kompakt die Struktur des Dungeons generiert wird, also wie eng Räume beieinander platziert werden und wie groß die Lücken zwischen diesen sind. Der Wert lässt sich von 0 (keine Komprimierung) bis 1 (maximale Komprimierung) einstellen.

allowPathCrossing Diese Option ermöglicht es einzustellen, ob sich kreuzende Wege erlaubt sind oder nicht. Kreuzungen erhöhen den Vernetzungsgrad im Dungeon deutlich und ermöglichen eine kompaktere Struktur. Der Parameter wird unter Abschnitt 3.4 noch einmal eingehender diskutiert.

connectionDegree Legt fest, wie stark die Räume im Dungeon miteinander vernetzt sein sollen. Wird der Parameter auf 0 gesetzt, wird die minimale Anzahl an Verbindungswegen im Dungeon generiert. Bei der Erhöhung des Faktors werden zusätzliche Verbindungen hinzugefügt.

shape Mithilfe dieses Parameters, lässt sich die Form des Dungeons beeinflussen. Das Erhöhen des Werts führt zur Verengung des Generationsbereichs, wodurch sich das Dungeon nicht in alle Richtungen im gleichen Maße ausbreiten kann und somit eher eine langezogene Form bekommt (siehe Abb. 3.1 links und mittig)

dungeonShapeBlueprint Die Angabe eines Blueprints ermöglicht es, die Form des Dungeons genauer zu bestimmen. Sollte ein Blueprint angegeben werden, so wird der shape Parameter bei der Generation ignoriert. Auf die genaue Funktionsweise von Blueprints wird im Unterabschnitt 3.5.1 genauer eingegangen.

Raum-Parameter

Die Parameter ermöglichen das Konfigurieren der Räume und deren Struktur innerhalb des Dungeons.

roomDimensionX / roomDimensionY Durch diese Parameter kann die minimale und maximale Ausdehnung von Räumen in x- bzw. y-Richtung festgelegt werden. Somit lässt sich nicht nur die Größe eines Raums einstellen, sondern auch Einfluss auf dessen Form nehmen. Werden z. B. für die Ausdehnung in y-Richtung größere Werte als in x-Richtung gewählt, so wird ein Großteil der Räume eine rechteckige Form aufweisen, die sich in y-Richtung streckt.

roomDistance Bestimmt den minimalen bzw. maximalen Abstand zwischen zwei verbundenen Räumen

roomBlueprints Blueprints ermöglichen eine genauere Kontrolle über die Raumform. Dabei lassen sich verschiedene Blueprints und deren Häufigkeit angeben. Zudem kann gewählt werden, ob die normale Raumgeneration mit genutzt werden soll oder ob Räume nur über die angegebenen Blueprints generiert werden. Der Einsatz von Blueprints wird im Unterabschnitt 3.5.1 eingehender diskutiert.

Weg-Parameter

Mithilfe dieser Parameter lässt sich die Positionierung des Start- und Endraums innerhalb des Dungeons konfigurieren.

startRoomPosition / endRoomPosition Diese Optionsfelder ermöglichen es, einen Positions-Typ für den Start- bzw. Endraum auszuwählen. Dabei stehen die Optionen *Center*, *Edge* und *Random* zur Auswahl. Diese bestimmen die Positionierung des Raums innerhalb des Dungeons. Die Positions-Typen lassen sich dabei frei miteinander kombinieren, wobei die Auswahl von *Center* für beide Räume jedoch zu keinem sinnvollen Ergebnis führt.

minPathLength / maxPathLength Bestimmt wie lang der Weg zwischen Start- und Endraum minimal und maximal sein soll. Die dabei angegebenen Grenzen werden dabei durch den Algorithmus so gut wie möglich erfüllt. Die angegebene Länge bezieht sich dabei nicht auf die absolute Distanz zwischen den beiden Räumen, sondern wird dabei als die Anzahl der Räume angenommen, die durchschritten werden müssen, um vom Anfang bis zum Ende zu gelangen, inklusive des Endraums.

useFullDungeonSize Wird diese Option aktiviert, werden die anderen Weg-Parameter deaktiviert. Die Aktivierung bewirkt, dass zwei Räume ausgewählt werden, die auf gegenüberliegenden Seiten des Dungeons liegen und somit eine möglichst große Distanz aufweisen. Besonders nützlich ist diese Option bei der Generation eines durch den *shape* Parameter verengten Dungeons, da diese meist darauf ausgelegt sein sollen, von einer Seite zur anderen zu gelangen und dabei das gesamte Dungeon zu nutzen (siehe Abb. 3.1 rechts)

Tile-Parameter

Die folgenden Parameter ermöglichen die Konfiguration der Asset-Tiles, die genutzt werden, um das Dungeon zu generieren.

tileSetTable Ermöglicht es eine zuvor durch den Game-Designer angelegte Struktur an den Algorithmus zu übergeben, in der verschiedene Tiles definiert sind, die zur Generation des Dungeons genutzt werden.

generateCeiling Durch die Option lässt sich konfigurieren, ob eine Decke im Dungeon generiert werden soll oder nicht. Somit lassen sich Dungeons für Top-Down Spiele aber auch First- oder Third-Person Spiele generieren.

dungeonHeight Definiert die Höhe des Dungeons und somit, wie hoch die Wände generiert werden soll und in welcher Höhe sich die Decke befindet, sollte diese generiert werden.

Die gegebenen Parameter geben dem Benutzer größtmögliche Kontrolle über den Algorithmus, sodass sich das Resultat gut manipulieren lässt. Jedoch lassen sich die Parameter nicht komplett frei kombinieren, da zwischen ihnen eine gewisse Abhängigkeit herrscht. Wird zum Beispiel der Parameter *dungeonSize* im Verhältnis zum Parameter *roomCount* sehr klein gewählt, kann nicht die geforderte Anzahl an Räumen generiert werden. Ähnlich verhält sich die Beziehung zwischen den Parametern *maxPathLength* und *roomCount*. Die Anzahl der generierten Räume

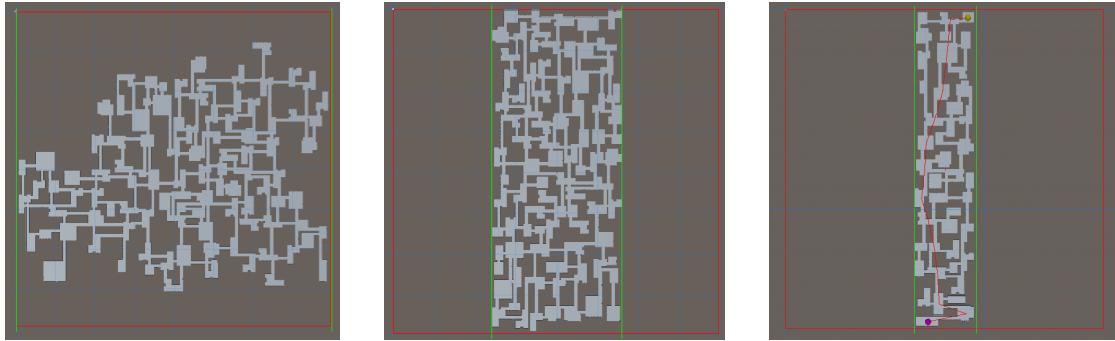


Abbildung 3.1: Links und Mitte: Mithilfe des Parameters `shape` lässt sich die Form des Dungeons in einfacher Weise beeinflussen. Wird der Parameter auf 0 gesetzt, wird die gesamte Fläche zur Generation genutzt, wie auf dem linken Bild zu sehen ist. Auf dem mittleren Bild wurde hingegen ein Wert von 0.6 für den `shape` Parameter verwendet, was zu einer deutlichen Verengung des Generationsbereichs führt. Rechts: Hier wurde ein Dungeon mit einem `shape` Parameter von 0.75 erzeugt und für die Generation des Weges der Parameter `useFullDungeonSize` aktiviert. Dadurch werden Start- und Endraum auf gegenüberliegenden Seiten des Dungeons generiert, sodass die volle Länge durchquert werden muss.

begrenzt maßgeblich die maximal mögliche Länge des Weges zwischen Start und Endpunkt. Solche Konflikte lassen sich nur sehr schwer verhindern, da die Abhängigkeiten zwischen den Parametern erst nach der Generation aufgelöst werden können. Besonders die Parameter, die sich auf Prozesse auswirken, die nach der eigentlichen Generation des Dungeons erfolgen, werden stark durch die Parameter beeinflusst, die sich direkt auf die eigentliche Generation beziehen. Diese Problematik wird im Unterabschnitt 3.3.2 noch einmal eingehender diskutiert. Da das Verfahren jedoch auf eine semi-automatisierte Arbeitsweise ausgelegt ist und die Parameter vor dem automatisiertem Einsatz des Algorithmus in einem Spiel durch den Game-Designer sorgfältig erprobt und ausgewählt werden sollen, stellen diese Abhängigkeiten kein schwerwiegendes Problem dar.

3.3 Grundlegende Arbeitsweise des Algorithmus

Das Verfahren verfolgt einen einfachen konstruktiven Ansatz, um dem Kriterium der Laufzeit bestmöglich nachzukommen. Auch wenn, wie im Kapitel 2 herausgestellt, such-basierte Verfahren ein potenziell besseres Ergebnis liefern, können diese das Kriterium der geringen Laufzeit nicht erfüllen und würden zu hohen Ladezeiten im Spiel führen. Das entwickelte Verfahren setzt auf einen rekursiven Generationsansatz von einem zentralen Raum aus und richtet sich dabei nach den gegebenen Parametern. Parallel dazu wird ein Graph aufgespannt, der die Struktur des Dungeons widerspiegelt und zur anschließenden Auswertung verwendet werden kann. Die Generation kann dabei in drei große Phasen unterteilt werden:

1. Generation der Dungeonstruktur
2. Setzen von Start- bzw. Endraum
3. Platzieren der Asset-Tiles

Phase Eins und Zwei arbeiten dabei nur theoretisch und erzeugen basierend auf den gegebenen Parametern die Struktur des Dungeons. Erst die dritte Phase baut auf Grundlage dieser beiden

Schritte das Dungeon aus den Asset-Tiles auf. Im Folgenden soll auf die einzelnen Phasen eingegangen und deren Abläufe genauer analysiert werden.

3.3.1 Generation der Dungeonstruktur

Im ersten Schritt der Generation geht es hauptsächlich um die Erzeugung der Dungeonstruktur, also der Anordnung der Räume und Gänge im Gesamtbild des Dungeons. Dieser Prozess lässt sich in drei Schritte unterteilen: (1) rekursiver Generationsprozess der Dungeonstruktur, (2) iterative Verbesserung der Struktur und (3) Erzeugen von zusätzlichen Verbindungen zwischen den Räumen. Am Ende dieser drei Schritte ist die Struktur vollständig generiert. Die Räume und Gänge werden dabei in einer Bitmatrix gespeichert, wobei ein Feld der Matrix einem Tile entspricht. Diese ermöglicht es zu überprüfen, ob ein neu generierter Raum mit einem bereits existierenden Raum kollidieren würde, wodurch Überschneidungen verhindert werden können. Zusätzlich wird ein Graph erzeugt, der die Struktur des Dungeons widerspiegelt und somit zur Evaluation in den weiteren Schritten genutzt werden kann.

Rekursiver Generationsprozess der Dungeonstruktur

Dieser Schritt stellt das Kernstück des Algorithmus dar, da hier ein Großteil der Struktur erzeugt wird. Der folgende C#-Code zeigt vereinfacht das grundlegende Prinzip des Verfahrens und dient dem besseren Verständnis. Die eigentliche Implementierung im Code ist komplexer, da die hier gezeigte vereinfachte Variante einige Problematiken aufweist, die unter Abschnitt 3.4 eingehender diskutiert werden:

```

1 int currentRoomCount = 1;
2 void GenerateRecursively(Room parentRoom){
3     for (int direction = 0; direction < 4; direction++){
4         if (currentRoomCount >= roomCount) continue;
5         Room newRoom = null;
6         for (int attempt = 0; attempt < 1 + compressFactor * 10; attempt++){
7             newRoom = GenerateNewRoom(parentRoom, direction);
8             if (newRoom != null){
9                 currentRoomCount++;
10                SaveRoomToBitMatrix(newRoom);
11                GenerateRecursively(newRoom);
12                break;
13            }
14        }
15    }
16 }
```

Um den rekursiven Generationsprozess zu starten, wird ein initialer Startraum an die Funktion übergeben. Dieser wird im Zentrum der zur Verfügung stehenden Fläche erzeugt. Von diesem parentRoom aus werden neue Räume in vier Richtungen generiert. Für die Generation eines neuen Raums wird zuerst ein Raum mit zufälliger Größe berechnet. Dieser wird anschließend vom Startraum aus in eine Richtung verschoben, wobei die Entfernung durch die Parameter

3 Architektur des entwickelten Generationsverfahrens

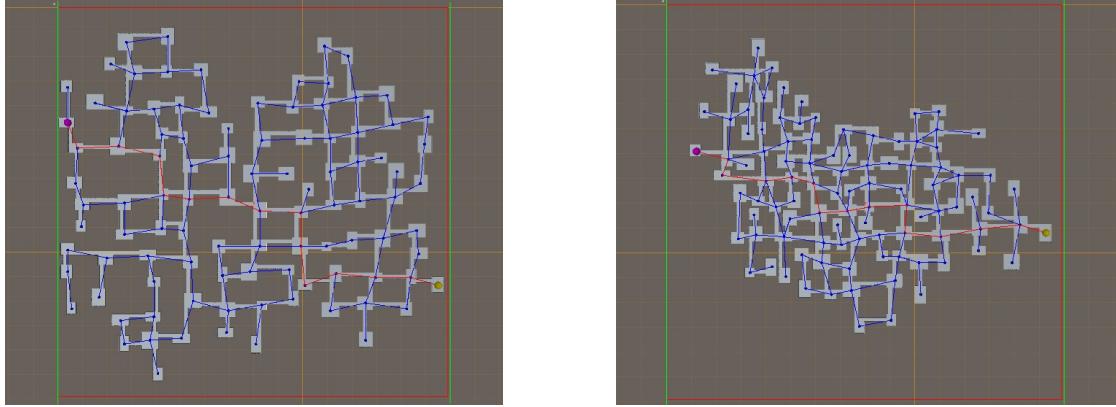


Abbildung 3.2: Die Bilder zeigen die Auswirkungen des `compressFactor`s auf die Struktur des Dungeons. Im linken Bild wurde dieser mit 0 gewählt, sodass die Lücken zwischen den Räumen bei der Generation deutlich größer sind. Im Rechten Bild wurde ein Faktor von 1 gewählt, wodurch die Lücken zwischen den Räumen deutlich kleiner ausfallen und somit auch das gesamte Dungeon eine deutlich kleinere Fläche einnimmt.

`minPathLength` und `maxPathLength` begrenzt ist. Zusätzlich wird eine seitliche Verschiebung um einen zufälligen Wert vorgenommen und ein Gang zwischen dem neu generierten Raum und dem Startraum erzeugt. Diese Funktionalität wird im Code-Beispiel in Zeile 7 in der Funktion `GenerateNewRoom()` gekapselt. Ein Raum wird dabei nach dem objektorientierten Prinzip durch eine eigene Klasse repräsentiert, die Position, Größe und Verbindungsräume speichert. In Zeile 10 wird die neu erzeugte Struktur in einer Bitmatrix gespeichert, um zu jedem Zeitpunkt überprüfen zu können, ob sich an einer bestimmten Position bereits ein Raum oder ein Gang befindet. Dadurch können Kollisionen vermieden werden. Jeder dadurch neu erzeugte Raum wird rekursiv erneut an die Funktion übergeben. Als Abbruchbedingung für die Rekursion wird die Anzahl der Räume überprüft, wobei die maximal erlaubte Anzahl durch `roomCount` festgelegt ist.

Ein weiterer wichtiger Parameter, der einen großen Einfluss auf den Generationsprozess nimmt, ist der `compressFactor`, der an verschiedenen Stellen im Code zum Einsatz kommt. So wird er bei der Kollisionserkennung in der Bitmatrix genutzt, um die erlaubten Abstände zwischen zwei Räumen zu bestimmen. Wird ein neuer Raum generiert, so wird dessen Position in der Bitmatrix überprüft. Die Größe der zu überprüfenden Fläche hängt dabei vom `compressFactor` ab. Wird dieser größtmöglich mit 1 gewählt, so ist die überprüfte Fläche um ein Tile-Größer als die Raumgröße. Dieser Abstand ist mindestens nötig, damit zwischen zwei benachbarten Räumen immer mindesten ein Feld frei liegt und die Räume nicht miteinander verschmelzen. Wird für den `compressFactor` der Wert 0 gewählt, so wird eine Fläche überprüft, die um 5 Felder größer ist, als der Raum. Somit existiert zwischen den Räumen ein größerer Abstand, wodurch sich das Dungeon auf einer deutlich größeren Fläche erstreckt.

Einen wichtigen Aspekt der Funktionsweise stellt zudem die Generation eines Raums in mehreren Anläufen dar, was im Code in Zeile 6 zu sehen ist. Wenn die Generation eines Raums scheitert, da dieser z. B. mit einem anderen Raum kollidiert, wird die Generation mehrmals mit anderen Parametern erneut versucht. Die Anzahl der Versuche wird dabei ebenso durch den Parameter `compressFactor` gesteuert, der mit einem Faktor von Zehn multipliziert wird. Dieser

3 Architektur des entwickelten Generationsverfahrens

Faktor	1	5	10	15	20	25
Raumanzahl $\bar{\Omega}$	59, 51	64, 49	67, 46	68, 79	69, 95	71, 33
Laufzeit $\bar{\Omega}$	120, 4 ms	152, 9 ms	179, 2 ms	210, 9 ms	243, 3 ms	282, 4 ms

Tabelle 3.1: Die Tabelle zeigt die Testreihe zur Ermittlung des Multiplikationsfaktors bei sonst identischer Parameterauswahl. Es wird versucht 100 Räume in einem Dungeon mit einer `dungeonSize` von 100 zu generieren. Die Generation wird 100 Mal durchgeführt und die durchschnittliche Anzahl der generierten Räume und die durchschnittliche Laufzeit betrachtet. Die gemessene Standardabweichung beträgt dabei bei jedem Versuch in etwa 3, 4 Räume.

Wert hat sich dabei durch mehrere Tests als geeignet erwiesen, da eine möglichst große Anzahl an Räumen in einem beengtem Gebiet bei so geringer Laufzeit wie möglich erreicht werden konnte. Für die Tests wurden bei der Generation von 100 Räumen und einer `dungeonSize` von 100, verschiedene Werte bei sonst identischer Parameterwahl untersucht. Die Ergebnisse sind in Tabelle 3.1 zu sehen. Dabei wird deutlich, dass sich die durchschnittliche Laufzeit bei der Erhöhung des Faktors um 5 um etwa 30 ms steigert, was einem linearen Wachstum entspricht. Bis zum Faktor 10 steigt sich die Anzahl der im Durchschnitt generierten Räume in etwa um 13, 4%. Bei weiterer Erhöhung des Faktors um 10 auf 20, steigt der Prozentwert nur auf 17, 5% an. Es wird also deutlich, dass die Verbesserung, die mit einem höheren Faktor erreicht werden kann, mit steigendem Wert immer geringer ausfällt. Aus diesem Grund wurde der Faktor 10 gewählt, da dieser bei möglichst geringer Erhöhung der Laufzeit noch eine deutliche Verbesserung bringt.

Sollte es auch nach mehrmaligen Generationsversuchen nicht gelingen, einen Raum in die entsprechende Richtung zu generieren, wird diese übersprungen und mit der Generation des nächsten Raums fortgefahren. Somit spannt sich ein einfaches Netz aus Räumen und Gängen auf, wie in Abbildung 3.3 links zu sehen ist. Zudem werden für jeden Raum die Nachbarräume gespeichert, mit denen er verbunden ist. Somit kann jeder Raum als ein Knoten in einem Graphen aufgefasst werden, welche über Kanten mit weiteren Knoten verbunden ist. Dieser Graph wird in Abbildung 3.3 links durch die blauen Linien und Punkte veranschaulicht. Die entstandene Struktur des Dungeons erweist sich noch als recht einfach, da sich vom zentralen Raum verzweigte Äste aufspannen, die jedoch nicht miteinander verbunden sind. Es existieren somit keine Rundwege oder Verknüpfungen, die erst noch in den weiteren Schritten hinzugefügt werden müssen.

Iterative Verbesserung der Struktur

Dieser Schritt hat nur Auswirkungen auf den Generationsprozess, wenn die Größe des Dungeons im Verhältnis zur festgelegten Raumanzahl zu klein ist, also nach dem ersten Generationsschritt nicht die durch den Parameter `roomCount` festgelegte Anzahl an Räumen erzeugt wurde. Um die Anzahl zu erhöhen, wird mehrere Male über alle zuvor generierten Räume iteriert und diese erneut als initialer Parameter an die rekursive Funktion aus Schritt Eins übergeben, wie im folgenden Code-Abschnitt zu sehen ist:

```

1 void StartIterativeImproving(){
2     if (roomsGraph.Count >= roomCount) return;
3     for (int attempts = 0; attempts < compressFactor * 10; attempts++) {
4         for (int i = 0; i < roomsGraph.Count; i++) {

```



Abbildung 3.3: Links: Das Bild zeigt ein generiertes Dungeons bestehend aus 100 Räumen und einer `dungeonSize` von 200. Mitte und Rechts: Diese beiden Dungeons wurde mit gleichen Parametern auf beengtem Raum generiert. Die `dungeonSize` beträgt dabei 100 und der angestrebte `roomCount` ebenfalls 100. Im mittleren Dungeon wurde für die Generation keine iterative Verbesserung verwendet, wodurch nur eine Anzahl von 59 Räumen generiert werden konnte. Im rechten Dungeon wurde hingegen die iterative Verbesserung im Generationsprozess angewendet, wodurch 83 Räume erzeugt werden können.

```

5     Room startRoom = roomsGraph[i];
6     GenerateRecursively(startRoom);
7     if (roomsGraph.Count >= roomCount) return;
8   }
9 }
10 }
```

Wie oft diese Iteration erfolgt, wird durch den Parameter `compressFactor` festgelegt, wie in Zeile 3 zu sehen ist. Anschließend wird in Zeile 4 – 8 über alle Knoten des Graphen iteriert und diese erneut an die rekursive Generation übergeben. Sollte die geforderte Anzahl an Räumen erreicht werden, wird der Prozess in Zeile 7 vorzeitig beendet. Durch dieses Verfahren kann die Raumanzahl deutlich erhöht werden (siehe Abb. 3.3 mittig und rechts).

Erzeugen von zusätzlicher Verbindungen

Damit eine komplexere Struktur entsteht, werden zusätzliche Verbindungen zwischen benachbarten Räumen erstellt. Wie stark die Vernetzung erfolgt, kann durch den Parameter `connectionDegree` beeinflusst werden. Für den Prozess wird das Dungeon als ein Graph G aufgefasst, der ungerichtet ist und keine Kantengewichte aufweist. Jeder Raum wird dabei als ein Knoten und jeder Gang als Kante aufgefasst, wodurch die Gesamtheit aller Räume des Dungeons die Knotenmenge V und alle Gänge die Kantenmenge E bilden. Somit ist der Graph durch

$$G = (Räume, Gänge)$$

definiert. Das Verfahren iteriert über jeden Knoten des Graphen und berechnet dessen Grad $d(v)$. Liegt dieser unter einer gewissen Grenze, die durch den Parameter `connectionDegree` gesteuert wird, werden Knoten in der direkten Umgebung gesucht, die über das Einfügen einer neuen Kante miteinander verbunden werden können. Wird ein `connectionDegree` von null gewählt, werden keine zusätzlichen Verbindungen erzeugt. Damit eine neue Kante und damit ein neuer

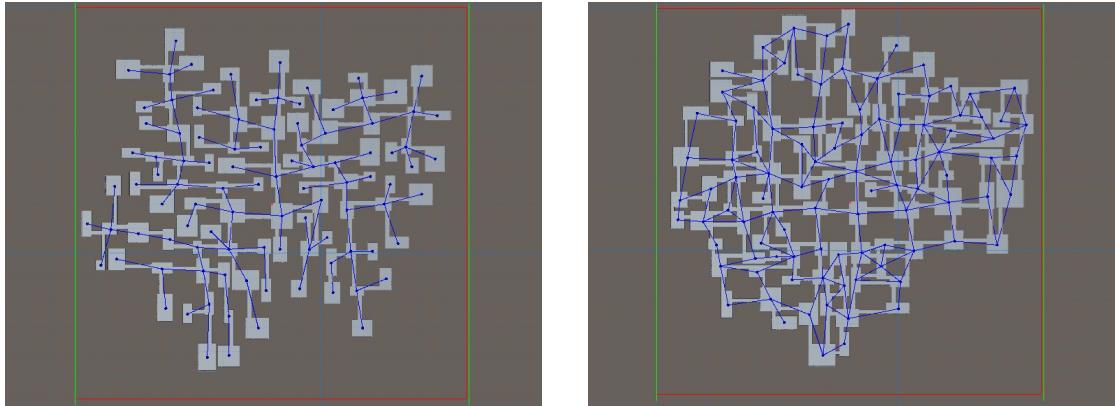


Abbildung 3.4: Links: Das Dungeon wurde ohne die Generation von zusätzlichen Verbindungen erzeugt. Rechts: Für dieses Dungeon wurde der Parameter `connectionDegree` auf 1 gesetzt, sodass die größtmögliche Anzahl an zusätzlichen Verbindungen erzeugt wird. Durch die neuen Verbindungen gewinnt die Struktur des Dungeons deutlich an Komplexität.

Weg im Dungeon erstellt werden kann, muss überprüft werden, ob zwischen den beiden Räumen in x- oder y-Richtung ein entsprechend großer Überschneidungsbereich für die Breite des Weges vorhanden ist und ob die Entfernung zwischen den zwei Räumen dem Parameter `roomDistance` entspricht. Sollte eine Verbindung theoretisch möglich sein, wird zusätzlich durch einen Zufallswert entschieden, ob diese generiert wird oder nicht. Die Wahrscheinlichkeit hängt auch hier vom Parameter `connectionDegree` ab.

Anschließend werden die Grid-Positionen für die Platzierung des Ganges berechnet, in die Bitmatrix eingetragen und der Graph um die zusätzliche Kante ergänzt. Somit erhöht sich der Grad $d(v)$ des entsprechenden Knotens um eins. Durch diesen Prozess kann der Vernetzungsgrad im Dungeon deutlich erhöht werden, wie in Abbildung 3.4 zu sehen ist. Nach der Formel 2.1 zur Berechnung des durchschnittlichen Grades des Dungeons ergibt sich für einen `connectionDegree` von 0 ein durchschnittlicher Grad von ca. 1,9 und für einen `connectionDegree` von 1 ein Grad von ca. 4,1. Dies entspricht einer Steigerung der Verbindungsanzahl um mehr als 100 %.

Dieser Prozess hat großen Einfluss auf die Struktur des Graphen und auch auf dessen Eigenschaften. Werden keine zusätzlichen Verbindungen generiert, handelt es sich um einen kreisfreien Graphen der keine Zyklen enthält. Erst durch das Hinzufügen von zusätzlichen Verbindungen entsteht ein zyklischer Graph. Somit kann davon ausgegangen werden, dass bei der Wahl des `connectionDegree` mit 0 immer nur genau ein Weg zwischen zwei ausgewählten Räumen existiert.

3.3.2 Setzen von Start- bzw. Endraum

Dieser Schritt nutzt ebenso die Repräsentation des Dungeons als Graph um einen Start- bzw. Endraum auszuwählen. Da es sich um ein konstruktives Verfahren handelt und somit das Dungeon zu diesem Zeitpunkt bereits vollständig generiert wurde, kann es dazu kommen, dass die geforderten Parameter nicht vollständig erfüllt werden können. So wird es in einem Graphen mit $|G| = 20$ niemals möglich sein einen Weg der Länge 100 zu generieren, da wie unter Ab-

schnitt 2.1 definiert wurde, ein Weg einen Teilgraphen P von G darstellt und somit alle Knoten voneinander verschieden sein müssen. Der Algorithmus versucht dabei jedoch immer das bestmögliche Ergebnis innerhalb der geforderten Parameter zu erzielen. Zudem handelt es sich um ein Verfahren, das auf eine semi-automatisierte Arbeitsweise ausgelegt ist. Die Parameter sollen also zuvor durch den Game-Designer erprobt und entsprechende Grenzen ausgewählt werden, bevor das System in ein Spiel integriert werden kann. Da bei jedem Generationsprozess mit gleichen Parametern ähnliche Ergebnisse erzielt werden, lassen sich durch einfaches Ausprobieren problematische Parameterkombinationen schnell ausfindig machen. Um die Analyse zu erleichtern, existiert ein Debug-System, das sich einschalten lässt und nützliche Informationen über die Eigenschaften des Dungeons gibt. Somit lassen sich Parameter gut aufeinander abstimmen und das Ergebnis der Generation kann durch den Game-Designer im Voraus erprobt und konfiguriert werden.

Für die Auswahl des Start- bzw. Endraums wird der Graph des Dungeons betrachtet. Die Auswahl der entsprechenden Knoten wird auf Grundlage von zwei Faktoren entschieden. Zum einen durch die Auswahl des entsprechenden Positions-Typs für Start- und Endraum und zum anderen durch die minimale und maximale Distanz des Weges. Diese zwei Faktoren werden durch die im Unterabschnitt 3.2 vorgestellten *Weg-Parameter* gesteuert. Für die Auswahl des Positions-Typs stehen jeweils *Center*, *Edge* und *Random* zur Verfügung. Wird als Typ *Center* gewählt, so wird der zentrale Knoten, der initial für die Generation des Dungeons diente, verwendet. Wird als Typ *Edge* gewählt, so wird zufällig ein Knoten aus dem Randbereich des Dungeons genutzt, wobei die Menge der möglichen Knoten durch die Berechnung der Konvexen Hülle mithilfe des im Abschnitt 2.2 vorgestellten Gift-Wrapping-Algorithmus bestimmt wird. *Random* als Typ führt zur Auswahl eines zufälligen Knotens aus $V(G)$. In Abbildung 3.5 links ist die Kombination aus *Center-Edge* und mittig die Kombination *Random-Random* zu sehen.

Um den Faktor der Distanz bei der Auswahl mit berücksichtigen zu können, muss zuerst der Start- oder der Endraum festgelegt werden. Von diesem aus wird der Graph anschließend mithilfe einer vereinfachten Variante des im Unterabschnitt 2.2 vorgestellten Dijkstra-Algorithmus ausgewertet, sodass für jeden Raum bzw. Knoten ein Weg mit der minimalen Entfernung zum festgelegten Start-Knoten bekannt ist:

```

1 static Queue<Room> callQueue = new Queue<Room>();
2 static void SimplifiedDijkstra(Graph graph, Room parentNode){
3     float distance = parentNode.pathDistance + 1;
4     foreach (var node in parentNode.connections){
5         if (node.pathDistance <= distance) continue;
6         node.pathDistance = distance;
7         node.pathParent = parentNode;
8         callQueue.Enqueue(node);
9     }
10    while (callQueue.Count > 0)
11        SimplifiedDijkstra(graph, callQueue.Dequeue());
12 }
```

Zu Beginn ist der initiale Startknoten mit einer Weg-Distanz von 0 und alle anderen Knoten mit einer Distanz von ∞ initialisiert. Die Funktion wird mit dem Startknoten als *parentNode* aufgerufen und iteriert in Zeile 4 – 9 über alle benachbarten Knoten. Falls die in diesen zu-

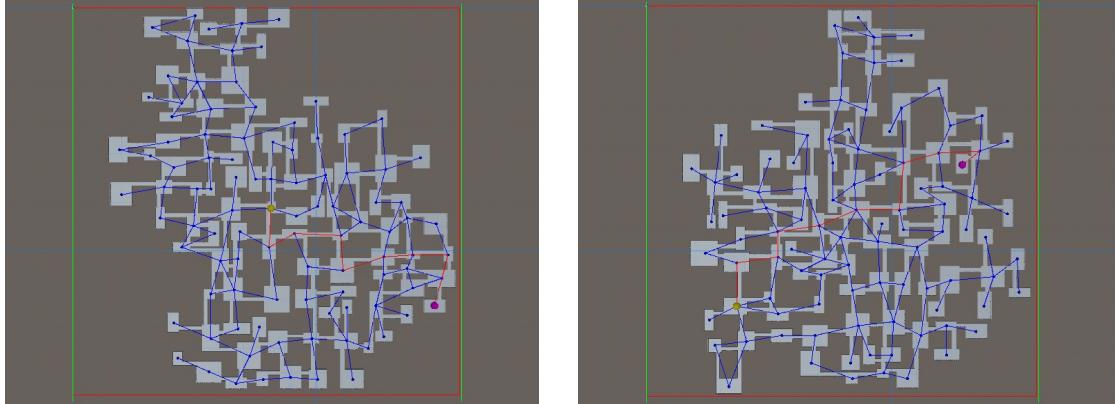


Abbildung 3.5: In beiden Dungeons wird der Starraum mit einem gelben und der Endraum mit einem magentafarbenen Punkt markiert. Die rote Linie zeigt den kürzesten Weg zwischen diesen beiden Räumen. Für das linke Dungeon wurde für den Starraum *Center* und für den Endraum *Edge* als Position gewählt. Im rechten Dungeon wurde für beide Räume *Random* als Positionierung gewählt.

vor gespeicherte Weg-Distanz größer als die in Zeile 3 berechnete Distanz der `parentNode +1` ist, wird die neue Distanz und der `parentNode` in diesem gespeichert. Jeder der Knoten wird anschließend rekursiv als neuer Startknoten an die Funktion übergeben. Um zu verhindern, dass der Algorithmus unnötig in die Tiefe geht, wird eine in Zeile 1 definierte Queue² für den rekursiven Aufruf in Zeile 11 verwendet, wobei es sich um die durch C# bereitgestellte Datenstruktur handelt. Nachdem der gesamte Graph ausgewertet wurde, ist für jeden Knoten die kürzeste Distanz zum initialen Startknoten bekannt. Nun kann entsprechend des `minPathLength` und `maxPathLength` Parameters zufällig der andere Raum ausgewählt werden. Sollte keiner der Räume das Distanzkriterium erfüllen, wird ein Raum gewählt, der dieses bestmöglich erfüllt.

Als Starraum für den Dijkstra-Algorithmus wird der Raum mit dem spezifischeren Positions-Typen ausgewählt, wobei gilt: *Center* ist spezifischer als *Edge* ist spezifischer als *Random*. Während für den Typ *Center* nur ein Raum zur Verfügung steht, können bei Typ *Edge* alle Räume, die am Rand des Dungeons liegen, genutzt werden. Der Typ *Random* ermöglicht jedoch die freie Auswahl aus allen verfügbaren Räumen, wodurch dieser am unspezifischsten ist. Sollte der Endraum einen spezifischeren Raum-Typ als der Starraum aufweisen, wird dieser zuerst festgelegt und als Startpunkt für den Dijkstra-Algorithmus verwendet. Durch diese Herangehensweise wird sichergestellt, dass das Verfahren ein möglichst gutes Ergebnis erzielt, ohne dass alle Knoten des Graphen mithilfe des Dijkstra-Algorithmus ausgewertet werden müssen.

Wird der Parameter `useFullDungeonSize` verwendet, haben die anderen Weg-Parameter keine Auswirkungen mehr auf den Prozess. Wie bereits unter Abschnitt 3.2 herausgestellt, kann somit die volle Länge bzw. Breite des Dungeons genutzt werden, was vor allem beim Einschränken der Generationsfläche durch den `shape` Parameter nützlich sein kann. Für die Auswahl werden zuerst der nördlichste, südlichste, östlichste und westlichste Raum im Dungeon gesucht, was sich einfach über den Vergleich der x- und y-Werte erzielen lässt. Anschließend wird aus diesen das Paar mit dem größeren Abstand als Start- und Endraum ausgewählt.

²<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1>

3.3.3 Platzierung der Asset-Tiles

In diesem Schritt geht es darum, die zuvor nur theoretisch erzeugten Strukturen, bestehend aus Räumen und Gängen, mithilfe von Assets in die Spielwelt zu überführen. Diese müssen durch den Game-Designer bereitgestellt werden und der durch den Parameter `tileSize` festgelegten Größe entsprechen. Zudem muss die Ausrichtung und Positionierung des Pivot-Punkts bei jedem Asset richtig eingestellt werden. Für die Generation iteriert das Verfahren zunächst über alle Räume. Für jeden Raum wird dabei ein Tile-Set aus einer Liste ausgewählt, in welchem verschiedene Assets definiert sind. Auf die Definition von Assets und wie genau diese an den Algorithmus übergeben werden wird im Unterabschnitt 3.5.2 noch einmal genauer eingegangen. Da jeder Raum im Raster der `tileSize` liegt, kann über alle Rasterpunkte eines Raums iteriert und entsprechend Tiles platziert werden. Für das Setzen von Wänden müssen zusätzlich für jede Position die vier benachbarten Rasterpunkte überprüft werden. An allen Stellen, an denen ein Boden-Tile an eine leere Position grenzt, muss eine Wand gesetzt werden. Nachdem alle Räume generiert wurden, wird in gleicher Weise mit den Gängen verfahren.

3.4 Problemfelder und deren Lösungsansätze

Im folgenden Abschnitt sollen verschiedene Problemfelder beleuchtet werden und wie diese im Verfahren gelöst wurden. Zudem sollen noch nicht gelöste Probleme aufgegriffen und Methoden zur Lösung diskutiert werden.

Implementierung der rekursiven Generation

Die im Unterabschnitt 3.3.1 vorgestellte, vereinfacht rekursive Arbeitsweise des Algorithmus führt zu der Problematik, dass sich die Struktur nur in eine Richtung ausbreitet, da der rekursive Aufruf immer sofort auf dem neu generierten Raum ausgeführt wird. Somit wird ein einzelner Gang aus Räumen generiert, bis die maximale Raum-Anzahl erreicht ist (siehe Abb. 3.6 links). Um dieser Problematik entgegenzuwirken, wurde die Struktur im Code so angepasst, dass vom `parentRoom` aus zuerst alle vier Räume in alle Richtungen generiert und diese in einer Queue³ gespeichert werden. Über diese wird anschließend iteriert und jeder Raum als initialer Parameter für den rekursiven Aufruf genutzt. Durch die Queue wird sichergestellt, dass sich das Dungeon gleichmäßig in alle Richtungen ausbreitet.

Aus diesem Lösungsansatz ergibt sich jedoch eine weitere Problematik. Die dadurch erzeugte Form weist eine zu gleichförmige Struktur auf. Jedes generierte Dungeon besitzt dabei ein ähnliches Aussehen (siehe Abb. 3.6 mittig). Vom zentralen Raum aus erstrecken sich Gänge in alle vier Himmelsrichtungen, von denen erneut Arme abgehen. Diese Struktur ist in jedem generierten Dungeon mehr oder weniger deutlich zu erkennen. Zudem führt der längste Weg immer gerade durch die Mitte des Dungeons, von einer Seite zur anderen. In der finalen Implementierung wird anstelle einer Queue eine einfache Liste⁴ verwendet. In diese werden wie in dem

³<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1>

⁴<https://docs.microsoft.com/de-de/dotnet/api/system.collections.generic.list-1>

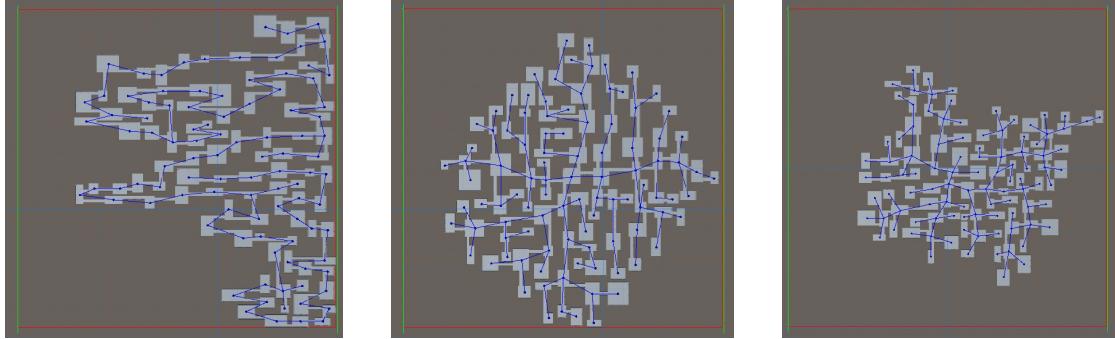


Abbildung 3.6: Zur besseren Übersicht ist in allen drei generierten Dungeons der `connectionDegree` auf null gesetzt. Links: Dieses Dungeon wurde durch den reinen rekursiven Aufruf ohne die Verwendung einer Datenstruktur zur Kontrolle der Generationsreihenfolge erzeugt. Das entstehende Dungeon besteht nur aus einem Gang mit sehr wenigen Abzweigungen. Mitte: Für dieses Dungeon wurde der rekursive Aufruf durch Nutzung einer Queue erweitert, wodurch eine sehr gleichförmige Struktur erzeugt wird. Rechts: Anstelle der Queue als Datenstruktur wird hier eine Liste verwendet, aus der Elemente zufällig ausgewählt und an die rekursive Funktion übergeben werden. Dadurch entsteht ein interessanter Dungeonkomplex mit einer ungleichmäßigen Form.

zuvor verwendeten Verfahren die vier generierten Räume gespeichert. Anschließend wird jedoch nicht über die Liste iteriert, sondern zufällig ein Raum aus dieser ausgewählt, der rekursiv an die Funktion als Parameter übergeben wird. Die dadurch entstehenden Dungeons weisen deutlich ungleichmäßige und dadurch interessantere Strukturen auf (siehe Abbildung 3.6 rechts).

Umgang mit Wegkreuzungen

Wird ein neuer Raum generiert, so wird zuerst überprüft, ob die entsprechende Position im Grid frei ist. Sollte dies der Fall sein, wird ein Weg zwischen dem neu platzierten Raum und dem Verbindungsraum berechnet. Auch bei diesem muss überprüft werden, ob die Positionierung erlaubt ist. Kollidiert der Gang beispielsweise mit einem anderen Raum, so fällt die Validierung negativ aus und beide Strukturen werden verworfen. Es kann jedoch auch dazu kommen, dass der Gang mit einem anderen Gang kollidiert und sich diese kreuzen. Ein Ansatz bestände darin, dies zu verbieten und jegliche Kollision zu vermeiden. Jedoch können sich aus den Wegkreuzungen interessante Strukturen ergeben, die den Gesamtkomplex des Dungeons dynamischer gestalten. Deswegen wurde entschieden, den Parameter `allowPathCrossings` einzuführen, der dem Game-Designer die Möglichkeit gibt diese Entscheidung je nach Situation selbst zu treffen.

Um nach der Generation den Weg durch das Dungeon berechnen zu können, werden für jeden Raum die entsprechenden Nachbarräume gespeichert, sodass ein Graph entsteht. Eine Wegkreuzung führt nun jedoch zu einer Überschneidung, wodurch neue Verbindungen im Dungeon entstehen. Werden diese im Graphen nicht mit berücksichtigt, spiegelt dieser die Struktur des Dungeons nur fehlerhaft wieder (siehe Abb. 3.7 links). Dies kann zu Problemen bei der Berechnung des Weges im Bezug auf die Erfüllung der minimalen Distanz führen. So können für Start- und Endpunkt Räume ausgewählt werden, die an einer Wegkreuzung beteiligt sind. Diese erfüllen möglicherweise die minimale Distanz ohne Beachtung der Wegkreuzung. Durch diese wird jedoch eine neue, direkte Verbindung eingefügt, wodurch es im Extremfall vorkommen kann, dass Start- und Endraum direkt nebeneinander liegen. Um dies zu vermeiden, musste der Algo-

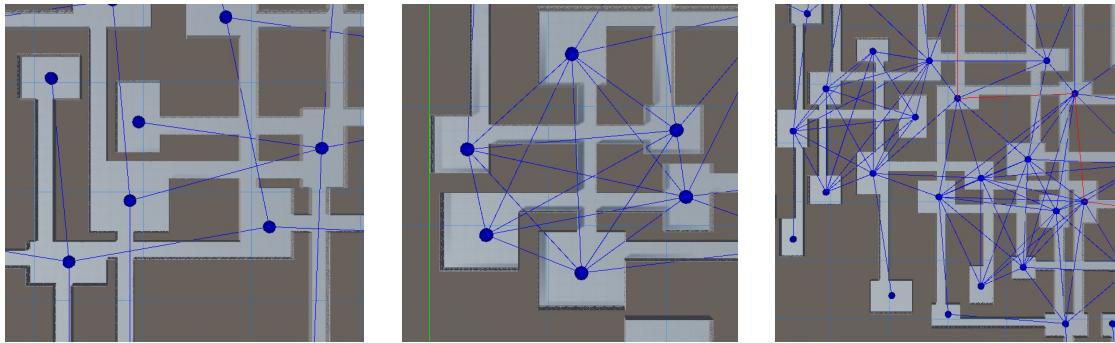


Abbildung 3.7: Links: Die Wegkreuzungen werden im Graph nicht mit beachtet, wodurch direkt aneinander grenzende Räume keine Verbindung aufweisen, was im Prozess der Weg-Berechnung zu schwerwiegenden Problemen hinsichtlich der minimalen Distanz führen kann. Mitte: Verbindungen bei Wegkreuzungen werden separat berechnet und in den Graph überführt, wodurch ein Netz aus Verbindungen entsteht. Rechts: Kreuzen sich mehrere Wege, entstehen ein komplexes Netz aus Verbindungen, das sich über große Flächen des Dungeons erstrecken kann.

rithmus dahingehend erweitert werden, dass er die zusätzlichen Verbindungen berechnet, sobald eine Wegkreuzung erkannt wird. Schneiden sich zwei Wege, werden die vier daran beteiligten Räume als benachbart gewertet. Schneidet ein weiterer Weg einen der beiden Gänge, so werden diese Räume mit in das bestehende Verbindungsnetz integriert (siehe Abb. 3.7 mittig). Dadurch können komplexe Vernetzungsstrukturen entstehen (siehe Abb. 3.7 rechts).

Nun kann darüber diskutiert werden, ob zwei Räume, die über mehrere Kreuzungen miteinander verbunden sind, noch als benachbart gewertet werden können. Als Kriterium für die Entscheidung wurde angenommen, dass ein Raum potenziell Gefahren wie Gegner oder Fallen enthalten kann, wohingegen ein Gang nur eine Verbindung zwischen zwei Räumen darstellt. Bewegt man sich nur zwischen den Kreuzungspunkten, so muss kein Raum betreten werden. Aus diesem Grund werden auch Räume, die über mehrere Kreuzungen verbunden sind und weiter auseinander liegen, als benachbart eingestuft. Jedoch wurde ein Grenzwert eingebaut, da sich ein solches Kreuzungs-Netz theoretisch durch das ganze Dungeon ziehen kann. Dafür wird die Distanz zwischen den beiden Räumen in Form der Luftlinie gemessen. Ist diese kleiner als die doppelte maximale `roomDistance`, werden die Räume als benachbart markiert. Andernfalls wird keine direkte Verbindung zwischen ihnen in den Graphen eingetragen.

Berechnung des bestmöglichen Weges

Dieser Aspekt zählt zu den Problemen, die im aktuellen Verfahren nicht optimal gelöst werden. So wird der Parameter zur Definition der Weglänge während der Generation des Dungeons nicht berücksichtigt. Dies kann dazu führen, dass die geforderte minimale Weglänge innerhalb des generierten Dungeons nicht eingehalten werden kann. Dies stellt aufgrund der semi-automatisierten Arbeitsweise des Verfahrens kein schwerwiegendes Problem dar, da die geschickte Auswahl der Parameter in der Hand des Game-Designers liegt. Allerdings stellt das aktuelle Verfahren nicht sicher, dass innerhalb des gegebenen Dungeons der bestmögliche Weg gefunden wird. Es kann also vorkommen, dass es potenziell einen Weg im Dungeon gibt, der die Kriterien der Weglänge besser erfüllt, dieser aber durch den Algorithmus nicht gefunden wird. Dies liegt daran, dass

der Start- oder Endraum festgelegt wird und von diesem aus der Graph ausgewertet wird. Innerhalb dieses Prozesses wird der bestmögliche Weg gefunden, jedoch nur für den zur Auswertung verwendeten Raum.

Um den optimalen Weg zu finden, müsste die maximale minimal Distanz im Graph gefunden werden, welche auch als Durchmesser des Graphen bezeichnet wird. Für die Lösung dieses Problems existieren verschiedene Verfahren. So lässt sich zum Beispiel der *Dijkstra-Algorithmus* für jeden Knoten im gesamten Graphen ausführen, was jedoch mit steigender Raumanzahl sehr rechenintensiv wird. Ein weiterer Algorithmus ist der *Floyd-Warshall-Algorithmus*, der das Problem für einen gerichteten, gewichteten Graphen löst. Jedoch besitzt auch dieser im schlechtesten Fall eine Komplexität von $\mathcal{O}(n^3)$, wobei n für die Anzahl der Knoten im Graph steht [4].

Da die möglichst geringe Laufzeit ein wichtiges Kriterium für das Verfahren darstellt, wurde die Entscheidung getroffen, auf die Berechnung des optimalen Weges zu verzichten. Eine Möglichkeit mit dieser Problematik umzugehen, wäre die Einführung eines weiteren Parameters zur Kontrolle der Priorität des Algorithmus. Durch diesen ließe sich einstellen, ob die Generation die bestmögliche Erfüllung der Parameter anstreben soll oder eine möglichst geringe Laufzeit das oberste Ziel darstellt. Somit könnten auch zeitaufwendigere Verfahren implementiert werden, die ein besseres Ergebnis erzielen. Jedoch mit der Möglichkeit diese zu deaktivieren, wenn die Laufzeit im Vordergrund steht.

3.5 Mechanismen zur Konfiguration

Wie in der Anforderungsanalyse herausgestellt, soll das Verfahren dem Game-Designer größtmöglichen Einfluss auf das Resultat der Generation geben, ohne dabei jedoch die Vorteile des Automatismus zu verlieren. Dies wird zum einen durch die vielen Parameter gewährleistet, die unter Abschnitt 3.2 vorgestellt wurden. Diese ermöglichen es Einfluss auf die Struktur und das Aussehen des Dungeons zu nehmen. Jedoch lassen sich mit ihnen nur schwer komplexere Einstellungen vornehmen. Zudem handelt es sich um abstrakte Zahlen und Werte, deren Auswirkungen auf die Struktur meist nur durch Ausprobieren erprobt werden können. Aus diesen Gründen bietet das Verfahren weitere Möglichkeiten zu Konfiguration, die in diesem Abschnitt genauer beleuchtet werden sollen.

3.5.1 Nutzung von Blueprints als Vorlage

Bei den sogenannten Blueprints handelt es sich um einfache Strukturvorlagen, die zum einen für die bessere Konfiguration der Form des gesamten Dungeons, als auch für einzelne Räume eingesetzt werden können. Sie ermöglichen es dem Game-Designer präzisen Einfluss auf die Struktur zu nehmen. Grundlegend handelt es sich bei den Blueprints um Bilder, die wie Masken in Bildbearbeitungsprogrammen aus schwarzen und weißen Pixeln bestehen und somit eine Struktur definieren. Die Bilder werden dabei in Form einer 2D-Textur an den Algorithmus übergeben. Der Vorteil von Blueprints besteht darin, dass sie ein mächtiges Werkzeug für den Game-Designer darstellen und dennoch sehr einfach zu handhaben sind. Im vorgestellten Verfahren

kommen Blueprints in zwei Aufgabenbereichen zum Einsatz. Zum einen für die Definition von speziellen Raumformen und zum anderen um die Form des gesamten Dungeons zu beeinflussen. Die Blueprints müssen vom Game-Designer nach gewissen Regeln angelegt und an den Algorithmus übergeben werden. Im Folgenden soll auf die zwei Einsatzgebiete genauer eingegangen werden.

Blueprints zur Kontrolle von Raumformen

Während es mit dem normalen Verfahren nur möglich ist rechteckige Räume zu generieren, ermöglichen es Raum-Blueprints interessante Raumformen ins Dungeon zu integrieren. Wichtig ist dabei, dass es sich bei den Blueprints um keine komplett vorgefertigten Räume handelt, sondern lediglich eine Struktur für den Algorithmus bereitgestellt wird, nach deren Vorbild er die Räume generieren kann (siehe Abb. 3.8 links und mittig).

Dabei können beliebig viele verschiedene Blueprints, zusammen mit einer Häufigkeit, als Parameter übergeben werden. Die Größe der durch Blueprints erzeugten Räume wird dabei nicht skaliert, sondern entspricht der Größe des vorgegebenen Blueprints. Ein Pixel entspricht dabei genau einem Tile im Grid. Soll dieselbe Raumform innerhalb des Dungeons in verschiedenen Größen auftreten, so müssen mehrere Blueprints an den Algorithmus übergeben werden. Zusätzlich kann für jede Vorlage eingestellt werden, ob bei der Generation eine zufällige Rotation auf den Raum angewendet werden soll. Bei dem Rotationswinkel handelt es sich immer um einen zufällig bei der Generation gewählten Wert, der ein Vielfaches von 90 Grad darstellt. Da es in Unity keine einfache Möglichkeit gibt Texturen und somit die Blueprints zu rotieren, musste stattdessen eine andere Möglichkeit gefunden werden, die Rotation auf den Raum anzuwenden. Bei jeder Abfrage eines Pixelwertes des Blueprints, werden die Koordinaten der Abfrage um den Winkel rotiert, wodurch eine Rotation des Raums simuliert wird:

```

1 private (int x, int y) TransformCoordinates(float x, float y, float sizeX,
      float sizeY){
2     float x2, y2;
3     x -= sizeX / 2f;
4     y -= sizeY / 2f;
5     x2 = x*Mathf.Cos(Mathf.Deg2Rad*angle)-y*Mathf.Sin(Mathf.Deg2Rad*angle);
6     y2 = x*Mathf.Sin(Mathf.Deg2Rad*angle)+y*Mathf.Cos(Mathf.Deg2Rad*angle);
7     bool swapOffset = (angle == 90 || angle == -90);
8     x2 += swapOffset ? sizeY / 2f : sizeX / 2f;
9     y2 += swapOffset ? sizeX / 2f : sizeY / 2f;
10    return (Mathf.RoundToInt(x2), Mathf.RoundToInt(y2));
11 }
```

Zuerst erfolgt auf den Zeilen 3 und 4 eine Translation der Abfrage-Koordinaten, damit der Pivot-Punkt der Rotation im Zentrum des Bildes liegt. Anschließend kann die Rotation mit den Operationen der Rotationsmatrix durchgeführt werden. Bei der inversen Translation muss nun jedoch darauf geachtet werden, dass bei einem Rotationswinkel von 90 Grad die Höhe und Breite des Bildes vertauscht vorliegen. Durch die Kapslung des Verfahrens kann der Generator normal über die Raumgröße iterieren, ohne dabei auf die Rotation des Raums zu achten. In Abbildung 3.8 sind zwei Blueprints zu sehen, die anschließend für die Generation eines Dungeons eingesetzt wurden.

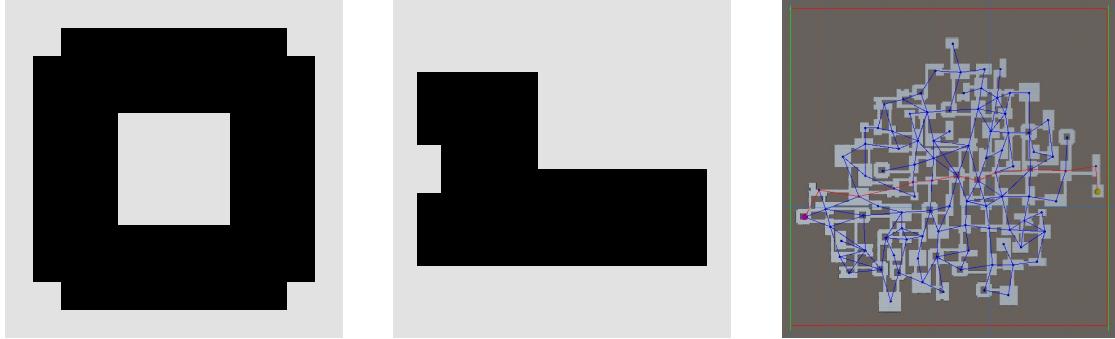


Abbildung 3.8: Links und Mitte: Die zwei Bilder zeigen verschiedene Blueprints, die als Vorlagen für Räume bei der Generation verwendet werden können. Die schwarzen Pixel bestimmen dabei die Form des Raums. Rechts: Für die Generation dieses Dungeons wurde sowohl die normale rechteckige Raumform, als auch die zwei Blueprints genutzt.

Theoretisch sollten die an den Algorithmus übergebenen Bilder nur aus schwarzen und weißen Pixeln bestehen und keine überschüssigen weißen Ränder enthalten. Zur besseren Darstellung wurden die in Abbildung 3.8 gezeigten Blueprints jedoch leicht angepasst. Die Raumform wird dabei durch die schwarzen Pixel bestimmt. Das Verfahren ermöglicht es sehr einfach komplexe Raumstrukturen zu erstellen und somit ein Dungeon mit einer großen Vielfalt an unterschiedlichen Räumen zu generieren (siehe Abb. 3.8 rechts).

Blueprints zur Konfiguration der Form des Dungeons

Ähnlich wie die Room-Blueprints Einfluss auf die Struktur und das Aussehen von einzelnen Räumen nehmen, ermöglichen es die Dungeon-Blueprints die Form des gesamten Dungeons zu beeinflussen. Zwar existiert bereits der `shape` Parameter, der jedoch nicht für komplexe Strukturen ausgelegt ist. Dabei funktionieren die Vorlagen nach einem ähnlichen Prinzip wie die Room-Blueprints. Mithilfe eines Bildes, bestehend aus weißen und schwarzen Pixeln, kann eine Form für das Dungeon festgelegt werden. Jedoch dient die Vorlage hier weniger als genauer Bauplan, sondern vielmehr als begrenzende Schablone. Es lässt sich also ein Bereich definieren, in welchem das Dungeon generiert werden darf. Weiße Pixel stehen dabei für zugelassene Bereiche in denen Räume generiert werden dürfen und schwarze Pixel verhindern das Platzieren von Strukturen. Wird ein neuer Raum generiert, wird dessen Position mit der Vorlage abgeglichen. Sollte sich die Position mit einem schwarzen Bereich im Blueprint überschneiden, wird der Raum verworfen. Somit wächst das Dungeon in Form des angegebenen Bereichs.

Um zu gewährleisten, dass die gewünschte Form bei der Generation erreicht wird muss jedoch darauf geachtet werden, dass die Anzahl der zu generierenden Räume groß genug ist, um den angegebenen Bereich auszufüllen. Zudem müssen die weißen Bereiche ein zusammenhängendes Gebiet bilden und groß genug für die angegebenen Raumgrößen sein. Zusätzlich zu den schwarzen und weißen Pixeln können rote Pixel verwendet werden, um die Bereiche zu markieren, in denen der initiale Startraum platziert werden darf. Dabei können mehrere Bereiche markiert werden, wobei sich der Algorithmus zufällig für eine der Möglichkeiten entscheidet. Wird kein rotes Pixel gesetzt, sucht sich der Algorithmus zufällig eine Position aus allen weißen Pixeln aus.

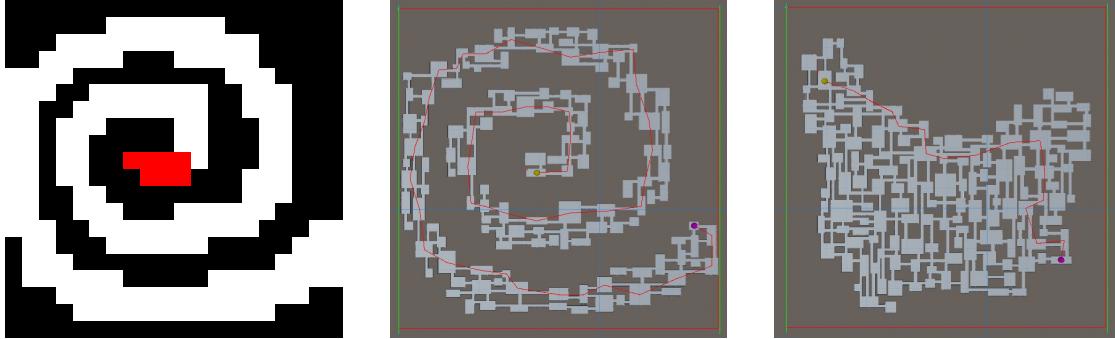


Abbildung 3.9: Links: Dieses Blueprint kann für die Generation eines spiralförmigen Dungeons genutzt werden. Der weiße Bereich gibt dabei die Fläche vor, in der das Dungeon generiert werden darf. Die roten Bereiche markieren die erlaubten Positionen für den Starraum. Mitte: Dieses Dungeon wurde unter Verwendung der spiralförmigen Vorlage generiert. Damit der Weg die volle Länge des Dungeons verwendet, wurde als Starraum *Center* und als Endraum *Random* gewählt. Zusätzlich muss für den Parameter `minPathLength` ein großer Wert verwendet werden, sodass der Raum mit der größtmöglichen Distanz für den Endraum gewählt wird. Rechts: Für dieses Dungeon wurde ein Blueprint mit einer organischen, höhlenartiger Form gewählt.

Sollten rote Pixel verwendet werden, so müssen diese direkt an ein weißes Gebiet angrenzen.

3.5.2 Tile-Sets zur Konfiguration der verwendeten Assets

Neben der eigentlichen Generation der Struktur des Dungeons übernimmt das Verfahren ebenso die Überführung dieser Struktur in die Spielwelt mithilfe von zuvor festgelegte Asset-Tiles. Das Grundlegende Verfahren wurde bereits im Unterabschnitt 3.3.3 vorgestellt. Auch hier soll es für den Game-Designer möglich sein auf das Aussehen Einfluss nehmen zu können. Der durch das Verfahren verfolgte Ansatz besteht darin, dass jeder Raum als eine Einheit angesehen und aus einer Menge zuvor ausgewählten Assets aufgebaut wird. Um die Assets an den Algorithmus übergeben zu können, muss zuerst ein sogenanntes Tile-Set angelegt werden. In diesem können sortiert nach Wand, Boden und Decke jeweils verschiedene Tiles mit einer entsprechenden Häufigkeit definiert werden (siehe Abb. 3.10 links). Ein solches Tile-Set legt die Assets fest, die anschließend für die Generation eines Raums oder eines Weges genutzt werden. Um die Vielfalt im Dungeon zu erhöhen, lassen sich mehrere solcher Tile-Sets anlegen, die anschließend in einer Tile-Set-Table gespeichert und so an den Algorithmus als Parameter übergeben werden können (siehe Abb. 3.10 mittig). Der Algorithmus iteriert über alle Räume und deren Positionen im Grid, wobei für jeden Raum ein Tile-Set aus der Tile-Set-Table ausgewählt wird. Entsprechend der Häufigkeiten der einzelnen Assets wird dann für jede Position des Raums im Grid eines ausgewählt und in der Spielwelt platziert. Diese Verfahren ermöglicht es, Räume mit kohärentem Aussehen zu erzeugen und gleichzeitig eine größere Vielfalt an verschiedenen Designs zu generieren (siehe Abb. 3.10 rechts).

3 Architektur des entwickelten Generationsverfahrens

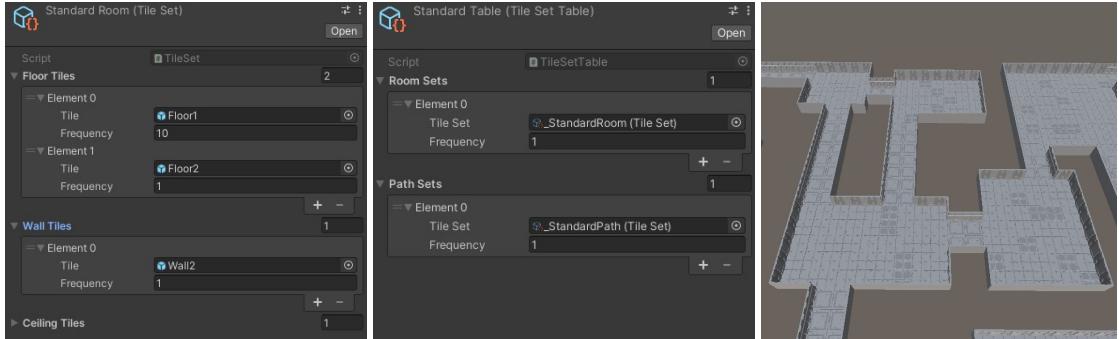


Abbildung 3.10: Links: Die Abbildung zeigt die Unity Ansicht eines Tile-Sets. In diesem werden die verschiedenen Assets, die für die Überführung der Dungeon-Struktur in die Spielwelt verwendet werden, definiert. Für den Boden wurden hier zwei verschiedene Assets angegeben, wobei die erste Bodenplatte zehnmal so häufig verwendet werden soll wie die zweite Bodenplatte. Mitte: In diesem Bild ist die Unity Ansicht einer Tile-Set-Table zu sehen, in welcher verschiedene Tile-Sets getrennt nach Raum und Gang angegeben werden können. Das im linken Bild gezeigte Tile-Set wurde hier für die Generation der Räume angegeben. Rechts: Zu sehen ist ein Ausschnitt aus einem Dungeon, für dessen Generation die im mittleren Bild zu sehende Tile-Set-Table verwendet wurde. Dabei ist deutlich die Unterscheidung zwischen Gängen und Räumen hinsichtlich der verwendeten Assets zu sehen. Auch in den Räumen wurden die zwei verschiedenen Assets für den Boden verwendet, wobei das eine in etwa zehnmal so häufig auftritt.

4 Experimente und Auswertung

Im folgenden Abschnitt soll das vorgestellte Verfahren ausgewertet und diskutiert werden, ob es die unter Abschnitt 3.1 gestellten Anforderungen erfüllt. Dabei sollen auch die Auswirkungen von einzelnen Parametern auf die Generation untersucht werden. Um eine nachvollziehbare Testumgebung zu schaffen und die Ergebnisse besser vergleichen zu können, werden für die folgenden Tests Standardparameter festgelegt, die in Abbildung 4.1 zu sehen sind. Bei der Untersuchung eines einzelnen Parameters wird lediglich dieser angepasst, ohne die restlichen Werte zu verändern.

4.1 Praktische Laufzeitanalyse

Eine der wichtigsten Anforderungen an das Verfahren besteht in der geringen Laufzeit, die es ermöglichen soll, während eines Spiels Dungeons ohne große Ladezeiten zu generieren. Die Laufzeit hängt dabei vor allem von der Anzahl der zu generierenden Räume ab. Aber auch andere Parameter wie der `compressFactor` oder die Nutzung von größeren Räumen oder Gängen führt zu einer deutlichen Erhöhung der Berechnungszeit, da zur Validierung der Position und zum Eintragen des Raums in die Bitmatrix über dessen volle Größe iteriert werden muss. Beide Faktoren skalieren dabei mit der Anzahl der Räume, da sich die Erhöhung der Berechnungszeit auf jeden einzelnen Raum im Dungeon auswirkt. Um ein aussagekräftiges Ergebnis für unterschiedliche Parameter zu erhalten, wird die durchschnittliche Laufzeit des Algorithmus ermittelt. Dazu wird dieser jeweils 1000 Mal ausgeführt und der Mittelwert der einzelnen Ausführungszeiten gebildet.

Der Test wird dabei für unterschiedliche Dungeon-Größen ausgeführt und untersucht die Auswirkungen des `compressFactors` und der Raumgröße auf die Laufzeit. Als Vergleichswert wird zusätzlich die Laufzeit des Algorithmus mit den in Abbildung 4.1 gezeigten Standardparametern ermittelt. Zusätzlich zur Veränderung der untersuchten Parameter wird mit steigender Raumanzahl auch die `dungeonSize` vergrößert, da sich dieser Faktor nicht messbar auf die Laufzeit auswirkt und bei einer sehr großen Anzahl an Räumen sonst nicht genügend Platz für die volle Anzahl vorhanden wäre.

Die Ergebnisse zeigen, dass besonders die Raumgröße sehr großen Einfluss auf die Laufzeit des Verfahrens hat. Dies liegt an der Implementierung der Validierung der Raumposition. Da jede Grid-Position des Raums auf Überschneidungen überprüft wird, skaliert die Laufzeit mit steigender Raumgröße sehr stark. Die Auswirkung des `compressFactors` hält sich hingegen in Grenzen. Erst bei höheren Raumzahlen macht sich der Unterschied deutlicher bemerkbar. Dies

4 Experimente und Auswertung

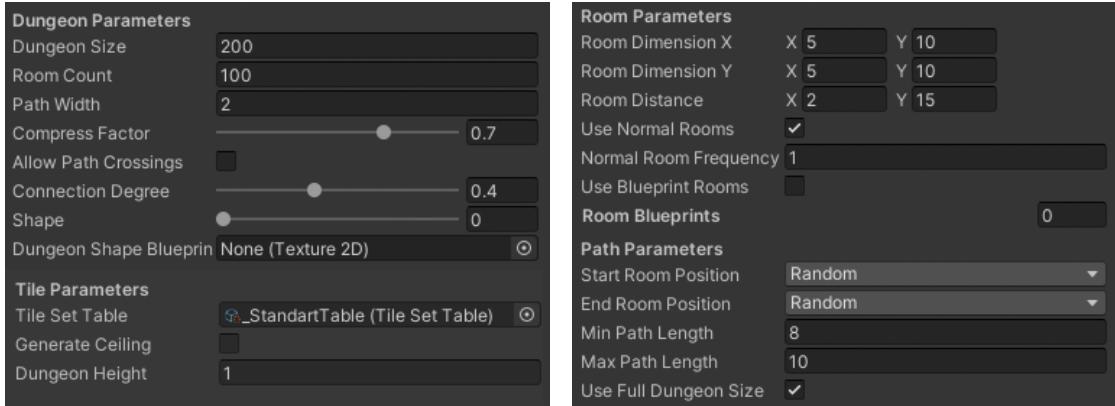


Abbildung 4.1: Zu sehen sind die wichtigsten Parameter des Generators mit ihren Basiswerten. Diese wurden im Hinblick auf ein durchschnittliches Dungeon gewählt, ohne dabei spezielle Features wie Blueprints zu verwenden. Die Parameter `compressFactor` und `connectionDegree` wurden so gewählt, dass die entstehenden Dungeons eine interessante und abwechslungsreiche Struktur aufweisen.

liegt daran, dass eine Veränderung des Wertes sowohl negative als auch positive Auswirkungen auf die Laufzeit hat. Zum einen führt eine Erhöhung des Wertes dazu, dass ein Raum mit mehreren Versuchen generiert wird, was sich negativ auf die Laufzeit auswirkt. Zum anderen verringert sich jedoch die Anzahl der zu überprüfenden Grid-Positionen pro Raum, da die Abstände zwischen den Räumen deutlich kleiner sein dürfen. Jedoch überwiegt der Einfluss des zweiten Faktors, was zu einer Reduktion der Laufzeit bei Erhöhung des `compressFactors` führt.

Bei der Generation von sehr großen Dungeons mit einer hohen Anzahl an Räumen in Kombination mit weiteren Parametern die eine erhöhte Laufzeit herbeiführen steigt die Generationsdauer stark an. Allerdings stellen Dungeons dieser Größenordnung eher eine Ausnahme dar, da auch aus Sicht des Level-Designs eine solche Größe für den Spieler zu unübersichtlich wird. Sollte der Algorithmus dennoch zur Generation von Dungeons im Bereich von 500+ Räumen genutzt werden, so ließe sich die Generationszeit mit einem kurzen Ladebildschirm überbrücken. Eine andere Möglichkeit bestände darin, die Generation nicht auf einen Schlag durchzuführen, sondern auf einen größeren Zeitraum zu verteilen. Jeder rekursive Aufruf der Generation könnte dabei im Update-Loop der Spiel-Engine geschehen, sodass jeden Frame ein kleiner Teil berechnet wird. Somit verteilt sich die Last der Berechnung auf einen größeren Zeitraum, wodurch diese im Hintergrund ablaufen könnte, ohne dass der Spieler davon etwas mitbekommt. Somit ließen sich auch Generationszeiten von mehreren Sekunden ins Spielgeschehen einbinden, ohne dass diese den Spielfluss stören. Auch ist anzumerken, dass die Implementierung in einigen Aspekten durch Nutzung von effektiverem Speichermanagement oder der Optimierung von iterativen Loop-Prozessen noch deutlich effizienter gestaltet werden könnte, wodurch sich die Laufzeit noch einmal verringern ließe.

Die Analyse zeigt, dass das unter Abschnitt 3.1 gestellte Kriterium der Laufzeit für Dungeons bis zu einer Größe von 500 Räumen vollständig erfüllt wird. Wird die Laufzeit des Algorithmus mit der Laufzeit des vorgestellten genetischen Verfahrens verglichen, so können qualitativ ähnliche Dungeons in deutlich geringerer Laufzeit generiert werden. Erst sehr große Dungeons oder die Nutzung von sehr großen Räumen bei der Generation führt zu einer deutlichen Erhöhung der

Raumanzahl	Standardparameter	CF 0	CF 1	doppelte Raumgröße
50	74, 625ms	78, 559ms	72, 221ms	211, 863ms
100	154, 999ms	172, 331ms	152, 28ms	434, 754ms
200	328, 743ms	362, 651ms	310, 510ms	910, 571ms
500	874, 129ms	957, 229ms	853, 375ms	2465, 604ms
1000	1901, 883ms	2074, 54ms	1867, 1ms	5664, 327ms

Tabelle 4.1: Die Tabelle zeigt die praktisch ermittelten Werte für die Laufzeit unter Nutzung verschiedener Parameter. Dafür wurde die Generation jeweils 1000 Mal durchgeführt und ein Mittelwert aus allen Laufzeiten gebildet. Der Parameter `compressFactor` wird dabei durch CF abgekürzt. Während die Raumgröße bei der Nutzung der Standardparameter 5 – 10 Tiles beträgt, wird bei doppelter Raumgröße dieser Wert auf 10 – 20 Tiles erhöht.

Laufzeit und einer Überschreitung des gesetzten Wertes.

4.2 Levelanalyse

Um die durch den Algorithmus entstehenden Dungeons besser miteinander vergleichen und einordnen zu können, müssen Metriken zur Bewertung der Struktur gefunden werden. Das Ziel eines Spielers innerhalb eines Dungeon-Levels besteht vor allem darin, den Weg durch dieses zum Ausgang zu finden. Aus diesem Grund soll der Fokus bei der Auswertung vor allem auf Metriken liegen, mit deren Hilfe sich die Schwierigkeit dieser Aufgabe für jedes Dungeon abschätzen lässt. Dabei sollen Maße betrachtet werden, die sich erst durch die Generation ergeben und sich nicht direkt durch konkrete Parameter einstellen lassen. So führt die Auswahl eines größeren `roomCounts` und einer höheren `pathLength` potenziell auch zu einem komplexerem Dungeon. Diese Werte lassen sich jedoch direkt beeinflussen und können somit einfach durch den Game-Designer eingestellt werden.

Ein Maß, das starken Einfluss auf die Komplexität des Dungeons hat, ohne dabei direkt über einen konkreten Wert eines Parameters gesteuert zu werden, ist die Anzahl und Tiefe der Sackgassen im Dungeon. Diese machen es für den Spieler schwieriger den Ausgang zu finden und geben damit ein gutes Kriterium, nach welchem sich verschiedene generierte Dungeons miteinander vergleichen lassen. Eine Sackgasse führt dazu, dass sich der Spieler neu orientieren und einen anderen Weg suchen muss. Je tiefer dabei die Sackgasse ist, also um so mehr Räume ohne weitere Abzweigungen in einer Reihe angeordnet sind, um so weiter muss der Spieler einen bereits erkundeten Weg zurück laufen, um eine neue Richtung einzuschlagen. Als Sackgasse wird im folgenden jeder Knoten des Graphen betrachtet, der das Kriterium $d(v) = 1$ erfüllt und bei dem es sich nicht um den Start- oder Endraum handelt. Zusätzlich wird jeder Knoten als Sackgasse gewertet, dessen Grad dem Faktor 2 entspricht, wobei einer der Nachbarknoten eine Sackgasse darstellt.

Bei dieser Betrachtung wird jedoch vorausgesetzt, dass der Spieler eine Möglichkeit zur Orientierung innerhalb des Dungeons hat. Die meisten Spiele mit prozedural generierten Dungeons

4 Experimente und Auswertung

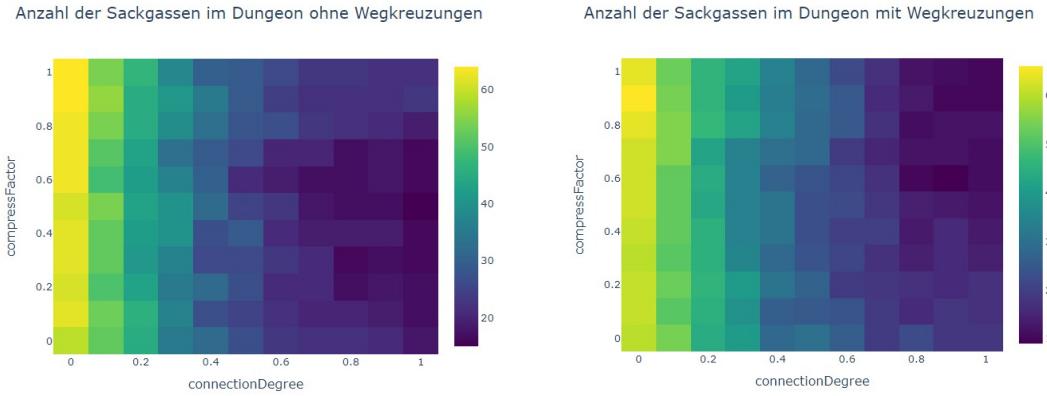


Abbildung 4.2: In den zwei Diagrammen ist die Anzahl der Sackgassen im Dungeon-Komplex in Abhängigkeit von `connectionDegree` und `compressFactor` zu sehen. Im Diagramm auf der rechten Seite wurden durch den Parameter `allowPathCrossings` die Generation von Wegkreuzungen erlaubt. Alle anderen Parameter entsprechen den in Abbildung 4.1 gezeigten Standardwerten. Helle Farben stehen dabei für eine hohe Anzahl von Sackgassen und damit potenziell schwierigere Dungeons.

verfügen über eine Karte, auf der bereits erkundete Räume und Gänge aufgedeckt werden. Dadurch ist es deutlich einfacher sich in einem stark vernetzten Dungeon zurechtzufinden, in welchem sonst leicht die Orientierung verloren gehen kann. Somit werden stark vernetzte Dungeons, die weniger Sackgassen besitzen im folgenden als einfacher betrachtet und die Auswertung der Ergebnisse geschieht unter diesem Gesichtspunkt.

Auch wenn sich dieses Kriterium nicht direkt über Parameter mit genauen Werten steuern lässt, haben einige Einstellungen einen deutlich größeren Einfluss auf das Ergebnis als andere. Den größten Einfluss weisen dabei der `compressFactor`, der `connectionDegree` und die Einstellung `allowPathCrossing` auf. In den Diagrammen in Abbildung 4.2 ist deutlich die Auswirkung des Parameter `connectionDegree` auf die im Dungeon vorhandene Anzahl an Sackgassen zu erkennen. Bei einem `roomCount` von 100 und der Wahl des Parameters `connectionDegree` mit 0, sind mehr als 60 % aller Räume Sackgassen. Wird der Parameter hingegen auf 1 gesetzt, sind weniger als 20 % der Räume Sackgassen. Die Auswirkungen des `compressFactors` fallen deutlich geringer aus, führen aber gerade bei Nutzung eines größeren `connectionDegrees` zu stärkeren Abstufungen in der Anzahl der Sackgassen. Interessant dabei ist, dass bei der Generation ohne Wegkreuzungen, wie im Diagramm links zu sehen ist, ein mittlerer `compressFactor` zu der geringsten Anzahl an Sackgassen führt. Bei weiterer Erhöhung über den Wert 0.7 steigt die Anzahl wieder an.

Generell kann also festgestellt werden, dass ein mittlerer bis hoher `compressFactor` in Kombination mit einem hohen `connectionDegree` zu deutlich weniger Sackgassen führt und damit auch zu einer potenziell leichteren Dungeon-Struktur.

Auch wenn diese Metriken keine perfekte Einschätzung der Schwierigkeit des Dungeons ermöglichen, geben sie einen guten Einblick, wie sich die Parameter auf die Struktur und damit auch auf das Erlebnis für den Spieler auswirken. Da das Finden des Weges zum Ausgang weniger eine Sache des Könnens als vielmehr eine Sache des Glücks darstellt, ist ein allgemeingültiges Maß der Schwierigkeit nur schwer zu finden. So kann ein strukturell potenziell schwieriges Dungeon

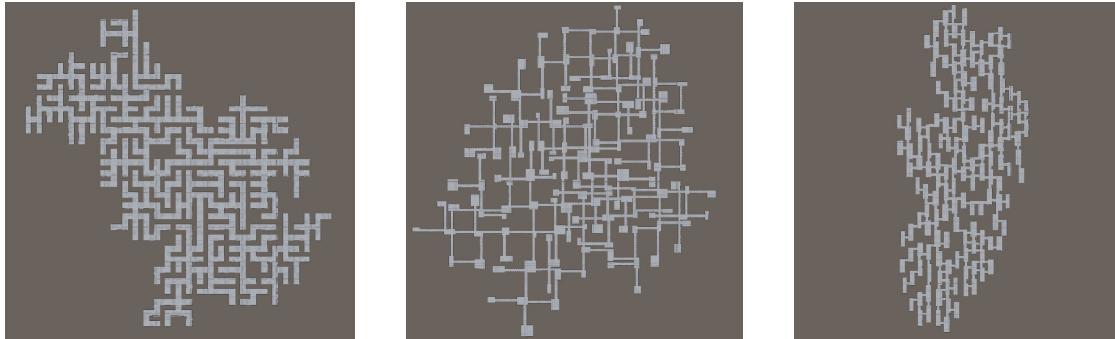


Abbildung 4.3: Die Bilder zeigen, wie mithilfe der zur Verfügung stehenden Parameter die Struktur des Dungeons beeinflusst werden kann und welche Bandbreite an Variationen dadurch entsteht. Für die Beispiele wurden nur die Wert-basierten Parameter und keine Blueprints verwendet, welche noch ein größeres Spektrum an Möglichkeiten bieten. Links: Durch die Wahl der Raumgröße identisch zur Wegbreite ergibt sich ein Dungeon, bestehend nur aus Wegen, das einem Irrgarten gleicht. Mitte: Für dieses Dungeon wurden Wegkreuzungen zugelassen und ein Raumabstand von 20–30 Tiles gewählt. Rechts: Hier wurden sehr kurze Raumabstände gewählt und die Raumgröße ist in vertikale Richtung deutlich größer. Zusätzlich wurde die Fläche zu den Seiten durch den Shape-Parameter begrenzt, wodurch die längliche Form entsteht.

durch zufällig richtige Entscheidungen sehr einfach werden. Ein System, welches eine auf den Fähigkeiten des Spielers basierende Einschätzung der Schwierigkeit ermöglicht, zeichnet sich dadurch aus, dass der Spieler durch das Training der Spielmechaniken in diesen besser wird und somit auch Inhalte mit höherem Schwierigkeitsgrad einfacher lösen kann. Dies trifft auf das Finden des richtigen Weges im Dungeon nicht zu, da zufällig richtige oder falsche Entscheidungen über den Erfolg entscheiden und somit keine konstante Verbesserung der Fähigkeiten möglich ist.

Jedoch zeigen die Ergebnisse, dass eine große Bandbreite an unterschiedlichen generierbaren Strukturen vorhanden ist und sich somit die Komplexität des Dungeons je nach Einsatzgebiet individuell einstellen lässt. Generell kann herausgestellt werden, dass sich die Struktur sehr einfach durch verschiedene Parameter und andere Methoden modifizieren lässt und eine große Bandbreite an verschiedenen Strukturen erzeugt werden können, wie eine Auswahl an verschiedenen Strukturen in Abbildung 4.3 zeigt. Somit lässt sich auch das Anforderungskriterium der bestmöglichen Kontrolle des Verfahrens durch den Game-Designer mit möglichst einfachen Mitteln als erfüllt einstufen.

4.3 Einordnung in die Taxonomie

In diesem Abschnitt soll das vorgestellte Verfahren in die unter Abschnitt 2.3 eingeführte Taxonomie, unter Berücksichtigung der zuvor durchgeführten Tests, eingeordnet werden.

Online Generation

Das vorgestellte Verfahren lässt sich in die online Kategorie einordnen, da die gemessene Laufzeit, wie unter Abschnitt 4.1 herausgestellt, relativ gering ist und sich damit für den Einsatz in

4 Experimente und Auswertung

Spielen während der Ausführung eignet. Selbst bei sehr großen Dungeons mit mehr als 500 oder sogar 1000 Räumen hält sich die Berechnungszeit im Verhältnis zur Größe im Rahmen und kann mit kurzen Ladezeiten überbrückt werden. Potenziell ließe sich das Verfahren aber auch zur offline Nutzung einsetzen, um Strukturen bereits im Voraus zu generieren und diese als fertiges Element ins Spiel zu integrieren.

Verpflichtender Inhalt

Unter diesem Punkt erfolgt eine Einordnung in den Bereich des verpflichtenden Contents, da ein Dungeon einen Level darstellt, der vom Spieler durchschritten werden muss, um im Spiel voranzukommen. Als kritischer Punkt gilt dabei vor allem, dass ein generiertes Dungeon immer über einen Ein- und Ausgang verfügen muss, welche über Gänge und Räume miteinander verbunden sind. Für den Spieler muss es also möglich sein, den Ausgang des Dungeons zu erreichen. Dieses Kriterium wird durch den Algorithmus immer erreicht. Die Erfüllung der Parameter, wie die geforderte Raumanzahl, sind für diesen Punkt nicht essenziell, da sie keine Beeinträchtigung hinsichtlich der Machbarkeit des Dungeons darstellen.

Verwendung von Parametern

Als Konfigurationsmöglichkeit nutzt das Verfahren Parameter, um dem Game-Designer eine präzise Möglichkeit der Kontrolle zu geben. Diese müssen jedoch sinnvoll gewählt werden, da nicht jede Kombination von Werten zu einem sinnvollen Ergebnis bei der Generation führt. Darüber hinaus können komplexere Strukturen wie Blueprints zur Kontrolle genutzt werden.

Generischer Ansatz

Da das Verfahren nur Parameter einbezieht, die zuvor durch den Game-Designer festgelegt wurden, und nicht durch den Spieler zur Laufzeit beeinflusst werden kann, wird es als generisch eingestuft.

Stochastisches Ergebnis

Das Verfahren arbeitet stochastisch und erzeugt somit bei jedem Generationsprozess, auch bei identischer Parameterwahl, ein anderes Ergebnis. Dadurch kann eine große Variation an verschiedenen Strukturen erzeugt werden. Jedoch wird gewährleistet, dass zwei mit gleichen Parametern generierte Dungeons in etwa dieselben Eigenschaften aufweisen. Dadurch kann sichergestellt werden, dass das Ergebnis innerhalb des Erwartungsbereichs liegt und keine zu großen Abweichungen entstehen.

Konstruktiver Ansatz

Wie bereits herausgestellt, verfolgt das Verfahren einen konstruktiven Ansatz. Das Dungeon wird also nur einmal erzeugt, wobei eine bestmögliche Erfüllung der gegebenen Parameter angestrebt wird. Somit kann eine möglichst geringe Berechnungszeit gewährleistet werden, die für den Einsatz in Spielen zur Laufzeit essenziell ist.

Semi-automatisierte Generation

Das Verfahren verfolgt einen semi-automatisierten Ansatz und gibt dem Game-Designer verschiedene Möglichkeiten das Resultat zu konfigurieren. Neben den zur Verfügung stehenden Parametern existieren noch die unter Abschnitt 3.5 vorgestellten Mechanismen, um einen gezielten Eingriff in das Verfahren zu ermöglichen.

5 Zusammenfassung und Ausblick

Die prozedurale Generation von Dungeons und ähnliches Strukturen stellt ein komplexes Feld der PCG dar und wird sich in den kommenden Jahren stetig weiterentwickeln. Die grundlegend verschiedenen Anforderungen, die je nach Spiel an einen Dungeonkomplex gestellt werden, lassen sich nur schwer mithilfe eines einzigen Algorithmus in vollem Umfang abdecken. Das vorgestellte Verfahren hat jedoch den Anspruch, diese Aufgabe bestmöglich zu erfüllen. Dem Game-Designer wird dafür eine Vielzahl an Möglichkeiten an die Hand gegeben, mit deren Hilfe sich das Aussehen und die verschiedenen Eigenschaften manuell beeinflussen lassen, ohne dabei die Vorteile der automatischen Generation zu verlieren. Besonders die Nutzung von Blueprints, aber auch die Vielzahl der zur Verfügung stehenden Parameter, ermöglichen es direkten Einfluss auf das Ergebnis nehmen zu können. Somit kann eine große Bandbreite an verschiedenen Strukturen mit unterschiedlichen Eigenschaften generiert werden.

Das vorgestellte Verfahren ließe sich in verschiedenen Gebieten einsetzen. Der Algorithmus kann ohne Probleme in ein vollwertiges Spiel integriert werden, da alle Funktionalitäten gekapselt in einer eigenen Klasse vorliegen und die Laufzeit hinsichtlich kleiner bis mittelgroßer Dungeons relativ gering ausfällt. Die Parameter können durch den Game-Designer im Vorfeld festgelegt oder durch einen Automatismus in zuvor festgelegten Schranken zufällig gewählt werden. Durch die große Bandbreite an verschiedenen Strukturen kann damit eine abwechslungsreiche Auswahl an verschiedenen Maps generiert werden. Zudem könnte das Verfahren auch in einen Spiele-Editor integriert werden, um in diesem Dungeons offline zu generieren und diese anschließend in einem Spiel zu verwenden. In diesem Fall stellen auch große Dungeons mit mehr als tausend Räumen keine Schwierigkeit dar, da die offline Generation keine zeitliche Begrenzung aufweist.

Im Hinblick auf mögliche Verbesserungen liegt der Schwerpunkt vor allem auf der Optimierung der Laufzeit, da die Implementierung an vielen Stellen noch deutlich effizienter gestaltet werden könnte. Trotz dessen befindet sich die aktuelle Laufzeit für kleinere bis mittelgroße Dungeons bereits im Rahmen der Anforderungen. Nur für sehr große Strukturen mit sehr großen Räumen wird die geforderte maximale Laufzeit leicht überschritten. Dies ließe sich jedoch einfach durch einige Optimierungen im Quellcode beheben.

Zusammenfassend kann festgestellt werden, dass der Algorithmus die im Voraus gestellten Anforderung erfüllt und somit ein voll funktionsfähiges Verfahren zur Generation von komplexen und interessanten Dungeon-Strukturen darstellt. Er bietet die Effizienz von konstruktiven Verfahren und ermöglicht dennoch ein präzises Eingreifen in die Struktur des Ergebnisses.

Literaturverzeichnis

- [1] Daniel Ashlock. *Evolutionary Computation for Modeling and Optimization*. Springer-Verlag GmbH, 1 edition, 2006. pp. 17-18.
- [2] Thomas H. (Dartmouth College) Cormen, Charles E. (MIT) Leiserson, Ronald L. (MIT) Rivest, and Clifford (Columbia University) Stein. *Introduction to Algorithms*. MIT Press Ltd, 3 edition, 2009.
- [3] Reinhard Diestel. *Graphentheorie*. Springer, Berlin Heidelberg, 5 edition, 2010.
- [4] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [5] Michael Cerny Green, Ahmed Khalifa, Athoug Alsoughayer, Divyesh Surana, Antonios Liapis, and Julian Togelius. Two-step constructive approaches for dungeon generation. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*. ACM, 2019.
- [6] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCGames '10*. ACM Press, 2010.
- [7] Gerard Kempen and Edward Hoenkamp. An incremental procedural grammar for sentence formulation. *Cognitive Science*, 11(2):201–258, 1987.
- [8] Antonios Liapis. Multi-segment evolution of dungeon game levels. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017. pp. 203–210.
- [9] David Plans and Davide Morelli. Experience-driven procedural music generation for games. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(3):192–198, 2012.
- [10] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games*. Springer International Publishing, 2016.
- [11] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Transactions on Graphics*, 30(2):1–14, 2011.
- [12] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. What is procedural content generation? In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games - PCGames '11*. ACM Press, 2011.
- [13] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation. In *Applications of Evolutionary Computation*, pages 141–150. Springer, Berlin Heidelberg, 2010.

Literaturverzeichnis

- [14] Valtchan Valtchanov and Joseph Alexander Brown. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. ACM Press, 2012.
- [15] Roland van der Linden. Designing procedurally generated levels. Master's thesis, Delft University of Technology, 2013.
- [16] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2014.
- [17] Breno M. F. Viana and Selan R. dos Santos. A survey of procedural dungeon generation. In *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 2019.

Link zu Projekt und Quellcode auf GitHub

https://github.com/NiclasMart/BA_Project_PDG

Selbstständigkeitserklärung

Ich versichere, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum: _____ Unterschrift: _____