

Hochschule für Technik und Wirtschaft Berlin
Fachbereich 4

Entwicklung eines dreidimensionalen zyklischen Dungeon-Generators mit Unity3D

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.), Internationaler Studiengang Medieninformatik

Nils Gawlik

Erstprüfer Prof. Dr. Tobias Lenz
Zweitprüfer Prof. Dr.-Ing. David Strippgen

Abgabetermin: 4.2.2019

Inhaltsverzeichnis

Abbildungsverzeichnis	IV
1 Einführung und Definitionen	1
1.1 Motivation	1
1.2 Genaue Beschreibung der Fragestellung	2
1.2.1 Definition von Dungeon	2
1.2.2 Definition von zyklisch	3
1.2.3 Dreidimensionalität	4
1.3 Die Unity Game Engine	4
2 Der Ersetzungs-Algorithmus	5
2.1 Formale Grammatiken und Graphgrammatiken für die Generierung	5
2.2 Spezieller Algorithmus	6
2.3 Andere Gitterformen	10
3 Implementierung	13
3.1 Ziele der Implementierung	14
3.2 Wichtige Unity-Features	14
3.3 Ordnerstruktur	15
3.4 Aufbau des Generator-Objekts	16
3.4.1 PatternView	16
3.4.2 PatternToPrefabs	17
3.4.3 RuleSet	18
3.4.4 ReplacementEngine	19
3.4.5 RandomReplacer und RecipeReplacer	19
3.4.6 RuleEditor	20

Inhaltsverzeichnis

3.5 Wichtige Klassen	22
3.5.1 Tile	22
3.5.2 Pattern	22
3.5.3 Rule	23
3.5.4 Utils	23
3.6 Optimierungen des Pattern Matchings	24
3.7 Editor UI	24
3.8 Rezept-Datei	28
3.9 Spielmechaniken	30
3.10 Raum-Modelle	31
3.11 Regelsystem für einen zyklischen Dungeon-Generator	32
4 Auswertung	35
4.1 Auswertung der generierten Dungeons	35
4.2 Mögliche Verbesserungen des Systems, Ausblick	37
4.2.1 Ausbauen des Regel- und Rezept-Systems	37
4.2.2 Verbessern der Räume	37
4.2.3 Modellierung der Muster	38
4.2.4 Eingabe der Regeln	38
5 Fazit	39
6 Verwendete Materialien	40
Literaturverzeichnis	41
Bildquellenverzeichnis	42

Abbildungsverzeichnis

1.1	Dungeon-Beispiel	2
1.2	Dungeon-Beispiel mit Graph	3
2.1	Beispiel für Graph im Gitter	7
2.2	Äquivalenz von Knoten und Raum	7
2.3	Generator mit prozeduralen Räumen	8
2.4	Beispiel einer Produktionsregel	9
2.5	Beispiel einer Regelanwendung	9
2.6	Beispiel einer Produktionsregel aus Levelteilen	10
2.7	Raum-Rotationen mit Symbolen	10
3.1	Ansicht des Unity-Editors	13
3.2	Verschiedene generierte Dungeons	14
3.3	Screenshot der Generator-Components	17
3.4	Dungeon-Teil im Unity Scene View	18
3.5	Inspektor des RuleEditor-Components	20
3.6	Beispiel einer Produktionsregel in Unity	21
3.7	Visualisierung gefundener Matches	25
3.8	Screenshot des RecipeReplacer Components	26
3.9	Beispiele für benutzerdefinierte Gizmos	27
3.10	Dungeonunterteilungs-Beispiel	30
3.11	Raum-Prefabs im Project-Window	31
3.12	Zyklischer Generationprozess 1	32
3.13	Zyklischer Generationprozess 2	33
4.1	Würfelbasierter Generator	36
4.2	Gewölbe-Generator	36

1 Einführung und Definitionen

1.1 Motivation

In einem Blog Post von 2016 beschreibt Joris Dormans einen Ansatz zur prozeduralen Level-Generierung, den er „Cyclic Dungeon Generation“ nennt. Dieses Verfahren präsentiert er als Gegensatz zu den Verzweigungs-Ansätzen die man in vielen anderen Dungeon-Generatoren findet [Dor16].

Zyklische Ansätze haben viele praktische Vorteile gegenüber Verzweigungs-Ansätzen. Zum einen vermeiden sie Sackgassen und damit verbundenes Backtracking und zum anderen ermöglichen sie das Generieren von typisch zyklischen Strukturen, unter anderem: „Einbahnstraßen“ mit anderem Rückweg, alternative Wege, Abkürzungen und direkte Rückkehr zum Startpunkt des Dungeons [Dor16].

Dormans Spiel „Unexplored“, dass im Blog Post als Beispiel verwendet wird, ist ein zweidimensionales Spiel in einer Top-Down-Perspektive. Besonders für zyklische Generierung ist es aber auch interessant einen dreidimensionalen Raum zu nutzen, da dies interessante Strukturen ermöglicht wie Brücken die bereits besuchte Teile des Dungeons überspannen.

Das Ziel dieser Bachelorarbeit ist es ein Verfahren zur zyklischen Generierung von Dungeons zu konzipieren und zu implementieren. Dieses Verfahren sollte im dreidimensionalen Raum funktionieren und die dritte Dimension für interessante Strukturen nutzen können. Die Implementierung findet in der Game Engine Unity statt.

1.2 Genaue Beschreibung der Fragestellung

Der Inhalt der Arbeit ist die Entwicklung eines dreidimensionalen zyklischen Dungeon-Generators mit Unity3D.

Im Folgenden soll erläutert werden, was unter Dungeons zu verstehen ist und wie die Einschränkungen „dreidimensional“ und „zyklisch“ im Kontext dieser Arbeit zu verstehen sind.

1.2.1 Definition von Dungeon

Der englische Begriff „Dungeon“ wurde für diese Arbeit bewusst gewählt, er hat im Bereich Spiele eine Bedeutung, die sich von der wörtlichen Übersetzung „Verlies, Kerker“ unterscheidet. Eine einheitlich etablierte Definition für Dungeon in diesem Kontext existiert nicht. In [LLB14] wird wie folgt definiert: „We define adventure and RPG dungeon levels as labyrinthic environments, consisting mostly of inter-related challenges, rewards and puzzles, tightly paced in time and space to offer highly structured gameplay progressions.“. Diese Definition wird auch für diese Arbeit verwendet.

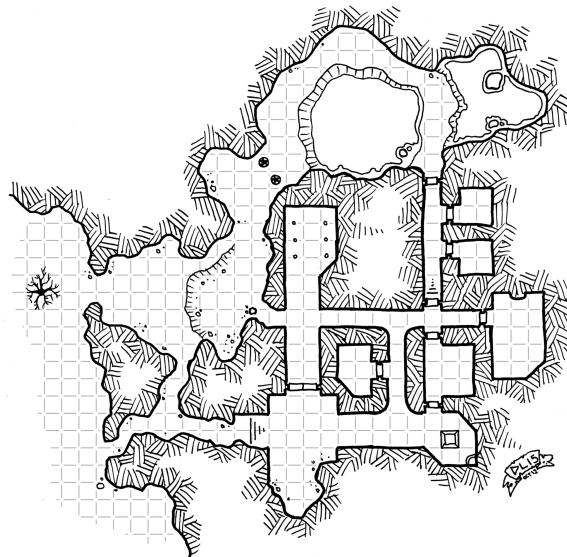


Abbildung 1.1: Beispiel eines Dungeons [Log15]

1.2.2 Definition von zyklisch

Der Begriff zyklisch ist als deutsches Äquivalent des englischen Begriffs „cyclic“ zu verstehen und ist aus dem bereits erwähnten Blog Post von Joris Dormans [Dor16] übernommen.

Für diese Arbeit wird folgende Definition verwendet: Ein Dungeon-Generator ist zyklisch wenn A) der generierte Dungeon Zyklen enthält und B) Zyklen als Bausteine während der Generierung dienen.

Im folgenden wird der Begriff Dungeon-Graph verwendet. Der Dungeon-Graph ist eine Graph-Repräsentation des Dungeons. Die Knoten des Graphen repräsentieren Räume, die Kanten repräsentieren Übergänge zwischen Räumen.

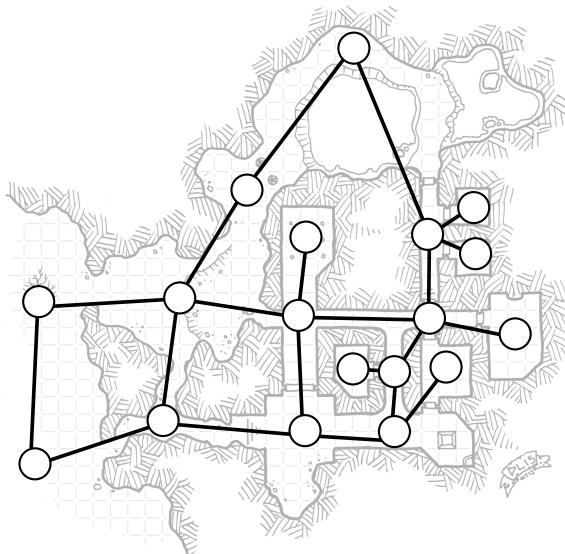


Abbildung 1.2: Bild eines Dungeons, mit möglichem Dungeon-Graph, der Graph ist ungewichtet und ungerichtet ([Log15], bearbeitet)

Ist der Dungeon-Graph zyklisch, so ist auch der Dungeon zyklisch. Betrachtet man das Beispiel in Abbildung 1.2 kann man mehrere Zyklen erkennen. Praktisch gesehen bedeutet das, dass der Spieler im Kreis laufen kann.

Die andere Bedingung, dass Zyklen während der Generierung als Bausteine des Dungeons dienen, ist wichtig. So wird von einer anderen Art der Generierung unterschieden, wie man sie häufig in Level-Generatoren findet: Man generiert ein Labyrinth ohne Zyklen mit

1 Einführung und Definitionen

einem Verzweigungs-Ansatz und fügt anschließend willkürlich Verbindungen zwischen Räumen hinzu, bzw. – etwas bildlicher – „reißt Wände ein“. So entstehen auch Zyklen, diese sind aber willkürlich und schlecht kontrollierbar. Sind die Zyklen ein Baustein des Dungeons, so dass mehrere vordefinierte Arten von Zyklen durch Aneinanderreihung oder Verschachtelung kombiniert werden, hat man Kontrolle über die Natur der Zyklen. Variablen wie Größe, Richtung und Anordnung der Zyklen sind konfigurierbar.

1.2.3 Dreidimensionalität

Dreidimensionalität bedeutet im Kontext dieser Arbeit, dass alle generierten Strukturen räumlich sind. Ein Spieler kann sich durch den Dungeon sowohl parallel zum Boden als auch vertikal (z.B. über Treppen oder Leitern bewegen). Es soll die Möglichkeit bestehen, dass Strukturen über andere Strukturen hinwegführen.

1.3 Die Unity Game Engine

Unity ist eine Game Engine für die Entwicklung von 2D und 3D-Spielen.

Die Verwendung einer Game Engine ist dadurch begründet, dass der Implementierungs-Teil der Arbeit auf den Generator selbst und das User-Interface des Generators beschränkt sein soll. Die Implementierung von Engine-typischen Features wie Rendering oder Kollision soll nicht Teil der Arbeit sein.

Die Wahl einer Game Engine für ein bestimmtes Projekt ist immer auch eine subjektive Entscheidung. Hauptargumente für Unity sind die weite Verbreitung der Game Engine unter Indie-Entwicklern, das Entity-Component-System, dass einen modularen Aufbau ermöglicht und das Editor Scripting, dass es erlaubt ein User Interface für den Generator innerhalb der Engine zu schreiben.

2 Der Ersetzungs-Algorithmus

Im Folgenden wird der der Arbeit zugrunde liegende Algorithmus theoretisch erklärt.

2.1 Formale Grammatiken und Graphgrammatiken für die Generierung

Dem Algorithmus liegt ein Ersetzungssystem zu Grunde. Dieses ähnelt der Generierung durch eine Graphgrammatik, weswegen kurz auf die Prozedurale Generierung mit Hilfe von Formalen Grammatiken und im speziellen Graphgrammatiken eingegangen wird.

Formale Grammatiken werden in der prozeduralen Generierung gerne verwendet um Wörter und Texte zu generieren oder andere eindimensionale Ketten von Symbolen, wie z.B. Melodien. „Procedural Content Generation In Games“ sagt folgendes:

„Generative grammars were originally developed to formally describe structures in natural language. These structures—phrases, sentences, etc.—are modelled by a finite set of recursive rules that describe how larger-scale structures are built from smaller-scale ones, grounding out in individual words as the terminal symbols.“ [STN16, Kap. 3.5, S. 45]

Damit eine nicht-deterministische Grammatik zur Generierung verwendet werden kann muss eine Entscheidung getroffen werden, in welcher Reihenfolge Regeln angewendet werden. Eine mögliche Methode ist jeden Generations-Schritt eine zufällige Regel aus der Menge an anwendbaren Regeln auszuwählen. [STN16, Kap. 5.2, S. 75] Eine Terminierung des

2 Der Ersetzungs-Algorithmus

Algorithmus ist hier nicht garantiert, da Regeln rekursiv sein können. Aber bei einer zufälligen Auswahl ist eine Terminierung in einer akzeptablen Laufzeit sehr wahrscheinlich, angenommen die Formale Sprache enthält keine rekursiven Regeln ohne Abbruchbedingung, was zu einer unvermeidbaren Endlosschleife führen würde.

Eine Graphgrammatik verhält sich wie eine Formale Grammatik, allerdings sind die linken und rechten Seiten Mustergraphen. Hierbei ist es wichtig, dass die individuellen Knoten auf der linken Seite jeweils individuelle Knoten auf der rechten Seite zugewiesen werden können, damit die Ersetzung stattfinden kann. [STN16, Kap. 5.5.1, S. 80] Durch wiederholte Ersetzung von Teilgraphen kann hier aus einem Start-Graph ein End-Graph erzeugt werden. Die Menge der produzierbaren Graphen bildet die durch die Graphgrammatik definierte Formale Sprache.

In [Dor10] benutzt Dormans Graphgrammatiken um Missionsstrukturen zu generieren und Figur-Grammatiken (engl. Shape Grammars) um den zugehörigen Raum zu generieren.

2.2 Spezieller Algorithmus

Der Generator den ich diese Bachelorarbeit entwickelt habe, ist durch wiederholtes Iterieren und Experimentieren entstanden. Diese Implementierung wird später ausführlich in Kapitel 3 beschrieben.

Wie bereits erwähnt ähnelt das Verfahren einer Graphgrammatik. Es ist inspiriert von Joris Dormans Ansatz in [Dor10], allerdings werden keine separaten Modelle für Missionsstruktur und Raum verwendet. In [Dor11] geht Dormans genauer auf „mission and space“ ein und dass diese beiden Modelle von Level-Designern häufig als isomorphisch angenommen werde. Er argumentiert jedoch, dass sie als separate Modelle verstanden werden sollten. Mission bezeichnet dabei ein Modell der Aufgaben die der Spieler bewältigen muss, Space bezeichnet den Raum in dem diese Aufgaben ausgeführt werden müssen.

Für mein Verfahren wurden keine separaten Modelle für Mission und Space verwendet, sondern direkt ein Raum generiert, der eine Missionsstruktur implizit beinhaltet. Für die Zwecke dieser Arbeit, wurde diese Vereinfachung als „gut genug“ befunden.

2 Der Ersetzungs-Algorithmus

Der Generierung dieses Raumes würde ich mich gerne über das bereits bekannte Konzept der Graphgrammatik annähern. Hierfür stelle man sich einen mit Hilfe einer Graphgrammatik generierten Dungeon-Graph vor. Dieser ist allerdings an ein dreidimensionales kartesisches Gitter gebunden. Verbindungen zwischen Knoten des Graphen sind nur zwischen direkt (nicht diagonal) benachbarten Zellen möglich, wie in Abbildung 2.1 dargestellt.

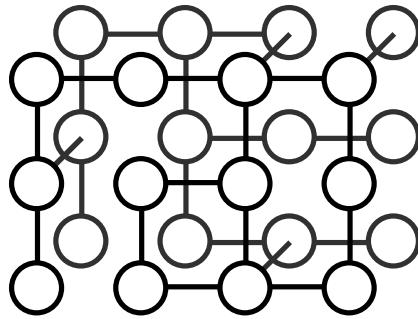


Abbildung 2.1: Beispiel für einen dreidimensionalen Graph innerhalb der Restriktionen des Gitters

Zu Jedem Knoten und den mit ihm verbundenen Kanten kann man sich einen äquivalenten Raum vorstellen. Als Verbindung zwischen den Räumen dient ein Ausgang, die äquivalent zu einer halben Kante ist (siehe Abb. 2.2).

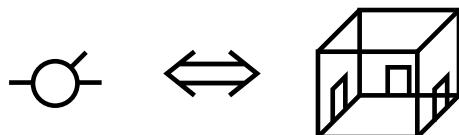


Abbildung 2.2: Knoten mit Halb-Kanten und äquivalenter Raum

Würde man einen solchen Raum, der genau eine Gitterzelle einnimmt für alle Ausrichtungen manuell entwerfen, etwa in einem 3D-Programm, so würden $2^6 = 64$ verschiedene Raum-Vorlagen benötigt, da es sechs Ausgänge pro Raum gibt, die jeweils zwei Zustände („Tür“ oder „keine Tür“) haben können. Durch Nutzen von Rotationssymmetrie könnte man diese Zahl verringern, dennoch ist dies nicht ideal, da ein Level-Designer trotzdem eine Mindestzahl an Räumen entwerfen muss. Außerdem will man für einen abwechslungsreichen Dungeon verschiedene „Themes“ haben, verschiedene Arten von

2 Der Ersetzungs-Algorithmus

Raumübergängen und andere Arten von Variationen. Diese Variationen kann man nicht alle in allen möglichen Raumvariationen bereitstellen.

Alternativ könnte man auch die einzelnen Räume prozedural generieren. Dieser Ansatz wurde ausgetestet, aber nicht weiter verfolgt, da die Räume dadurch zu gleich aussahen, wie man in Abbildung 2.3 sehen kann. Einen abwechslungsreichen Raum-Generator zu entwickeln ist ein interessantes Problem, aber nicht Inhalt dieser Arbeit.

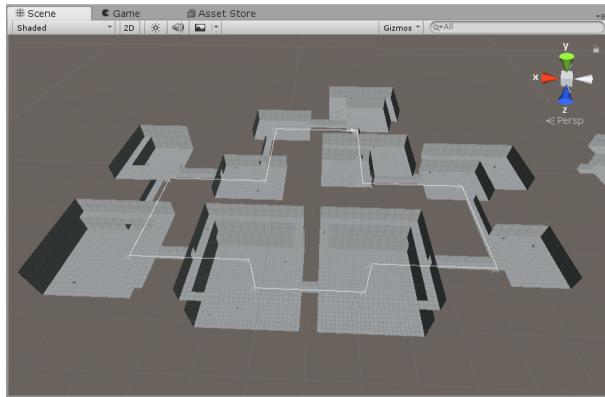


Abbildung 2.3: Screenshot einer frühen Version des Generators mit prozedural generierten Räumen. Die Räume sehen sehr ähnlich aus.

Die Lösung dieses Problems ist die Umformulierung des Modells. Anstatt einen Dungeon-Graphen zu generieren, und diesen in einem zweiten Schritt in Räume zu übersetzen, werden die Räume direkt produziert.

Dies bedeutet, dass keine Graphgrammatik im eigentlichen Sinne verwendet wird sondern stattdessen eine dreidimensionale Array-Grammatik. Ein Eintrag im Array lässt sich über die Indizes einer Gitterzelle zuweisen, dies führt dazu, dass das Gitter räumlich durch die Größe des Arrays begrenzt ist.

Für diesen Algorithmus wird eine Array-Grammatik verwendet, deren Produktionsregeln als linke und rechte Seite dreidimensionale Arrays haben. Die Seiten sind zueinander identisch in Länge, Höhe und Breite, somit kann ein Vorkommen der linken Seite durch die zugehörige rechte Seite ersetzt werden. Im Folgenden wird außerdem nicht zwischen Terminal- und Nichtterminalsymbolen unterschieden. Dies hat den Zweck, dass man die Generierung zu jeder Zeit abbrechen kann, und trotzdem eine korrekte Dungeon-Darstellung hat. Eine Unterscheidung von Terminalen und Nichtterminalen wäre aber einfach vorstellbar.

2 Der Ersetzungs-Algorithmus

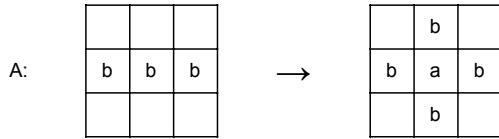


Abbildung 2.4: Beispiel einer Produktionsregel¹

Anders als bei Zeichenketten-basierten Grammatiken wo sich die Länge des Wortes bei der Ersetzung ändern kann, ändert sich die Größe des Arrays dabei nicht, da linke und rechte Seite gleich groß sind, es wird nur das relevante Teilstück ersetzt.

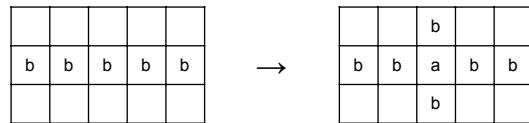


Abbildung 2.5: Anwendung der Produktionsregel A (eine von drei möglichen Positionen der Anwendung)

Um mit dieser Array-Grammatik einen Dungeon zu erstellen müssen wir lediglich statt einem Alphabet aus Buchstaben ein Alphabet aus Raumteilen verwenden. Verwendet man ein solches Alphabet, so kann eine Graphgrammatik implizit in den Arrays enthalten sein, wie in Abbildung 2.6 illustriert. In der Praxis sind diese Raumteile 3D Modelle, die modular zusammenpassen und so kann ein ganzer Dungeon mit Regeln transformiert werden.

Hierbei ist dem Autor/der Autorin der Regeln überlassen, dass die Regeln selbst einen korrekten Dungeon produzieren. Eine Regel, die z.B. eine Verbindung ins Nichts führen lässt, sollte nicht erstellt werden.

Ein großer Vorteil dieser Methode ist, dass das Alphabet aus beliebig vielen oder wenigen Raumteilen bestehen kann, ein Mangel an Raumteilen schränkt lediglich die Menge der

¹ Arrays werden im Folgenden in 2D dargestellt, um die Grafiken übersichtlicher zu gestalten, sämtliche Grafiken gelten im gleichen Maße für dreidimensionale Arrays

2 Der Ersetzungs-Algorithmus

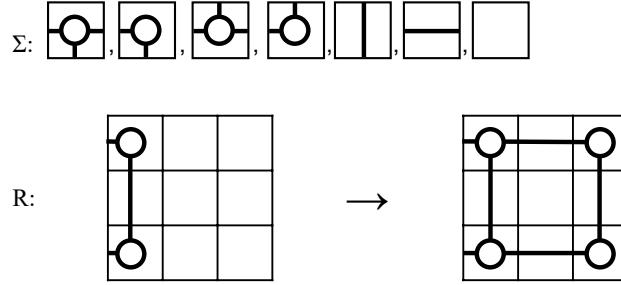


Abbildung 2.6: Alphabet Σ mit Dungeon-Graph-Teilen und einer Produktionsregel R

sinnvollen Regeln ein. Symbole für verschiedene Rotationen des gleichen Modells können automatisch erzeugt werden (siehe Abb. 2.7).

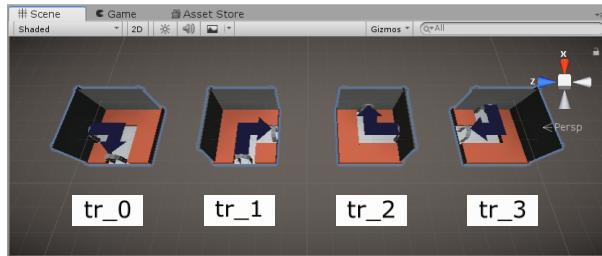


Abbildung 2.7: Das gleiche Raum-Modell in verschiedenen Rotationen, mit jeweils anderem Symbol.

2.3 Andere Gitterformen

In 2.2 wurde der Algorithmus für eine kartesisches Gitter beschrieben. Im folgenden soll ein Überblick gegeben werden wie eine Verallgemeinerung auf andere Gitter möglich wäre.

Um den Algorithmus für alle Gitter verallgemeinern, muss eine allgemeine Definition für ein Muster gefunden werden. Diese Definition kann sowohl für den gesamten Dungeon als auch für die linken und rechten Seiten der Produktionsregeln verwendet werden.

2 Der Ersetzungs-Algorithmus

Definieren wir zuerst die Symbolmenge und ein Null-Symbol. Das Null-Symbol bedeutet, dass die Gitterzelle außerhalb des Musters liegt.

V ist die Symbolmenge

$$V_0 = V \cup \{0\}$$

Nehmen wir die Zellen des Musters M sind über ein n -Tupel ganzer Zahlen p eindeutig adressierbar. Dies kann durch folgende Funktion ausgedrückt werden.

$$M : \mathbb{Z}^n \rightarrow V_0$$

So kann ein zu suchendes Teilstück T des Musters ebenfalls als Funktion gesehen werden.

$$T : \mathbb{Z}^n \rightarrow V_0$$

Ein Teilstück T ist dann ein Match im Muster M an Position $p \in \mathbb{Z}^n$ wenn gilt:

$$\forall x \in \mathbb{Z}^n : M(p + x) = T(x) \vee T(x) = 0$$

Ein Array erfüllt die Definition, nimmt man an, dass alle Elemente außerhalb der Grenzen des Arrays das Null-Symbol sind. Elemente innerhalb des Arrays sind entweder das Null-Symbol oder ein Symbol der Symbolmenge.

Um einen Strukturen zu platzieren, müssen wir der Position im Gitter eine Position im Raum zuweisen. Definieren wir hierzu eine Funktion G .

$$G : \mathbb{Z}^n \rightarrow \mathbb{R}^n$$

für ein dreidimensionales kartesisches Gitter mit der Gitterzellengröße $d \in \mathbb{R}^3$ sähe diese Funktion so aus:

$$G_k : \mathbb{Z}^3 \rightarrow \mathbb{R}^3$$

$$G_k : G_k(x) = \begin{pmatrix} d_1 \cdot x_1 \\ d_2 \cdot x_2 \\ d_3 \cdot x_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ 0 \\ 0 \end{pmatrix} \cdot x_1 + \begin{pmatrix} 0 \\ d_2 \\ 0 \end{pmatrix} \cdot x_2 + \begin{pmatrix} 0 \\ 0 \\ d_3 \end{pmatrix} \cdot x_3$$

Man kann eine Verallgemeinerung vornehmen, wobei das Gitter durch die Vektoren $\vec{d}_1, \dots, \vec{d}_n$ aufgespannt wird. Bei kartesischen Gittern sind diese Vektoren achsenparallel.

2 Der Ersetzungs-Algorithmus

$$G : \mathbb{Z}^n \rightarrow \mathbb{R}^n$$

$$G : G(x) = \sum_{i=1}^n \vec{d}_i \cdot x_i$$

Der in [2.2](#) beschriebene Algorithmus kann also für alle Gitter, die diese Form haben angewandt werden.

3 Implementierung

In diesem Kapitel wird die praktische Umsetzung des in 2.2 beschriebenen Algorithmus behandelt. Die Implementierung beinhaltet viele einzelne Komponenten und Konfigurationsmöglichkeiten. Sie ist objektorientiert programmiert und läuft als Erweiterung der Unity Engine.

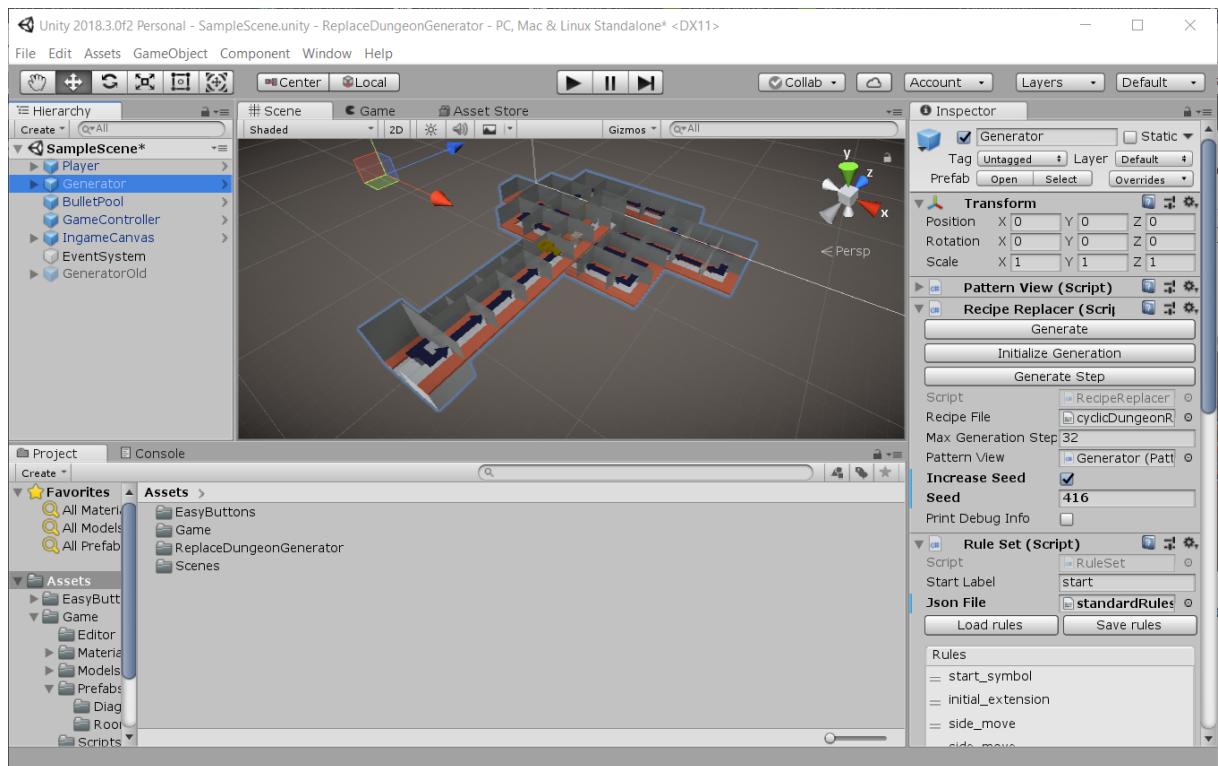


Abbildung 3.1: Ansicht des Unity-Editors in einer typischen Szene

3 Implementierung

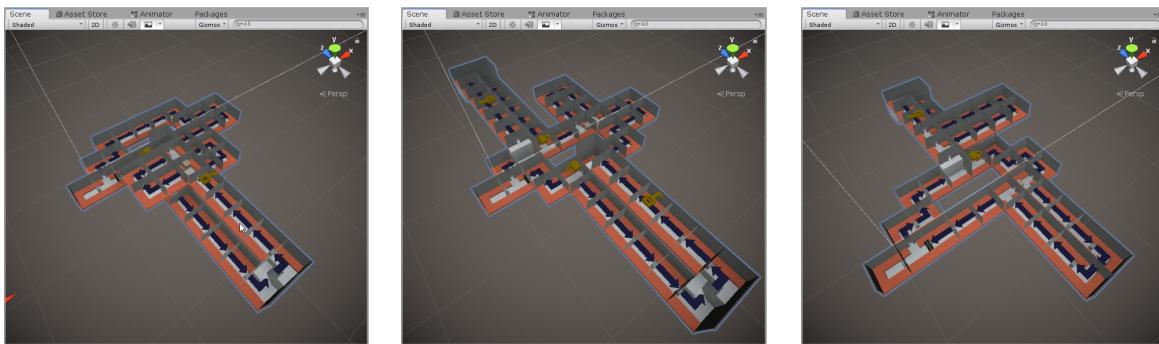


Abbildung 3.2: Verschiedene in Unity generierte Dungeons

3.1 Ziele der Implementierung

Es sind viele verschiedene Anwendungsszenarien für diesen Generator denkbar.

Ein Ziel der Implementierung ist es, dass das Programm nicht einen spezifischen Generator darstellt, sondern ein Werkzeug mit dem sich verschiedene Generatoren der gleichen Klasse konfigurieren lassen. Es sollten möglichst große Teile des Algorithmus nicht hard-coded sondern konfigurierbar sein, indem sie als Daten vorliegen. Insbesondere gilt das für das Alphabet und die Produktionsregeln der Array-Grammatik.

Ein weiteres Ziel ist es die Features der Unity-Engine zu nutzen, insbesondere die Entity-Component-Architektur und die Möglichkeiten den Unity-Editor mit eigener UI zu erweitern.

3.2 Wichtige Unity-Features

Unity verwendet eine Entity-Component-Architektur. So werden an ein GameObject beliebig viele Components angehängt, die verschiedene Funktionen erfüllen und miteinander interagieren können. Dies ermöglicht einen modularen Aufbau von Objekten. [Uni18b, Seite: GameObjects] Neue Components können in Unity programmiert werden, indem man eine Klasse schreibt, die von der Klasse MonoBehaviour erbt. [Uni18b, Seite: CreatingAndUsingScripts]

3 Implementierung

Ein Beispiel hierfür kann man in Abbildung 3.3 sehen, nur durch das Zusammenspiel der einzelnen Components ergibt sich ein Generator. Durch den modularen Aufbau verhindert man zum einen Code Duplication (Es wird z.B. der PatternView auch in anderen Objekten wiederverwendet), zum anderen ist es eine Anpassungsmöglichkeit: So lässt sich nach Belieben ein RandomReplacer oder ein RecipeReplacer verwenden um verschiedenes Verhalten zu erzeugen.

Manche Components ließen sich prinzipiell zusammenfassen, wie z.B. PatternView und PatternToPrefabs, aber wurden trotzdem getrennt implementiert, da sie unterschiedliche Aufgaben erfüllen. Dies macht auch das Ausbauen der Software in der Zukunft einfacher, da einzelne Components ausgetauscht oder umgeschrieben werden können, solange die öffentlichen Funktionen und Variablen dieser Components gleich bleiben.

In Abbildung 3.3 sieht man auch gut ein weiteres wichtiges Unity-Feature, nämlich die Darstellung von öffentlichen Variablen im Inspektor. Ist ein Field public (oder private und mit dem SerializeField-Attribute gekennzeichnet) so kann der Wert dieses Fields im Inspektor gesetzt werden [Uni18b, Seite: VariablesAndTheInspector] [Uni18a, Seite: SerializeField]. Unity unterstützt hierbei die häufigsten Datentypen von Haus aus. Es wurde bei der Implementierung der Components darauf geachtet möglichst viele Einstellungen so im Inspektor editierbar zu machen.

Außerdem wurde das User Interface über selbst geschriebene Editor-UI ausgebaut, mehr hierzu in Abschnitt 3.7.

3.3 Ordnerstruktur

Die Ordnerstruktur des Projektes ist auf der höchsten Ebene in drei Ordner unterteilt: EasyButtons, ReplaceDungeonGenerator und Game. Die Ordner EasyButtons und ReplaceDungeonGenerator haben ihren eigenen Namespaces, die jeweils mit dem Ordnernamen identisch sind.

EasyButtons ist ein Open-Source-Package, dass das einfache Erstellen von Buttons im Inspektor zum Aufrufen von Methoden von MonoBehaviour- und ScriptableObject-Klassen erlaubt. [Mad18]

3 Implementierung

ReplaceDungeonGenerator enthält die Kern-Assets des Generators und kann in jedes Projekt eingefügt werden. Das Package hat lediglich EasyButtons als Dependency. Somit lässt sich der Generator einfach in anderen Projekten einsetzen ohne unnötige Spiel-spezifische Assets wie Modelle, PlayerController, etc. mit zu importieren. Der Ordner enthält eine Beispielszene mit einer simplen Grammatik, die auf Würfeln basiert.

Game enthält alle Spiel-spezifischen Assets, die nicht Teil des Generator-Kerns sind. Dazu gehört unter anderem der Programmcode der Spielmechaniken (siehe [3.9](#)) und die Regeln, die Räume und das Rezept des zyklischen Dungeon-Generators (siehe [3.11](#)).

Innerhalb dieser Ordner wird nach Asset-Typ unterschieden (Scenes, Models, Scripts, ...). Ordner, die Editor heißen, werden nur für den Editor gebraucht und darin enthaltene Assets sind im fertigen Spiel nicht verwendbar [[Uni18b](#), Seite: SpecialFolders]

Weitere Unterordner wurden nach Bedarf erstellt, z.B. um die Modelle von zwei Generationen-Stilen (Rooms und DiagonalRooms) zu unterscheiden.

3.4 Aufbau des Generator-Objekts

Das wichtigste GameObject ist der „Generator“, dieser ist üblicherweise zusammengesetzt aus folgenden Components: Transform (Bestandteil von jedem GameObject), PatternView, RandomReplacer oder RecipeReplacer, RuleSet, RuleEditor (optional), ReplacementEngine, PatternToPrefabs (siehe Abb. [3.3](#)).

Es folgt eine kurze Beschreibung der einzelnen Components. Es wird dabei hauptsächlich auf die Funktion und Bedienung der Components eingegangen, nicht auf Implementierungsdetails.

3.4.1 PatternView

Das PatternView repräsentiert einen dreidimensionalen Array von Labels. Diese Labels sind äquivalent zu den Symbolen der Graphgrammatik, sie können verschiedene Bedeutungen haben, können aber prinzipiell jede Zeichenkette sein. Das PatternView enthält außerdem einen dreidimensionalen Vector „Display Delta“. Dieser legt die Größe einer

3 Implementierung

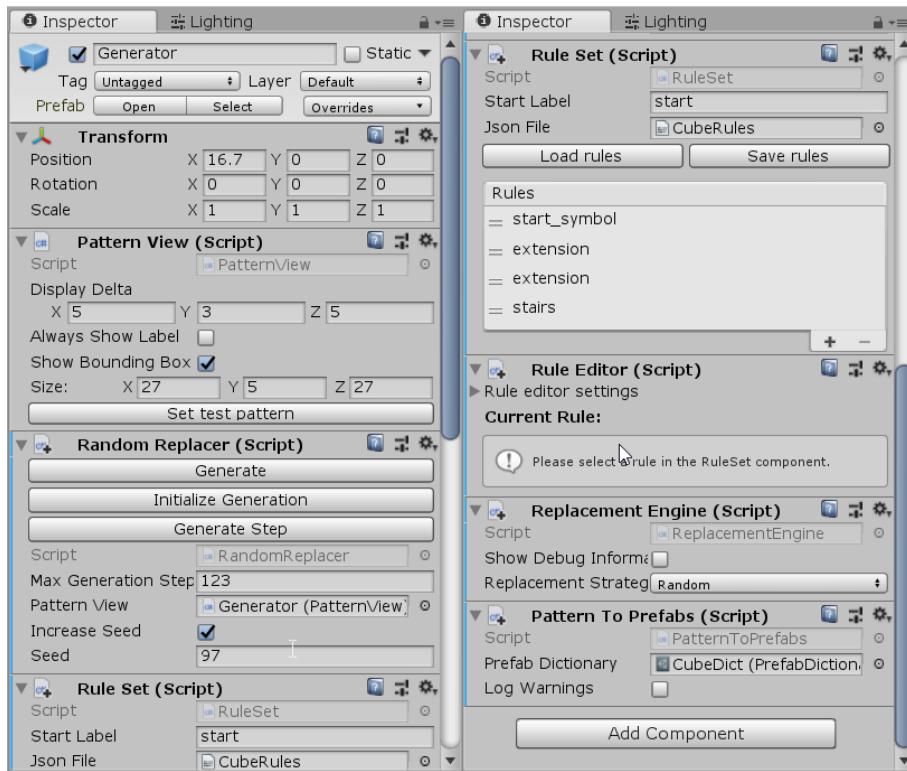


Abbildung 3.3: Screenshot des Inspektors eines Generator-GameObjects, man sieht alle vorhandenen Components und deren Einstellungen

Gitterzelle des dreidimensionalen kartesischen Gitters fest, mit dem die Räume in der Szene angeordnet werden. Im Scene View¹ von Unity werden die Labels als Text im dreidimensionalen Raum angezeigt (erkennbar in Abb. 3.4).

3.4.2 PatternToPrefabs

Der PatternToPrefabs Component ist ein Component, der für jedes Label im PatternView ein oder mehrere Prefabs in der Szene erstellt. Dies geschieht wenn eine Änderung im PatternView stattfindet und einmal bei Spielstart. In einem ScriptableObject² der Klasse PrefabDictionary wird jedem Raumnamen wie „st“ ein Prefab³ zugeordnet.

1 Übersicht der Unity-UI: [[Uni18b](#), Seite: LearningtheInterface]

2 Scriptable Objects in Unity: [[Uni18b](#), Seite: class-ScriptableObject]

3 Prefabs in Unity: [[Uni18b](#), Seite: LearningtheInterface]

3 Implementierung

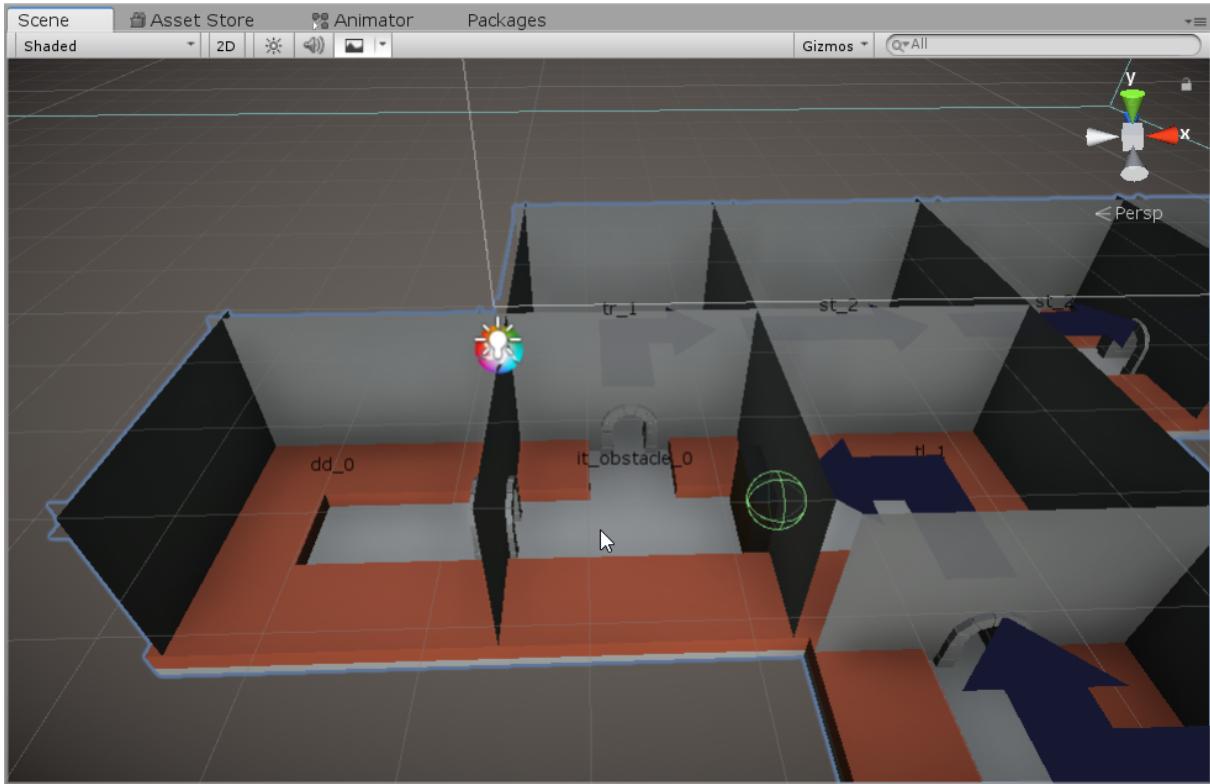


Abbildung 3.4: Sicht auf einen Teil eines generierten Dungeons im Unity Scene View

Es können beliebig viele Raumnamen mit Unterstrichen getrennt in einem Label vorkommen, die zugehörigen Prefabs werden dann übereinander generiert. die Kette kann mit einer Zahl beendet werden, die die Rotation in 90°-Schritten um die y-Achse angibt. Ein Beispiel hierfür ist „it_obstacle_0“ in Abbildung 3.4. Es wird das Kreuzung-Prefab „it“ mit einem Hindernis-Prefab „obstacle“ generiert und um 0 mal 90° gedreht.

Der PatternToPrefabs Component verlangt ein PatternView Component mit RequireComponent⁴.

3.4.3 RuleSet

Das RuleSet ist der Component in dem alle Produktionsregeln enthalten sind. Die Inspektor-UI dieses Components enthält die Möglichkeit Regeln im JSON-Format zu laden und zu speichern. Es lassen sich Regeln entfernen, hinzufügen und auswählen. Ist eine Regel

4 RequireComponent in Unity: [Uni18a, Seite: RequireComponent]

3 Implementierung

ausgewählt, kann sie im RuleEditor-Component editiert werden (vorausgesetzt es ist ein RuleEditor an das GameObject angehängt).

Das RuleSet ist nur im Namen ein Set, nicht im informatischen Sinne. Ein Set wäre ungeordnet, das RuleSet allerdings enthält eine Liste an Regeln. Die Reihenfolge der Regeln in der Liste kann in bestimmten Fällen eine Rolle spielen, etwa wenn in der ReplacementEngine die Replacement Strategy auf „First“ oder „Last“ gesetzt ist.

3.4.4 ReplacementEngine

Die ReplacementEngine ist verantwortlich für die im Hintergrund stattfindende Ersetzung. Sie findet im PatternView Matches für Regeln aus dem RuleSet. Erhält sie von einem anderen Component wie RandomReplacer das Signal, wendet sie eine Produktionsregel an. Dabei kann eingestellt werden ob an einer zufälligen Position, der zuerst gefundenen, oder zuletzt gefundenen Position ersetzt werden soll. Es kann außerdem ein Filter übergeben werden, sodass nur bestimmte Regeln ersetzt werden. Dieser Filter wird vom RecipeReplacer verwendet.

Der PatternToPrefabs Component verlangt die PatternView und RuleSet Components mit RequireComponent.

3.4.5 RandomReplacer und RecipeReplacer

Es gibt zwei verschiedene Replacer: Den RandomReplacer und den RecipeReplacer. Beide Replacer erfüllen die selbe Funktion, sie dienen als Haupt-Schnittstelle des Generators. Von ihnen kann die Generation ausgelöst werden. Dies ist zu Testzwecken im Unity Editor möglich, indem man auf den Generate-Knopf drückt, oder zur Laufzeit indem man die Generate-Methode aufruft. Die Generate-Methode erzeugt einen ganzen Dungeon auf einmal. Alternativ kann man auch einmal die InitializeGeneration-Methode aufrufen und im Anschluss die GenerateStep-Methode um einzelne Ersetzungen Schritt für Schritt vorzunehmen.

Der RandomReplacer ist die simplere Variante der beiden Replacer. Er wählt immer eine zufällige Ersetzung für den nächsten Produktionsschritt. Dabei gewichtet er die

3 Implementierung

einzelnen möglichen Ersetzungen gemäß der Gewichtung der zugehörigen Regel im RuleSet.

Der RecipeReplacer verwendet eine Rezept-Datei des Datentypen TextAsset⁵. In der Text-Datei ist festgelegt in welcher Reihenfolge die Regeln aus RuleSet angewandt werden. Das genaue Format der Rezept-Datei ist in [3.8](#) beschrieben.

3.4.6 RuleEditor

Der RuleEditor ist ein Component der zum editieren von Regeln verwendet wird.

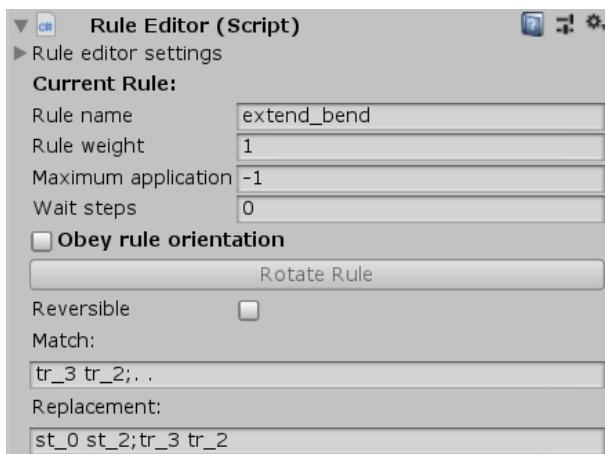


Abbildung 3.5: Inspektor des RuleEditor-Components

Ist im RuleSet eine Regel ausgewählt, enthält der RuleEditor-Inspektor die Einstellungen für diese Regel. Es können Parameter wie Name und Gewichtung der Regel, eine maximale Anzahl an Anwendungen dieser bestimmten Regel und eine Anzahl an „Wait Steps“ gesetzt werden. Der Generator wendet die Regel nicht an bevor nicht so viele Generationsschritte vergangen sind.

Ist der Toggle „Obey rule orientation“ aktiv, werden für diese Regel intern vier Regeln generiert, eine für jede Himmelsrichtung. Dieses Feature existiert, da Menschen instinktiv nicht zwischen Himmelsrichtungen unterscheiden, und somit die meisten Regeln um die y-Achse rotiert auch angewendet werden können. Ist der Toggle angeschaltet kann die Rotation auch per Knopfdruck geändert werden.

5 Text Assets in Unity: [[Uni18b](#), Seite: class-TextAsset]

3 Implementierung

Ist der Toggle „Reversible“ aktiv, so kann die Regel nicht nur angewendet, sondern auch rückgängig gemacht werden. Hierfür wird im Hintergrund eine inverse Regel erstellt, bei der die linke und rechte Seite vertauscht sind. Dies kann schnell zu Fluktuationen im Generator führen, wobei eine Regel in aufeinanderfolgenden Schritten wiederholt angewandt und rückgängig gemacht wird. Deshalb sollte dieses Feature bewusst verwendet werden.

Die linke und rechte Seite der Regeln werden hier in Textfeldern angegeben. Da diese Seiten dreidimensionale Arrays sind, ist die Frage der Eingabemethode für diese Arrays nicht trivial. Es wird hier eine eindimensionale Darstellung des dreidimensionalen Arrays gewählt; die Symbole Semikolon, Komma und Leerzeichen brechen jeweils in die nächste Dimension um. Die in Abbildung 3.6 abgebildete Regel liegt z.B. in der xz-Ebene, da Semikolon in x-Richtung trennt und Leerzeichen in z-Richtung. Die Regeln lesen sich auf dem Screenshot von unten nach oben (z) und von links nach rechts (x).

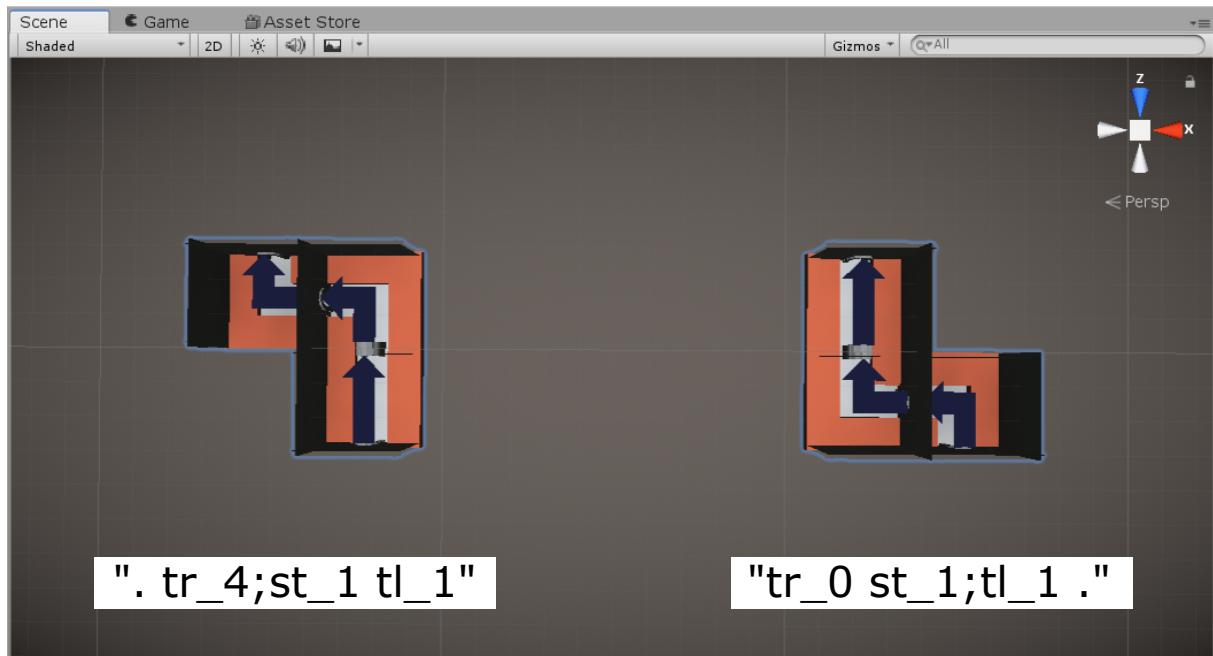


Abbildung 3.6: Beispiel einer Produktionsregel in Unity mit linker und rechter Seite und den dazugehörigen String-Darstellungen. Die verwendeten Symbole sind „st“ (straight), „tl“ (turn left), „tr“ (turn right) und „.“ (kein Raum). Die Pfeile bezeichnen die vorgesehene Begehungsrichtung und sind nur im Scene View sichtbar.

Dieses Format ist nicht wirklich intuitiv, wurde aber so belassen, um den Programmieraufwand akzeptabel zu belassen. Eine mögliche Verbesserung findet sich in 4.2.

3.5 Wichtige Klassen

Es gibt auch einige Klassen, die keine Components, aber trotzdem erwähnenswert sind. Die Klassen Tile, Pattern und Rule modellieren für den Ersetzungsalgorithmus essenzielle Objekte und sind mit dem Attribut System.Serializable markiert, was bedeutet, dass sie serialisiert werden können, ohne von MonoBehaviour oder ScriptableObject zu erben. Die Serialisierung ist notwendig, damit diese Klassen von Unity intern gespeichert werden können.⁶

3.5.1 Tile

Das Tile repräsentiert einen einzelnen Raum im dreidimensionalen Array. Diese Klasse ist ein Wrapper um einen read-only string, der das Label des Raumes darstellt, und hat zusätzliche Convenience-Funktionen, wie z.B. statische read-only Variablen für besondere Tiles (Empty, Wildcard, OutOfBounds).

3.5.2 Pattern

Das Pattern stellt einen dreidimensionalen Array von Tiles dar. Diese Wrapper-Klasse erledigt einige Aufgaben. Sie implementiert die Serialisierung des mehrdimensionalen Arrays, die Unity nicht standardmäßig durchführt. Die Pattern-Klasse besitzt mehrere Konstruktoren, z.B. um ein Pattern mit einer bestimmten Größe, gefüllt mit einem bestimmten Tile zu erzeugen, oder ein Pattern als Kopie eines anderen Patterns zu erzeugen.

Außerdem hat die Klasse die Methode RotatePattern90Y, die eine um 90° um die y-Achse rotierte Kopie des Patterns zurückgibt. Diese Methode wird benutzt um die in 3.4.6 erwähnte „obey rule orientation“ Funktionalität des RuleEditors zu ermöglichen. Diese Methode könnte auch für andere Permutationen ausgebaut werden, aber für den Großteil der Anwendungsszenarien ist dies ausreichend, da Gravitation üblicherweise in y-Richtung wirkt, was bedeutet, dass Räume häufig um die y-Achse drehbar sein müssen, jedoch nicht in andere Richtungen.

6 Serialisierung in Unity: [[Uni18b](#), Seite: script-Serialization]

3 Implementierung

Des weiteren stellt das Pattern die Methoden GetTile und SetTile zur Verfügung um Tiles an gegebenen Positionen im Gitter erhalten und zu modifizieren. Diese Methoden führen einen Bounds-Check durch, um IndexOutOfRangeExceptions zu vermeiden, ein Bounds-Check in anderen Teilen des Programms wird damit überflüssig. Wird eine Position außerhalb des Patterns abgefragt wird stattdessen das spezielle OutOfBounds-Tile zurückgegeben.

Patterns finden sowohl als Datenstruktur des gesamten Dungeons Verwendung, als auch als auch als Datenstruktur der linken und rechten Seiten in Regeln. Sie werden auch zur puren Visualisierung verwendet z.B. um im Editor die linke und rechte Seite einer Regel zu visualisieren, die geschieht dann mit Hilfe eines PatternViews (siehe [3.4.1](#)).

3.5.3 Rule

Die Klasse Rule modelliert eine einzelne Produktionsregel. Wie schon im theoretischen Teil beschrieben besteht eine Produktionsregel aus einer linken und rechten Seite, diese Seiten sind jeweils ein Objekt der bereits beschriebenen Pattern-Klasse.

Des weiteren enthält sie ein Field für den Namen der Regel und folgende Felder für Einstellungsmöglichkeiten: weight, strictRotation, maximumApplications.

Die Klasse ist Serializable, dennoch gibt es außerdem die Klasse SerializedRule; diese wird verwendet, wenn die Klasse für Menschen lesbar serialisiert werden soll, insbesondere für das Speichern eines RuleSets im JSON-Format.

3.5.4 Utils

Die Klasse Utils ist eine statische Utility-Klasse, die häufig verwendete Funktionen beinhaltet, die zu keiner bestimmten anderen Klasse zugehörig sind.

Sie enthält verschiedene Funktionen für Vector3Int[[Uni18a](#), Seite: Vector3Int]: Clamping, Bounds-Checking, Bounds-Überschneidung und Iteration über ein einen dreidimensionales Gitter.

3 Implementierung

Außerdem enthält sie eine Funktion für gewichteten Zufall, eine Funktion um ein Child-Objekt in der Hierarchie einzigartig zu erzeugen, eine Funktion um Labels im Scene View im dreidimensionalen Raum zu zeichnen und ein Tool, dass Mesh Collider für Objekte mit bestimmter Benennung hinzufügt (Speziell für mit der Software Asset Forge erstellte Modelle).

3.6 Optimierungen des Pattern Matchings

Der in [2.2](#) beschriebene Ersetzungs-Algorithmus verlangt, dass bei der Anwendung einer Produktionsregel die linke Seite einer Regel im Dungeon-Array gefunden werden muss.

In einem Generations-Schritt müssen alle möglichen Ersetzungen bekannt sein, damit eine zufällige ausgewählt werden kann. Dies wird getan indem über den Dungeon-Array, über alle Regeln und über die linke Seite der jeweiligen Regel iteriert wird. Diese dreifache Verschachtelung kann sehr rechenaufwändig sein, vor allem für eine große Anzahl an Regeln oder einen großen Dungeon-Array.

Eine Art und Weise die Generierung insgesamt schneller zu machen ist diese Suche nicht jeden Schritt auf dem ganzen Dungeon durchzuführen. Stattdessen wird nur im ersten Schritt das gesamte Dungeon-Pattern durchsucht, gefundene Matches werden in einer Liste gespeichert. Wird in darauf folgenden Schritten eine Ersetzung durchgeführt wird nur die Gegend um das eben ersetzte Stück als „schmutzig“ angesehen. Matches die mit dieser Gegend überschneiden werden aus der Liste gelöscht und die Gegend wird erneut nach Matches durchsucht.

Die Liste der momentan gefundenen Matches kann außerdem gut visualisiert werden, wie in Abbildung [3.7](#) zu erkennen.

3.7 Editor UI

Zur einfacheren Konfiguration und zum schnelleren iterieren wurde der Unity-Editor mit zusätzlicher UI ausgestattet und der Generator so geschrieben, dass er im Editor

3 Implementierung

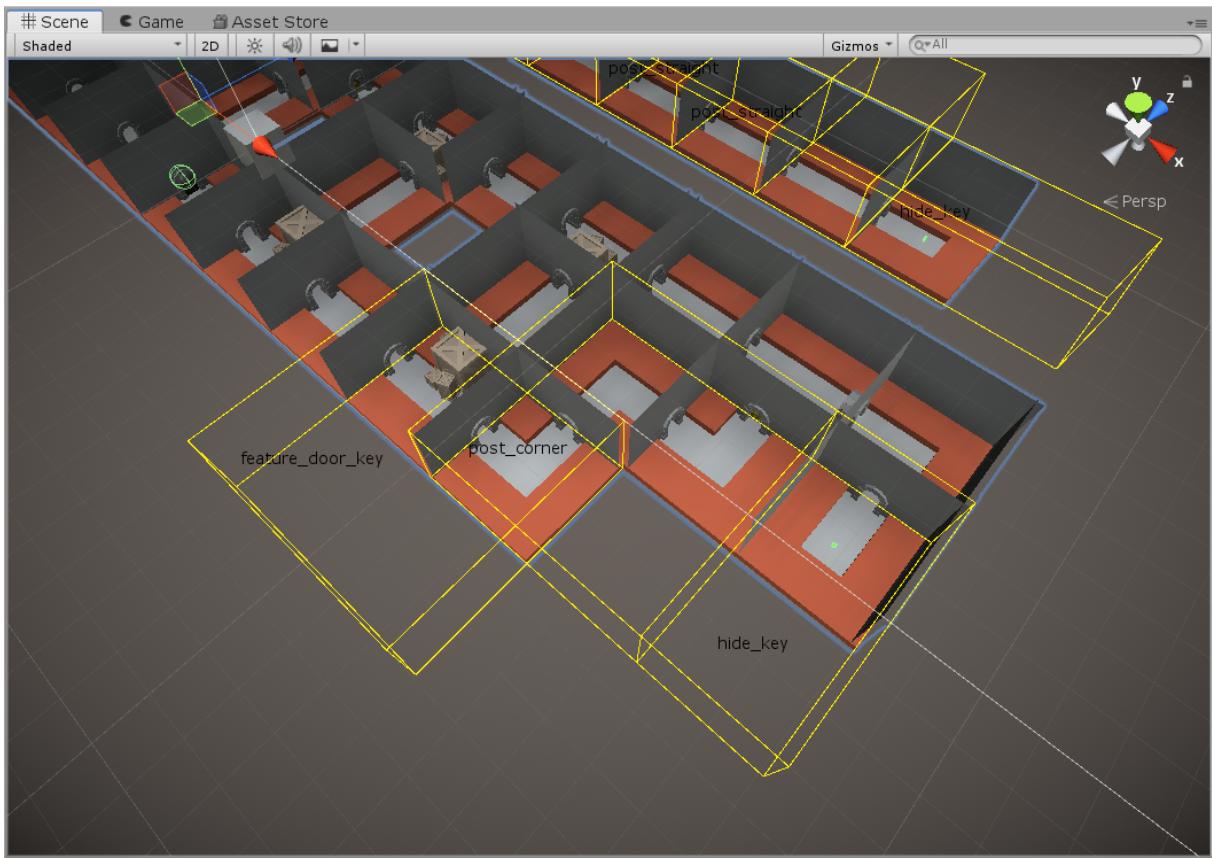


Abbildung 3.7: Visualisierung aller momentan gespeicherten Matches als gelbe Boxen mit zugehöriger Regel.

ausgeführt werden kann.

Die einfachste Art und Weise dies zu ermöglichen ist das Open Source Package EasyButtons. Es ermöglicht Buttons über ein Attribut zu Funktionen hinzuzufügen.

Programmausdruck 3.1: Beispiel eines EasyButtons-Attributs

```
1 [EasyButtons.Button]
2 public void InitializeGeneration()
3 {
4     // ...
5 }
```

3 Implementierung

Diese Buttons werden automatisch am Anfang des Inspektors eingefügt (siehe Abb. 3.8). Dies ist eine einfache Variante einen Inspektor auszubauen, ohne eine zusätzliche Klasse zu definieren.

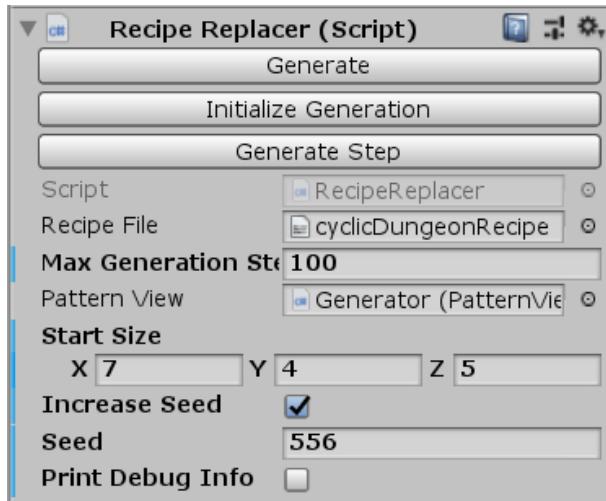


Abbildung 3.8: Screenshot des RecipeReplacer Components. Am Anfang des Inspektors sind die mit EasyButtons generierten Buttons zu sehen

Für andere Editor-Funktionalität musste der Inspektor weiter ausgebaut werden. In diesem Fall schreibt man einen Custom Editor und implementiert die Methode `OnInspectorGUI`⁷.

Die drei Components PatternView, RuleEditor und RuleSet haben benutzerdefinierte Inspektoren. PatternView hat einen simplen benutzerdefinierten Inspektor zum ändern der Größe des Patterns, da der normale Vector3Int Property Drawer hierfür nicht verwendet werden konnte.

Für das RuleSet wurde eine Editor geschrieben um die Unity-Klasse ReorderableList zu nutzen. Mit dieser Klasse kann eine Listenansicht erzeugt werden, die hinzufügen, entfernen und umordnen von Einträgen einer Liste möglich macht. Diese Ansicht wird für die Produktionsregeln verwendet (Listenansicht zu sehen in Abb. 3.3). Außerdem kann der Benutzer eine Regel auswählen. Die ausgewählte Regel kann dann im RuleEditor-Component editiert werden.

Der RuleEditor hat auch einen Custom Editor, da hier die Eigenschaften der aktuell ausgewählten Regel angezeigt werden. Da sich die momentan ausgewählte Regel stän-

⁷ Benutzerdefinierte Editoren in Unity: [Uni18b, Seite: editor-CustomEditors]

3 Implementierung

dig ändern kann, wurde ein User Interface geschrieben, das die eingegebenen Werte für die momentan ausgewählte Regel setzt. Außerdem werden hier die in [3.4.6](#) bereits erklärten String-Darstellungen der linken und rechten Seite der Regel in Arrays umgewandelt.

Des weiteren implementieren einige Components die OnDrawGizmos Methode (siehe [[Uni18a](#), Seite: MonoBehaviour.OnDrawGizmos]) um ihre eigenen Grafiken im SceneView zu zeichnen.

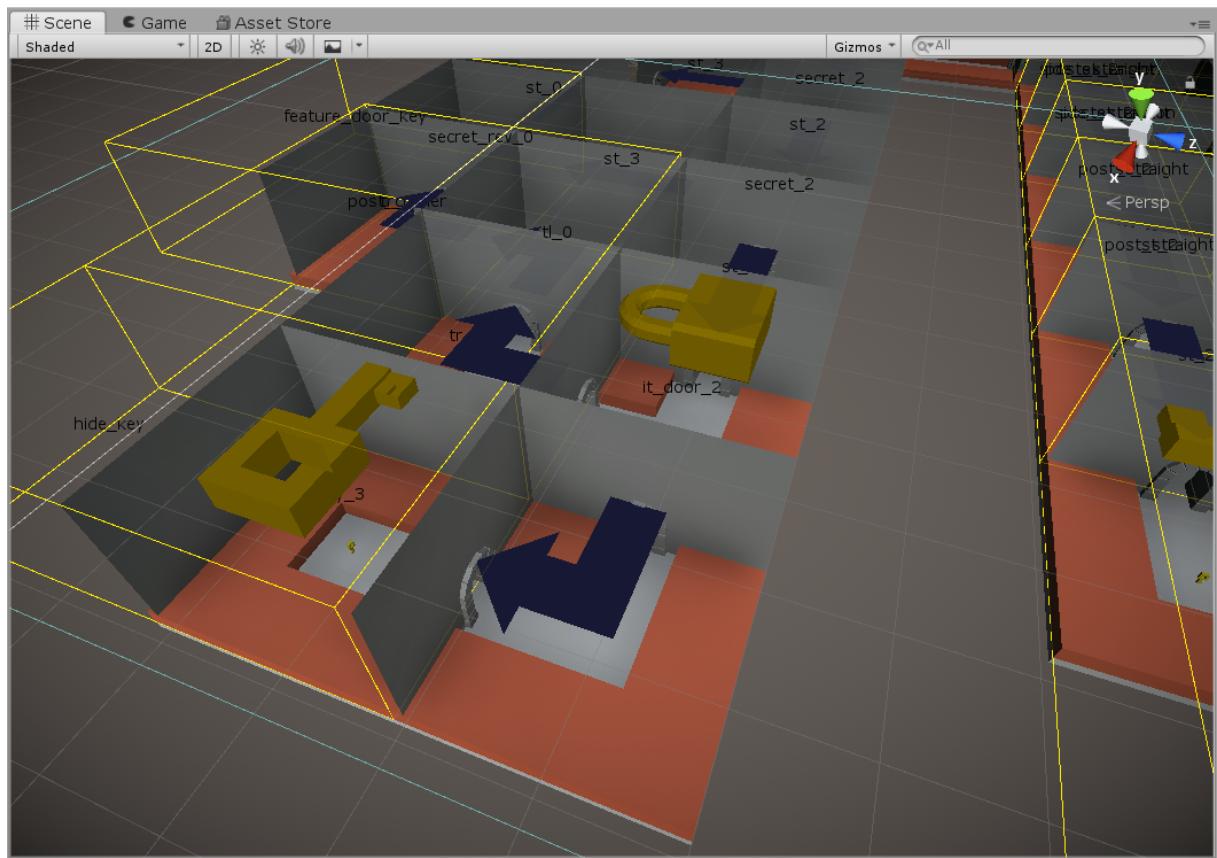


Abbildung 3.9: Beispiel von benutzerdefinierter Visualisierung im Scene View: Pfeile, die die Raumdurchquerungsrichtung angeben, Schlosser- und Schlüssel-Indikatoren, Boxen die die Positionen von bestimmten Patterns anzeigen und Labels, die Matches und Räume bezeichnen.

3 Implementierung

3.8 Rezept-Datei

Wie bereits in [3.4.5](#) erwähnt verwendet der RecipeReplacer eine Text-Datei in der die Reihenfolge der Regelanwendungen definiert werden kann. Will man eine stärkere Kontrolle über den Generierungsprozess haben, ist dies notwendig. So kann man z.B. einschränken, dass erst ein grober Aufbau des Dungeons mit einer eingeschränkten Anzahl an Regeln erzeugt wird, bevor der Dungeon im Detail ausgearbeitet wird.

In der Rezept-Datei ist jede Zeile genau eine Anweisung, es existieren die in Tabelle [3.1](#) aufgelisteten Anweisungen (<> sind Indikatoren für einen Platzhalter).

<Name>	Einmalige Anwendung einer Regel oder Funktion
<Name> <n>	n-malige Anwendung
<Name> <n> <m>	Zufällige Zahl an Anwendungen zwischen n und m
<Name> !	Globale Anwendung / „Rule Sweep“
subdivide <x> <y> <z>	Unterteilung / Vergrößerung
def <Funktionsname>	Beginn einer Funktionsdefinition
end	Ende einer Funktionsdefinition
# <beliebiger Text>	Kommentar

Tabelle 3.1: Arten von Anweisungen in der Rezept-Datei

Die Namen der Regeln sind als Filter zu verstehen, da verschiedene Regeln den gleichen Namen haben können; haben mehrere Regeln den gleichen Namen so wird eine zufällige ausgewählt. Dies ist nützlich wenn zwei Regeln konzeptionell identisch sind, aber als unterschiedliche Regeln aufgeschrieben werden müssen. Im Bezug auf die in [3.6](#) beschriebene Implementierung bedeutet das, dass die vorhandene Liste an möglichen Matches bezüglich des Regelnamens gefiltert wird und alle andersnamigen Matches ignoriert werden.

In einer Anweisung kann der Name einer Regel auch das Format <Regelanfang>* haben. In diesem Fall muss eine Regel nur mit <Regelanfang> anfangen um angewendet werden zu können. So kann z.B. die Anweisung *feature_** sowohl die Regel *feature_alternative* als auch die Regel *feature_shortcut* anwenden.

3 Implementierung

Die Anweisungen der Form $\langle Name \rangle !$ und $subdivide \langle x \rangle \langle y \rangle \langle z \rangle$ sind nicht Teil des zugrunde liegenden Algorithmus (siehe [2.2](#)), sondern sind pragmatische Erweiterungen die in bestimmten Anwendungsfällen nützlich sind.

Ein „Rule Sweep“ der Form $\langle Name \rangle !$ wird verwendet um eine Regel global anzuwenden. Die definierte Regel wird für alle momentan bekannten Matches angewendet, ohne die Liste der Matches zwischenzeitlich neu zu berechnen. Das heißt wenn sich Matches überschneiden wird der Dungeon unter Umständen in einen Zustand transformiert, der so durch die Produktionsregeln nicht definiert wurde.

Dies ist ein Kompromiss der eingegangen wurde, da diese Anweisung sehr viel performanter ist als eine widerholte sequenzielle Anwendung mit ständiger Neuberechnung.

Die Unterteilungs-Anweisung der Form $subdivide \langle x \rangle \langle y \rangle \langle z \rangle$ wird verwendet um das Gitter zu vergrößern. Hierzu wird für jedes Tile im Dungeon genau $\langle x \rangle, \langle y \rangle, \langle z \rangle$ Tiles in x -, y -, z -Richtung eingeschoben (bildlich veranschaulicht in Abbildung [3.10](#)). Diese Anweisung kann verwendet werden um erst die Makro-Struktur des Dungeons mit wenigen Tiles zu generieren, und anschließend den Dungeon zu strecken bevor weiter-generiert wird. Diese Anweisung muss üblicherweise mit einer globalen Anwendung kombiniert werden, die die entstandenen Lücken füllt. Um das zu tun erstellt man eine Anzahl an Lückenfüller-Regeln, die gleich anfangen (z.B. $subdiv_*$...) und führt diese global aus (z.B. $subdiv_* !$).

Programmausdruck 3.2: Beispiel einer Funktionsdefinition in der Rezept-Datei

```
1 def subdiveFill
2     subdivide 1 0 1
3     subdiv_* !
4 end
5
6 subdiveFill 2
```

Ein weiteres Feature der Rezept-Datei ist das definieren von Funktionen (siehe Programmausdruck [3.2](#)). Ist eine Funktion definiert kann sie wie eine normale Regel angewandt werden, allerdings sind Funktionen nicht durch Anweisungen der Form $\langle Regelanfang \rangle^*$ ausführbar.

3 Implementierung

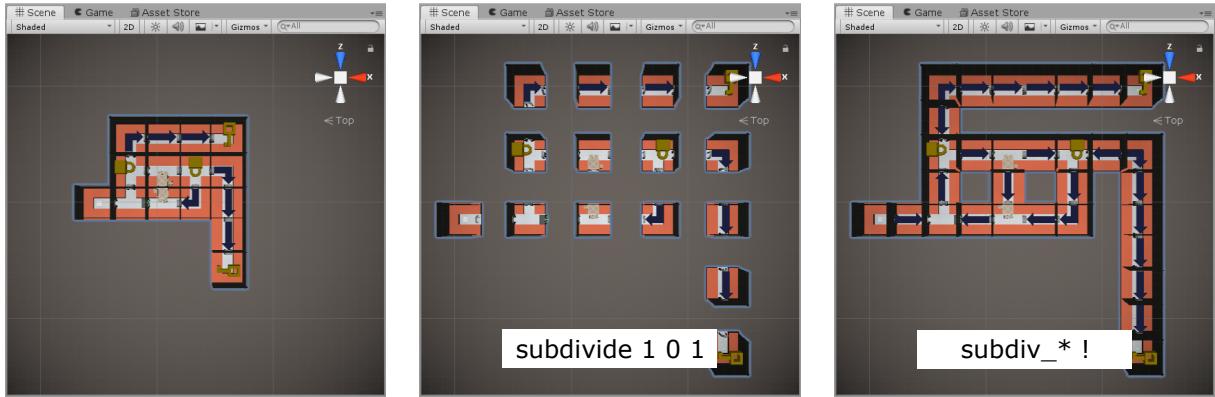


Abbildung 3.10: Anwendungsbeispiel von Unterteilung und Füllen von Lücken.

3.9 Spielmechaniken

Um den Dungeon-Generator zu demonstrieren wurden ein paar rudimentäre Spielmechaniken implementiert. Startet man das Spiel wird ein Dungeon generiert und der/die Spieler/in erscheint im Start-Raum.

Der/die Spieler/in kann sich in einer First-Person-Perspektive mit der Maus umschauen und mit dem WASD-Tasten herumlaufen. Mit der Leertaste kann gesprungen werden (auch in der Luft, was nur zu Testzwecken getan werden sollte).

Mit der rechten Maustaste kann mit Objekten interagiert werden, interaktive Objekte werden mit einem Text auf dem Bildschirm gekennzeichnet. Mit der linken Maustaste können Würfel geschossen werden, die nur zum Austesten von Kollision oder zum hinterlassen von Markierungen gedacht sind, aber keine Spiel-Funktion besitzen. Der Spieler kann so den Dungeon durchsuchen und die Geometrie von innen beurteilen.

Es wurde ein Schlüssel-Schloss-System implementiert. Der Spieler kann im Dungeon Schlüssel finden und aufheben, die Anzahl der Schlüssel wird auf dem Bildschirm abgebildet. Hat der Spieler einen Schlüssel, so kann er diesen benutzen um eine Tür zu öffnen. Dieses System kann verwendet werden um zu testen, ob die Schlüssel-Schloss-Regeln richtig geschrieben worden, und keine Tür erzeugt wird ohne das davor im Dungeon auch ein Schlüssel erzeugt wurde.

Ein weiteres interaktives Objekt ist eine schwarze Box, die über einer Tür erzeugt und nur von einer Seite entfernt werden kann. Diese Box wird in einem Durchgang platziert um

3 Implementierung

diesen Durchgang nur in eine Richtung zu versperren.

Diese Spielmechaniken bilden natürlich kein richtiges Spiel ab, reichen aber aus um die Idee der erzeugten Dungeon-Struktur zu erkunden. Würde man den Generator für ein tatsächliches Spiel verwenden würde man die Dungeons mit diesem Spiel testen.

3.10 Raum-Modelle

Ähnlich wie die Spielmechaniken sind auch die Modelle der einzelnen Räume zur Veranschaulichung gedacht und enthalten keine Monster, Fallen oder ähnliches. Diese Räume müssten für jedes Spiel individuell modelliert und gestaltet werden.

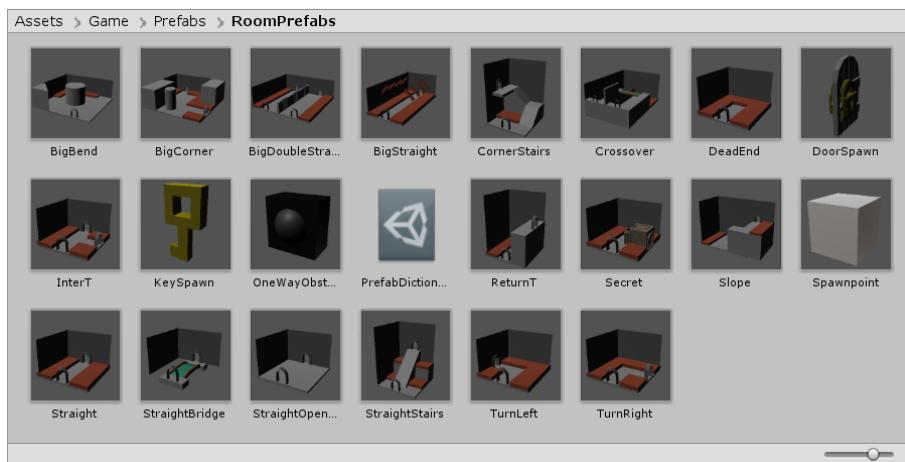


Abbildung 3.11: Screenshot der Raum-Prefabs im Project-Window

Die Räume liegen als Prefabs vor, was heißt, dass eine mit Unity vertraute Person kein Problem haben sollte diese Räume zu erstellen, es muss lediglich darauf geachtet werden, dass die Räume and den Übergängen perfekt zusammenpassen. Für diese Arbeit wurde das Programm Asset Forge verwendet um die Räume zu erstellen, aber die Räume hätten auch mit einem herkömmlichen 3D-Editor oder in Unity erstellt werden können. Die Raumteile für Decken, Wände und Türbögen sowie die Modelle für Schlüssel und Schloss wurden in Blender modelliert. Es wurden vereinzelt Modelle aus einem öffentlich verfügbaren Modell-Set verwendet (siehe 6).

3 Implementierung

3.11 Regelsystem für einen zyklischen Dungeon-Generator

Um das Ziel der Arbeit zu erfüllen einen zyklischen Generator zu entwickeln wurden die beschriebenen Werkzeuge verwendet um einen Generator zusammenzustellen. Dieser ist in der Szene „SampleScene“ zu finden. Der Generator verwendet einen RecipeReplacer-Component um den Dungeon zu generieren und 29 Regeln (im Hintergrund erstellte Rotationen von Regeln nicht eingeschlossen). Die Rezept-Datei ist cyclicRecipe.txt.

Im folgenden wird kurz die Generation anhand der Rezept-Datei (3.3) aufgebrochen.

Programmausdruck 3.3: Rezept-Datei des Generators

```
1 basic_structure
2 basic_transform_* 3 8
3 add_floor_* 1
4 basic_transform_* 3 7
5 feature_* 3 6
6 feature_lock
7
8 subdivide 1 0 1
9 subdiv_* !
10
11 basic_transform_* 0 10
12 move_key 0 10
13 post_* 10 20
```

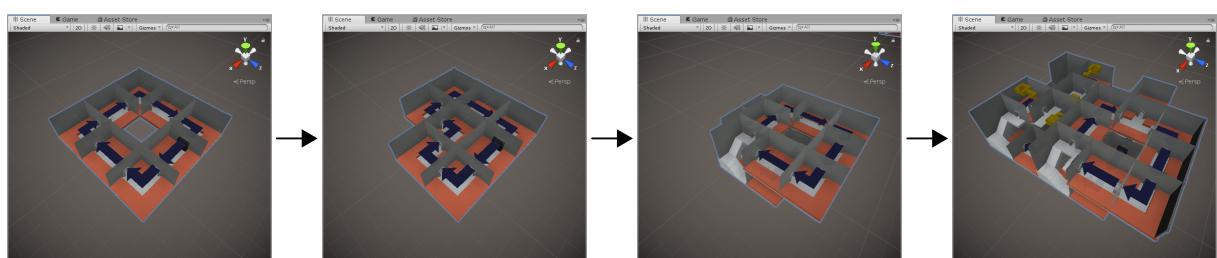


Abbildung 3.12: Screenshots aus dem zyklischen Generationsprozess (erste Hälfte)

3 Implementierung

In Zeile 1 wird die Grundstruktur des Dungeons platziert. Die Regel *basic_structure* hat als linke Seite das Start-Label „start“ und als rechte Seite einen drei mal drei Tiles großen Zyklus mit einem „Einbahnstraßen“-Hindernis am Start. Mit *basic_transform_** wird eine zufällige Zahl an Transformationen angewandt, die das Aussehen des Dungeons interessanter machen, indem z.B. Ecken umgeknickt werden, wie in Abbildung 3.12, Bild 2 zu sehen.

Mit der Anweisung *add_floor_** wird an einer der Ecken ein Treppenaufgang mit zweitem Stockwerk hinzugefügt, das zweite Stockwerk hat die selbe Struktur wie das untere. Für mehr Stockwerke könnte man diese Regel häufiger anwenden. Anschließend wird erneut *basic_transform_** angewandt um das obere Stockwerk interessanter zu machen.

In Zeilen 5 und 6 werden sogenannte Features hinzugefügt. Dies kann ein Zyklus mit verschlossener Tür, eine alternativer Weg oder eine abzweigende Sackgasse sein.

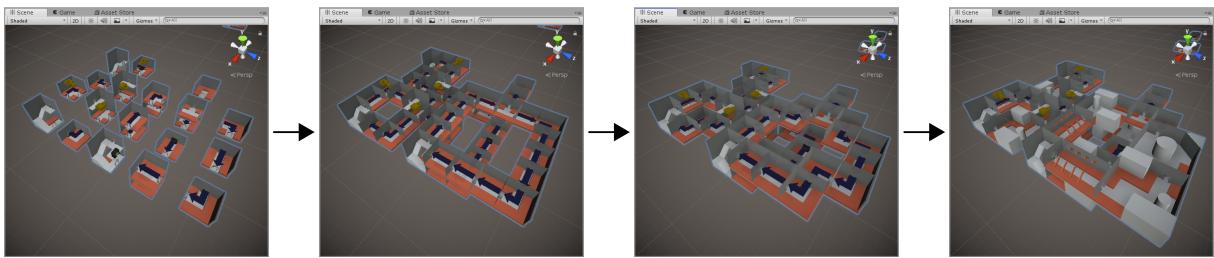


Abbildung 3.13: Screenshots aus dem zyklischen Generationsprozess (zweite Hälfte)

In Zeile 8 wird der Dungeon unterteilt um zusätzlichen Platz zu schaffen. In Zeile 9 wird eine Sammlung an Regeln global angewendet, die Zwischenstücke einfügt. Dieser Prozess ist in 3.8 beschrieben (siehe Abb. 3.13 Bild 1 und 2).

Anschließend wird ein weiterer *basic_transform_** durchgeführt und die Schlüssel werden mit der Regel *move_key* etwas bewegt (siehe Abb. 3.13 Bild 3).

Die letzte Anwendung *post_** 10 20 wendet wiederholt Regeln der Form *post_...* an. Diese sind sogenannte Post-Processing-Regeln. Eine Post-Processing-Regel nimmt einen kleinen Teil des Dungeons, bestehend aus Grundbausteinen und ersetzt ihn durch einen interessanteren handgemachten großen Raum. Dies geschieht zum Schluss, denn diese Räume können nicht weiter transformiert werden, da es zu umständlich wäre für sie extra

3 Implementierung

Regeln zu schreiben. Den Stand des Dungeons nach diesem Schritt sieht man in Abbildung [3.13 Bild 4](#).

Ein interessantes Ergebnis erhält man auch, wenn man den Recipe-Replacer durch einen RandomReplacer austauscht. Die Dungeons sind kompakter, da keine Unterteilung stattfindet, aber sehr viel abwechslungsreicher, da weniger Struktur vorgegeben ist.

4 Auswertung

4.1 Auswertung der generierten Dungeons

Es ist zuerst zu sagen, dass eine objektive Bewertung der Dungeons schwer möglich ist. Dies hat mehrere Gründe.

Ein direkter Test mit Spielern und eine Auswertung über Fragebögen ist nicht möglich, da der Dungeon-Generator nicht für ein bestimmtes Spiel konzipiert würde. Mit einem Test-Spiel wäre dies jedoch ein guter Ansatz, wenn auch etwas aufwändig und nicht im Rahmen dieser Arbeit machbar.

Zweitens ist die Qualität von Dungeons prinzipiell subjektiv, was eine rein statistische Auswertung erschwert.

Drittens gibt es keine akzeptierten Standards für die Auswertung von Dungeons, was einen Vergleich mit anderen Verfahren schwer macht. Eine Vergleich von verschiedenen Verfahren wäre natürlich interessant, aber auch nicht Inhalt dieser Arbeit.

Es entsteht allerdings der subjektive Eindruck, dass die Dungeons so wie sie in 3.11 generiert werden, zueinander relativ ähnlich sind. Dies ist der Tatsache geschuldet, dass nur eine kleine Anzahl an Feature-Regeln erstellt wurden. Diese Strukturen kehren immer wieder, und es fällt einem Spieler/einer Spielerin auf, dass z.B. auf eine Tür häufig ein Treppenaufgang auf der linken Seite folgt.

Davon abgesehen ist dennoch gezeigt, dass Dungeons aus Zyklen auf diese Art und Weise gebaut werden können und es ist denkbar, dass mit einem komplexeren Regelsystem noch bessere Ergebnisse erzielt werden können. Mögliche Verbesserungen, die die generierten Dungeons abwechslungsreicher gestalten werden in 4.2 betrachtet.

4 Auswertung

Das entstandene Array-Ersetzungs-System ist darüber hinaus äußerst flexibel. Es ist auch für nicht-zyklische Strukturen sehr effektiv, ein Beispiel hierfür ist im Projekt in der Szene CubeScene gezeigt (Abb. 4.1) und in der Szene DoorSpaceScene (Abb. 4.2)

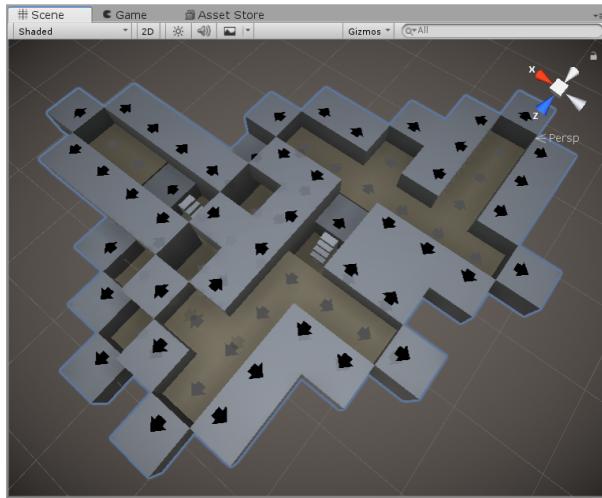


Abbildung 4.1: Nicht-zyklischer würfelbasierter Generator, der mit wenig Raumteilen und Regeln interessante Strukturen erzeugt.

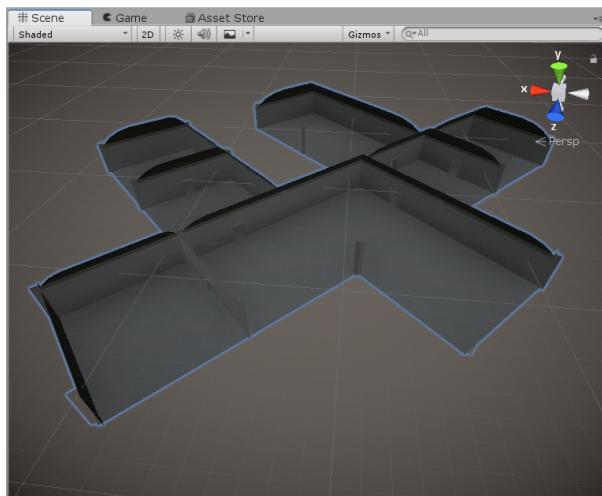


Abbildung 4.2: Nicht-zyklischer Gewölbe-Generator, bei dem nicht die Räume, sondern stattdessen die Wände, Böden und Türen als Symbole verwendet werden.

4.2 Mögliche Verbesserungen des Systems, Ausblick

4.2.1 Ausbauen des Regel- und Rezept-Systems

Damit die Dungeons abwechslungsreicher ausfallen könnte das Regel- und Rezept-System weiter ausgebaut werden. Eine Möglichkeit wäre Regeln nicht nur automatisch zu rotieren, sonder auch zu spiegeln. Hierzu müsste für die Räume allerdings Symmetrieinformation existieren (z.B. Raumteil *tl* ist spiegelverkehrt zu Raumteil *tr*).

Eine nützliche Erweiterung des Rezept-Systems wäre das Definieren von Mengen an Regeln. Man könnte Regeln bestimmten Mengen zuweisen und dann zufällige Regeln einer Menge ausführen. Dies wäre ein Kompromiss zwischen RandomReplacer und RecipeReplacer, da sich innerhalb einer Menge der RecipeReplacer wie ein RandomReplacer verhält – er sucht aus einer Menge an Regeln komplett zufällig aus. Anstatt Mengen in der Rezept-Datei zu definieren wäre auch eine Lösung mit mehreren RuleSets denkbar.

4.2.2 Verbessern der Räume

Die Räume, die die Bausteine des Dungeons sind, sind uninteressant und sehen immer gleich aus. Eine einfache Möglichkeit dies zu verbessern wäre einen einfachen Raum-Generator zu schreiben, der Räume mit Objekten dekoriert. Man könnte in den Räumen Spawnpoint-Objekte platzieren, die zufällig Möbel, Fallen oder Monster erzeugen.

Ein weiterer wichtiger Schritt wäre eine große Anzahl an Räumen zu erzeugen, die alle mit Post-Processing-Regeln (siehe 3.11) platziert werden und Abwechslung für häufig auftauchende Strukturen erzeugen ohne die Komplexität des Regelwerks zu beeinträchtigen. Anstatt von Post-Processing-Regeln könnte man diesen Schritt auch in das PrefabDictionary integrieren und dort mehrere Varianten für ein Raum-Symbol einstellbar machen, aus denen zufällig ausgewählt wird.

4 Auswertung

4.2.3 Modellierung der Muster

Da die Generierung auf einem Array stattfindet ist die Größe des Dungeons durch den Array begrenzt. Häufig sind dennoch große Teile des Arrays nicht gefüllt. Des weiteren wäre es interessant den Generator für beliebige Gitter-Strukturen wie in [2.3](#) beschrieben auszubauen.

Es ist unter Umständen sinnvoll die Datenstruktur eines Arrays aufzugeben und eine flexiblere zu wählen. Diese Datenstruktur müsste eine Abbildung von ganzzahligen Tupeln auf Symbole ermöglichen. Eine offensichtliche Wahl wäre ein HashSet, dass Integer-Vektoren auf Tiles abbildet. Die Hash-Funktion der Integer-Vector-Klasse müsste einzigartig im Bezug auf die Koordinaten sein, damit an der selben Position nicht zwei Tiles liegen können. Das HashSet würde nur die Positionen abbilden, an denen nicht-leere Tiles liegen, der Rest des Raumes wird als leere Tiles angenommen.

Nach dieser Änderung könnte der Dungeon beliebig groß wachsen. Es ist außerdem eine Performance-Verbesserung denkbar, da nun in keinem Fall das ganze Array durchsucht werden muss.

4.2.4 Eingabe der Regeln

Wie bereits in [3.4.6](#) erwähnt ist die textbasierte Eingabe der Regeln nicht intuitiv und selbst dann umständlich wenn man mit dem System vertraut ist. Es würde sehr viel Sinn machen ein besseres Eingabesystem zu konzipieren.

Es ist schwer zu sagen wie ein solches System aussehen sollte, da es das editieren in drei Dimensionen ermöglichen muss. Denkbar wären unter anderem ein intuitiveres textbasiertes System, ein dreidimensionaler Editor, oder ein zweidimensionaler Editor in dem man die einzelnen Scheiben eines Patterns editiert.

5 Fazit

TODO: noch schreiben

6 Verwendete Materialien

Unity Engine von Unity Technologies <https://unity3d.com/>

Blender <https://www.blender.org/>

EasyButtons von Mads Bang Hoffensetz <https://github.com/madsbangh/EasyButtons>

Asset Forge von Kenney <https://assetforge.io/>

Modular Dungeon 2 - 3D Models von Keith at Fertile Soil Productions <https://opengameart.org/content/modular-dungeon-2-3d-models>

LATEX-Vorlage von Martin Bretschneider <https://www.bretschneidernet.de/tips/thesislatex-vorlagen.html#vorlagen>

Literaturverzeichnis

- [Dor10] Dormans, Joris: Adventures in level design: Generating missions and spaces for action adventure games. (2010), 01. <http://dx.doi.org/10.1145/1814256.1814257>. – DOI 10.1145/1814256.1814257
- [Dor11] Dormans, Joris: Level design as model transformation: A strategy for automated content generation. (2011), 01. <http://dx.doi.org/10.1145/2000919.2000921>. – DOI 10.1145/2000919.2000921
- [Dor16] Dormans, Joris: *A Handcrafted Feel: ‘Unexplored’ Explores Cyclic Dungeon Generation.* <http://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/>, 2016. – letzter Zugriff: 01.02.2019
- [LLB14] Linden, Roland; Lopes, Ricardo; Bidarra, Rafael: Procedural Generation of Dungeons. In: *Computational Intelligence and AI in Games, IEEE Transactions on* 6 (2014), 03, S. 78–89. <http://dx.doi.org/10.1109/TCIAIG.2013.2290371>. – DOI 10.1109/TCIAIG.2013.2290371
- [Mad18] Mads Bang Hoffensetz: *EasyButtons Source Code und Dokumentation.* <https://github.com/madsbangh/EasyButtons>, 2018. – letzter Zugriff: 01.02.2019
- [STN16] Shaker, Noor; Togelius, Julian; Nelson, Mark J.: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016
- [Uni18a] Unity Technologies: *Unity Scripting Reference.* <https://docs.unity3d.com/ScriptReference/index.html>, 2018. – letzter Zugriff: 01.02.2019
- [Uni18b] Unity Technologies: *Unity User Manual.* <https://docs.unity3d.com/Manual/index.html>, 2018. – letzter Zugriff: 01.02.2019

Bildquellenverzeichnis

[Log15] Logos, Dyson: *Map of the fictional Warrek's Nest dungeon.* https://commons.wikimedia.org/wiki/File:Warrek%20%99s_Nest.jpg, 2015. – letzter Zugriff: 01.02.2019

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum

Unterschrift