

Entwicklung eines dreidimensionalen zyklischen Dungeon-Generators mit Unity3D

Bachelorarbeit

Nils Gawlik
Matrikel-Nummer 553449

Betreuer Prof. Lenz
Erstprüfer Prof. Lenz
Zweitprüfer Prof. Strippgen

Inhaltsverzeichnis

Abbildungsverzeichnis	IV
1 Einführung und Definitionen	1
1.1 Motivation	1
1.2 Dungeons	2
1.2.1 Definition von Dungeon	2
1.2.2 Wichtige Beispiele von Dungeons	2
1.2.3 Prozedurale Dungeon-Generierung in Spielen, Geschichte	3
1.3 Genaue Beschreibung der Fragestellung	3
1.3.1 Definition von zyklisch	3
1.3.2 Dreidimensionalität	5
1.4 Warum Unity?	5
2 Der Ersetzungs-Algorithmus	6
2.1 Formale Grammatiken und Graphgrammatiken für die Generierung	6
2.2 Spezieller Algorithmus	7
2.3 Allgemeiner Algorithmus	11
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 Erklärung der Komponenten	16
3.5 Wichtige Klassen	22
3.5.1 Tile	22
3.5.2 Pattern	23

Inhaltsverzeichnis

3.5.3 Rule	23
3.5.4 Utils	24
3.6 Lazy updating, suchen	25
3.7 Editor UI	25
3.8 Regelsystem	25
3.9 Recipe-Format	25
3.10 Modelle	25
4 Auswertung der generierten Dungeons	26
4.1 Einschränkungen	26
4.2 qualitative Auswertung	27
4.3 quantitative Auswertung	27
4.4 Mögliche Verbesserungen	27
4.5 Mögliche Anwendungsszenarien	27
4.5.1 Roguelike, Rogue-lite	27
4.5.2 Minigolf	27
4.5.3 Text adventure	27
5 Zusammenfassung	28
6 Verwendete Materialien	29
Literaturverzeichnis	30
Bildquellenverzeichnis	31
7 Erklärung	32

Abbildungsverzeichnis

1.1	Dungeon-Beispiel	2
1.2	Dungeon-Beispiel mit Graph	4
2.1	Beispiel für Graph im Gitter	8
2.2	Äquivalenz von Knoten und Raum	8
2.3	Generator mit prozeduralen Räumen	9
2.4	Beispiel einer Produktionsregel	10
2.5	Beispiel einer Regelanwendung	11
2.6	Beispiel einer Produktionsregel aus Levelteilen	11
3.1	Ansicht des Unity-Editors	13
3.2	Verschiedene generierte Dungeons	14
3.3	Screenshot der Generator-Komponenten	17
3.4	Dungeon-Teil im Unity Scene View	18
3.5	Inspektor des RuleEditor-Components	21
3.6	Beispiel einer Produktionsregel in Unity	22

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 Unity3D statt.

TODO: Quelle finden, dass viele designete Dungeons Zyklen aufweisen

1.2 Dungeons

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 einheitliche Definition für Dungeon in diesem Kontext existiert nicht, aber üblicherweise bezeichnet er einen weitläufigen, in sich geschlossenen Raum gefüllt mit Fallen und Monstern, den ein oder mehrere Spieler navigieren müssen, häufig mit einer großen Herausforderung am Ende, z.B. einem besonders starken Boss-Monster.

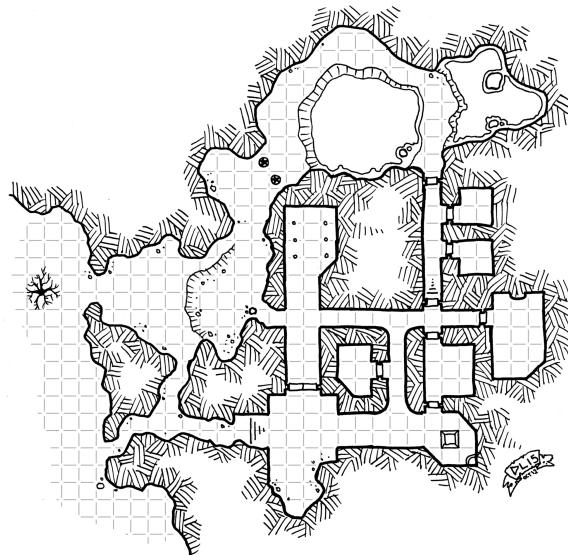


Abbildung 1.1: Beispiel eines Dungeons [Log15]

1.2.2 Wichtige Beispiele von Dungeons

Dungeons ist in einigen der populärsten Spielen vertreten. Beispiele sind „Dungeons & Dragons“, „The Legend Of Zelda“, „The Elder Scrolls“ und „Minecraft“.

TODO: Eine gute Auswahl finden, Bilder-Collage machen und einfügen

1 Einführung und Definitionen

1.2.3 Prozedurale Dungeon-Generierung in Spielen, Geschichte

In Computerspielen gibt es eine relativ lange Tradition der Dungeon-Generierung.

TODO: Research Roguelikes, Roguelites, TES Oblivion

1.3 Genaue Beschreibung der Fragestellung

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

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

1.3.1 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.

Zyklen im Dungeon erkennt man am Level-Graph, ist der Level-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 Übersetzung der Level-Geometrie in einen Graphen ist nicht eindeutig - es sind mehrere Graphen möglich, aber der Graph sollte die Topologie des Raumes wiedergeben. TODO: Evtl. etwas zu topologischer Graphentheorie schreiben

In der Realität kommt es selten vor, dass man beurteilen muss ob ein bereits existierender Dungeon zyklisch ist oder nicht. Häufiger ergibt sich ein Level-Graph während der

1 Einführung und Definitionen

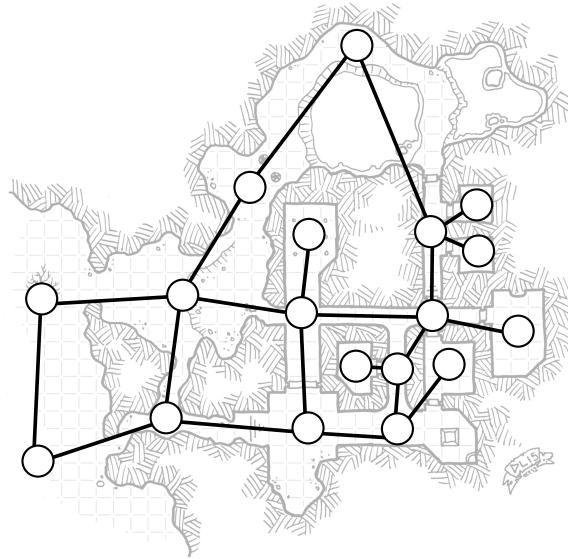


Abbildung 1.2: Bild eines Dungeons, mit möglichem Level-Graph, der Graph ist ungewichtet und ungerichtet ([Log15], bearbeitet)

Generierung, z.B. indem kontinuierlich Räume hinzugefügt werden. Die Räume können als Knoten, die Verbindungen zwischen den Räumen als Kanten verstanden werden.

Der 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 auch häufig in Level-Generatoren findet: Man generiert ein Labyrinth ohne Zyklen mit 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 allerdings 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 direkt konfigurierbar.

1 Einführung und Definitionen

1.3.2 Dreidimensionalität

Dreidimensionalität TODO: Wie definieren???

1.4 Warum Unity?

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

Es wurde eine Game Engine verwendet, um den Implementierungs-Teil der Arbeit auf den Generator selbst und das User-Interface des Generators zu beschränken. 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 (die häufig von prozeduraler Generierung Gebrauch machen TODO: Really?), 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

2 Der Ersetzungs-Algorithmus

des 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.

Eine Graphgrammatik verhält sich wie eine Formale Grammatik, allerdings sind die linken und rechten Seiten hier 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. Diese Missionsstrukturen sind zyklisch, der Raum allerdings nicht.

2.2 Spezieller Algorithmus

Der Generator den ich diese Bachelorarbeit entwickelt habe, ist durch wiederholte Iteration und Experimentation entstanden. Diese Implementierung wird später ausführlich in Kapitel 3 beschrieben.

Aber zum klareren Verständnis will ich zunächst den spezielleren Algorithmus formal beschreiben, und in 2.3 bei einer Verallgemeinerung ankommen.

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, 2.] redet Dormans von „mission and space“ und dass diese beiden Modelle von Level-Designern häufig als isomorphisch angenommen werde. Er argumentiert jedoch, dass sie als separate Modelle verstanden werden sollten.

2 Der Ersetzungs-Algorithmus

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

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 Level-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.

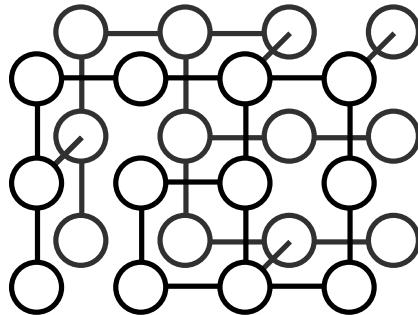


Abbildung 2.1: Beispiel für einen dreidimensionalen Graph innerhalb der Restriktionen des Gitters

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).

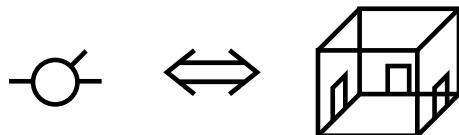


Abbildung 2.2: Knoten mit Halb-Kanten und äquivalenter Raum

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 geschicktes nutzen von

2 Der Ersetzungs-Algorithmus

Rotationssymmetrie könnte man diese Zahl verringern, dennoch ist dies nicht ideal, da ein Level-Designer trotzdem eine Mindestzahl an Räumen entwerfen müsste. Außerdem will man für einen abwechslungsreichen Dungeon verschiedene „Themes“ haben, verschiedene Arten von Raumübergängen und andere Arten von Variationen. Diese Variationen kann man nicht alle in allen möglichen Eingangs/Ausgangsvariationen bereitstellen.

TODO: Nochmal genauer darüber nachdenken, was genau diese Variation hier bedeuten, etc. (oder

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.

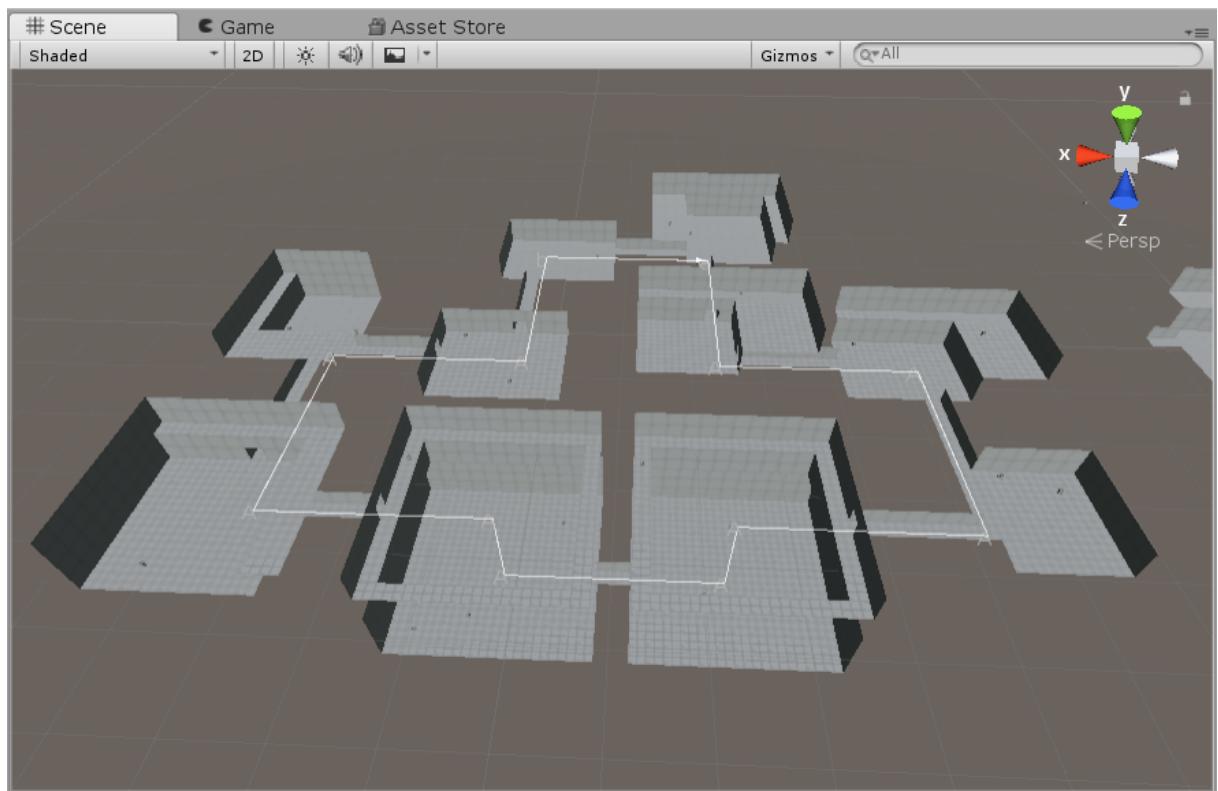


Abbildung 2.3: Screenshot einer frühen Version des Generators mit prozedural generierten Räumen. Die Räume sehen sehr ähnlich aus.

2 Der Ersetzungs-Algorithmus

Die Lösung dieses Problems ist die Umformulierung des Modells. Anstatt einen Level-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.

Für diesen Algorithmus wird eine Array-Grammatik verwendet, deren Produktionsregeln als linke und rechte Seite gleich große dreidimensionale Arrays haben. Ein Vorkommen der linken Seite kann 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.

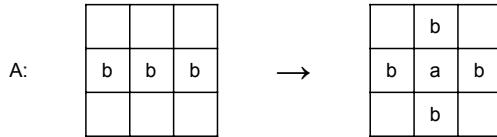


Abbildung 2.4: Beispiel einer Produktionsregel¹

Anders als bei eindimensionalen 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 lediglich das relevante Teilstück ersetzt.

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.

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

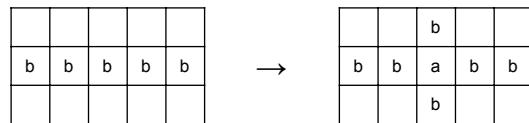


Abbildung 2.5: Anwendung der Produktionsregel A (eine von drei möglichen Positionen der Anwendung)

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. Symbole für verschiedene Rotationen des gleichen Modells können automatisch erzeugt werden.

TODO: Bild aus der Implementierung nehmen

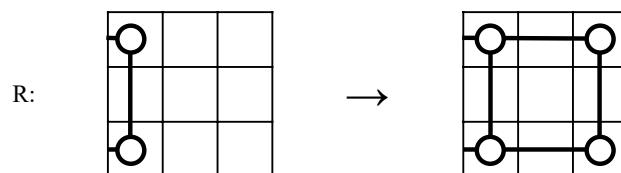


Abbildung 2.6: Alphabet Σ mit Level-Graph-Teilen und einer Produktionsregel R

2.3 Allgemeiner Algorithmus

Der in Abschnitt 2.3 beschriebene Algorithmus ist der in der Implementierung verwendete.

2 Der Ersetzungs-Algorithmus

Im Folgenden will ich diesen allerdings zu akademischen Zwecken (TODO: really?) verallgemeinern.

$$G = (V, T, P, S)$$

V ist die Symbolmenge

$T \subset V$ ist die Menge der Terminalsymbole

$P \subset (V^* \setminus t^*) \times V^*$ ist die Menge der Produktionsregeln

X^*

TODO: Mache ich später, wenn platz und bock. :P

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.

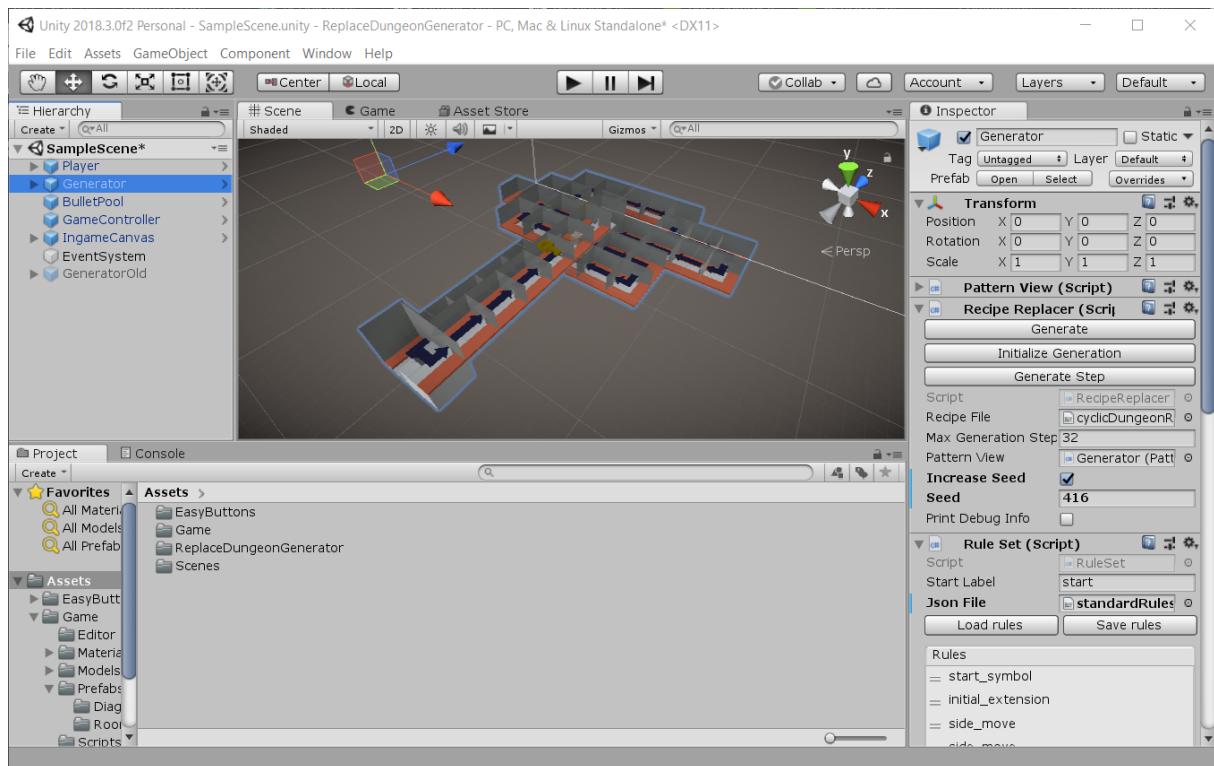


Abbildung 3.1: Ansicht des Unity-Editors in einer typischen Szene

3 Implementierung

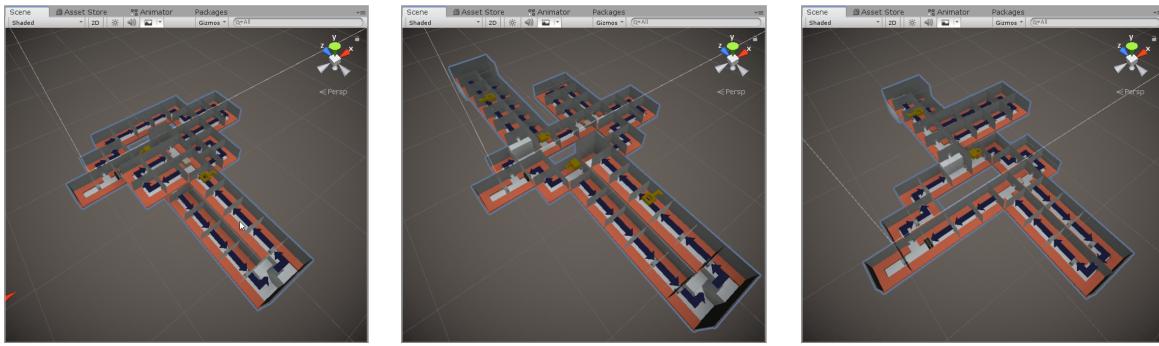


Abbildung 3.2: Verschiedene in Unity generierte Dungeons, im Scene View betrachtet

3.1 Ziele der Implementierung

Wie später in 4.5 beschrieben, sind viele verschiedene Anwendungsszenarien für diesen Generator denkbar.

Ein Ziel der Implementierung ist es, dass die Software nicht einen spezifischen Generator darstellt, sondern eine Software in der 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 Komponenten angehängt, die verschiedene Funktionen erfüllen und miteinander interagieren können. Dies ermöglicht einen modularen Aufbau von Objekten. [Uni18b, Seite: GameObjects] Neue Komponenten können in Unity programmiert werden,

3 Implementierung

indem man eine Klasse schreibt, die von der Klasse MonoBehaviour erbt. [[Uni18b](#), Seite: CreatingAndUsingScripts]

Ein Beispiel hierfür kann man in Abbildung [3.3](#) sehen, nur durch das Zusammenspiel der einzelnen Komponenten 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 Komponenten 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 Komponenten ausgetauscht oder umgeschrieben werden können, solange die öffentlichen Funktionen und Variablen dieser Komponenten 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; Unity unterstützt hierbei gewisse Datentypen von Haus aus, für andere kann man sogenannte Property Drawers definieren. Alle Generator-Komponenten haben public Fields, was den ganzen Generator sehr konfigurierbar macht.

Teilweise wurde außerdem das User Interface über selbst geschriebene Editor-UI ausgebaut, mehr hierzu in Abschnitt [3.7](#).

3.3 Ordnerstruktur

TODO: Namespaces!

Die Ordnerstruktur des Projektes ist auf der höchsten Ebene in drei Ordner unterteilt: EasyButtons, ReplaceDungeonGenerator und Game.

3 Implementierung

EasyButtons ist ein Package, dass das einfache Erstellen von Buttons im Inspektor zum Aufrufen von Methoden des Components erlaubt. TODO: ref zu materialien

ReplaceDungeonGenerator enthält die Kern-Assets des Generators. Dieser Ordner kann in jedes Projekt eingefügt werden. Dieses Package hat lediglich EasyButtons als Dependency. Somit lässt sich der Generator einfach in anderen Projekten einsetzen ohne unnötige Assets wie Modelle, PlayerController, etc. mit zu Importieren.

Game enthält alle Spiel-spezifischen Assets, die nicht Teil des Generator-Kerns sind.

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.

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

3.4 Aufbau des Generator-Objekts

Das wichtigste GameObject der Software ist der „Generator“, dieser ist üblicherweise zusammengesetzt aus folgenden Komponenten: Transform (Bestandteil von jedem GameObject), PatternView, RandomReplacer oder RecipeReplacer, RuleSet, RuleEditor (optional), ReplacementEngine, PatternToPrefabs. Auf diese Komponenten wird in [3.4.1](#) genauer eingegangen.

3.4.1 Erklärung der Komponenten

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

3 Implementierung

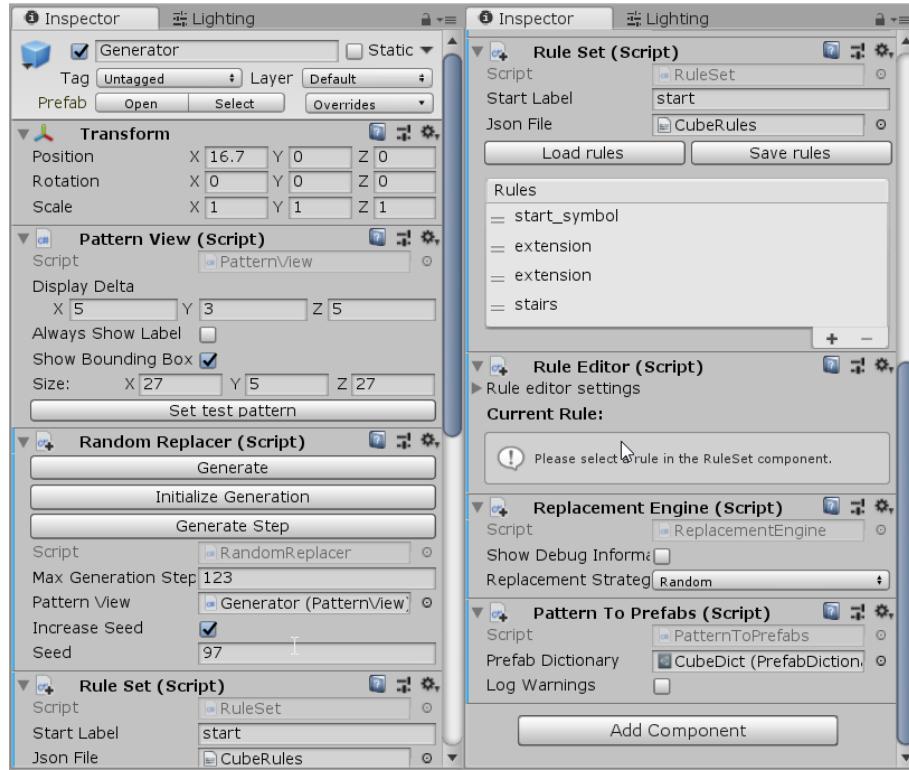


Abbildung 3.3: Screenshots des Inspektors eines Generator-GameObjects, man sieht alle vorhandenen Komponenten und deren Einstellungen

PatternView

Das PatternView repräsentiert einen dreidimensionalen Array von Labels. Diese Labels stehen üblicherweise für einen Raum und eine Rotation. Das Label „st_2“ steht beispielsweise für eine gerades Stück („st“ für „straight“), dass um zwei mal um die y-Achse rotiert wurde. Das PatternView enthält außerdem ein Display Delta, dass bestimmt mit wie viel Abstand die Räume im Raum verteilt sind. Im Scene View¹ von Unity werden die Labels als Text im dreidimensionalen Raum angezeigt (erkennbar in Abb. 3.4).

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

3 Implementierung

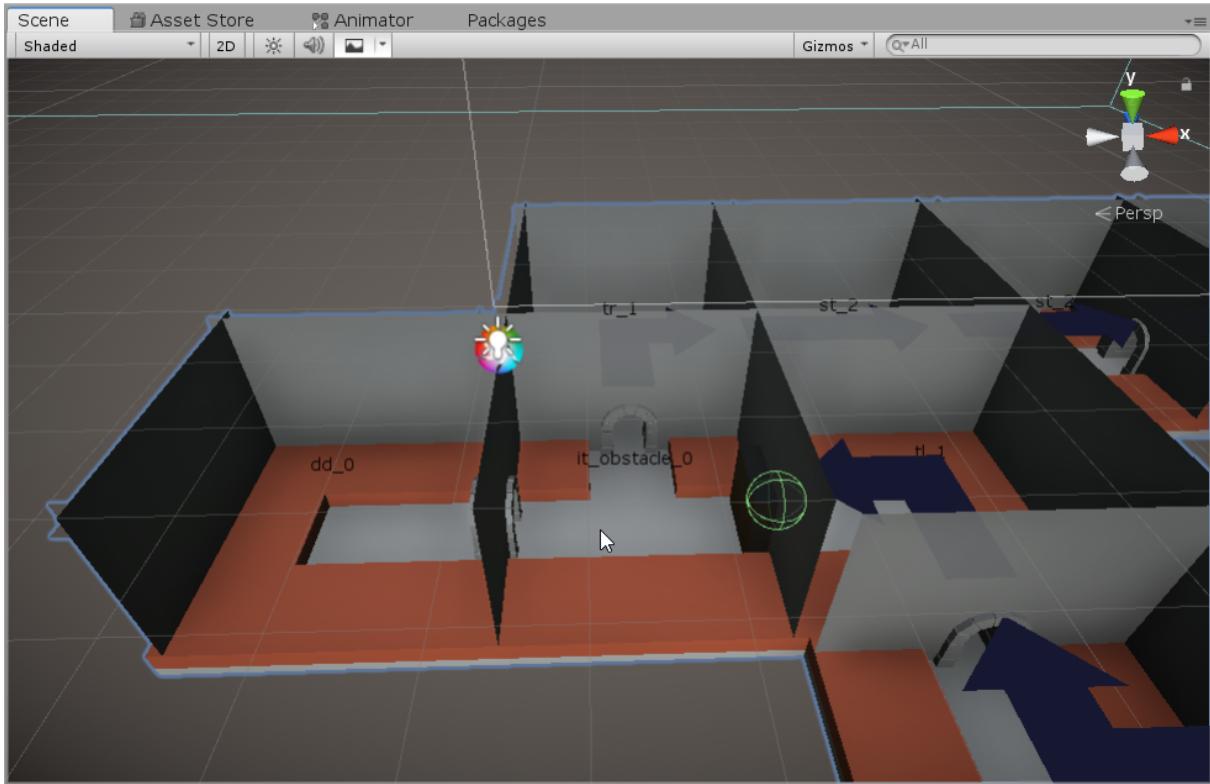


Abbildung 3.4: Sicht auf einen Teil eines generierten Dungeons im Unity Scene View

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.

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.

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

3 Prefabs in Unity: [Uni18b, Seite: LearningtheInterface]

3 Implementierung

Der PatternToPrefabs Component verlangt ein PatternView Component mit RequireComponent⁴

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.

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 generiert 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 Position und Regel für den nächsten Produktionsschritt. Dabei gewichtet er die einzelnen Positionen gemäß der Gewichtung der Regeln im RuleSet.

⁴ RequireComponent in Unity: [[Uni18a](#), Seite: RequireComponent]

3 Implementierung

Der RecipeReplacer verwendet eine Recipe-Datei des Datentypen TextAsset⁵. In der Text-Datei ist festgelegt in welcher Reihenfolge die Regeln aus RuleSet angewandt werden. Das genaue Format des Recipes ist in [3.9](#) beschrieben.

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 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.

RuleEditor

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

Ist im RuleSet eine Regel ausgewählt, enthält der RuleEditor-Inspektor die Einstellungen für genau diese Regel. Es können Parameter wie Name und Gewichtung der Regel gesetzt werden. Außerdem eine maximale Anzahl an Anwendungen dieser bestimmten Regel und eine Anzahl an „Wait Steps“. 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 instinkтив nicht zwischen Himmelsrichtungen unterscheiden, und somit die meisten Regeln um die

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

3 Implementierung

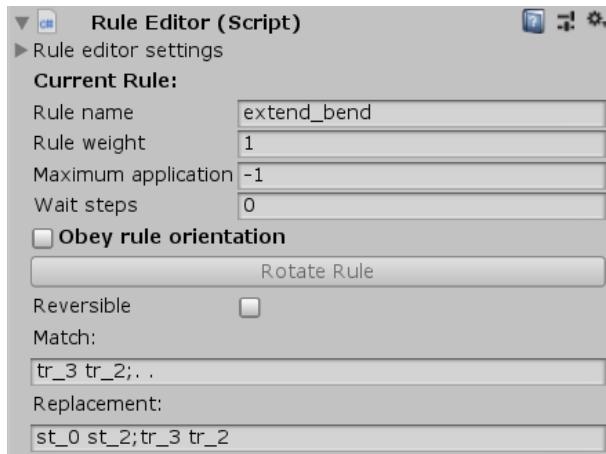


Abbildung 3.5: Inspektor des RuleEditor-Components

y-Achse rotiert auch angewendet werden können. Ist der toggle angeschaltet kann die rotation auch per Knopfdruck geändert werden.

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 Eingabemethode nicht trivial. Es wurde 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).

Dieses Format ist nicht wirklich intuitiv, wurde aber aus Gründen von Aufwands/Nutzen-Erwägungen so belassen. Eine mögliche Verbesserung findet sich in 4.4.

3 Implementierung

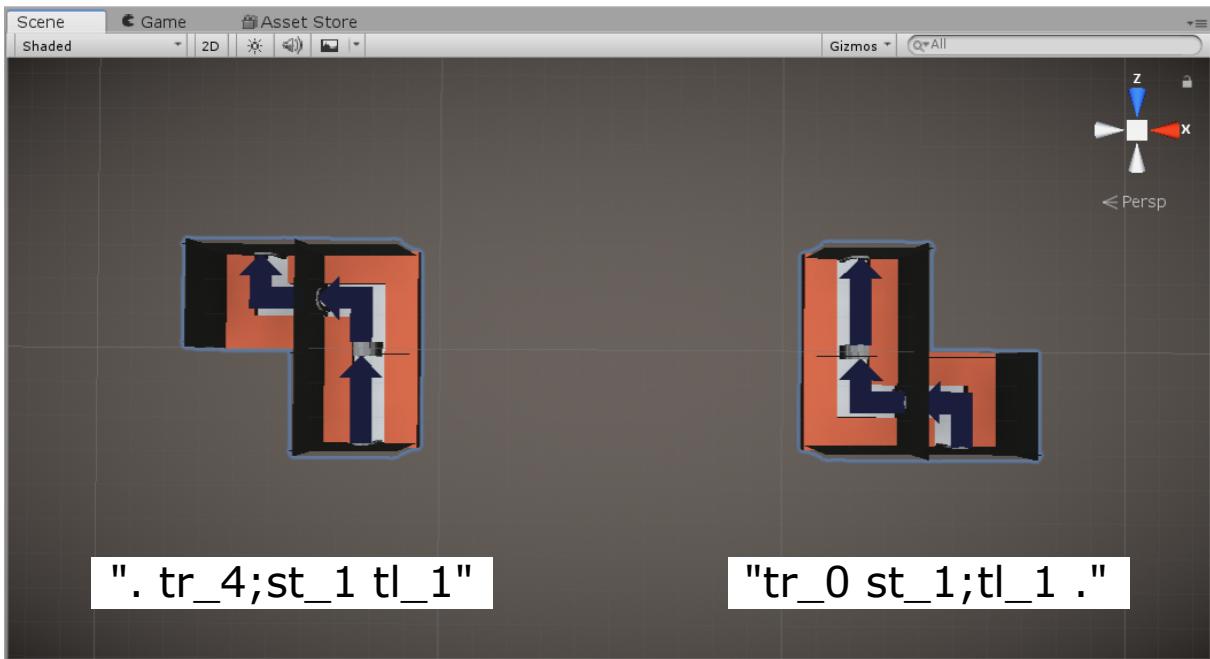


Abbildung 3.6: Beispiel einer Produktionsregel in Unity mit linker und rechter Seite und den dazugehörigen String-Darstellungen. Die verwendeten Symbole sind „st“ (straight), „tl“ (turn left) und „tr“ (turn right). Die Pfeile bezeichnen die vorgesehene Begehrungsrichtung und sind nur im Scene View sichtbar.

3.5 Wichtige Klassen

Es gibt auch einige Klassen, die keine Components, aber trotzdem erwähnenswert sind. Die Klassen Tile, Pattern und Rule 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`).

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

3 Implementierung

3.5.2 Pattern

Das Pattern stellt einen dreidimensionalen Array von Tiles dar. Diese Wrapper-Klasse erledigt einige Aufgaben. Zum einen implementiert diese Klasse 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 gibt es die Methode RotatePattern90Y, die eine um 90° um die y-Achse rotierte Kopie des Patterns zurückgibt. Diese Methode wird benutzt um die in [3.4.1](#) 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.

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 ([3.4.1](#): PatternView).

3.5.3 Rule

Die Klasse Rule implementiert eine einzelne Produktionsregel. Wie schon im theoretischen Teil beschrieben besteht eine Produktionsregel aus einer linken und rechten

3 Implementierung

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 ein paar häufig verwendet Funktionen beinhaltet, die zu keiner bestimmten anderen Klasse zugehörig sind.

Sie enthält verschiedene Funktionen für Vector3Int: Clamping, Bounds-Checking, Bounds-Überschneidung und Iteration über ein einen dreidimensionales Gitter.

Außerdem 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 Implementierung

3.6 Lazy updating, suchen

3.7 Editor UI

3.8 Regelsystem

3.9 Recipe-Format

3.10 Modelle

4 Auswertung der generierten Dungeons

Die Qualität der generierten Dungeons soll im Folgenden bewertet werden um das Verfahren zu beurteilen.

4.1 Einschränkungen

Es ist wohl 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 quasi unmöglich, bzw. wenig aussagekräftig macht.

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

TODO: Verweis auf Ausblick o.ä.

4 Auswertung der generierten Dungeons

4.2 qualitative Auswertung

4.3 quantitative Auswertung

4.4 Mögliche Verbesserungen

4.5 Mögliche Anwendungsszenarien

4.5.1 Roguelike, Rogue-lite

4.5.2 Minigolf

4.5.3 Text adventure

5 Zusammenfassung

6 Verwendete Materialien

LATEX-Vorlage von Martin Bretschneider

<https://www.bretschneidernet.de/tips/thesislatex-vorlagen.html#vorlagen>, 2006. –
letzter Zugriff: 08.01.2019

TODo: Easybuttons

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: 27.12.2018
- [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: 08.01.2018
- [Uni18b] Unity Technologies: *Unity User Manual*.
<https://docs.unity3d.com/Manual/index.html>, 2018. – letzter Zugriff: 08.01.2018

Bildquellenverzeichnis

[Log15] Logos, Dyson: *Map of the fictional Warrek's Nest dungeon.*

https://commons.wikimedia.org/wiki/File:Warrek%E2%80%99s_Nest.jpg,
2015. – letzter Zugriff: 25.12.2018

7 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