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

1 Einführung in das Thema

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 designte 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, das ein oder mehrere Spieler navigieren müssen, häufig mit einer großen Herausforderung am Ende, z.B. einem besonders starken Boss-Monster.

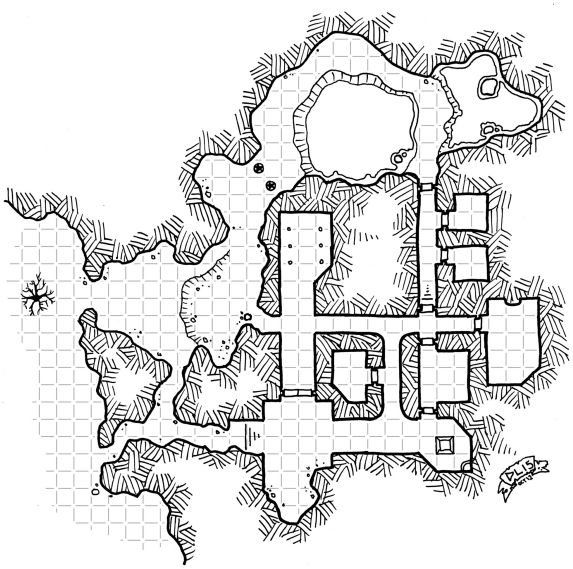

Abbildung 1.1: Beispiel eines Dungeons [Log15]

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

1.2.3 Prozedurale Dungeon-Generierung in Spielen, Geschichte

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

TODO: Research Rougelikes, 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.

Ein Dungeon-Generator is 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. 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 dabei durchaus subjektiv, 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 Generierung, z.B. indem kontinuierlich Räume hinzugefügt werden. Die Räume können

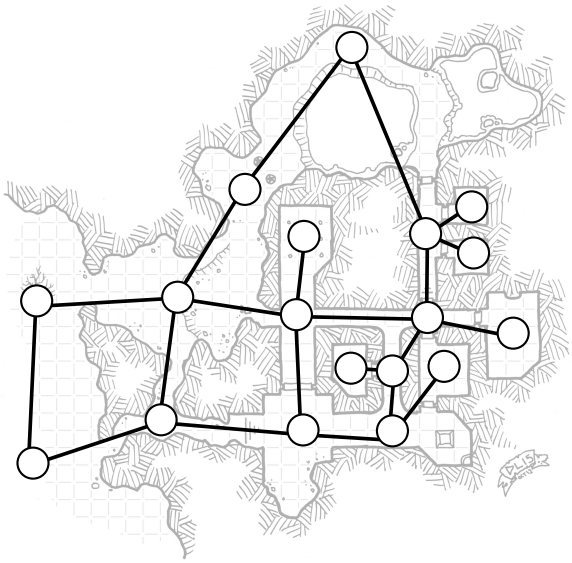

Abbildung 1.2: Bild eines Dungeons, mit möglichem Level-Graph, der Graph ist ungewichtet und ungerichtet ([Log15], bearbeitet)

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.

1.3.2 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.

1.4.1 Entity/Component

1.4.2 Editor Scripting

2 Erklärung des 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 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 widerholte 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 Modelle von Level-Designern häufig als isomorphisch angenommen werden, dabei argumentiert er, dass sie als separate Modelle verstanden werden sollten.

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.

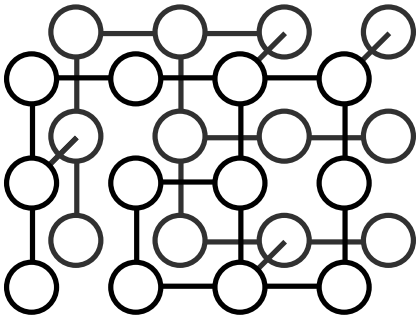

Abbildung 2.1: Beispiel für einen dreidimensionalen Graph innerhalb der Restriktionen des Gitters

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

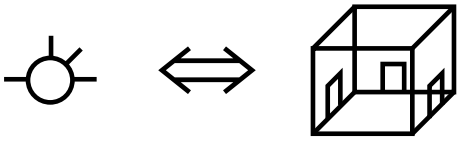

Abbildung 2.2: Knoten mit Halb-Kanten und äquivalenter Raum

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 (sechs Ausgänge, mit jeweils zwei Zuständen, „Tür“ oder „keine Tür“), durch geschicktes nutzen von Rotationssymmetrie könnte man

diese Zahl verringern. Dennoch ist dies nicht ideal, da ein Level-Designer trotzdem eine Mindestzahl an Räumen entwerfen müsste.

TODO: Ausführen, dass der Designer auch Variationen erstellen will, was sich multipliziert

Alternativ könnte man auch die einzelnen Räume prozedural generieren. Dieser Ansatz wurde ausgetestet, aber nicht weiter verfolgt, da die Räume dadurch zu gleich aussahen.
TODO: Screenshot von VoxelRooms

Die Lösung dieses Problems ist die Umformulierung des Modells. Anstatt einen Level-Graphen zu generieren, und dann Räume zuzuordnen, werden die Räume direkt produziert.

Dies bedeutet, dass keine Graphgrammatik im eigentlichen Sinne Verwendet wird sondern stattdessen eine „Gittergrammatik“.

TODO: Formale Definition Gittergrammatik

2.3 Allgemeiner Algorithmus

3 Implementierung

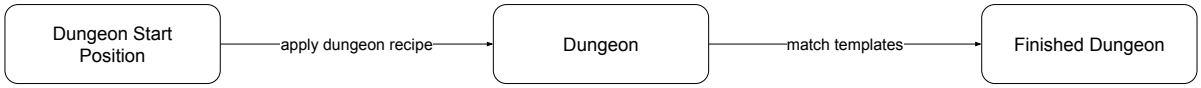

Abbildung 3.1: Test-Bild mit langer Bildunterschrift

3.1 Tool-Charakter der Implementierung

3.2 Wichtige Klassen

TODO: Tile, Pattern, ReplacementEngine, ...

3.3 Die Pattern-Datenstruktur

3.4 Lazy updating, suchen

4 Auswertung der generierten Dungeons

4.1 Einschränkungen

TODO: Subjektivität, Spiele-spezifisch

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
http://www.bretschneidernet.de/tips/thesislatex.html, 2006. – letzter Zugriff: 15.10.2018

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

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