Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Algorithmische Erzeugung eines Dungeon Crawler Levels

Procedural Generation of a Dungeon Crawler Level

Benedikt Pischinger

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr.-Ing. Christoph Lürig

Trier, den 20.08.2020

Vorwort

An dieser Stelle möchte ich der Hochschule Trier, meinem Betreuer Prof. Dr.-Ing. Christoph Lürig und generell den Professoren des Fachbereichs Informatik danken. Durch die COVID-19-Pandemie stellte sich 2020 als durchaus turbulentes Jahr heraus. Der schnelle und kompetente Wechsel auf Online-Lehre für das Sommersemester 2020 hatte es mir dennoch ermöglicht mich auch während dieser Krise auf mein Studium zu konzentrieren und relativ sorgenfrei an dem vorliegenden Werk zu arbeiten.

Kurzfassung

In der vorliegenden Arbeit wird mit Hilfe der Unity Engine ein Prototyp und eine Reihe von Skripten entwickelt, welche ein Dungeon Crawler Level algorithmisch erzeugen. Ziel ist es mit jedem Starten des Spiels ein neues Level mit einer zufälligen Karte zu generieren und dieses mit zufällig verteilten Gegner Typen und Hindernissen zu bestücken. Zunächst werden die generellen Ansätze und der theoretische Hintergrund zur algorithmischen Erzeugung diskutiert und mit dem in der Arbeit verwendeten Ansatz verglichen. Im weiteren Verlauf der Arbeit wird noch betrachtet welche Probleme sich aus den theoretischen Ansätzen ergeben haben, und wie diese bei der Umsetzung gelöst wurden. Die Arbeit schließt ab mit einem Ausblick auf die Anwendungsmöglichkeiten des entstandenen Prototypen und das Potenzial dessen Verwendung, um ein vollständiges Spiel zu entwickeln.

The following Paper is about the development of a prototype using the Unity Engine and several scripts, which can be used to algorithmically generate a dungeon crawler level. The purpose is to have a new, randomly generated dungeon layout each time the game is started and to have it randomly filled with different types of monsters and hazards. At first the general idea and theoretical approach of algorithmic content generation is discussed and compared to the implementation in this work. Next the difficulties that had to be dealt with, as well as the approach of their solutions, are examined. The theoretical section is followed by the technical aspects of the projects, such as the software and Assets that were used in the creation of the prototype. The paper concludes with an assessment of the potential usage of the developed scripts in a full fledged video game.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1

Einleitung und Problemstellung

Über das letzte Jahrzehnt ist das algorithmische Generieren von Inhalten für Videospiele in der Spieleindustrie immer beliebter geworden. Hierbei werden verschiedene Aspekte eines Videospieles von einem Algorithmus generiert, anstatt von einem Entwickler per Hand integriert. Anwendungsmöglichkeiten sind fast grenzenlos. So wurde diese Methodik zum Beispiel schon im Jahr 1991 in dem rundenbasierten Strategiespiel *Civilization* benutzt, um eine zufällige zweidimensionale Karte zu generieren [Wikc]. Das „Sandbox“ Spiel *Minecraft* generiert eine fast endlose dreidimensionale Spielwelt, komplett mit thematisch verschiedenen Binomen und Dungeons [Wikc]. Die Verwendung solcher Algorithmen sind aber nicht nur limitiert auf die Erzeugung von Spielwelten. Das „Survival Horror“ Spiel *Left 4 Dead* benutzt einen Algorithmus, um die statische Spielwelt mit algorithmisch generierten Wellen von Zombies zu bevölkern [Wikc] und in dem Action-Rollenspiel *Borderlands* werden Waffen in Millionen verschiedener Kombinationen generiert [Wikc].

Nun stellt sich natürlich die Frage, warum man sich als Entwickler mit der Algorithmischen Erzeugung auseinander setzten sollte. Zum einen kann bei der Spielentwicklung viel Zeit und Geld gespart werden. Da der Algorithmus einen großen Teil des Spiels generieren kann und somit nicht jedes einzelne Element von Hand platziert werden muss, können Ressourcen entweder gespart oder anderweitig verwendet werden. Ein weiterer Grund ist die Erhöhung des Wiederspielwerts. Eine statische Spielerfahrung ist in meisten Fällen vollkommen in Ordnung, besonders wenn ein Entwickler deren spezifische Vollstellung des Spiels produzieren will. Im Normalfall ist ein solches Spiel aber in dutzenden von Stunden durchgespielt und der Spieler hat selten einen Grund sich nochmals mit dem Spiel auseinander zu setzen. Der Wiederspielwert ist hier also recht gering. Manche Spiele machen von einem sogenannten „neues Spiel+“ Modus Gebrauch. Hierbei handelt es sich um das exakt selbe Spiel, behält aber seinen Charakterfortschritt aus dem ersten Durchlauf bei, während Gegner oftmals stärker und Items seltener werden. Allerdings noch nicht für Jeden Grund genug einen zweiten Durchlauf des Spieles zu starten. Das Konzept der algorithmischen Erzeugung kann aber benutzt werden, um den Wiederspielwert eines Spieles immens zu erhöhen. Besonders das Genre des „Roguelikes“ und „Roguelites“ profitieren stark von algorithmischer Erzeugung von Spielinhalten. Hierbei handelt es sich oft um ein Dungeon Crawler Spiel,

in welchem der Tod des Spielers permanent ist. Bei Versagen beginnt das Spiel von vorne. Wäre die Spielwelt also statisch würde ein solches Spiel recht schnell langweilig werden. Nicht nur, weil bei jedem Durchlauf in der selben Welt und gegen die selben Gegner gespielt wird, sondern auch, weil der Spieler mit jedem Durchlauf lernt, wo sich Items und andere Schätze befinden. Durch ein zufällig generiertes Spielerlebnis bietet jeder Start des Spiels allerdings eine neue Herausforderung, sowie Erfahrungen. Kein Durchlauf ist exakt wie der Vorherige und somit kann zum Beispiel ein eher simpler Dungeon Crawler, wie *Binding Of Isaac: Rebirth* [Wikc] für hunderte von Stunden immer wieder erneut gespielt werden (siehe Abb. 1.1), ohne genug von dem Spiel zu bekommen. Aus diesen Gründen ist das Konzept der algorithmischen Erzeugung unter Indie Entwicklern, welchen die Finanzierung oder Erfahrung für großangelegte Spielentwicklung fehlt, sehr beliebt [SA17]. In diesem Sinne war das Ziel dieser Arbeit ein Dungeon Crawler Level algorithmisch zu erzeugen. Die aus dem Prototypen resultierenden Skripte können vielseitig in Unity Projekte eingebunden und benutzt werden, um Spielinhalte zufällig zu generieren und den Spieler somit immer mit einer neuen Herausforderung zu unterhalten.

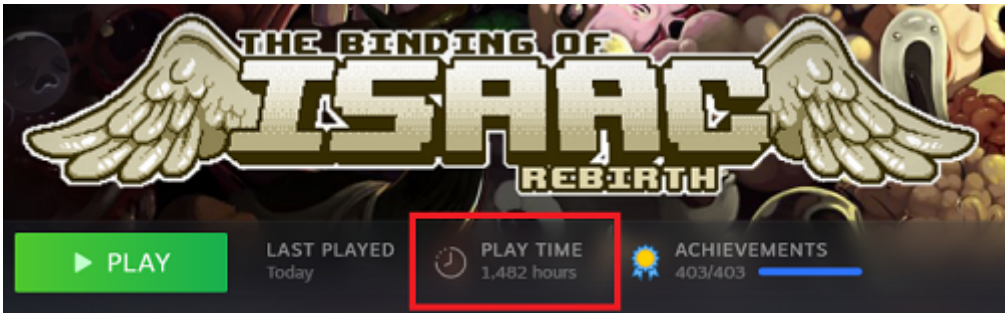

Abb. 1.1. *Binding Of Isaac: Rebirth* Spielzeit des Autors

2

Software & Assets

2.1 Programme

Zur Erstellung des im Folgenden dokumentierten Prototypen wurde die Unity Game Engine in der Version 2019.3 verwendet, da für das Projekt sehr gut von dem Scene Manager Gebrauch gemacht werden konnte. Die Skripte für tatsächliche algorithmische Erzeugung wurden in der Programmiersprache C-Sharp unter Benutzung von Visual Studio 2019 Community Edition geschrieben.

2.2 Assets

Zur graphischen Repräsentation des Dungeons wurden mehrere Asset Packs aus dem Unity Asset Store verwendet.

- **Ultimate Low Poly Dungeon v1.0** wurde benutzt für die Darstellung der Räume, aus welchen sich der Dungeon zusammensetzt. Es wurden hier spezifisch Prefabs von Wänden und Bodenplatten verwendet, aus denen die einzelnen Dungeon Räume aufgebaut sind.
- **Four Evil Dragons Pack HP v1.2** wurde für die Repräsentation von Gegnern in den Dungeon Räumen verwendet. Prefabs von verschiedenen Arten von Drachen werden während dem Ablauf des Algorithmus in die Dungeon Räume geladen.
- **Dungeon Traps v2.0** wurde für die Repräsentation von Hindernissen innerhalb der Dungeon Räume verwendet. Die Prefebs werden ebenfalls während des Ablaufs des Algorithmus geladen.
- **GUI Parts v1.0.** Aus diesem Asset Pack wurden nur Sprites verwendet, um Items innerhalb des Dungeons darzustellen, so wie ein Sprite für die Visualisierung eines Rasters auf dem Raumboden, wo Gegner geladen werden können.

3

Theoretische Ansätze

Dieses Kapitel beschäftigt sich mit den theoretischen Ansätzen der algorithmischen Erzeugung in Videospielen und deren Benutzung im *Dungeon Crawler* Genre. Es wird zunächst die grundlegende Idee betrachtet, welche Herausforderungen bei der Umsetzung entstehen und wie mit diesen umgegangen wird.

3.1 Algorithmische Erzeugung in Videospielen

Bei der algorithmischen Erzeugung in Videospielen geht es darum Teile eines Videospiels mit limitierter, oder indirekter Eingabe des Entwicklers zu erzeugen [STN16]. Dieses Konzept kann auf endlos viele Aspekte eines Spiels angewandt werden. Von der Erzeugung ganzer Spielwelten, einzelnen Leveln, Charakteren, Items, Waffen, Fahrzeugen, bis zu Aufgaben für den Spieler usw. Somit kann während der Entwicklung Zeit und Geld gespart werden, da nicht jeder Aspekt des Spiels von Hand kreiert werden muss. Die generelle Idee hinter einem solchen Algorithmus ist, dass verfügbare Elemente zufällig gewählt und aneinandergereiht werden, bis ein gewünschtes Ziel erreicht ist. In der Realität ist es leider nicht ganz so einfach. Jedes Verfahren sollte vordefinierten Regeln folgen, um eine glaubhafte Spielwelt zu erzeugen. Diese Regeln sind abhängig vom Spieldesign und müssen von dem Entwickler vor Beginn der algorithmischen Erzeugung festgelegt werden [SA17]. Hierbei kann es sich um logische Regeln handeln, wie Wasser sollte nicht einfach aus der Luft heraus laufen, oder Bäume sollten nicht aus Steinen heraus wachsen. Es kann sich aber auch um Regeln handeln, welche vom Spieldesign gefordert werden. Insbesondere die Spielziele müssen erreichbar sein. Der Spieler sollte nicht plötzlich auf eine Wand, welche den Weg komplett versperrt stoßen, oder auf Gegner, die noch nicht besiegt werden können, treffen.

3.2 Verfahren zur algorithmischen Erzeugung

Algorithmische Erzeugung ist nicht gleich Algorithmische Erzeugung. Generell kann unterschieden werden zwischen einer “prozeduralen“ und einer “zufälligen“ Erzeugung von Spielinhalten. Diese beiden Begriffe werden heutzutage oftmals als Synonym für einander benutzt, können aber auf zwei verschiedene Ansätze deuten. Für ein weiteres Verfahren kann künstliche Intelligenz (KI) benutzt.

3.2.1 Prozedurale Erzeugung

Unter der prozeduralen Erzeugung wird die Erzeugung von komplett neuen Spielinhalten verstanden. Hierbei wird ein Algorithmus mit einer detaillierten Liste von Anweisung von dem Entwickler geschrieben, welche bestimmen, wie der zu generierende Spielinhalt auszusehen hat. Das Spiel generiert dann die besagten, originalen Inhalte, basierend auf den Anweisung und zuvor festgelegten Regeln. Im Beispiel von *Spelunky* (ein 2D *Platoformer* in dem Höhlen erkundet werden) wurde hier zunächst vom Entwickler Derek Yu ein System entwickelt, welches definiert, was eine Höhle alles beinhalten sollte, sowie eine Reihe von festen und zufälligen Elementen. Das Spiel setzt dann alle Inhalte zusammen und generiert so jedes Level [Yu11]. Ein weiteres und wahrscheinlich bekannteres Beispiel für die prozedurale Erzeugung ist *Minecraft*. Der Algorithmus fängt hier zunächst mit einem *Seed* an. Hierbei handelt es sich um eine Folge aus Zahlen und/oder Buchstaben. Auf diesem *Seed* werden anschließend eine Reihe von vordefinierten mathematischen Rechenanweisung ausgeführt und das Resultat wird nun benutzt, um einen sogenannten *Perlin-Noise* (eine fraktale Rauschfunktion) zu generieren (siehe Abb. 3.1). Wie zu sehen ist hat der *Perlin-Noise* eine gewissen Struktur und ist nicht komplett zufällig. Der Wert jedes Pixels ist relativ zum Wert des vorhergegangenen Pixels. Somit kann es nicht passieren, dass ein stark weißer Pixel direkt neben einem komplett schwarzen Pixel liegt. Der entstandene *Perlin-Noise* kann nun dreidimensional visualisiert werden. Desto heller der Pixel, desto höher liegt er und umgekehrt (siehe ABb. 3.2). Im Fall von *Minecraft* wird dieser *Perlin-Noise* nun mit Blöcken gefüllt, um Abschnitte des Levels zu erzeugen. Auf diese Art und Weise werden in dem Spiel massive Welten, ohne großen Aufwand von Seiten des Entwicklers kreiert und können frei von den Spielern erkundet werden [V’a20]. Die Verwendung eines *Seeds* hat weiterhin den Vorteil, dass ein spezifischer *Seed* immer zum selben Ergebnis führt. Somit muss zum Beispiel nicht die gesamte Spielwelt, sondern nur der Fortschritt des Spielers, sowie der *Seed*, auf der Festplatte gespeichert werden und Spieler können ihre Welten untereinander austauschen.

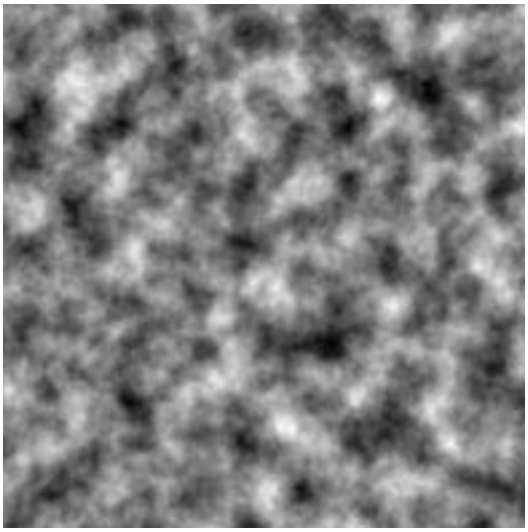

Abb. 3.1. *Perlin-Noise* [V'a20]

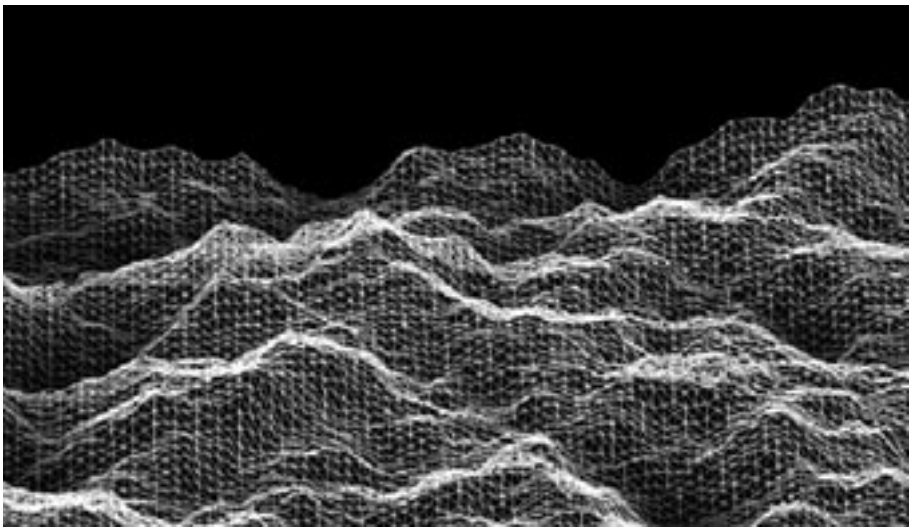

Abb. 3.2. *Perlin-Noise* visualisiert in 3D [V'a20]

3.2.2 Zufällige Erzeugung

Zufällige Erzeugung ist eine weitaus einfachere Methode. Hier werden Abschnitte des Spiels von den Entwicklern von Hand kreiert und anschließend mit Hilfe eines Algorithmus in zufälligen Kombinationen zusammengesetzt. Bei diesen Spielinhalten kann es sich um ganze Levelabschnitte, oder nur einzelne Räume handeln. Im Beispiel von *Path of Exile* (ein Action-Rollenspiel) wird zunächst eine Vielzahl von Räumen per Hand kreiert. In den Räumen werden anschließend Punkte definiert, an denen weitere Elemente während der Levelerzeugung zufällig platziert werden können. Anschließend generiert der Algorithmus einen zufälligen Pfad durch eine leere Karte und entlang diesen Pfades werden dann Räume platziert, um das Level zu generieren. Abschließend werden noch Gegner und andere Gegenstände auf den zuvor erwähnten Positionen platziert, um die Erzeugung des Levels abzuschließen

[Rog11]. Anders als bei der prozeduralen Erzeugung können bei dieser Methode mit genug Spielzeit einzelne Räume und Elemente wiedererkannt werden, auch wenn zunächst jeder Durchlauf anders aussieht. Desto mehr Arbeit aber in die Vielfalt der Räume gesteckt wird, desto besser kann die Illusion der Originalität des Spielinhaltes aufrecht gehalten werden. Die zufällige Erzeugung kann auch mit der prozeduralen Erzeugung kombiniert werden. In *Diablo* (ebenfalls ein Action-Rollenspiel) zum Beispiel sind Abschnitte, welche relevant für die Story sind fest eingebaut, während die Abschnitte zwischen diesen Storypunkten prozedural erzeugt werden.

3.2.3 Erzeugung mit künstlicher Intelligenz

Ein weiteres Verfahren der algorithmischen Erzeugung kann mit der Benutzung von künstlicher Intelligenz (KI), in diesem Fall neuronalen Netzen, umgesetzt werden. Kurz gesagt handelt es sich hierbei um ein vereinfachtes Modell eines Gehirns, komplett mit Neuronen und Synapsen. Diesem Modell kann nun durch das Einspeisen von Daten beigebracht werden, Muster nicht nur zu erkennen, sondern auch vorauszuberechnen [Wika]. Bei den besagten Daten kann es sich auch um Spielinhalte handeln. Das Netz würde so lernen, wie ein Level für ein Spiel aussehen muss und könnte dann Spielinhalte, basierend auf diesem Wissen erzeugen. Im Fall des Spiels *Fantasy Raiders* stellte sich das Trainieren eines solchen neuronalen Netzes zum Kreieren von spielbaren Leveln allerdings als extrem aufwendig heraus und erforderte oftmals noch den Eingriff der Entwickler, um brauchbare Ergebnisse zu erzielen [SP18]. Die Erzeugung von einem einfacheren Beispiel, wie einer 3D Welt sehr ähnlich zu *Minecraft*, konnte allerdings schon erfolgreich umgesetzt werden, ist aber dennoch mit einem hohen Aufwand verbunden [BC18]. Generell scheint es so, dass neuronale Netze noch mehr Fortschritt machen müssen, bevor sie ein brauchbares Werkzeug der Spielentwicklung werden. Valve wiederum hat in ihrem Survival-Horror Spiel *Left 4 Dead* eine einfache künstliche Intelligenz benutzt, um die statische Spielwelt, mit prozedural Erzeugten Wellen von Zombies zu füllen, während die Spieler sich durch das Level bewegen. Für diesen Zweck behält die KI die Spieler ständig im Überblick und definiert eine Fläche um die Spieler, in dem Zombies platziert werden können. In dieser Fläche schließt die KI Bereiche aus, die potentiell sichtbar für die Spieler sind und platziert dann Wellen basierend auf einem *Structured Unpredictability* System, welches nicht komplett zufällig, aber auch nicht deterministisch eindeutig ist. Es werden unter anderem der verfügbare Raum und Intervalle für Zombie-Platzierung, sowie die Frequenz mit denen bestimmte Zombies auftauchen in Betracht gezogen und basierend auf diesen Information Wellen erzeugt [Boo08].

3.3 Ansatz für die Umsetzung des Dungeon Crawler Levels

Bei einem *Dungeon Crawler* geht es darum, dass der Spieler sich durch eine Art Kerker navigiert, währenddessen Gegner bekämpft, Puzzles löst und Schätze fin-

det. Wie der Name schon andeutet befindet sich der Spieler also oftmals in einer geschlossenen Umgebung [Wikb]. Dies limitiert nicht nur das artistische Design, sondern auch das Spieldesign generell. Die Umgebung ist häufig kerker- oder höhlen-artig, der Raum innerhalb des *Dungeons* ist limitiert und somit ist auch die Anzahl und Art der Gegner limitiert. Auf Grund dieser Limitierung klingt das Konzept, sich durch einen öden Kerker zu kämpfen recht schnell als langweilig. Werden nun die etlichen Spiele, welche heutzutage im *Dungeon Crawler* Genre existieren betrachtet, so wird schnell festgestellt, dass meistens eine zufällige Erzeugung von Spielinhalten verwendet wird, damit Durchläufe des Spiels frisch und spannend bleibend.

3.3.1 Ansatz der Dungeon Karte

Wie oben erwähnt ist die Umgebung eines *Dungeons* oft sehr ähnlich zu einem Kerker und diese bestehen oftmals aus einer Vielzahl von kleinen Räumen. Aus diesem Grund wurde für die Entwicklung des Prototypen die zufällige Erzeugung aus Kapitel 3.2.2 gewählt. Die Idee ist es also, eine Reihe von Räumen zu entwickeln, welche mit Hilfe eines Algorithmus zufällig angeordnet und anschließend zufällig mit Gegnern und anderen Hindernissen bevölkert wird. Außerdem sollte jedes Level einen Itemraum haben, in dem der Spieler Schätze finden kann, sowie einen Boss, der besiegt werden muss, um das Level abzuschließen.

Wie in 3.3.1 erwähnt muss die algorithmische Erzeugung bestimmten Regeln folgen. Wird nun ein klassischer *Dungeon Crawler* betrachtet, so wird es schnell klar, wie diese Regeln für den Prototypen aussehen müssen, um einen funktionierenden Spieldurchlauf zu garantieren:

1. Räume dürfen nicht ineinander geladen werden.
2. Räume müssen neben einem schon existierenden Raum geladen werden, um einen zusammenhängenden Dungeon zu garantieren.
3. Räumen brauchen einen Eingang und einen Ausgang.
4. Es muss genug Räume für die benötigten Tätigkeiten des Spiels geben. Wenn der Spieler zum Beispiel drei Räume überleben, ein Item finden und einen Boss besiegen muss, um das Level zu beenden, dann müssen neben dem Startraum mindestens fünf weitere Räume existieren.
5. Es müssen Passagen zwischen den Räumen existieren, damit auch der komplette Dungeon durchquert werden kann.

Der Ansatz für den Algorithmus zum Erzeugen der Karte des Dungeons sieht also so aus: Zunächst wird eine zufällige Zahl gerollt, welche zwischen dem Minimum und Maximum der gewünschten Raumanzahl liegt. Als nächstes wird ein Startraum platziert. Basierend auf dem Startraum wird nun eine zufällige Richtung gewählt und ein weiterer Raum in dieser Richtung platziert. Dieser Prozess wird dann für den Start und alle anderen Räume wiederholt, bis die anfänglich gerollte Anzahl erreicht ist. Am Ende des Algorithmus werden noch Passgen zwischen den Räumen erzeugt, ein Itemraum und ein Bossraum platziert. Basierend auf diesem Ansatz könnte dann die folgende Dungeon Karte entstehen (siehe Abb. 3.3)

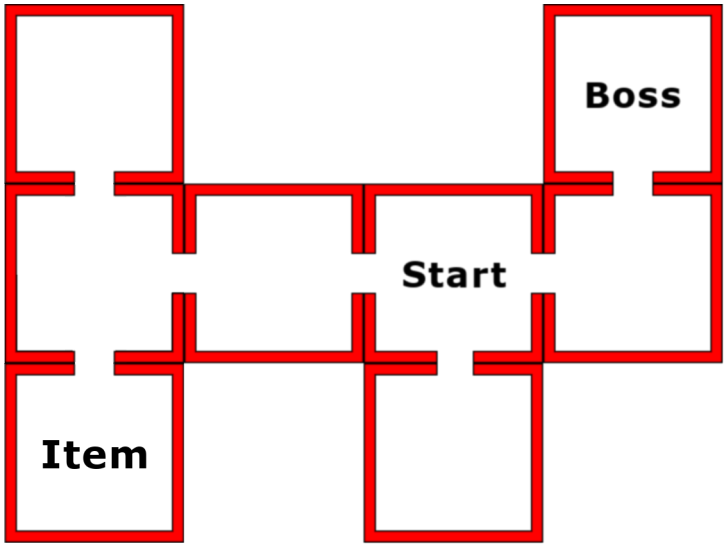

Abb. 3.3. Illustration eines möglichen Dungeon Levels

3.3.2 Ansatz der Bevölkerung des Dungeons

Das platzieren von Gegnern und Hindernisse muss ähnlicher weise bestimmten Regeln folgen. Der Spieler sollte nicht einfach von zu vielen und zu starken Gegnern überwältigt werden. Der Algorithmus muss hier also verschiedene Aspekte des Spieldesigns berücksichtigen.

1. Gegner und Hindernisse dürfen nicht ineinander platziert werden
2. Es muss ein Limit an möglichen Gegnern und Hindernissen für jeden Raum geben
3. Starke Gegner sollten nicht zu oft in einem einzelnen Raum vorkommen

Der Ansatz für diesen Teil des Algorithmus sieht also wie folgt aus: Zunächst wird ein Raster über jeden Raum gelegt, welches definiert wo und wie viele Gegner oder Hindernisse in jedem gegebenen Raum platziert werden können. Anschließend werden zufällige Zahlen für die Anzahl an Gegnern und Hindernissen gerollt. Freie Felder auf dem Raster werden nun mit Gegnern und Hindernissen bestückt. Zusätzlich muss die Chance für schwächere Gegner höher sein, als die für starke Gegner. Basierend auf diesem Ansatz könnte ein Raum nach dem Ablauf des Algorithmus wie in der folgenden Abbildung bestückt sein (siehe Abb. 3.4).

Abb. 3.4. Illustration eines möglichen Raumes, wobei grüne Felder frei und rote Felder bestückt sind

4

Architektur

Dieses Kapitel beschäftigt sich mit der tatsächlichen Umsetzung der algorithmischen Erzeugung des Dungeon Crawler Levels. Es werden zuerst die Prefabs betrachtet, welche das Dungeon Level in der Unity Engine repräsentieren und im Anschluss die technische Umsetzung der algorithmischen Erzeugung mit Hilfe der Skripte, welche im Rahmen dieser Arbeit entwickelt wurden, erklärt.

4.1 Visuelle Repräsentation

4.1.1 Dungeon Raum

Die Räume des Dungeons werden aus Boden und Wand Assets aus dem *Ultimate Low Poly Dungeon* Asset Pack aufgebaut. Ein Raum besteht aus 8 individuellen Wänden, sowie vier zusätzlichen Wand Assets welche als Passage dienen und 25 individuellen Bodenplatten. Im vorliegenden Prototyp sind die Räume vordefiniert als Prefab (siehe Abb. 4.1) und werden in einer Szene gespeichert, um sie als Teil des Dungeons laden zu können.

4.1.2 Gegner

Gegner sind repräsentiert durch 4 verschiedene Arten von Drachen aus dem *Four Evil Dragons Pack HP* Asset Pack. Diese sind individuell als Prefab (siehe Abb. 4.2) gespeichert und werden bei Bedarf in die Szene geladen.

4.1.3 Hindernisse

Hindernisse werden in der Form Bodenfallen, bestehend aus individuellen Assets aus dem *Dungeon Traps* Asset Pack, dargestellt und sind ebenfalls als Prefab 4.3 im Projekt verfügbar, um bei Bedarf in die Szene geladen werden zu können.

4.1.4 Items

Items wiederum sind durch simple Sprites aus dem *GUI Parts* Asset Pack dargestellt, und wie die vorherigen Elemente als Prefabs gespeichert.

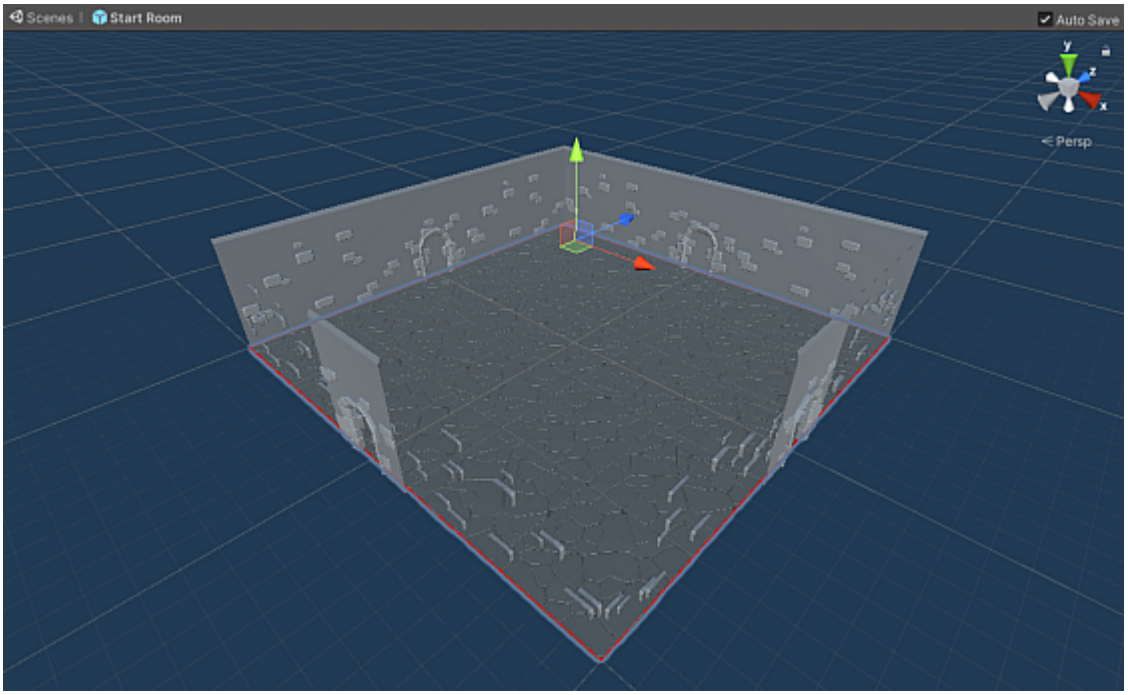

Abb. 4.1. Prefab eines Dungeon Raums

Abb. 4.2. Prefab der Gegnertypen

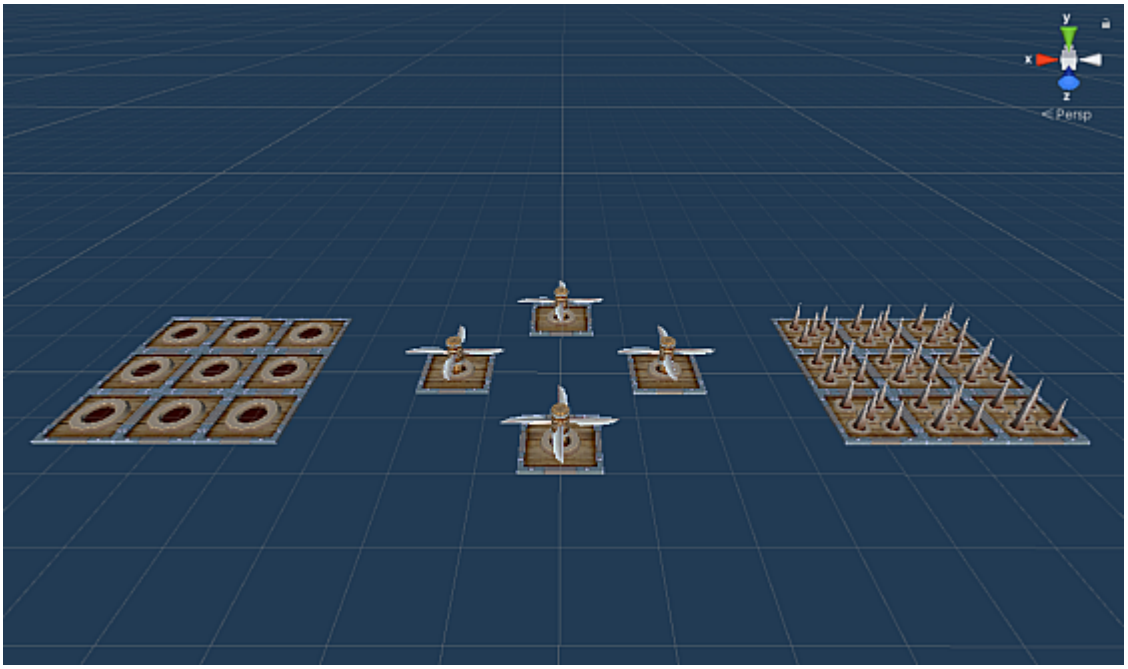

Abb. 4.3. Prefab der Fallen, welche als Hindernisse dienen

Abb. 4.4. Prefab der Items

4.2 Umsetzung der algorithmischen Erzeugung

4.2.1 Erzeugung der Dungeon Karte

Im Kern des Dungeon Levels befindet sich zunächst ein *ScriptableObject* welches ein *DungeonGenerationData Object* erzeugt. Dieses bestimmt die Größe des zu erzeugenden Levels und wie viele Item Räume in dem Level auftauchen sollen (siehe Abb. 4.8). In diesem *ScriptableObject* wird festgelegt wie viele Dungeons generiert werden und wie viele Itemräume das Level haben soll. Das Iterations-Minimum und Maximum bestimmt wie oft der Algorithmus durchlaufen wird. Ein einzelner Dungeon mit einem Minimum von 10 und einem Maximum von 20 Iterationen würde also eine Karte mit 10 bis 20 Räumen produzieren. Desto größer ein einzelner Dungeon ist, desto weiter wächst er in jede Richtung. Soll also ein mehr konzentrierter Dungeon erzeugt werden, so können zum Beispiel zwei Dungeons mit nur 5 bis 10 Durchläufen generiert werden.

Im weiteren Verfahren wird von Unitys *SceneManager* Gebrauch gemacht. Die einzelnen Räume des Dungeons werden individuell in einer eigenen Szene gespeichert (siehe Abb. 4.14). Während des Ablaufs des Algorithmus werden diese Szenen additiv aneinander gehängt, um das Level zu erzeugen. In der *DungeonMain* Szene, welches die Hauptszene des Levels repräsentiert, ist hierzu das *DungeonGenerator*, sowie das *RoomController* Skript mit Hilfe eines GameObjects eingebunden und mit der *DungeonGenerationData* verlinkt (siehe Abb. 4.7). Das *DungeonGenerator* Skript initialisiert den *DungeonController* und startet somit die Erzeugung des *Dungeons*.

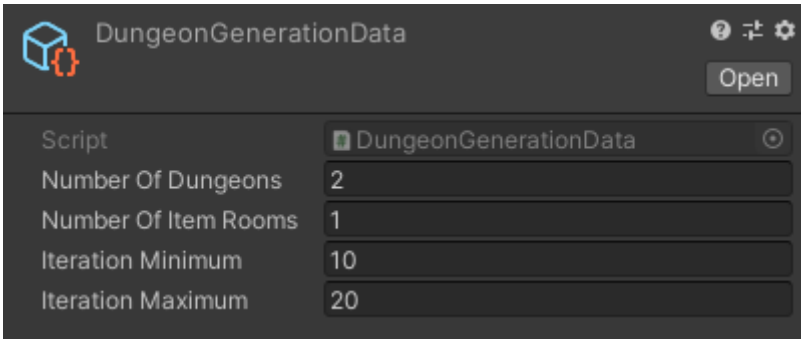

Abb. 4.5. DungeonGenerationData in Unity

Der *DungeonController* legt im nächsten Schritt die Richtungen fest, in denen Räume geladen werden können. Da der komplette Dungeon sich auf einer Ebene befindet, können Räume in Richtung der X und Z-Achse existieren (siehe Listing 4.1). Die *DungeonCrawler* Klasse initialisiert eine Startposition für den Dungeon und wählt zufällige Richtungen aus dem zuvor definierten Dictionary für die Erzeugung der Dungeon Karte aus. Der *DungeonController* legt mit der Random.Range()-Methode fest wie viele Iterationen zwischen dem in der *DungeonGenerationData* festgelegten Minimum und Maximum durchgeführt werden sollen

Abb. 4.6. Szenen mit verfügbaren Räumen

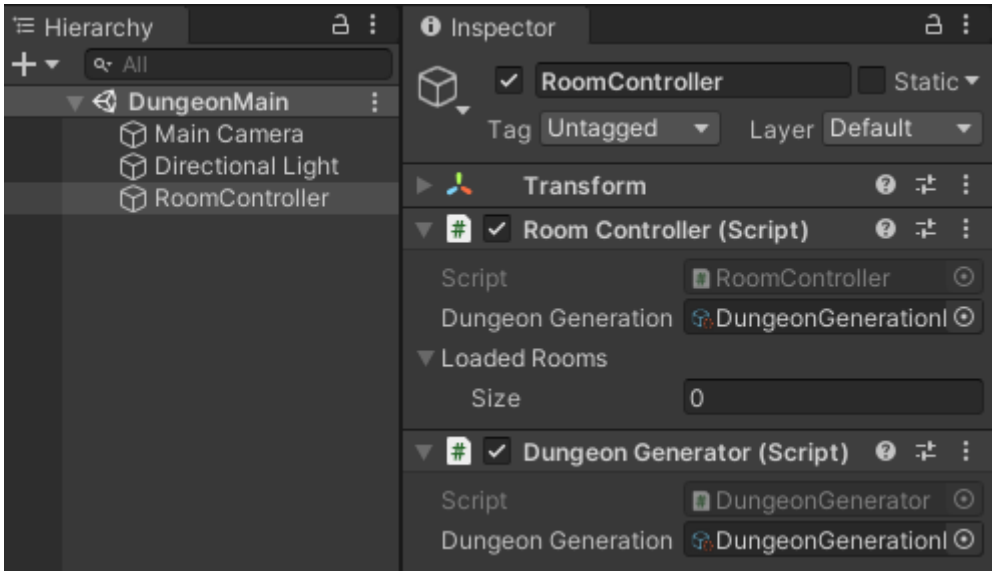

Abb. 4.7. GameObject mit eingebundenen Skripten und verlinkter Data

und bestimmt somit wie viele Räume der Dungeon am Ende des Algorithmus haben wird.

```
1
2  // Iterations for the dungeon generation based on the direction map
3  public class DungeonController : MonoBehaviour
4  {
5      private static readonly Dictionary<Direction, Vector3Int> directionMap
6          = new Dictionary<Direction, Vector3Int>
7      {
8          {Direction.xLeft, Vector3Int.left },
9          {Direction.xRight, Vector3Int.right },
10         {Direction.zUp, new Vector3Int (0, 0, 1)},
11         {Direction.zDown, new Vector3Int(0, 0, -1)}
12     };
13 }
```

Listing 4.1. Ausschnitt aus dem *DungeonConroller* Skript

Die letzten beiden Skripte, welche für die Erzeugung gebraucht werden sind das *Room* Skript und der dazugehörige *RoomController*. *Room* definiert einen Raum mathematisch, sprich die Dimensionen des Raums, sowie dessen Position. Das Skript ist außerdem für die Platzierung von Passagen zuständig, hierzu aber später mehr.

Der *RoomController* enthält eine Sammlung an Szenen mit verfügbaren Räumen und ist nun für das Laden von zufälligen Räumen, basierend auf den vorangegangenen Metriken, zuständig (siehe Listing 4.2). Da jeder Raum in einer eigene Szene ausgelagert ist, können hier Räume nach Bedarf einfach zur Liste hinzugefügt werden. Eine *RoomQueue* im *RoomController* enthält die Rauminformationen der zu ladenden Räume und wird in der Update()-Methode jede Frame aktualisiert. Solange sich Elemente in dieser Queue befinden werden Räume geladen und danach aus der Queue entfernt. Das Laden der Räume geschieht mit Hilfe einer Coroutine (LoadRoomRoutine()-Methode), welche parallel zur Füllung der *RoomQueue* abgearbeitet wird. Wie zuvor in Kapitel 3.3.1 erwähnt muss der Algorithmus gewissen Regeln folgen. Der Einhalt dieser Regeln wird ebenfalls von der *RoomController* Klasse gewährleistet. Um sicher zu gehen, dass Räume nicht ineinander geladen werden, prüft die Klasse mit einer CheckForOverlap()-Methode beim Laden eines neuen Raumes, ob an der selben Position schon ein Raum existiert. Sollte dies nicht der Fall sein, wird der neue Raum zu einer Liste von erfolgreich geladenen Räume hinzufügt und ansonsten wieder zerstört (siehe Listing 4.3). Wie in Abbildung 4.8 zu sehen ist, haben nach 25 Iterationen, also potenziell 25 Räumen (ausschließlich dem Start, Boss und Itemraum) nur 11 Räume die CheckForOverlap()-Methode überlebt. Dies demonstriert, wie häufig das Überlappen von Räumen geschieht, weshalb derartige Problemfälle abgefangen werden müssen.

```
1  // Returns a random room from a list of available scenes
2  public string GetRandomRoom()
3  {
4      string[] possibleRooms = new string[]
5      {
6          "StandardRoom",
7          "MonsterRoom"
8      };
9
10     return possibleRooms[Random.Range(0, possibleRooms.Length)];
11 }
```

Listing 4.2. Liste der verfügbaren Raumszenen aus dem *RoomController* Skript

```
1  // Checks if a room already exists at a position and destroys it if necessary
2  public void CheckForOverlap(Room room)
3  {
4      if(!DoesRoomExist(currentRoomData.x, 0, currentRoomData.z))
5      {
6          room.transform.position = new Vector3(currentRoomData.x
7          * room.width, 0, currentRoomData.z * room.depth);
8
9              room.roomX = currentRoomData.x;
10             room.roomY = currentRoomData.y;
11             room.roomZ = currentRoomData.z;
12
13             room.transform.parent = transform;
14
15             loadingRoom = false;
16
17             loadedRooms.Add(room);
18     }
19     else
20     {
21         Destroy(room.gameObject);
22         loadingRoom = false;
23     }
24 }
```

Listing 4.3. CheckForOverlap-Methode aus dem *RoomController* Skript

Abb. 4.8. Raumliste nach 25 Iterationen

Nachdem der *RoomController* nun mit dem erfolgreichen Laden aller Räume fertig ist, fehlen noch zwei Schritte, um die Erzeugung des Dungeons abzuschließen. Zum einen benötigt das Level einen Bossraum. Hierbei wird lediglich der letzte Raum, welcher geladen wurde, gewählt und wird durch einen Bossraum ersetzt. Das Laden des Itemraums funktioniert parallel. Mit dem Unterschied, dass hier ein zufälliger geladener Raum, ausschließlich dem Startraum und den letzten drei Räumen, mit dem Itemraum ersetzt wird. Zwei Räume vor dem Bossraum bleiben erhalten, um einen Puffer zwischen dem Itemraum und Bossraum zu gewährleisten.

Der letzte Schritt vor Vollendung der Dungeon Karte ist die Platzierung der Passagen zwischen den Räumen. Eine funktionierende Methode für diese Aufgabe zu entwickeln, stellte sich als unerwartet schwierig heraus. Diese Herausforderung konnte aber letztendlich gelöst werden, indem zunächst in jedem Raum sowohl eine Tür, als auch eine Wand an die Stelle der Passagen übereinander platziert werden. Mit Hilfe des *Room* Skripts wird nun jedes Asset mit einem Typ referenziert (siehe Abb. 4.9). Nachdem alle Räume erfolgreich geladen und sowohl der Itemraum, als auch der Bossraum platziert wurde, ruft der *RoomController* eine PlaceDoors()-Methode auf, woraufhin alle vorher definierten Assets in eine Liste geladen werden. Mit einer Reihe von Hilfsmethoden wird dann geprüft, ob ein Raum einen Nachbarn hat oder nicht. Basierend auf dieser Information wird für jedes Asset in der Liste eine Wand oder Tür aktiviert (siehe Listing 4.4). In Zusammenarbeit können dann alle bisher betrachteten Skripte folgende Dungeon Karte erzeugen (siehe Abb. 4.10 und 4.11).

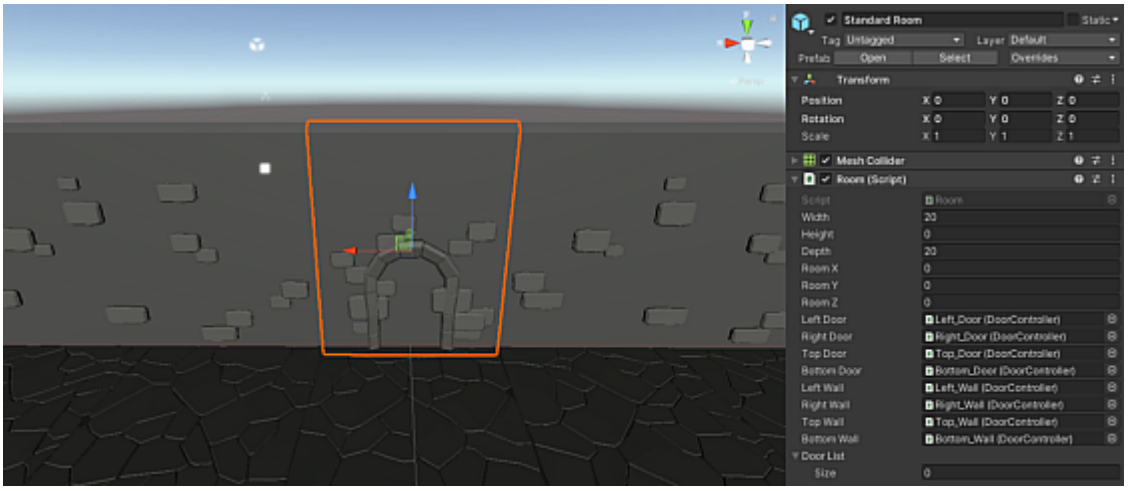

Abb. 4.9. Wände und Türen definiert im Raum Prefab

```
1  // Checks which sides of the room need doors or walls
2  //and activates the GameObjects accordingly
3  public void PlaceDoors()
4      {
5          foreach (DoorController door in doorList)
6          {
7              switch (door.doorType)
8              {
9                  case DoorController.DoorType.leftDoor:
10                     if (GetLeft() == null)
11                     {
12                         door.gameObject.SetActive(false);
13                         break;
14                     }
15
16                 case DoorController.DoorType.leftWall:
17                     if (GetLeft() != null)
18                     {
19                         door.gameObject.SetActive(false);
20                         break;
21                     }
22             }
23         }
24 }
```

Listing 4.4. Gekürzte PlaceDoors()-Methode aus dem *Room* Skript

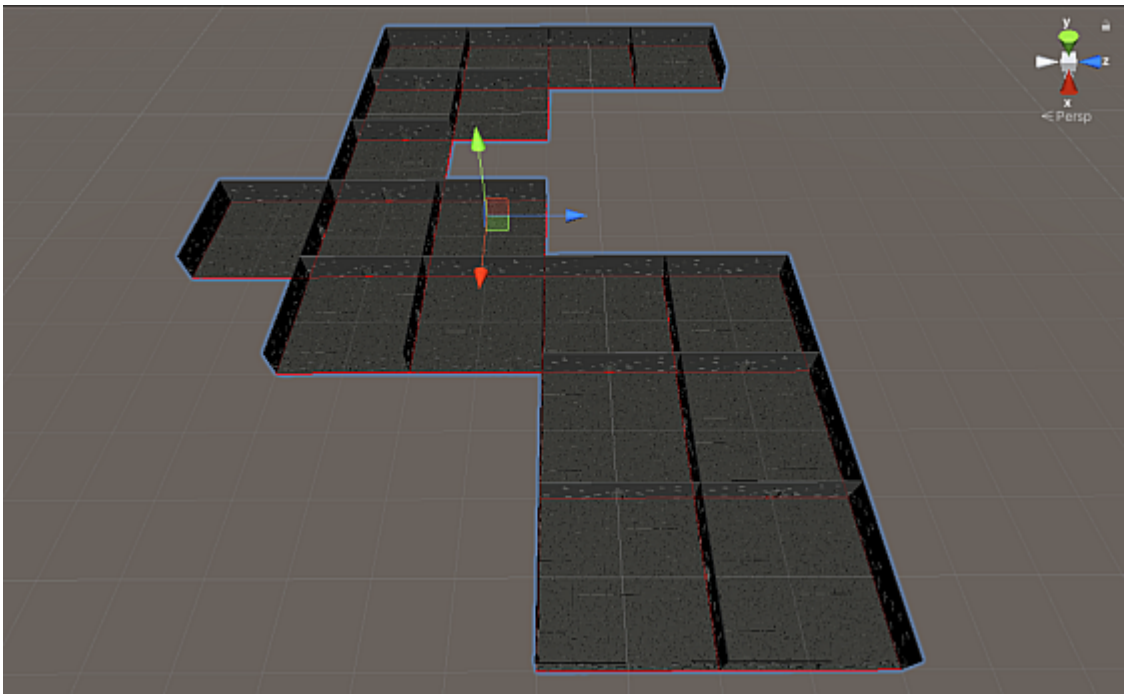

Abb. 4.10. Eine mögliche Dungeon Karte nach 25 Iterationen

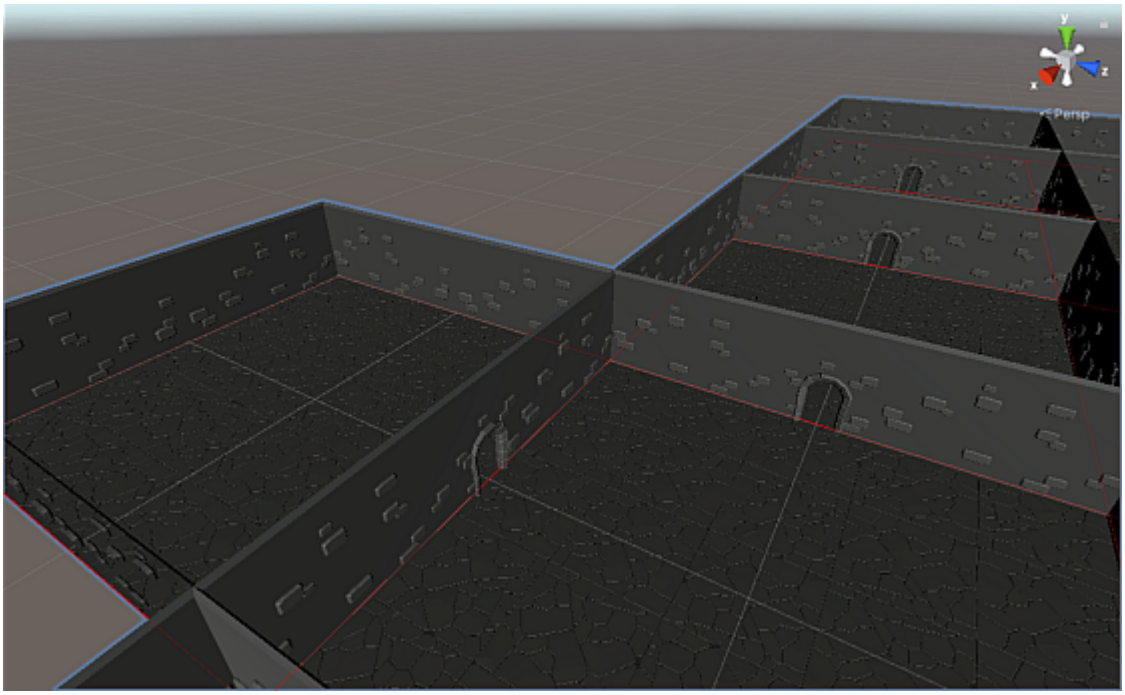

Abb. 4.11. Korrekt platzierte Wände und Türen innerhalb des Dungeons

4.2.2 Bevölkerung

Wie in Kapitel 3.3.2 erwähnt, ist es auch hier wichtig, dass Gegner nicht ineinander, oder auf Bodenfallen geladen werden. Für die Bevölkerung des Dungeon Levels wurde daher zunächst ein *GridController* Skript entwickelt. Dieses Skript legt ein Raster aus Sprites über einen Dungeon Raum. Jedes Sprite repräsentiert dann eine verfügbare Position, auf welcher ein Monster oder ein Hindernis platziert werden kann (siehe Abb. 4.12). In dem Skript werden zunächst die Anzahl der Spalten und Reihen mit Hilfe eines Structs definiert. Der horizontale und vertikale Offset-Wert dienen dazu, das komplette Raster innerhalb eines Raumes zu verschieben. In der Awake()-Methode holt sich das Skript zunächst den spezifischen Raum in dem das Raster ausgelegt werden soll, definiert die gewünschte Anzahl an Spalten und Reihen und startet anschließend die GenerateGrid()-Methode, welche für das Platzieren der Sprites zuständig ist. Die GenerateGrid()-Methode berechnet nun die Position der einzelnen Felder, so dass diese, basierend auf der Größe des Raumes, gleichmäßig verteilt werden. Außerdem wird die Position jeden Feldes in eine Liste von verfügbaren Feldern aufgenommen (siehe Listing 4.5). Die for()-Schleifen sehen zunächst etwas untypisch aus. Der X- und Z-Wert wird hier innerhalb der Schleife benutzt, um den Abstand zwischen den einzelnen Feldern festzulegen. Die Abbruch-Bedingung sollte daher immer ein vierfaches des Inkrements sein, da sonst die Felder nicht richtig platziert werden können. Das Skript selbst kann nun wiederum mit Hilfe eines GameObjects in einen Raum oder eine Szene eingebunden werden (siehe 4.13).

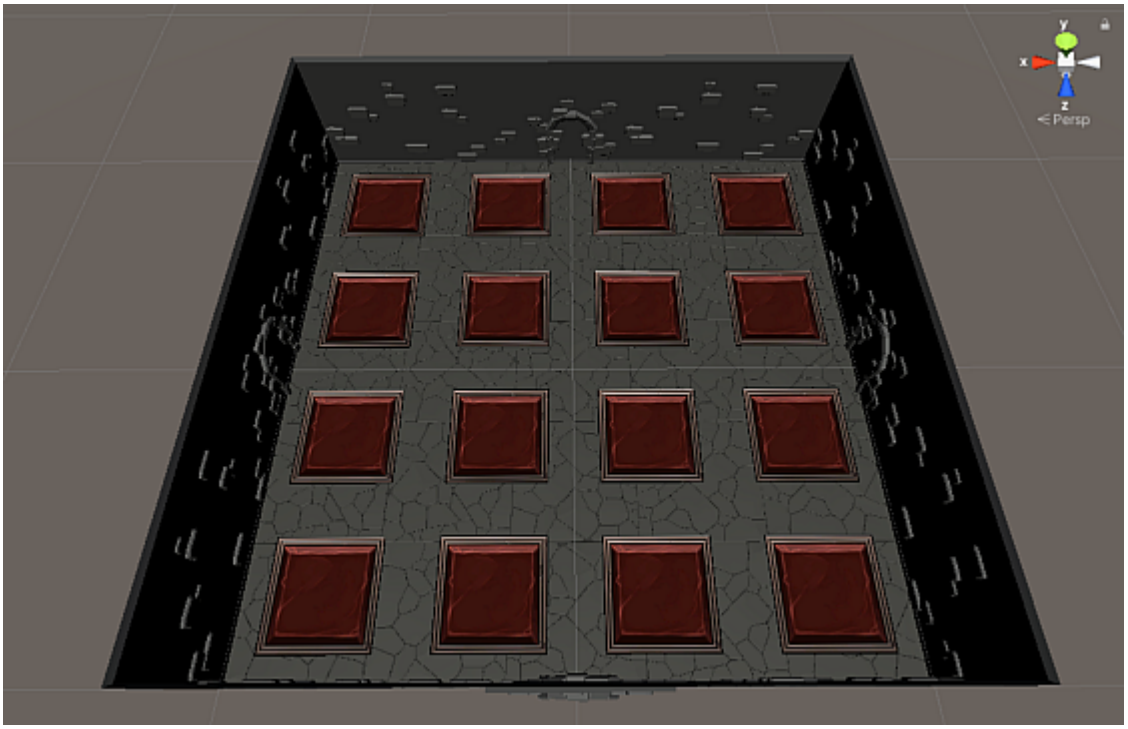

Abb. 4.12. Raster in einem Raum des Dungeons

```
1  // Generates the script by evenly spacing out
2  //and placing tiles based on the room size
3  public void GenerateGrid()
4  {
5      grid.horizontalOffset += room.transform.localPosition.x;
6      grid.verticalOffset += room.transform.localPosition.z;
7
8      for(int x = 0; x < 20; x+=5)
9      {
10         for(int z = 0; z < 20; z+=5)
11         {
12             GameObject gameObject = Instantiate(gridTile, transform);
13
14             gameObject.transform.position
15                 = new Vector3 (x - (grid.columns - grid.horizontalOffset)
16                     , 0.1f, z - (grid.rows - grid.verticalOffset));
17
18             gameObject.name = "x: " + x + ", y: 0, z: " + z;
19
20             availableTiles.Add(gameObject.transform.position);
21
22             gameObject.SetActive(true);
23         }
24     }
25     GetComponentInParent<SpawnerController>().InitialiseObjectSpawning();
26 }
```

Listing 4.5. GenerateGrid()-Methode aus dem *GridController* Skript

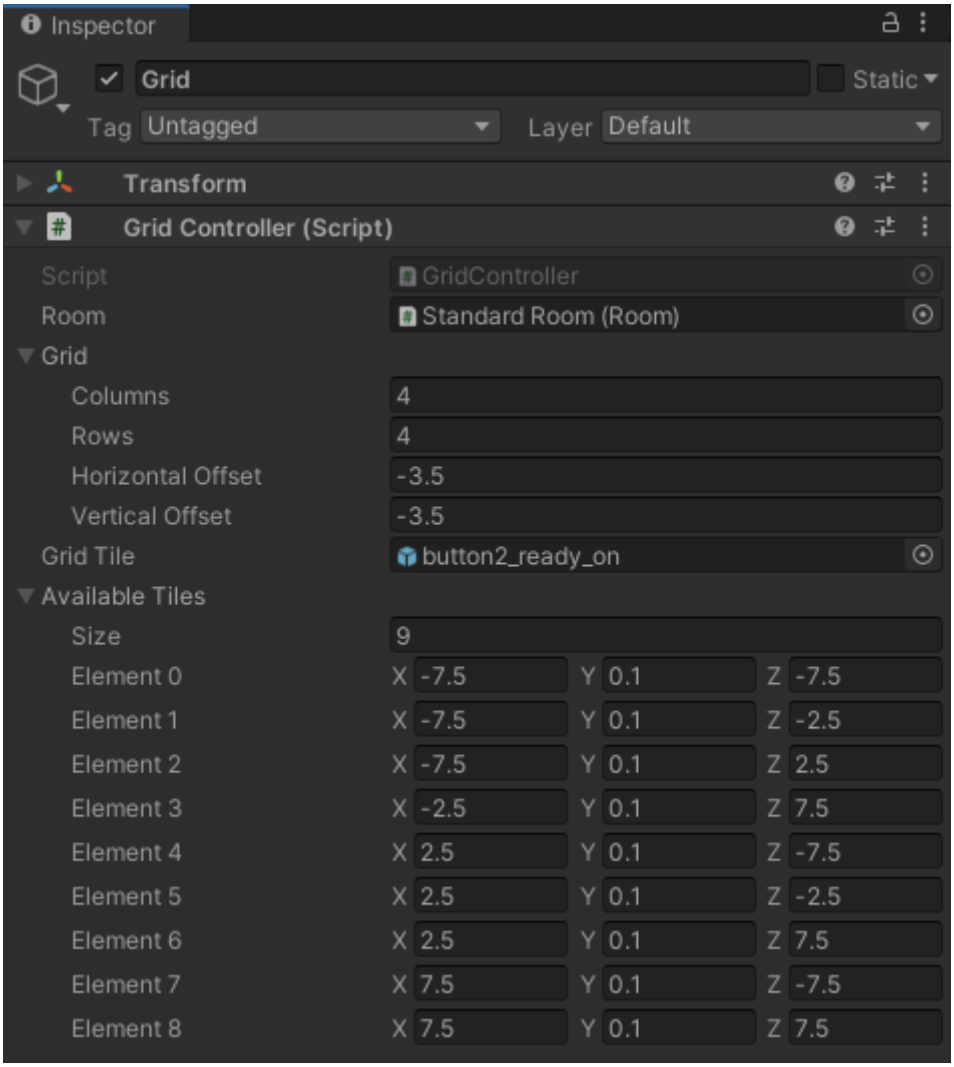

Abb. 4.13. Einbindung des *GridControllers* in einen Raum

Da nun ein Raster mit verfügbaren Positionen existiert, kann begonnen werden Gegner und Fallen in den Raum zu laden. Hierzu wurden *Spawner* kreiert, welche die Platzierung von GameObjects verwalten. Für das komplette Dungeon Level werden insgesamt vier verschiedene *Spawner* verwendet. Da diese aber alle auf die selbe Weise funktionieren, wird im folgenden Teil nur der *MonsterSpawner* betrachtet. Ein *Spawner* besteht aus drei Bausteinen. Die *SpawnerData*, einem *SpawnerController* und dem eigentlichen *Spawner*. Für die *Data* wird zunächst ein weiteres *ScriptableObject* definiert. Dieses enthält das GameObject, welches platziert werden soll, so wie ein Minimum und Maximum der zu platzierenden GameObjects. Mit Hilfe dieses *ScriptableObjects* kann nun eine *MonsterSpawnerData* kreiert werden (siehe Abb. 4.14). Als nächstes wurde eine *SpawnerController* Klasse entwickelt, welche das Raster mit dem *Spawner* verheiratet. Der *SpawnerController* platziert auf einer zufälligen Position aus der Liste der verfügbaren Felder des *GridControllers* einen *Spawner* und entfernt dann das benutzte Feld aus der

Liste. Das *MonsterSpawner* Skript, welches mit dem *SpawnerController* verlinkt wird, verwaltet die Daten der Gegner, welche den Dungeon bevölkern. Für diesen Zweck besitzt das Skript zunächst eine Liste mit den möglichen Gegner, welche in dem Level vorkommen können. Die Prefabs dieser Gegner können mit dem jeweiligen *Spawner* einfach verlinkt (siehe Abb.4.15) und jedem Gegner ein Gewicht zugeordnet werden. Das Gewicht der Gegner dient wiederum zum Kontrollieren der Chancen für jeden Gegnertypen. Wie zuvor in Kapitel 3.3.2 erwähnt sollten Gegner nicht einfach wahllos auftauchen. Abhängig von dem Schwierigkeitsgrad sollten schwache Gegner häufiger und starke Gegner seltener auftauchen. Daher existiert ein *Weight-System* innerhalb des *Spawners*, welches die Chancen für bestimmte Gegner festlegt. Es werden zunächst die Gewichte aller möglichen Gegner zusammen addiert, mit einer zufälligen Zahl zwischen 0.0 und 1.0 multipliziert und in der Variable *weightPool* gespeichert. Anschließend wird durch die möglichen Gegner iteriert, bis die Abbruchbedingung erreicht wird. Der Gegner an der erreichten Indexposition wird dann auf das Feld geladen, auf dem sich der *MonsterSpawner* befindet. Also desto geringer das Gewicht eines Gegners, desto geringer ist die Chance, dass der Gegner geladen wird. Der *SpawnerController* kann nun zu einem Raum hinzugefügt und konfiguriert werden. Hierzu muss lediglich das Raster des jeweiligen Raumes, sowie die benötigten *SpawnerDatas* verlinkt werden (siehe Abb. 4.16). In Abb. 4.17 und 4.18 sind jeweils ein möglicher normaler Raum und ein Bossraum zu sehen, welche von dem *SpawnerController* unter Benutzung aller *Spawner* erzeugt wurden.

Werden nun alle Skripte in Zusammenarbeit benutzt, so kann ein vollständiges zufälliges Dungeon Level, mit Räumen, Gegnern, Fallen, Itemraum und Bossraum generiert werden (siehe Abb. 4.19).

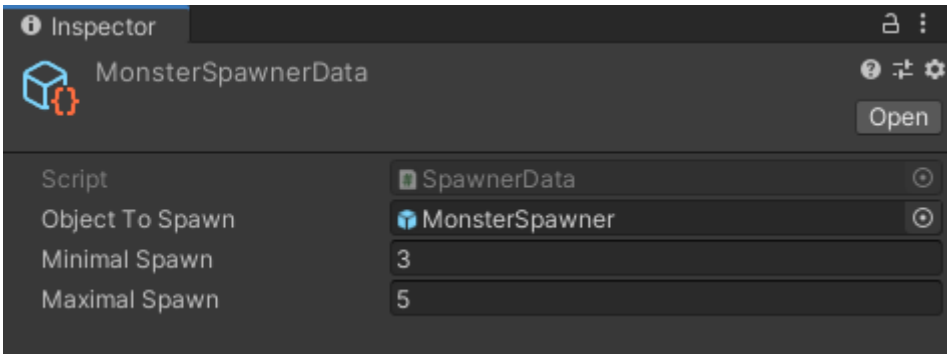

Abb. 4.14. *MonsterSpawnerData* für den *MonsterSpawner*

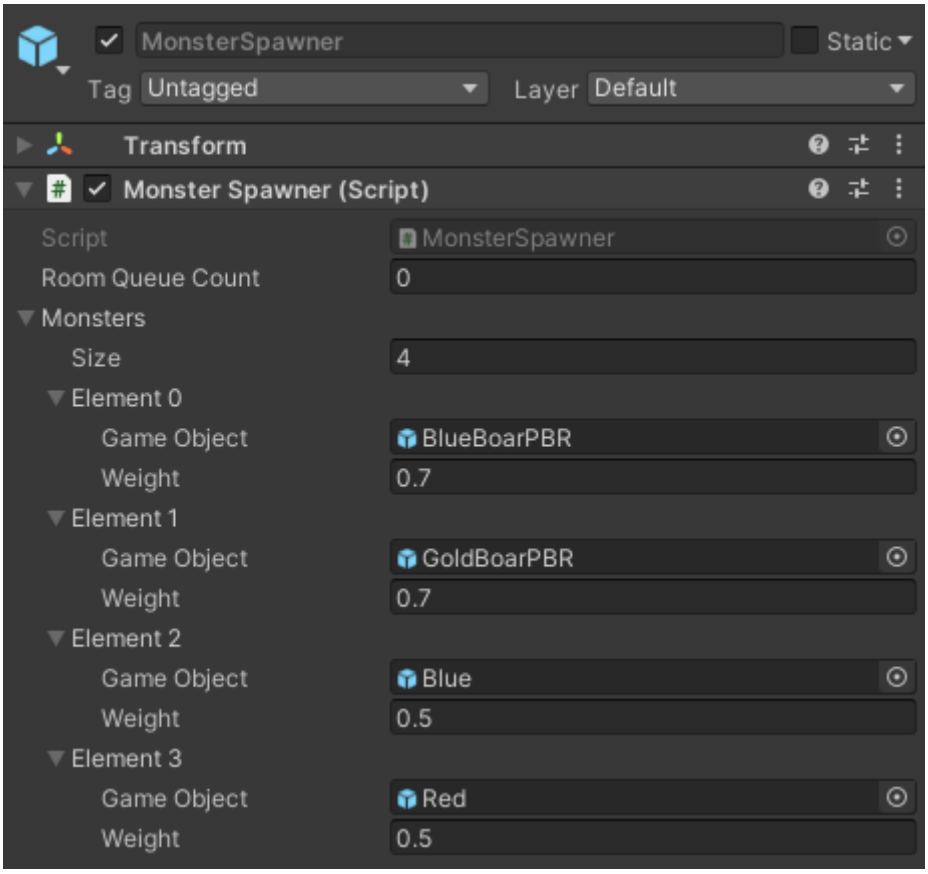

Abb. 4.15. *MonsterSpawner* verlinkt mit den möglichen Gegner-Prefabs

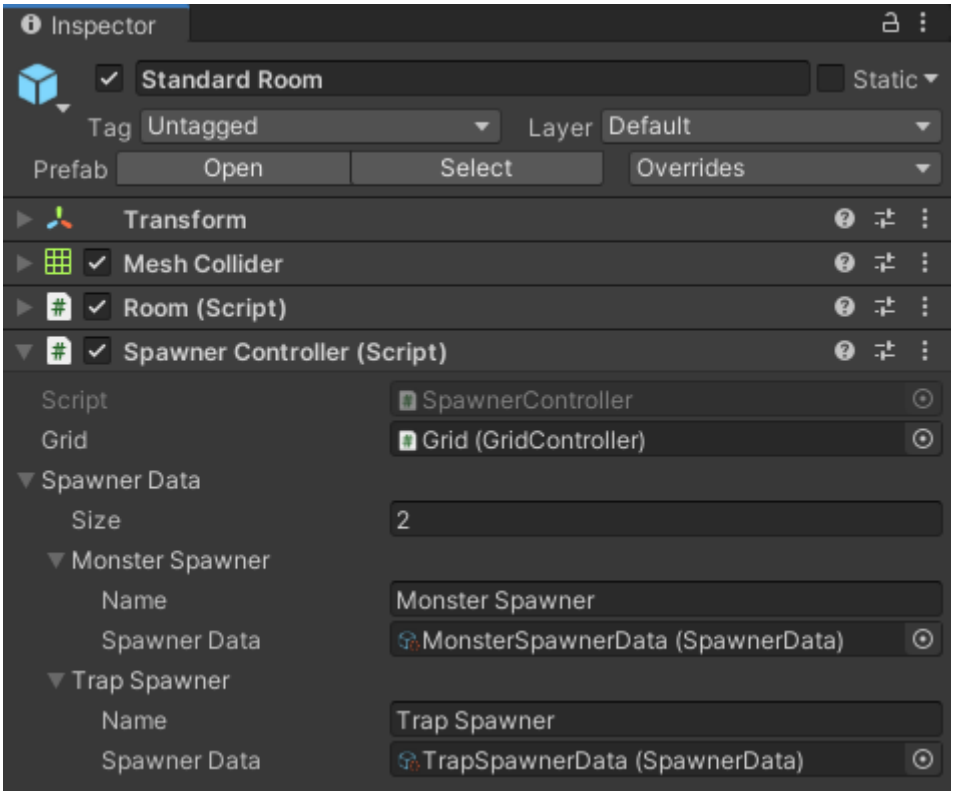

Abb. 4.16. Konfiguration des *SpawnerControllers*

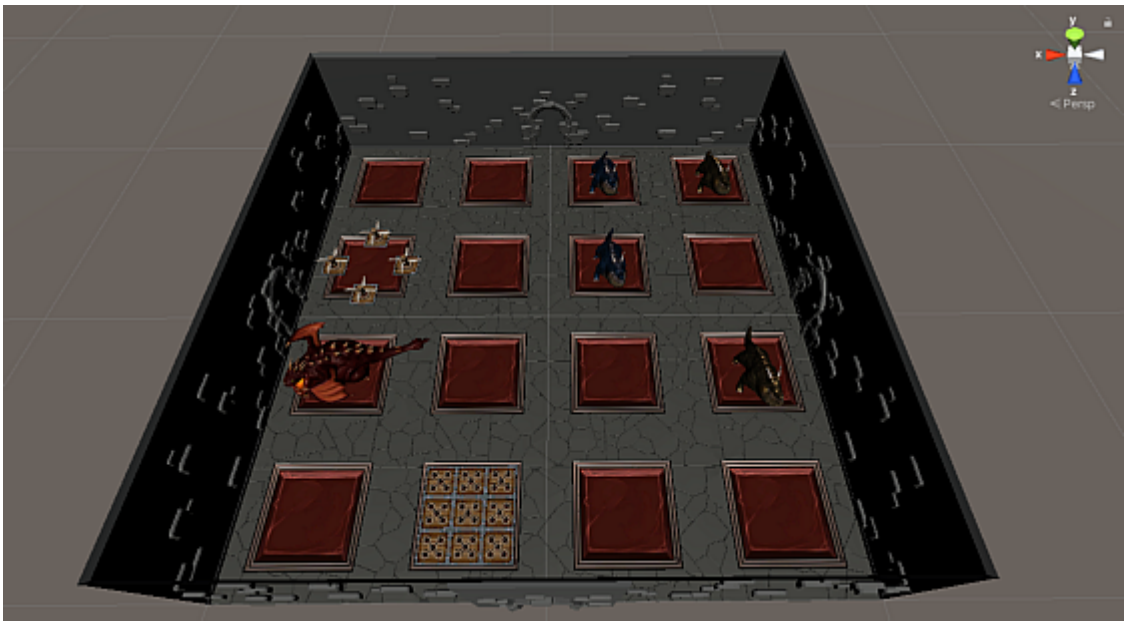

Abb. 4.17. Ein Raum, bevölkert von einem *Monster-* und *TrapSpawner*

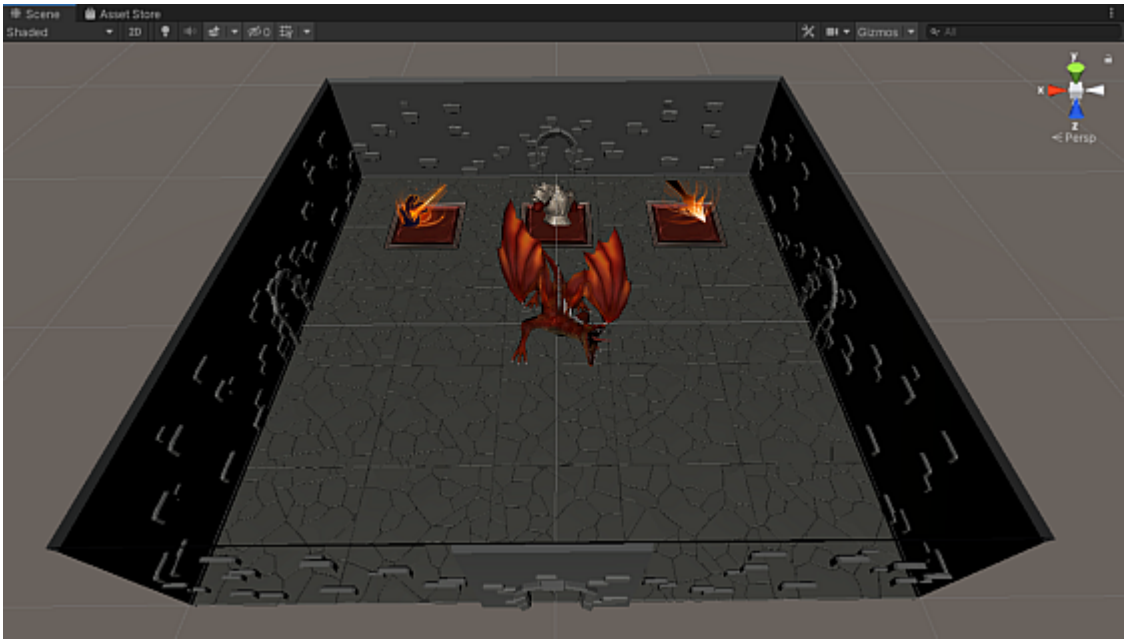

Abb. 4.18. Ein Bossraum, bevölkert von einem *Boss-* und *ItemSpawner*

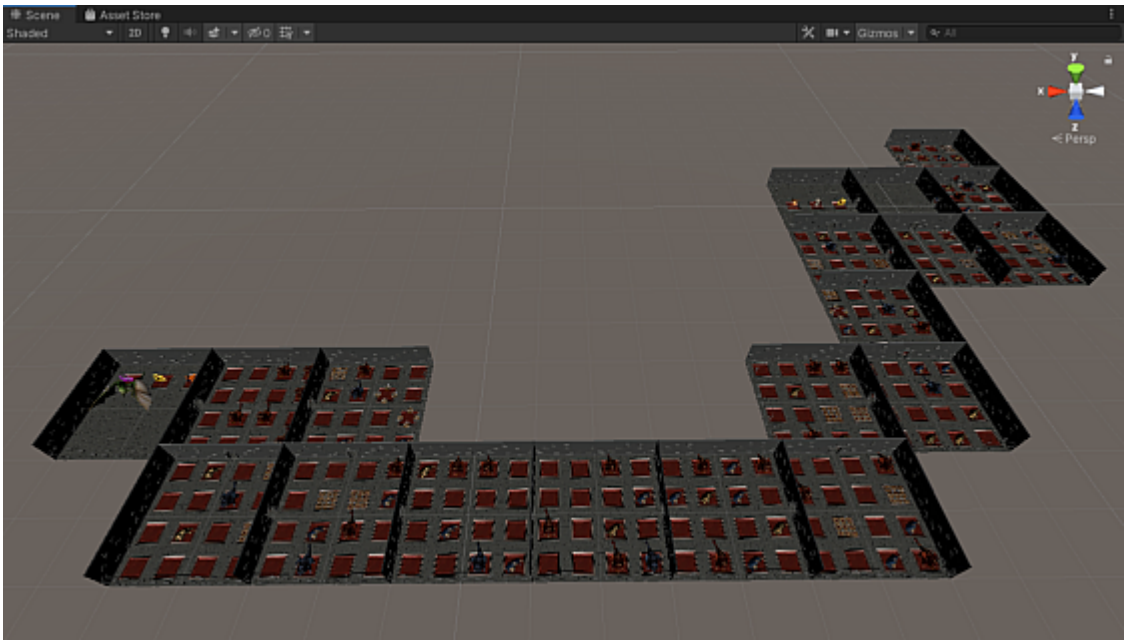

Abb. 4.19. Ein algorithmisch erzeugtes Dungeon Level

5

Zusammenfassung und Ausblick

Das zu Beginn dieser Arbeit angesetzte Ziel, einen Prototyp für die algorithmische Erzeugung eines Dungeon Crawler Levels zu schreiben, konnte im Wesentlichen erreicht werden. Mit Unity und dem SceneManager zu arbeiten war anfänglich eine Herausforderung, welche nach viel Einlesen und experimentieren überwältigt werden konnte. Als eine der größten Herausforderungen stellte sich unerwarteterweise die Organisation der Skripte heraus. Da in dem Prototyp sehr viele verschiedene Elemente zusammenarbeiten, kann es sehr schnell unübersichtlich werden. Wo welche Metrik definiert wurde, oder in welchem File sich eine bestimmte Methode befindet, war häufig mit einer kleinen Schnitzeljagd verbunden. Dies ist zurückzuführen auf die Tatsache, dass während der Entwicklung immer wieder Bedarf nach neuen Elementen entstand und diese dann in eine neue Datei ausgelagert wurden. Mit der nun vorhanden Einsicht und mehr Erfahrung mit dem Thema ist klar geworden, dass sehr viele Skripte zusammengefasst, oder vereinfacht hätten werden können. Zusätzliche Features, wie größere Räume, oder Räume mit verschiedenen Formen, stellten sich im Laufe der Arbeit allerdings als zu aufwendig heraus und wurden daher an das Ende der Liste geschoben, mit der Absicht sie zu implementieren sollte am Ende des Semesters noch Zeit dafür sein. Es ist besonders bedauerlich, dass das Seeding System mit unter den Opfern der nicht implementierten Features ist, da dieser Aspekt des Themas recht interessant und in dem zugrundeliegenden Genre häufig verwendet wird. Eine weitere unerwartete Herausforderung war die COVID-19-Pandemie. Durch soziale Distanzierung, Ausgangseinschränkungen und generell dem durch die Situation verursachten Stress, war es oftmals schwer einen klaren Kopf zum Arbeiten zu bewahren. Trotz diverser Hindernisse entstand im Rahmen der Arbeit aber ein Prototyp, welcher eine zufällige, algorithmisch erzeugte Dungeon Karte generiert und mit Gegnern und Items bevölkert. Die Skripte sind größtenteils modular und können auch in andere Projekte eingebunden werden, um Spielinhalte zufällig zu erzeugen. Dies lässt die Tür offen für zukünftige Erweiterungen an dem Projekt. Entweder durch die anderweitige Verwendung der Skripte, oder dem Ausbau des Prototypen in ein volles Spiel. Neben den oben genannten Features, kann allein durch die Integration eines Kampfsystems der Prototyp zum Beispiel schon spielbar gemacht werden. Von dort aus könnten dann weitere System, wie ein Stat- und Itemsystem entwickelt wer-

den, sowie weitere Raumarten hinzugefügt werden. Dank der Art und Weise, wie der Prototyp aufgebaut ist kann der Kreativität hier freien Lauf gelassen werden.

Literaturverzeichnis

BC18. Brummelen, Jessica Van und Bryan Chen:
Procedural Generation: Creating 3D World With Deep Learning, 2018.
http://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html.

Boo08. Booth, Michael:
The AI Systems of Left 4 Dead, 2008.
https://steamcdn-a.akamaihd.net/apps/valve/2009/ai_systems_of_l4d_mike_booth.pdf.

Rog11. Rogers, Jonathan:
Path of Exile - Random Level Generation Presentation. 2011.
https://www.youtube.com/watch?v=GcM9Ynfzll0, abgerufen am 14.07.2020.

SA17. Short, Tanya und Tarn Adams:
Procedural Generation in Game Design.
A K Peters/CRC Press, 2017.

SP18. Shin, Seungback und Sungkuk Park:
Game Level Generation Using Neural Networks, 2018.
https://www.gamasutra.com/blogs/SeungbackShin/20180227/315017/Game_Level_Generation_Using_Neural_Networks.php.

STN16. Shaker, Noor, Julian Togelius und Mark J. Nelson:
Procedural Content Generation in Games: A Textbook and an Overview of Current Research.
Springer, 2016.

V'a20. V'andrake:
Procedural Generation in Game Development, 2020.
https://www.davidepesce.com/2020/02/24/procedural-generation-in-game-development, abgerufen am 26.07.2020.

Wika. Wikipedia, Online Lexikon:
Artifical neural networks.
https://en.wikipedia.org/wiki/Artificial_neural_network, abgerufen am 28.07.2020.

Wikb. Wikipedia, Online Lexikon:
Dungeon crawl.
https://en.wikipedia.org/wiki/Dungeon_crawl, abgerufen am 26.07.2020.

Wikc. Wikipedia, Online Lexikon:
List of games using procedural generation.
https://en.wikipedia.org/wiki/List_of_games_using_procedural_generation, abgerufen am 26.07.2020.

Yu11. Yu, Derek:
Game Developers Conference: The Full Spelunky on Spelunky, 2011.
https://makegames.tumblr.com/post/4061040007/the-full-spelunky-on-spelunky.

A

Erklärung der Kandidatin / des Kandidaten

☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist ...

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser: ...

Datum	Unterschrift der Kandidatin / des Kandidaten