Erstellen eines Level Generators basierend auf Genetischen Algorithmen für ein 2D-Computer-Rollenspiel

Matutat André

05. März 2020

Matutat André (1119367)
Erstellen eines Level Generators basierend auf Genetischen Algorithmen für ein 2D-Computer-Rollenspiel
05. März 2020

Erstprüfer/in: Prof. Dr. Carsten Gips, FH Bielefeld
Zweitprüfer/in: Frau. Birgit Christina George, FH Bielefeld

Zusammenfassung

Dies ist die Zusammenfassung auf Deutsch.

Danksagung

Hier kommt die Danksagung hin.

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Motivation 1

1.2. Ziel und Struktur der Arbeit 2

2. Shattered Pixel Dungeon **3**

2.1. Rollenspiele 3

2.2. Rouge und Rouge-Like 5

2.3. Shattered Pixel Dungeon 6

3. Stand der Technik/Forschung, vergleichbare Arbeiten **9**

4. Grundlagen des Leveldesign **11**

4.1. Was ist Leveldesign 11

4.2. Regeln für gutes Leveldesign 11

5. Einführung in Genetische Algorithmen **13**

5.1. Herkunft 13

5.2. Grundbegriffe aus der Genetik 13

5.3. Ablauf 14

5.3.1. Kodierung 16

5.3.2. Bewertung 16

5.3.3. Selektion 17

5.3.4. Rekombination 19

5.3.5. Mutation 19

5.3.6. Abbruchbedingung 20

5.4. Vor und Nachteile 20

5.5. Anwengunsbeispiele 21

6. Eigene Ideen **23**

6.1. Anforderungen an das Projekt 23

6.2. Konzept zur prozeduralen Level Generierung basierend auf einen GA 24

6.2.1. GA 24

6.2.2. LevelParser 27

6.2.3. Locks and Keys 28

6.2.4. Methoden zur Auswertung und optimierung 29

6.3. Unterschied zu bekannten Methoden der prozeduralen Levelgenerierung . . . 29

7. Realisierung, Evaluation **31**
7.1. Level Generator . . . 31
7.1.1. generateLevel . . . 31
7.1.2. GenerateRandomLevel . . . 34
7.1.3. Place Start and End . . . 34
7.1.4. fitness1 . . . 35
7.1.5. fitness2 . . . 35
7.1.6. is Connected . . . 36
7.1.7. is reachable . . . 37
7.1.8. createRechableList . . . 37
7.1.9. roulettWheelSelection . . . 38
7.1.10. onePointCrossover . . . 39
7.1.11. multiPointCrossover . . . 40
7.1.12. Bit Flip Mutation . . . 40
7.1.13. fixRowMutation . . . 41
7.1.14. gameOfLife . . . 42
7.1.15. remove unrechable floors . . . 43
7.2. CodedLevel . . . 43
7.2.1. changeField . . . 43
7.3. LevelParser . . . 44
7.3.1. parseLevel . . . 44
7.3.2. place . . . 44
7.3.3. Texturen Generieren . . . 45
7.3.4. Room ZUsamensetzer . . . 47
7.4. Fazit . . . 53
7.5. Ausblick . . . 53

Literatur **55**

A. Appendix 1: Some extra stuff **57**
A.1. Bilder . . . 57
A.2. Code . . . 57
A.3. Tabellen . . . 57

B. Wuppie **59**

1. Einleitung

1.1. Motivation

Im Module Programmiermethoden des Studiengangs Informatik, an der Fachhochschule Bielefeld, sollen die Teilnehmer ihre Kenntnisse in der Programmiersprache Java erweitern und in gängige Methoden der Softwareentwicklung eingeführt werden. Die Teilnehmer sollen ein tieferes Verständnis für die Prinzipien der Objektorientierten Programmierung erwerben und grundlegende Architekturmuster kennen lernen. Die Teilnehmer sollen Git verwenden, um Erfahrungen in der Versionsverwaltung zu erlangen. Die Teilnehmer sollen grundlegende Programmiere Konventionen kennen und beachten lernen. [Prüfungsordnung]

Um diese Lernziele zu erreichen, findet neben der Vorlesung, welche den Lernstoff theoretisch bearbeitet, auch ein Praktikum statt, in dem die Teilnehmer regelmäßig vorgegebene Aufgaben in kleinen Gruppen bearbeiten müssen um dann ihre Lösungen vorzustellen. Die Teilnahme an den Praktikum ist für den Abschluss des Studienganges verpflichtend. [Prüfungsordnung]

Für das Somersemster 2020 wurde ein neues Praktikums Konzept entwickelt. Die sonst oft unabhängig voneinander gestellten Aufgaben wurden in ein größeres Gesamtprojekt eingearbeitet. Die Teilnehmer werden im laufe des Semester eine eigenes 2D Rouge Like Computerrollenspiel, angelehnt an das bereits existierende Shattered Pixel Dungeon, in der Programmiersprache Java entwickeln. Die Teilnehmer werden jede Woche ihr neu erlangtes Wissen nutzen, um das Spiel Stück für Stück aufzubauen und zu erweitern. Durch diese Umstrukturierung soll ein tieferes Verständnis für zusammenhänge einzelner Konzepte vermittelt werden. Ebenso soll die Motivation eigenständig an den Projekt weiterzuarbeiten der Teilnehmer dadurch gestiegen werden, das Erfolgserlebnisse nicht nur theoretisch Einfluss nehmen, sondern sich auch praktisch im Spiel aufzeigen. [MA Arbeit]

Das als Vorbild dienende Shattered Pixel Dungeon gehört zu den Genre der Rouge-Like Rollenspiele. Rouge-Like Videospiele zeichnen sich besonders dadurch aus, das ihre Inhalte, wie zum Beispiel Monster, Items oder Level, prozedural generiert werden.[https://de.wikipedia.org/wiki/Rogue-like]

Die Konzeptionierung und Implementierung solcher PCGs ist nicht trivial. Da das Module Programmier Methoden im zweiten Fachsemester angeordnet ist, ist die Entwicklung eines PCGs nicht nur nicht zielführend, sondern würde viele Teilnehmer auch fachlich überfordern, da diese noch ihre Grundkenntnisse festigen müssen.

1.2. Ziel und Struktur der Arbeit

Diese Arbeit präsentiert bekannte Verfahren zur prozeduralen Erzeugung von Videospiel Leveln. Die Arbeit beschäftigt sich mit Aspekten des Level Designs, jedoch nicht mit Aspekten der Level Arts. Ziel dieser Arbeit ist die Konzeptionierung und Umsetzung eines, für das Spiel der Teilnehmer angepassten, Leven Generator basierend auf Genetischen Algorithmen.

Zusätzlich beschäftigt sich die Arbeit mit der Frage, wie GAs Aspekte guten Leveldesign umsetzten ohne das diese direkten Einfluss auf den Algorithmus nehmen.

Der so erstellte Algorithmus soll den Teilnehmern zur Verfügung gestellt werden, damit diese ihn in ihre Implementation integrieren können. Der hier entwickelte Algorithmus, wird aufgrund der Natur von GAs, nicht in der Lage sein, Level in Echtzeit zu generieren. Die Konzeptionierung und Implementation der Spiellogik ist nicht Bestandteil dieser Arbeit.

Um auch nicht Spielern einen Überblick zu ermögliche, was genau ein Rouge-Like Videospiel ist und was es ausmacht, wird der erste Abschnitt grundlegende Eigenschaften dieses Genres anhand des bereits erwähnten Shattered Pixel Dungeon erläutert. Im zweiten Teil dieser Arbeit wird ein Einblick in die prozedurale Level Generierung gewährt. Es werden die wichtigsten Begriffe der PCG erläutert, ebenso werden bekannte Verfahren der PLG präsentiert. Mithilfe von veröffentlichten Quellcode sowie mithilfe von Fans durchgeführten Reverse Engineering, wird die Vorgehensweise zur Level Generierung einiger bekannter Videospiele beleuchtet. Der dritte Abschnitt dieser Arbeit liefert einen Überblick der Thematik Leveldesign. Es wird aufgezeigt was genau Leveldesign ist und auch was es nicht ist. Mithilfe der Aussagen von Spieleentwicklern und Spielejournalisten, wird eine Liste wichtige Faktoren für gutes Leveldesign aufgestellt. Der vierte Teil liefert eine Einführung in die Genetischen Algorithmen. Nachdem alle Grundlagen betrachtet wurden, führt der fünfte Abschnitt in die Konzeption des hier entwickelten Level Generators ein. Der darauffolgende Abschnitt zeigt die Umsetzung des Konzeptes, sowie eine Vielzahl an Schwierigkeiten, aufgrund dessen das Konzept regelmäßig überarbeitet werden musste. Ebenso werden die durch den Generator erzeugen Level mithilfe der in Abschnitt drei aufgestellten Liste bewertet. Zum Schluss folgt eine Zusammenfassung der Arbeit, sowie eine Liste möglicher Verbesserungen. Außerdem wird ein weiteres Konzept zur Generierung von Leveln basierend auf GAs präsentiert.

2. Shattered Pixel Dungeon

Um ein Verständnis dafür zu bekommen, welche Art von Spiel in den vom Generator erzeugten Level gespielt wird, werden in diesen Abschnitt die Grundkonzepte von Computer Rollenspiele insbesondere des Subgenres Rouge-Like beschrieben. Danach wird das Spiel: Shattered Pixel Dungeon noch einmal genauer beleuchtet, da dieses Inspiration für das neue Praktikums Konzept ist.

2.1. Rollenspiele

Das Genre der Computer Rollenspiele entstand aus den klassischen Pen and Paper Rollenspiel. Der Spiele schlüpft in die Rolle einer oder mehrere Spielfiguren mit unterschiedlichsten Fähigkeiten um verschiedene Aufgaben zu erledigen, Erfahrung zu sammeln und Ausrüstung zu erbeuten. Durch die Inspiration klassischen Pen and Paper Spiele, spielen unterschiedliche Charakterwerte, sowie die werte der Ausrüstung eine entscheidende Rolle. Neben der Erfüllung der Aufgaben, um in der Geschichte weiterzukommen. ist essenzieller Bestandteil, die Charakterwerte durch Erfahrung aufzuwerten, neue Fähigkeiten zu erlernen und die Ausrüstung zu verbessern. https://de.wikipedia.org/wiki/Computer-Rollenspiel

Viele Spiele lassen den Spieler bereits zu beginn eine von mehreren Charakterklassen wählen. Die Klasse der Figur bestimmt meist, welche Ausrüstung sie benutzen kann und welche Fähigkeiten sie erlernen kann. Vor allem seid den großen Erfolg der Elder Scrolls Spiele, ermöglichen es allerdings immer mehr Spiele, die Klasse während des eigentlichen Spielens zu bestimmen, indem Fähigkeitspunkte frei verteilt werden können. So kann der Spieler auch verschiedene Klasse miteinander vermischen.

Abbildung 2.1.: Charakter auswahl zu beginn des Spiels Dragon Age 2 <https://www.gamestar.de/artikel/dragon-age-2-tipps-tricks-zum-rollenspiel,2321599.html

Abbildung 2.2.: Fähigkeitenübersicht im Spiel Risen https://www.spieletipps.de/artikel/2018/1/

Klassische Computer Rollenspiele führen den Spieler so durch eine handgebaute Welt und erzählen dabei ihre Geschichte. Viele Rollenspiele ermöglichen es den Spieler, direkten Einfluss auf den Ausgang der Geschichte zu nehmen. So kann der Spieler im Rollenspiel The Witcher 2: Assassins of Kings, im ersten von drei Akten eine Entscheidung treffen, die Einfluss darauf nimmt, von welcher Seite der im Spiel gezeigte Konflikt betrachtet wird. Auch dürfen weniger wichtige Entscheidungen vom Spieler übernommen werden, die zwar weniger Einfluss auf das große ganze nehmen, jedoch die Immersion verbessern.

Neben der Geschichte spielt der Kampf die zweite Große Rolle in Rollenspiele. Im laufe der Zeit haben sich zwei Kampfsysteme im Markt etabliert. Rundenbasierte Kampfsysteme bleiben der

Pen and Paper Vorlage treu, und lassen Spieler und Monster nacheinander ihre Angriffe tätigen. In Actionbasierten Kamfpsystemen wird der Kampf in Echtzeit ausgetragen, hier spielen vor allem Treffervermögen und Ausweichtalent eine rolle.

Abbildung 2.3.: Rundenbasierte Kampfszene aus Final Fantasy 7

Abbildung 2.4.: Kampfszene aus den Actionrollenspiel Dark Souls

2.2. Rouge und Rouge-Like

https://en.wikipedia.org/wiki/Rogue_(video_game)

Das Adventure Rouge: Exploring the Dungoens of Doom, ist ein, in den 1980er entwickeltes, Videospiel. Der Spieler bewegt sich Rundenbasiert durch ein Levelsystem, den Dungeon, um am ende das Magische Amulett von Yendor zu erlangen. Auf den Weg dorthin gilt es die feindlichen Monster zu besiegen.

Abbildung 2.5.: Ein typisches Level aus den Spiel Rouge

https://www.gamasutra.com/view/feature/4013/the_history_of_rogue_have__you_.php?print=1 https://en.wikipedia.org/wiki/Rogue_(video_game)

Besonders machte Rouge der damals unübliche Permanenten Tod, stirbt der Spieler verlor er seinen gesamten Fortschritt und musste von vorne beginnen, als auch die Tatsache, dass das Dungeon jedes mal zufällig neu generiert wurde. Rogue setzte als eines der ersten Spiele auf prozedurale Level Generierung um die Abwechslung und den Wiederspielwert zu steigern. In Abschnitt drei dieser Arbeit, wird genauer auf prozedurale Level Generierung eingegangen.

Glenn R. Wichman schrieb in einen offenen Brief

"„But I think Rogue's biggest contribution, and one that still stands out to this day, is that the computer itself generated the adventure in Rogue. Every time you played, you got a new adventure. That's really what made it so popular for all those years in the early eighties."http://www.digital-eel.com/deep/A_Brief_History_of_Rogue.htm"

Da Rouge eines der ersten Spiele war die auf PCG setzten, waren sowohl Spieler als auch Entwickler von diesen Konzept begeistert. Wie Entwickler fühlten sich durch Inspiriert und entwickelten ihre eigenen Spiele mit PCG. https://www.gamasutra.com/view/feature/4013/the_history_of_rogue_have__you_.php?print=1

Heute bezeichnet man Spiele die das Spielprinziept des Permanenten Todes mit prozedural erzeugten Level kombiniert, als Rouge-Like. https://en.wikipedia.org/wiki/Roguelike

2.3. Shattered Pixel Dungeon

"**Shattered Pixel Dungeon** is a Roguelike RPG, with pixel art graphics and lots of variety and replayability. Every game is unique, with four different playable characters, randomized levels and enemies, and over 150 items to collect and use. The game is simple to get into, but has lots

of depth. Strategy is required if you want to win!" https://pixeldungeon.fandom.com/wiki/Mod-Shattered_Pixel_Dungeon

Zu beginn des Spiels wählt der Spieler eine der vier Charakterklassen aus. Zur Auswahl stehen:https://pixeldungeon.fandom.com/wiki/Mod-Shattered_Pixel_Dungeon/Classes#Rogue

- Krieger: Nahkämpfer mit Fokus auf hohe stärke und Lebensenergie
- Magier:Fernkämpfer der Zauber verwendet
- Schurke: Nahkämpfer der auf Planung und Hinterhalte fußt
- Jäger: Fernkämpfer der vor allem Bögen verwendet

Der Spieler startet auf der obersten ebene eines Dungeons. Ziel ist es, möglichst tief in das Dungeon einzudringen, je tiefer der Spieler ist, desto schwieriger wird das Spiel. In den einzelnen Ebenen begegnen den Spieler die unterschiedlichsten Monster, durch dessen Tötung der Spieler Erfahrung sammelt, wurde genug Erfahrung gesammelt, steigt der Spieler auf, dies ermöglicht ihn, seine Spielfigur mit neuen Fähigkeiten auszustatten. Ebenso lassen besiegte Gegner Ausrüstungsgegenstände wie Heiltränke oder Waffen fallen. Um Erfolg im Spiel zu haben, sollte man also nicht nur versuchen möglichst schnell möglichst tief in das Dungeon einzudringen, sondern auch viele Gegner zu besiegen um an Erfahrung und Items zu gelangen. Das Spiel wird genau wie Rouge Rundenbasiert gespielt.

Darüber hinaus bietet Shattered Pixel Dungeon noch viele weiter Features wie Subklassen oder Verzauberungen, welche hier nicht weiter beschrieben werden, da diese für das Verständnis des Spielablaufes nicht nötig sind.

Abbildung 2.6.: Beispielausschnitt für das Leveldesign in Shattered Pixel Dungeon https://www.holarse-linuxgaming.de/wiki/pixel_dungeon

3. Stand der Technik/Forschung, vergleichbare Arbeiten

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

4. Grundlagen des Leveldesign

4.1. Was ist Leveldesign

- Definintion
- unterschied zu level art

4.2. Regeln für gutes Leveldesign

1. Lösbarkeit
2. Immersion
3. Interaktionsvermögen
4. Pacing
5. Navigation
6. Schwierigkeit
7. Risk and Reward
8. Einzigartigkeit
9. Unterstützt Gameplay

5. Einführung in Genetische Algorithmen

5.1. Herkunft

“Evolutionäre Algorithmen (EA) sind Optimierungsverfahren, die sich am Vorbild der biologischen Evolution orientieren.” https://ls11-www.cs.tu-dortmund.de/lehre/SoSe03/PG431/Ausarbeitungen/GA_Selzam.pdf

Vereinfacht ausgedrückt: Evolutionäre Algorithmen sind ein Verfahren zur kontrollierten und gesteuerten Zufalls suche.

Es gibt vier, aus der Historie entstandenen unterscheidungsformen:

- Genetische Algorithmen (GA)
- Genetische Programmierung (GP)
- Evolutionsstrategien (ES)
- Evolutionäre Programmierung (EP)

GAs wurden Anfang der 60er Jahre von John Holland vorgestellt. [Holland, John: Adaptation in Natural and Artificial Systems; The University of Michigan Press, 1975] Zur selben Zeit entwickelte Hans-Paul-Schwefel und Ingo Rechenberg die Evolutionsstragien. [Rechenberg, Ingo: Evolutionsstrategie. Optimierung technischer Systeme nach Prinzipien der biologischen Evolution; Frommann-Holzboog-Verlag, Stuttgart 1973].

Zwar gibt es zwischen den einzelnen Formen inhaltliche unterschiede, allerdings haben sich die Verfahren heutzutage so miteinander vermengt, das eine Unterscheidung in dieser Arbeit nicht weiter Zielführend ist. Im weiteren Verlauf wird von Genetischen Algorithmen gesprochen, auch dann wenn teils verfahren aus den Evolutionären Strategien besprochen oder angewandt werden.

5.2. Grundbegriffe aus der Genetik

https://ls11-www.cs.tu-dortmund.de/lehre/SoSe03/PG431/Ausarbeitungen/GA_Selzam.pdf

Da Genetische Algorithmen sich an der Evolution orientieren, haben viele Fachbegriffe der Genetik ihren Weg in die Informatik gefunden. Im Folgenden werden die, für Genetische Algorithmen und diese Arbeit, relevanten Begriffe in ihrer Bedeutung inerhalb der Informatik, erläutert.

Individuum / Chromosom

"Ein Individuum im biologischen Sinne ist ein lebender Organismus, dessen Erbinformationen in einer Menge von Chromosomen gespeichert ist. Im Zusammenhang mit genetischen Algorithmen werden die Begriffe Individuum und Chromosom jedoch meistens gleichgesetzt." https://ls11-www.cs.tu-dortmund.de/lehre/SoSe03/PG431/Ausarbeitungen/GA_Selzam.pdf

Gen

Ein Gen ist genau eine Sequenz im Individuum. Je nach Kontext kann es sich hierbei um eine einzelne Stelle oder mehrere Stellen im Individuum handeln.

Allel

Allel beschreibt den exakten Wert eines Gens.

Population / Generation

Einer Menge gleichartiger Individuen wird als Population bezeichnet. Die Anzahl der Individuen gibt die Populationsgröße an. Sterben Individuen oder werden neune geboren, verändert sich die Größe der Population. Betrachtet man eine Population über mehrerer Zeitpunkte, spricht man von Generationen.

5.3. Ablauf

GAs folgen einer Reihe an Subrutinen, die sich solange Wiederholen bis eine Abbruchbedingung erreicht ist.

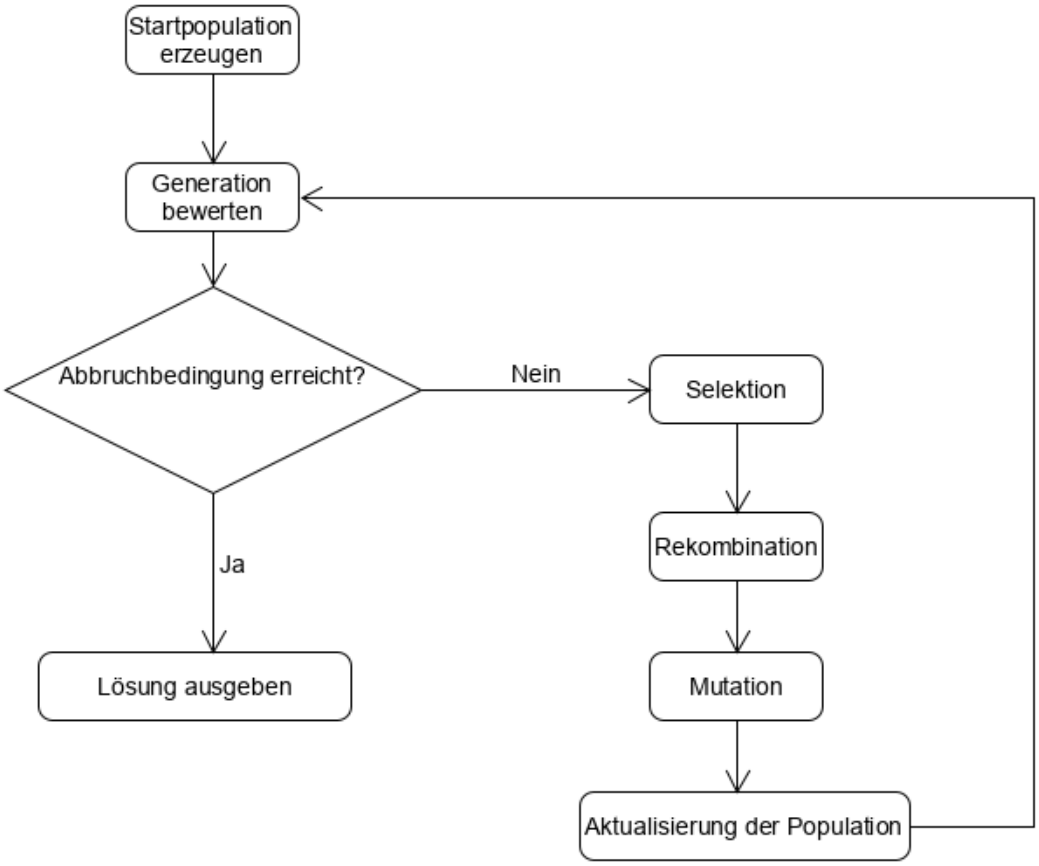


Abbildung 5.1.: Ablauf eines generischen GAs

Abbildung … zeigt den zugrunde legenden Ablauf von GAs.

1. Kodierte Startpopulation wird erzeugt
2. Aktuelle Generation wird bewertet
3. Es wird geprüft, ob die Abbruchbedingung erreicht ist
4. Wenn nein

    1. Es werden Individuen für die neue Generation ausgewählt
    2. Manche der Ausgewählten Individuen werden miteinander Rekombiniert und erzeugen so neue Individuen
    3. Einige der Gene der neuen Generation werden Mutiert
    4. Die neue Generation wird zu Start Generation für den nächsten durchlauf
    5. Go to 2

5. Ausgabe der Lösung

Im folgenden werden die Einzelnen Subroutinen genauer beschrieben, sowie gängige Implementationen gezeigt.

5.3.1. Kodierung

Zu beginn muss das betrachtete Problem kodiert werden. Das bedeutet das alle relevanten Aspekte auf ein Chromosom abgebildet werden müssen. Hierbei gibt es zwei Kodierungsverfahren, die je nach Problem ausgewählt werden müssen.

Bei der Binären Kodierung besteht ein Chromosom aus n vielen Genen. Jeden Gen wir ein binärer Wert zugewiesen und repräsentiert dabei eine Problemvariable. Aus den Genen wird dann ein Bitstring erzeugt.

$$g = (g_0 ....g_m) \in \{0,1\}^m$$

Verwendet man die Binäre Kodierung, muss man am ende die Lösung mithilfe einer Dekodierungsfunktion zurück wandeln.

$$T : \{0,1\}^m \to R^n$$

Eine andere Variante ist die reellwertige Kodierung. Sie funktioniert ähnlich zu der Binären Kodierung, nur wird hier jeden Gen ein reellwertige Wert zugewiesen. Eine Dekodierung ist nicht nötig.

$$g = (g_0 ....g_m) \in \{R\}^m$$

Die Allel der Gene der Startpopulation werden zufällig bestimmt.

5.3.2. Bewertung

Die Bewertung erfolgt mithilfe der sogenannten Fitnessfunktion, angelehnt an Darwins Survival of the Fittest. Die Fitnessfunktion bewertet die güte eines Individuums, also wie nahe es schon an einer möglichen Lösung ist. Dabei weißt die Fitnessfunktion den Individuum eine reelle Zahl zu, die die Fitness darstellt. Im Regelfall ist eine höhere Fitness besser als eine geringer.

Die Implementation der Fitnessfunktion ist stark mit der eigentlichen Problemstellung verwoben. Da die Fitnessfunktion großen Einfluss darauf hat, in welche Richtung sich die Population entwickelt, sind die Bewertungskriterien so zu wählen, das sie zur Erreichung der Lösung beitragen. Da die Fitnessfunktion während der Laufzeit jedes einzelne Individuum jeder Population jeder Generation betrachtet, sollte bei der Implementierung auf Laufzeit Optimierung geachtet werden, eine komplexe Fitnessfunktion kann den gesamten GA verlangsamen.

5.3.3. Selektion

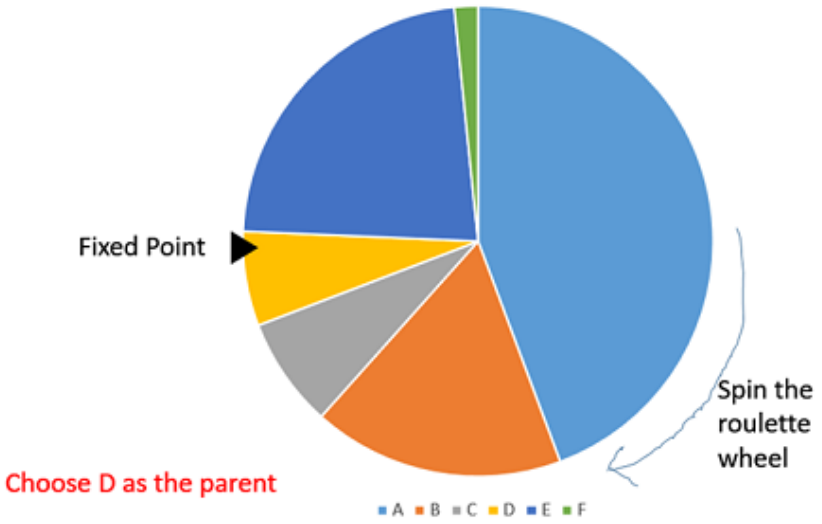
Bei der Selektion werden die Individuen ausgewählt, welche die nächste Generation bilden. Die gängigsten verfahren selektieren nach der Fitness der Individuen. Jedoch sollte man nicht nur die besten Individuen der Population auswählen, da nicht nachvollzogen werden kann, ob sich das Individuum in der nähe des globalen Hochpunkts befindet oder sich lediglich einen lokalen Hochpunkt nährt. Würde man nur die besten Individuen erlauben sich zu vermehren, würde sich die Population in eine Richtung festfahren.

Es folgt eine Auflistung und Erklärung bekannter Selektionsverfahren. Bei jeden Verfahren werden die ausgewählten Individuen auch zurück in die Ursprungspopulation gelegt, es ist also möglich das selbe Individuum mehrfach auszuwählen. Es wird solange Ausgewählt bis die neue Population die gewünschte Größe erreicht hat, diese ist in der Regel genauso groß oder größer als die Ursprungspopulation.

Fitness Proportionate Selection

Bei der Fitness Proportionate Selection hat jedes Individuum die Chance ausgewählt zu werden. Die Chance ausgewählt zu werden ist abhängig von der FItness des Individuums. Besonders gute Lösungen haben also hohe Chance ausgewählt zu werden, schlechtere Lösungen können dennoch ausgewählt werden um so die vielfallt der Population zu gewährleisten.

Eine gängige Art der Umsetzung dieses Verfahren ist die **Roulett Wheel Selection**. Angelehnt an Glücksräder, wird ein ein Rad in n Teile zerteilt, wobei n die Summe der Fitness der Population entspricht. Jedes Individuum der Population enthält entsprechend seiner Fitness Anteile am Rad. Am Rad wird ein Fix Punkt angesetzt, das Rad wird rotiert und das Individuum ausgewählt auf dessen Anteil der Fix Punkt stehenbleibt.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Abbildung 5.2.: Bildliche Darstellung der RWS https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm

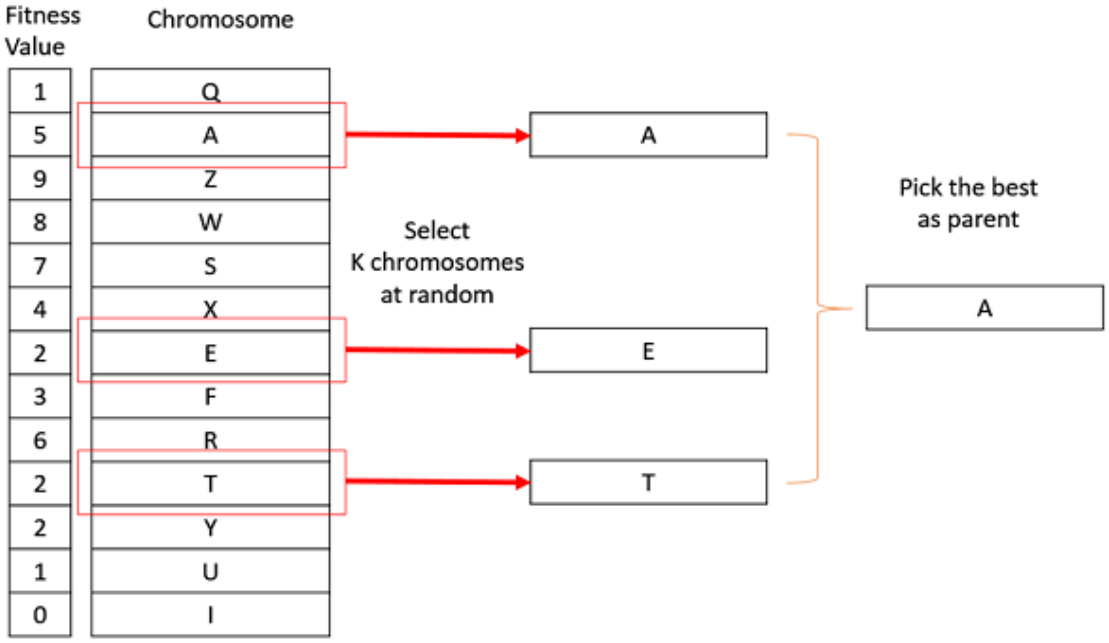
Das **Stochastic Universal Sampling** erweitert die Roulett Wheel Selection um einen zweiten

Fixpunkt. So können zeitgleich zwei Individuen ausgewählt werden.

Fitness Proportionate Selektionsverfahren funktionieren nicht in Fällen, in dem Fitnesswerte negativ sein können.

Tournament Selektion

Bei der Tournamen Selektion werden zufällig k Individuen aus der Ursprungspopulation ausgewählt, das Individuum mit der höchsten Fitness wird in die nächste Generation aufgenommen. Dieses Verfahren ermöglicht zwar auch schlechteren Lösungen Ausgewählt zu werden, versichert aber das die schlechtesten k-1 Lösungen nicht ausgewählt werden können und die beste Lösung auf jeden Fall ausgewählt wird. Tournament Selektion funktioniert auch bei negativen Fitnesswerten.



Rank Selektion

Bei der Rank Selektion wird die Population anhand der Fitnesswerte der Lösungen sortiert. Für die Auswahl spielt nicht mehr der Fitnesswert sondern die Platzierung der Lösung eine Rolle. Höher platzierte Lösungen haben eine höhere Chance ausgewählt zu werden als niedrig platzierte Lösungen. Der Chancenunterschied ist je nach Problemstellung zu wählen. Rank Selektion kann auch bei negativen Fitnesswerten verwendet werden.

Zufällige Selektion

Bei der zufälligen Selektion werden zufällig Individuen aus der Population ausgewählt. Dieses Verfahren wird für gewöhnlich vermieden, da es keinerlei Filter Mechanismen gibt und die suche nicht gesteuert werden kann.

5.3.4. Rekombination

Bei der Rekombination werden zwei Selektierte Lösungen neu zusammengesetzt. Die beiden Ursprünglichen Lösungen bezeichnet man als Eltern, die neu erzeugten Lösungen als Kinder.

Ob zwei Eltern miteinander Rekombiniert werden, ist Abhängig von der festgelegten Crossoverchance. In GAs liegt diese für gewöhnlich bei $\approx$ 60%. Die Rekombination vermischt den Genpool der Eltern und soll so für eine möglichst diverse Population sorgen.

Es folgt eine Erläuterung einiger verbreitetet Crossoververfahren. Je nach Problemstellung kann auch ein Individuelles verfahren Zielführend sein.

Point Crossover Beim Point Crossover verfahren werden beide Eltern an einer oder mehrere Stellen in Segmente geteitl. Die Segmente der Eltern werden miteinander vertauscht um die Kinder zu erzeugen.

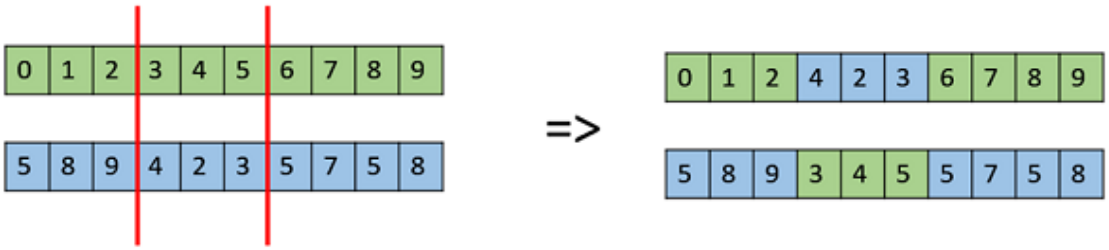


Abbildung 5.3.: Bildliche Darstellung des Multi Point Crossoververfahren
https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm

Uniform Crossover

Beim Unform Crossover wird jedes Gen eines Elternteils betrachtet, es gibt eine 50% Chance dass das Gen mit dem entsprechenden Gegenstück des anderen Elternteils ausgetauscht wird.

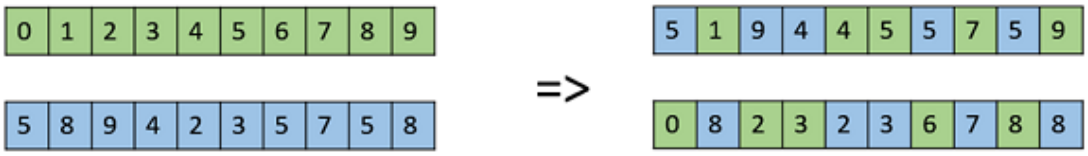


Abbildung 5.4.: Bildliche Darstellung des Uniform Crossoververfahren
https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm

5.3.5. Mutation

Die Mutation ist eine Zufällige Veränderung der Gene um eine neue Lösung zu erhalten. Genau wie die Rekombination wird sie genutzt um eine möglichst große Diversität der Population zu

erzeugen. Durch Crossover erzeugte Kinder liegen im Suchbaum in der nähe der Eltern, durch die Mutation werden die Kinder von den Eltern weiter entfernt.

Die Mutationschance pmut ist gering anzusetzen, da eine zu hohe Mutationschance den GA zu auf eine zufällige Suche reduzieren würde.

Es folgt eine Auflistung und Beschreibung bekannter Mutationsverfahren. Je nach Problemstellung kann auch eine Spezifische Mutationsmethode Zielführend sein. Je nach Kontext kann ein einzelnes Gen oder eine Sequenz an Genen mutiert werden. Für die folgenden Verfahren muss man dann beachten, das eine Sequenz auch als Gen bezeichnet wird. Je nach Bedarf kann nur eine Mutation pro Lösung durchgeführt werden oder es kann jedes Gen mit einer Chance von pmut mutiert werden.

Bit Flip Mutation

Bei der Bit Flip Mutation wird ein oder mehrere bits gewechselt. Bit Flip Mutation kann daher nur bei Binär Kodierten GAs verwendet werden. Abwandlungen für Reellwertige Kodierungen sind aber möglich.

Swap Mutation

Bei der Swap Mutation werden zwei Zufällig ausgewählte Gene miteinander vertauscht.

Scramble Mutation

Bei der Scramble Mutation wird eine Auswahl an zusammenhängenden Genen vermischt um so die Reihenfolge zu ändern.

Inversion Mutation

Bei der Scramble Mutation wird die Reihenfolge einer Auswahl an zusammenhängenden Genen umgedreht. Das erste Gen in der Sequenz wird zum letzten und umgedreht.

5.3.6. Abbruchbedingung

Die Abbruchbedingung ist für gewöhnlich dann erreicht, wenn eine gültige Lösung gefunden wurde. Je nach Problemstellung kann es auch das überschreiten eines gewissen Fitnessschwellwertes sein oder der durchlauf einer bestimmten Generationenanazahl. Es bietet sich an einen Neustart des GAs vorzunehmen, sollte nach einer gewissen Generationsanzahl keine Lösung gefunden worden sein. Der entsprechende Schwellwert sollte so gewählt werden, dass Erfahrungsgemäß keine Steigerung der Fitness zu erwarten ist und ist dementsprechend für jede Implementierung unterschiedlich.

5.4. Vor und Nachteile

Pro	Contra
Gut geeignet für Probleme in großen Suchräumen	Laufzeit in kleinen Suchräumen oft länger als andere Verfahren
Benötigt keine abgeleiteten Informationen des Problems	Laufzeit stark von der Komplexität der Fitnessfunktion abhängig
Lässt sich gut parallelisieren	Lösungen werden Zufällig gefunden, es gibt keine Garantie die beste Lösung zu finden.
Liefert eine Menge an guten Lösungen, nicht nur eine Lösung	
Verläuft sich nicht in lokalen Hochpunkten	

5.5. Anwengunsbeispiele

http://wwwisg.cs.uni-magdeburg.de/sim/vilab/2007/papers/12_genetisch_sharbich.pdf

https://www.degruyter.com/view/j/auto.1995.43.issue-3/auto.1995.43.3.110/auto.1995.43.3.110.xml

GAs werden beispielsweise genutzt um das Profil eines Flügels und die Form des Rumpfes von Flugzeugen zu optimieren. Beim Brückenbau lassen sich Positionierung und Gewicht einzelner Bauteile optimieren. GAs werden auch für Schwellwertanpassung bei verschieden Algorithmen und Neuronalen Netzten genutzt. Sie können auch für Scheduling-Probleme verwendet werden um so zum Beispiel Stundenpläne in Schulen zu gestalten oder Routenplanung für Paketdienste. Im allgemeinen können GAs genutzt werden um NP-Schwere Probleme zu lösen.

6. Eigene Ideen

Im folgenden wird das erste Konzept zur Erstellung eines Level Generators basierend auf GAs. erläutert. Zuerst werden die Anforderungen an das Projekt definiert und die Zielsetzung spezifiziert. Danach folgt die Erklärung des Konzeptes. Am ende des Abschnittes wird das erstellte Konzept mit den aus Abschnitt 3 bekannten Verfahren verglichen.

6.1. Anforderungen an das Projekt

Der Level Generator muss in der Programmiersprache Java geschrieben sein, da diese die durch die Prüfungsordnung festgelegte Modul Sprache ist. Der Levelgenerator muss lösbare 2D Level generieren. Level bestehen aus unterschiedlichen Räumen die mit Fluren verbunden sind. Als Lösbar gilt ein Level dann, wenn vom Startpunkt der Endpunkt aus erreicht werden kann. Die Größe des Levels soll durch den User aus wählbar sein. Um das Level grafisch darstellen zu können, muss eine Levelgrafik erzeugt werden. Im Level müssen Monster und Items von den Usern platziert werden können. Im Level sollen Türen und dazugehörige Schlüssel so platziert werden, das kein Softlock entsteht. Der Level Generator soll den Teilnehmer möglichst viel Freiheit in der Umsetzung der Spielinhalte, abseits der Level, gewähren. Die Teilnehmer sollen in der Lage sein, neben den in den Aufgaben vordefinierten Elemente, eigen konzeptionierte Elemente in die Level zu integrieren. Grundlegende Level müssen erzeugt werden können, ohne Informationen über Layout oder Inhalt zu bekommen. Die Teilnehmer müssen in der Lage sein, die erzeugten Level in ihre Implementation der Spiellogik zu verwenden.

Den Generator sollen möglichst wenig Informationen über die Regeln guten Leveldesigns gegeben werden, damit in der abschließenden Evaluierung geschaut werden kann, wie viele Regeln passiv beachtet wurden.

Der Generator muss nicht in der Lage sein, neue Spielregeln, wie das Sprengen von Wänden durch Bomben, in den Generierungsprozess zu beachten. Der Generator muss Texturen nicht selbständig erzeugen oder die Größenverhältnisse übergebener Texturen anpassen. Der Generator muss keine Quests oder Boss Gegner erzeugen. Der Generator muss keine Level in Echtzeit erzeugen können. Es wird kein Fokus auf die Optimierung der Laufzeit gelegt.

6.2. Konzept zur prozeduralen Level Generierung basierend auf einen GA

Der Generator wird aus zwei Teilen zusammensetzt. Der erste Teil ist der GA selbst, er generiert Level die aus Wänden, Böden einen Start und einen Ziel bestehen. Der zweite Teil ist ein Parser, der für die Integration der erzeugten Level in die Spiellogik zuständig ist. Der Parser ist auch dafür verantwortlich, das Monster und Items im Level platziert werden, er soll die Möglichkeit bieten, einzelne Wände und Böden gegen andere Oberflächen auszutauschen. Er ist für die Generierung der Levelgrafik verantwortlich.

Alle Konstanten werden von der Klasse Constants als statische Variablen zur Verfügung gestellt. Um Konstanten in der weiteren Beschreibung des Konzeptes darzustellen, werden alle verwendeten Konstanten in Großbuchstaben geschrieben. Die Werte der Konstanten werden erst durch praktische Tests festgelegt werden können.

6.2.1. GA

6.2.1.1. Kodierung

Die Kodierung der Level passt sich der vorgesehenen Implementation in die Spielogik an. Ein Level kann als zwei Dimensionales Array verstanden werden, die einzelnen Felder im Array sind Felder im Level. Der Spieler bewegt sich pro Zug genau um ein Feld.

Die Level werden als zwei Dimensionales Char Array kodiert, jedes Feld im Array repräsentiert ein Gen. Die Gene repräsentieren eine Oberfläche, also ob das Feld eine Wand, ein Boden, der Start oder Ausgang sind. Wände werden als nicht passierbare Felder betrachtet.

Ein Kodiertes Level könnte ausgegeben also so aussehen:

```
1 WWWWWWW
2 WFFFFFW
3 WFFFFSW
4 WFXFFFW
5 WWWWWWW
6
7 W=Wall      S=Start
8 F=Boden     X=Ende
```

6.2.1.2. CodedLevel

Eine Instanz der Klasse CodedLevel ist ein Chromosom, also eine mögliche Lösung bzw. ein Level. Neben den Char Array welches den Levelaufbau entspricht und Informationen über die

Größe des Levels, besitzen CodedLevel eine Fitness welche die Güte der Lösung angibt sowie Informationen über den Standort der Start- bzw. Endpunktes.

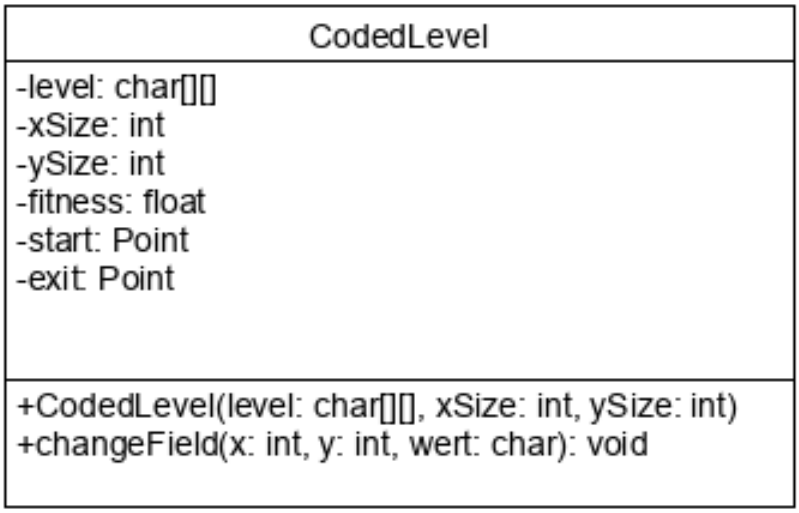


Abbildung 6.1.: UML CodedLevel. Eigene Grafik

Neben Getter und Setter verfügt die Klasse über die changeField Methode. Diese verändert den Allel eines Gens auf den übergebenen Wert. Zwar könnte diese Änderung auch direkt am Array vorgenommen werden, dann würden allerdings Änderungen an der Position der Start und Ausgänge evtl. verloren gehen. Sollte ein Start bzw. Ausgang gesetzt werden, obwohl schon einer Vorhanden ist, wird stattdessen ein Boden gesetzt, ist keiner vorhanden werden die Koordinaten dem entsprechenden Attribut zugewiesen.

6.2.1.3. LevelGenerator

Die Klasse Level Generator beinhaltet die Implementation des GA. Sie beinhaltet die Implementationen aller Subroutinen. Sie verfügt über die Methode generateLevel, welche als Einstiegspunkt in den GA gesehen werden kann. Ihr werden die gewünschte Level Größe übergeben und dann kümmert Sie sich um die Durchführung des GA Ablaufes.

Erzeugen der Startpopulation Um die Startpopulation zu erzeugen werden die Level zufällig mit Oberflächen gefüllt. Die CHANCE_TO_BE_FLOOR gibt an, mit welche Chance eine Oberfläche ein Boden wird. Alle Außenfelder des Levels werden mit Wänden gefüllt, um einen Level Abschluss darzustellen. Ein und Ausgang werden zufällig auf Böden gesetzt.

Fitnessfunktion Die Fitnessfunktion bewertet die Level. Die Bewertungskriterien sollen dabei helfen, dass das Level in größere Bodenflächen, den Räumen verbunden durch Wandketten, den Fluren.

Jedes Bewertungskriterium hat dabei eine andere Wertigkeit. Die Bewertungskriterien sind

1. Wie viele Böden sind begehbar?
2. Ist das Level lösbar?
3. Wie viele Wände sind Verbunden?

Die Anzahl der begehbaren Böden ist gleichzusetzten mit der Spielfläche und der daraus folgenden Interaktion mit den Level. Sind Böden nicht erreichbar, weil sie von Wänden eingeschlossen sind, können diese Flächen nicht bespielt werden und der restliche Dungeon wird kleiner. Dadurch sollte die Erstellung von Raumflächen begünstigt werden.

Die Lösbarkeit ist eines Levels ist Kernvoraussetzung um als gültige Lösung zu gelten. Daher nimmt dieses Kriterium großen Einfluss auf die Fitness.

Das dritte Kriterium soll vor allem einzeln im Level platzierte Wände vermeiden, da diese in der Logik des Spiels keinen Sinn erfüllen und daher vom Spieler als störend empfunden werden und die Immersion mindern. Für jede Wand die direkt oder indirekt über Nachbarn mit der außen Wand verbunden sind gibt es Fitnesspunkte. Dadurch sollen Wandketten gefördert werden. Die Verbindung mit den Außenwänden ist vom klassischen Hausaufbau inspiriert, da dort in der Regel auch jede Wand in irgendeiner Form mit der Außenwand verbunden ist. Da die hier generierten Level auch größere Dungeon darstellen sollen, könnten auch Säulenartige Strukturen oder Stützwände wünschenswert sein. Daher gibt es für Wände, die zwar keine Anbindung an den Levelrand haben, jedoch mit anderen Wänden verbunden sind, Teilpunkte.

Selektion und Rekombination Als Selektionsverfahren wird das im Abschnitt …. beschreibende Roulette Wheel Selection Verfahren genutzt. Da keine negative Fitness erreicht werden kann als auch von einer großen Spannbreite an Bewertungen ausgegangen werden kann, bietet sich ein Rank Selection Verfahren nicht an. Alternativ wäre auch die Verwendung der Tournament Selektion denkbar.

Sollte die es zu einen Crossover, abhängig von der CHANCE_FOR_CROSSOVER, kommen, werden beide Eltern mithilfe des One-Point-Crossover rekombiniert. Dieses Verfahren hat eine geringer Chance, gute Level Strukturen zu zerstören, da nur ein großer und nicht viele kleine Eingriffe am Level durchgeführt werden.

Das Uniform Crossover Verfahren würde wieder eine komplett Zufällige Anordnung von Böden und Wänden zu folge ziehen und ist daher für die Generierung von Leveln nicht geeignet.

Mutation Zur Levelgenerierung bieten sich fast alle bekannten Mutationsverfahren an. In dieser Implementierung wird eine angepasste Version der Bit-Flip Mutation verwendet. Ignorieren wir bei der Mutation Start und Ausgangspunkt, bleiben noch Felder die entweder Böden oder Wände sind. Es wird für jedes Gen überprüft ob es zur Mutation kommt, und wenn ja, wird der Allel des Gen geändert. Wände werden zu Böden und Böden zu Wände.

6.2.1.4. Abbruchkriterium

Die GA wird dann beendet, wenn ein lösbares Level den Fitnessschwellwert überschreitet. Der Fitnessschwellwert ergibt sich aus der maximal Erreichbaren Fitness. Da die maximal erreichbare Fitness aufgrund von Zufallsfaktoren nicht exakt bestimmt werden kann, wird sich ihr angenähert:

```
1 AnzahlBoeden \approx \text{ CHANCE_TO_BE_FLOOR } * \Sum{  }{  }{
    }Felder \linebreak
2 AnzahlWaende \approx \text{ 1-CHANCE_TO_BE_FLOOR } * \Sum{  }{  }{
     }Felder \linebreak
3 MaxFitness\approx \text{PUNKTE_FUER_ERREICHBARKEIT} * \text{
    AnzahlBoeden } + \text{PUNKTE_FUER_VERBUNDEN} * \text{
    AnzahlWaende } + \text{ PUNKTE_FUER_LOESBAR }
```

Um Zufallswerte auszugleichen, wird der Schwellwert unter den berechneten Wert angesiedelt.

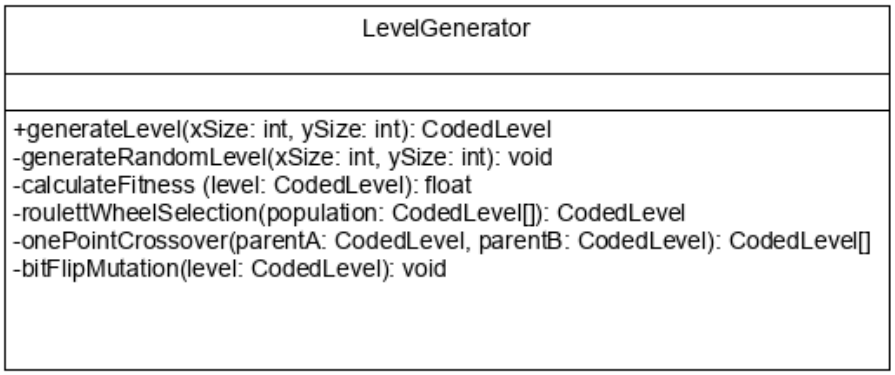


Abbildung 6.2.: UML LevelGenerator. Eigene Grafik

6.2.2. LevelParser

Die Klasse LevelParser stellt alle Methoden zur Verfügung die benötigt werden um von einen generierten CodedLevel zu einen richtigen Level mit Monstern und Items zu gelangen. Um den Parser so zu gestalten, das er für möglichst alle Implementierungen der Teilnehmer funktionsfähig ist, werden eine Reihe an Interfaces vorgegeben welche von den Teilnehmer in ihre Implementation integriert werden müssen.

Mithilfe des Interfaces ILevel wird sichergestellt, das die Klasse Level die Methoden getXSize und getYSize, welche die Maße der Level zurückgeben sowie die Methode getLevel welches ein zwei Dimensionales ISurface Array zurückliefert, welches Analog zu den aus CodedLevel bekannten Array den Levelaufbau darstellt. Das Interface ISurface muss von jeder Oberfläche implementiert werden, es versichert Methoden zur Platzierung von Monstern und Items. Ebenso

muss die Methode getTexture implementiert werden, welche den Pfad zu einer Grafik liefert, die auf der Levelgrafik die jeweilige Oberfläche darstellen soll.

Die parseLevel Methode verwandelt ein CodedLevel in ein richtiges Level um. Die chars welche bisher als Referenzen für Oberflächen gedient haben, werden durch Instanzen der entsprechenden Oberflächen ausgetauscht.

Der Parser nutzt die von ISurface bereitgestellte Methoden um übergebene Monster oder Items auf zufällige Felder zu verteilen. Sollen neben Wänden oder Böden auch andere Oberflächen integriert werden, bietet der Parser eine Funktion zum austauschen einer Zufällig gewählten Instanz des Oberflächentypes A um eine Instanz des Oberflächentypes B zu platzieren.

Die Funktion generateTextureMap itteriert über das zwei Dimensionale ISurface Array und holt sich mithilfe der getTexture Methode die Texturen der einzelnen Oberflächen und fügt diese nacheinander zusammen und speichert das erzeugte Bild ab. Dadurch das die Textur nicht Typ weise sondern Instanz weise ausgelesen wird, wird es den Teilnehmern ermöglicht, Wände mit unterschiedlichen Texturen zu verwenden.

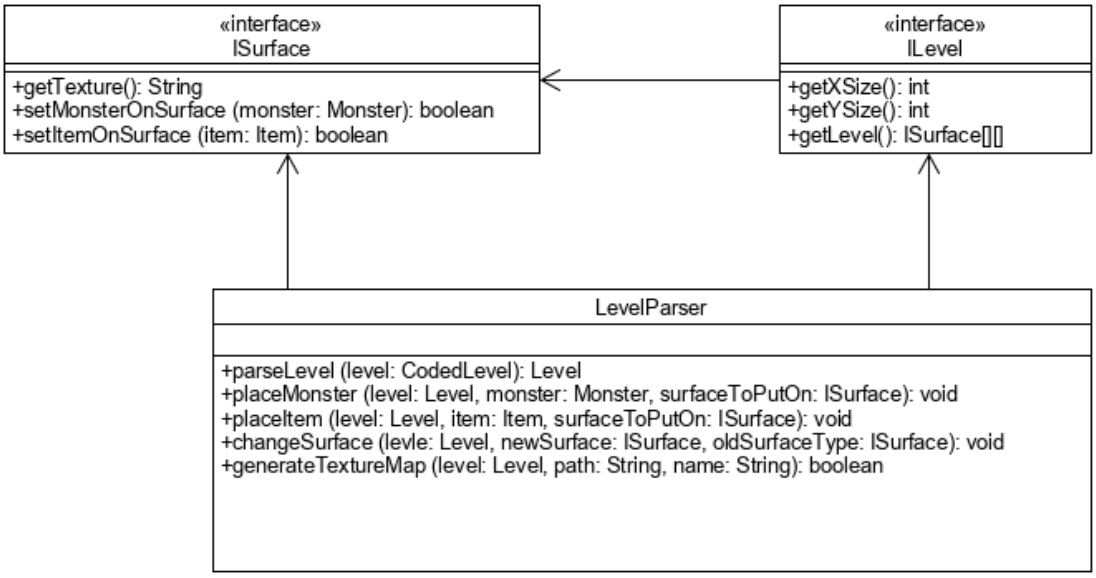
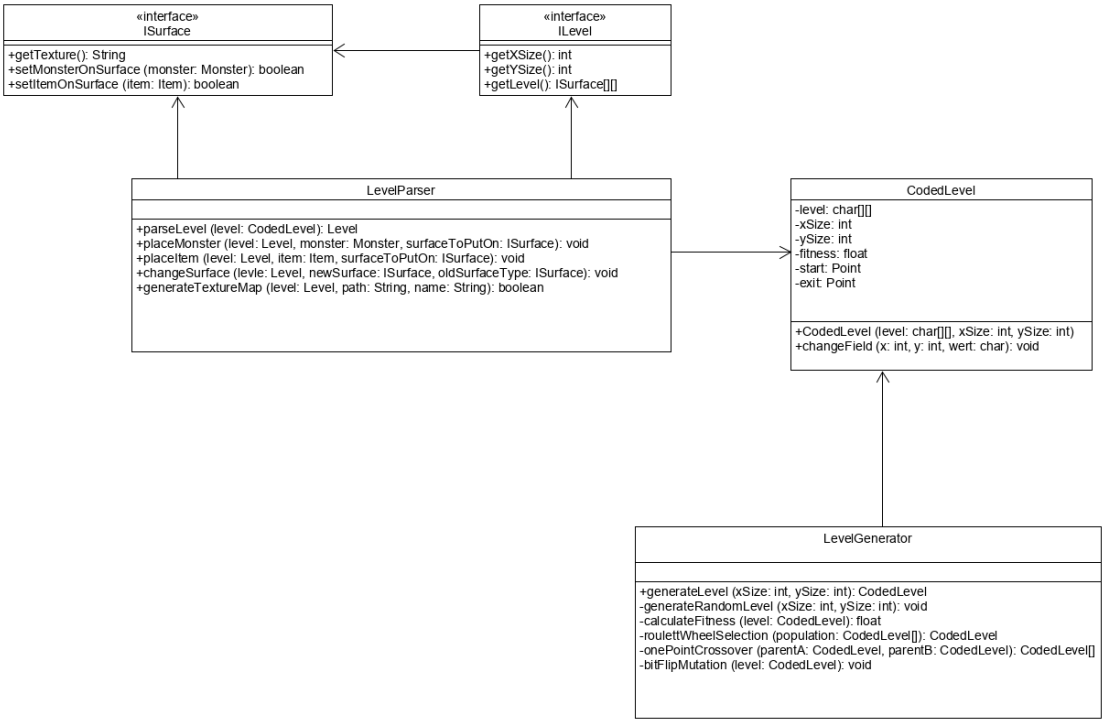


Abbildung 6.3.: UML LevelParser und Interfaces. Eigene Grafik

6.2.3. Locks and Keys

Um ein sinnvolles Konzept zur Platzierung von Türen und Schlüsseln zu entwickeln, muss erst ein Eindruck erlangt werden, wie die generierten Level aufgebaut sind. Grundsätzlich müssen Raumähnliche Strukturen erkannt werden, welche sich dadurch auszeichnen, das sie vom Start Punkt aus nur über ein Feld erreichbar sind, der Tür. Auf diesen Feld kann die Tür platziert werden, der Key wird zwischen Start und Level verteilt.

Das Klassendiagramm für den kompletten Generator ist in Abbildung … zu sehen. .



6.2.4. Methoden zur Auswertung und optimierung

Um den Einfluss der verschiedenen Parameter nachvollziehen zu können, wird ein Logger in den GA implementiert. Es werden alle relevanten Parameter sowie Ausgabe Daten geloggt. Dazu zählen:

- Größe der Level
- Populationsgröße
- Punkte Verteilung der für die einzelnen Fitness Kriterien
- Chance für Rekombination und Mutation
- Durchschnittlich erreichte Fitness der Lösung
- Durchschnittlich gebrauchte Generationen für die beste Lösung

Bevor der Fitnessschwellwert implementiert wird, wird die Abbruchbedingung so bestimmt, das eine feste Anzahl an Generationen durchlaufen werden, die zurück gelieferte Lösung entspricht der Lösung mit der höchsten Fitness über den gesamten Generierungsprozess. Ohne Schwellwert wird sichergestellt das der Generator nicht frühzeitig abbricht. Mithilfe der so erlangten Daten lassen sich mangelhafte Mutations und Rekombinations Methoden erkennen.

6.3. Unterschied zu bekannten Methoden der prozeduralen Levelgenerierung

7. Realisierung, Evaluation

7.1. Level Generator

7.1.1. generateLevel

```
1  public CodedLevel generateLevel(final int xSize, final int ySize,
       final int fitnessVersion,
2              final int parentSlectionVersion, final int
                   crossoverVersion, final int mutationVersion)
3              throws IllegalArgumentException {
4          if (xSize < Constants.MINIMAL_XSIZE || ySize < Constants.
               MINIMAL_YSIZE)
5              throw new IllegalArgumentException(
6                      "Size must be at least " + Constants.
                           MINIMAL_XSIZE + "x" + Constants.
                           MINIMAL_YSIZE);
7
8
9          this.generationLog = 0;
10         CodedLevel bestLevel = null;
11
12         // Startgeneration erzeugen
13         CodedLevel[] startPopulation = new CodedLevel[Constants.
               POPULATIONSIZE];
14         for (int i = 0; i < Constants.POPULATIONSIZE; i++)
15             startPopulation[i] = (generateRandomLevel(xSize, ySize
                   ));
16
17         // Durchlauf
18         for (int generation = 0; generation < Constants.
               MAXIMAL_GENERATION; generation++) {
19
20             // Start und Exit platzieren, Fitness pruefen
21             for (CodedLevel lvl : startPopulation) {
22                 placeStartAndEnd(lvl);
23
24                 float fitness;
25                 switch (fitnessVersion) {
26                 case 1:
27                     fitness = fitness1(lvl);
28                     break;
29                 case 2:
30                     fitness = fitness2(lvl);
31                 default:
```

```
32                      fitness = fitness1(lvl);
33                  }
34
35                  lvl.setFitness(fitness);
36                  lvl.resetList();
37
38                  if ((bestLevel == null || bestLevel.getFitness() <
                        fitness)
39                          && isReachable(lvl, lvl.getExit().x, lvl.
                            getExit().y)) {
40                      bestLevel = lvl.copyLevel();
41                      generationLog = generation + 1;
42                  }
43
44              }
45
46              // Kombinieren
47              CodedLevel[] newPopulation = new CodedLevel[Constants.
                    POPULATIONSIZE];
48              for (int i = 0; i < startPopulation.length; i += 2) {
49
50                  // Elternpaar Auswaehlen
51                  CodedLevel parentA;
52                  CodedLevel parentB;
53
54                  switch (parentSlectionVersion) {
55                  case 1:
56                      parentA = roulettWheelSelection(
                            startPopulation);
57                      break;
58                  default:
59                      parentA = roulettWheelSelection(
                            startPopulation);
60                  }
61
62                  do {
63                      switch (parentSlectionVersion) {
64                      case 1:
65                          parentB = roulettWheelSelection(
                                startPopulation);
66                          break;
67                      default:
68                          parentB = roulettWheelSelection(
                                startPopulation);
69                      }
70                  } while (parentA == parentB);
71
72                  if (Math.random() <= Constants.
                        CHANCE_FOR_CROSSOVER) {
73                      CodedLevel combined[];
74                      switch (crossoverVersion) {
75                      case 1:
76                          combined = onePointCrossover(parentA,
                                parentB);
77                          newPopulation[i] = combined[0];
78                          newPopulation[i + 1] = combined[1];
```

```
79                          break;
80                      case 2:
81                          combined = multipointCrossover(parentA,
                                parentB);
82                          newPopulation[i] = combined[0];
83                          newPopulation[i + 1] = combined[1];
84                          break;
85                      default:
86                          combined = onePointCrossover(parentA,
                                parentB);
87                          newPopulation[i] = combined[0];
88                          newPopulation[i + 1] = combined[1];
89                      }
90
91                  } else {
92                      newPopulation[i] = parentA;
93                      newPopulation[i + 1] = parentB;
94                  }
95              }
96
97              // Mutieren
98              for (CodedLevel lvl : newPopulation) {
99                  switch (mutationVersion) {
100                 case 1:
101                     bitFlipMutation(lvl);
102                     break;
103                 case 2:
104                     fixRowMutation(lvl);
105                     break;
106                 case 3:
107                     gameOfLifeMutation(lvl);
108                     break;
109                 default:
110                     bitFlipMutation(lvl);
111                     break;
112                 }
113             }
114
115             // Neue Population ist die Startpopulation fuer die
                    naehste Generation
116             startPopulation = newPopulation;
117
118
119
120             for (CodedLevel lvl : newPopulation)
121                 if ((bestLevel == null || bestLevel.getFitness() <
                        lvl.getFitness())
122                         && isReachable(lvl, lvl.getExit().x, lvl.
                            getExit().y)) {
123                     bestLevel = lvl.copyLevel();
124                     this.generationLog = generation + 1;
125                 }
126
127
128
129         }
```

```
130         removeUnreachableFloors(bestLevel);
131         return bestLevel;
132     }
```

7.1.2. GenerateRandomLevel

```
1  char[][] level = new char[xSize][ySize];
2          for (int y = 0; y < ySize; y++) {
3              for (int x = 0; x < xSize; x++) {
4
5                  if (x == 0 || y == 0 || y == ySize - 1 || x ==
                       xSize - 1)
6                      level[x][y] = Constants.REFERENCE_WALL;
7
8                  else if (Math.random() <= Constants.
                       CHANCE_TO_BE_FLOOR)
9                      level[x][y] = Constants.REFERENCE_FLOOR;
10                 else
11                     level[x][y] = Constants.REFERENCE_WALL;
12
13             }
14         }
15
16         return new CodedLevel(level, xSize, ySize);
17     }
```

7.1.3. Place Start and End

```
1  boolean change = false;
2
3          if (!lvl.hasStart()) {
4              do {
5                  // xSize druch 3 damit der Eingang im linken
                       drittel spawnt
6                  int x = (int) (Math.random() * lvl.getXSize() / 3)
                       ;
7                  int y = (int) (Math.random() * lvl.getYSize());
8                  if (y != 0 && y != lvl.getYSize() - 1 && x != 0 &&
                        x != lvl.getXSize() - 1) {
9                      lvl.changeField(x, y, Constants.
                           REFERENCE_START);
10                     change = true;
11                 }
12             } while (!change);
13             change = false;
14         }
15         if (!lvl.hasExit()) {
16             do {
17                 // Exit soll im rechten drittel Spawnen
18                 int x = (int) (Math.random() * lvl.getXSize() / 3)
                       + 2 * (int) (lvl.getXSize() / 3);
```

```
19                  int y = (int) (Math.random() * lvl.getYSize());
20                  if (y != 0 && y != lvl.getYSize() - 1 && x != 0 &&
                        x != lvl.getXSize() - 1) {
21                      lvl.changeField(x, y, Constants.REFEERNCE_EXIT
                            );
22                      change = true;
23                  }
24              } while (!change);
25          }
26      }
```

7.1.4. fitness1

```
1       private float fitness1(final CodedLevel level) {
2           float fitness = 0f;
3           level.resetList();
4           for (int x = 1; x < level.getXSize() - 1; x++) {
5               for (int y = 1; y < level.getYSize() - 1; y++) {
6                   if (level.getLevel()[x][y] == Constants.
                        REFERENCE_WALL) {
7                       if (isConnected(level, x, y))
8                           fitness += Constants.WALL_IS_CONNECTED;
9                       else if (level.getCheckedWalls().size() > 1) {
10                          fitness += Constants.WALL_HAS_NEIGHBOR;
11                      }
12                      level.resetWallList();
13                  } else if (level.getLevel()[x][y] == Constants.
                        REFEERNCE_EXIT) {
14                      if (isReachable(level, x, y))
15                          fitness += Constants.EXIT_IS_REACHABLE;
16
17                  } else if (level.getLevel()[x][y] == Constants.
                        REFERENCE_FLOOR && isReachable(level, x, y))
18                      fitness += Constants.FLOOR_IS_REACHABLE;
19              }
20
21          }
22
23          return fitness;
24      }
```

7.1.5. fitness2

```
1       private float fitness2(final CodedLevel lvl) {
2           float fitness = 0;
3           lvl.resetList();
4           for (int x = 1; x < lvl.getXSize() - 1; x++) {
5               for (int y = 1; y < lvl.getYSize() - 1; y++) {
6                   if (lvl.getLevel()[x][y] == Constants.
                        REFERENCE_WALL) {
```

```
7                       if (isConnected(lvl, x, y))
8                           fitness += Constants.WALL_IS_CONNECTED;
9                       if (lvl.getLevel()[x - 1][y] != Constants.
                            REFERENCE_WALL)
10                          fitness += Constants.
                                WALL_NEIGHBOR_IS_FLOOR;
11                      if (lvl.getLevel()[x + 1][y] != Constants.
                            REFERENCE_WALL)
12                          fitness += Constants.
                                WALL_NEIGHBOR_IS_FLOOR;
13                      if (lvl.getLevel()[x][y - 1] != Constants.
                            REFERENCE_WALL)
14                          fitness += Constants.
                                WALL_NEIGHBOR_IS_FLOOR;
15                      if (lvl.getLevel()[x][y + 1] != Constants.
                            REFERENCE_WALL)
16                          fitness += Constants.
                                WALL_NEIGHBOR_IS_FLOOR;
17                  } else if (lvl.getLevel()[x][y] == Constants.
                        REFEERNCE_EXIT) {
18                      if (isReachable(lvl, x, y))
19                          fitness += Constants.EXIT_IS_REACHABLE;
20                  } else if (lvl.getLevel()[x][y] == Constants.
                        REFERENCE_FLOOR && isReachable(lvl, x, y))
21                      fitness += Constants.FLOOR_IS_REACHABLE;
22
23              }
24
25          }
26          if (fitness <= 0)
27              fitness = 1;
28          return fitness;
29      }
```

7.1.6. is Connected

```
1       private boolean isConnected(final CodedLevel level, final int
            x, final int y) {
2
3           if (level.getLevel()[x][y] != Constants.REFERENCE_WALL)
4               throw new IllegalArgumentException("Surface must be a
                    wall");
5           // Wenn Wand Ausenwand ist -> return true
6           if (x == level.getXSize() - 1 || x == 0 || y == level.
                getYSize() - 1 || y == 0)
7               return true;
8
9           // Hinzufuegen der Wall um loops zu verhindern.
10          level.getCheckedWalls().add(x + "" + y);
11          boolean connected = false;
12          // Rekursiver aufurf mit allen Nachbarn
13          if (level.getLevel()[x - 1][y] == Constants.REFERENCE_WALL
14                  && !level.getCheckedWalls().contains((x - 1) + ""
```

```
                    + y))
15              if (isConnected(level, x - 1, y))
16                  connected = true;
17          if (!connected && level.getLevel()[x + 1][y] == Constants.
                REFERENCE_WALL
18                  && !level.getCheckedWalls().contains((x + 1) + ""
                        + y))
19              if (isConnected(level, x + 1, y))
20                  connected = true;
21          if (!connected && level.getLevel()[x][y - 1] == Constants.
                REFERENCE_WALL
22                  && !level.getCheckedWalls().contains(x + "" + (y -
                        1)))
23              if (isConnected(level, x, y - 1))
24                  connected = true;
25          if (!connected && level.getLevel()[x][y + 1] == Constants.
                REFERENCE_WALL
26                  && !level.getCheckedWalls().contains(x + "" + (y +
                        1)))
27              if (isConnected(level, x, y + 1))
28                  connected = true;
29
30          return connected;
31      }
```

7.1.7. is reachable

```
1       private boolean isReachable(final CodedLevel level, final int
            x, final int y) {
2           if (level.getLevel()[x][y] != Constants.REFERENCE_FLOOR &&
                level.getLevel()[x][y] != Constants.REFEERNCE_EXIT
3                   && level.getLevel()[x][y] != Constants.
                        REFERENCE_START)
4               throw new IllegalArgumentException("Surface must be a
                    floor. Is " + level.getLevel()[x][y]);
5
6           if (level.getReachableFloors().size() <= 0)
7               createReachableList(level, level.getStart().x,level.
                    getStart().y);
8
9           return level.getReachableFloors().contains(x + "_" + y);
10
11      }
```

7.1.8. createRechableList

```
1   private void createReachableList(final CodedLevel level, final int
        x, final int y) {
2           if (level.getLevel()[x][y] != Constants.REFERENCE_FLOOR &&
                level.getLevel()[x][y] != Constants.REFEERNCE_EXIT
```

```
3                 && level.getLevel()[x][y] != Constants.
                      REFERENCE_START)
4             throw new IllegalArgumentException("Surface must be a
                  floor Is " + level.getLevel()[x][y]);
5
6         level.getReachableFloors().add(x + "_" + y);
7
8         if ((level.getLevel()[x - 1][y] == Constants.
              REFERENCE_FLOOR
9                 || level.getLevel()[x - 1][y] == Constants.
                      REFEERNCE_EXIT
10                || level.getLevel()[x - 1][y] == Constants.
                      REFERENCE_START)
11                && !level.getReachableFloors().contains((x - 1) +
                      "_" + y))
12            createReachableList(level, x - 1, y);
13
14        if ((level.getLevel()[x + 1][y] == Constants.
              REFERENCE_FLOOR
15                || level.getLevel()[x + 1][y] == Constants.
                      REFEERNCE_EXIT
16                || level.getLevel()[x + 1][y] == Constants.
                      REFERENCE_START)
17                && !level.getReachableFloors().contains((x + 1) +
                      "_" + y))
18            createReachableList(level, x + 1, y);
19
20        if ((level.getLevel()[x][y - 1] == Constants.
              REFERENCE_FLOOR
21                || level.getLevel()[x][y - 1] == Constants.
                      REFEERNCE_EXIT
22                || level.getLevel()[x][y - 1] == Constants.
                      REFERENCE_START)
23                && !level.getReachableFloors().contains(x + "_" +
                      (y - 1)))
24            createReachableList(level, x, y - 1);
25
26        if ((level.getLevel()[x][y + 1] == Constants.
              REFERENCE_FLOOR
27                || level.getLevel()[x][y + 1] == Constants.
                      REFEERNCE_EXIT
28                || level.getLevel()[x][y + 1] == Constants.
                      REFERENCE_START)
29                && !level.getReachableFloors().contains(x + "_" +
                      (y + 1)))
30            createReachableList(level, x, y + 1);
31    }
```

7.1.9. roulettWheelSelection

```
1     private CodedLevel roulettWheelSelection(final CodedLevel[]
          population) {
2         int fitSum = 0;
```

```
3           for (CodedLevel lvl : population) {
4               fitSum += lvl.getFitness();
5           }
6           int target = (int) (Math.random() * fitSum);
7           int p = 0;
8
9           for (CodedLevel lvl : population) {
10              p += lvl.getFitness();
11              if (p >= target)
12                  return lvl;
13          }
14
15          // non reachable code
16          return null;
17      }
```

7.1.10. onePointCrossover

```
1  private CodedLevel[] onePointCrossover(final CodedLevel lvl1,
       final CodedLevel lvl2) {
2
3           CodedLevel newLevelA = new CodedLevel(new char[lvl1.
               getXSize()][lvl1.getYSize()], lvl1.getXSize(),
4                   lvl1.getYSize());
5
6           for (int x = 0; x < lvl1.getXSize(); x++) {
7               for (int y = 0; y < lvl1.getYSize(); y++) {
8
9                   if (x >= lvl1.getXSize() / 2)
10                      newLevelA.changeField(x, y, lvl2.getLevel()[x
                            ][y]);
11                  else
12                      newLevelA.changeField(x, y, lvl1.getLevel()[x
                            ][y]);
13              }
14          }
15
16          CodedLevel newLevelB = new CodedLevel(new char[lvl1.
               getXSize()][lvl1.getYSize()], lvl1.getXSize(),
17                  lvl1.getYSize());
18          for (int x = 0; x < lvl1.getXSize(); x++) {
19              for (int y = 0; y < lvl1.getYSize(); y++) {
20                  if (x >= lvl1.getXSize() / 2)
21                      newLevelB.changeField(x, y, lvl1.getLevel()[x
                            ][y]);
22                  else
23                      newLevelB.changeField(x, y, lvl2.getLevel()[x
                            ][y]);
24              }
25          }
26          CodedLevel c[] = { newLevelA, newLevelB };
27
28          return c;
```

```
29      }
```

7.1.11. multiPointCrossover

```
1  private CodedLevel[] multipointCrossover(final CodedLevel lvl1,
       final CodedLevel lvl2) {
2          int cut1 = ((int) (Math.random() * lvl1.getYSize()));
3          int cut2 = ((int) (Math.random() * lvl1.getYSize()));
4
5          if (cut1 > cut2) {
6              int temp = cut1;
7              cut1 = cut2;
8              cut2 = temp;
9          }
10         CodedLevel newLevelA = new CodedLevel(new char[lvl1.
               getXSize()][lvl1.getYSize()], lvl1.getXSize(),
11                 lvl1.getYSize());
12
13         for (int y = 0; y < lvl1.getYSize(); y++) {
14             for (int x = 0; x < lvl1.getXSize(); x++) {
15                 if (y >= cut1 && y <= cut2)
16                     newLevelA.changeField(x, y, lvl2.getLevel()[x
                           ][y]);
17                 else
18                     newLevelA.changeField(x, y, lvl1.getLevel()[x
                           ][y]);
19             }
20         }
21
22         CodedLevel newLevelB = new CodedLevel(new char[lvl1.
               getXSize()][lvl1.getYSize()], lvl1.getXSize(),
23                 lvl1.getYSize());
24         for (int y = 0; y < lvl1.getYSize(); y++) {
25             for (int x = 0; x < lvl1.getXSize(); x++) {
26                 if (y >= cut1 && y <= cut2)
27                     newLevelB.changeField(x, y, lvl1.getLevel()[x
                           ][y]);
28                 else
29                     newLevelB.changeField(x, y, lvl2.getLevel()[x
                           ][y]);
30             }
31         }
32
33         CodedLevel c[] = { newLevelA, newLevelB };
34         return c;
35     }
```

7.1.12. Bit Flip Mutation

```
1  private void bitFlipMutation(final CodedLevel lvl) {
```

```
        for (int y = 1; y < lvl.getYSize() - 1; y++) {
            for (int x = 1; x < lvl.getXSize() - 1; x++) {
                if (Math.random() <= Constants.CHANCE_FOR_MUTATION
                    ) {
                    if ((lvl.getLevel())[x][y] == Constants.
                        REFERENCE_WALL)
                        lvl.changeField(x, y, Constants.
                            REFERENCE_FLOOR);
                    else
                        lvl.changeField(x, y, Constants.
                            REFERENCE_WALL);
                }
            }
        }
    }
```

7.1.13. fixRowMutation

```
private void fixRowMutation(final CodedLevel lvl) {
        for (int y = 2; y < lvl.getYSize() - 2; y++) {
            if (Math.random() < Constants.CHANCE_FOR_MUTATION) {
                for (int x = 2; x < lvl.getXSize() - 2; x++) {
                    if (lvl.getLevel()[x][y] == Constants.
                        REFERENCE_WALL) {
                        lvl.resetWallList();
                        int actX = x;
                        boolean first = true;

                        while (!isConnected(lvl, actX, y)) {

                            // Nach rechts schieben
                            if (actX >= lvl.getXSize() / 2) {
                                // Vertauschen

                                char c = lvl.getLevel()[actX + 1][
                                    y];
                                lvl.changeField(actX + 1, y,
                                    Constants.REFERENCE_WALL);
                                lvl.changeField(actX, y, c);

                                actX++;

                                // Wenn ein Surface nach rechts
                                    geschoben wird, muss die alte
                                    Position erneut
                                // ueberprueft werden
                                // da evtl. dort wieder eine Wand
                                    steht.
                                if (first) {
                                    first = false;
                                    --x;
                                }
                            }
```

```
30                                  // Nach links schieben
31                                  else {
32                                      // vertauschen
33                                      char c = lvl.getLevel()[actX - 1][
                                            y];
34                                      lvl.changeField(actX - 1, y,
                                            Constants.REFERENCE_WALL);
35                                      lvl.changeField(actX, y, c);
36                                      // Beim naechsten Run selbes
                                            Surface an neuer Position
                                            ueberpruefen
37                                      actX--;
38
39                                  }
40
41                              }
42                          }
43                      }
44
45                  }
46              }
47          }
```

7.1.14. gameOfLife

```
1  private void gameOfLifeMutation(CodedLevel level) {
2          CodedLevel tempLevel = level.copyLevel();
3          for (int x = 1; x < tempLevel.getXSize() - 1; x++) {
4              for (int y = 1; y < tempLevel.getYSize() - 1; y++) {
5                  int lebendeNachbarn = 0;
6                  if (Math.random() <= Constants.CHANCE_FOR_MUTATION
                       ) {
7                      if (level.getLevel()[x + 1][y] == Constants.
                           REFERENCE_WALL)
8                          lebendeNachbarn++;
9                      if (level.getLevel()[x - 1][y] == Constants.
                           REFERENCE_WALL)
10                         lebendeNachbarn++;
11                     if (level.getLevel()[x][y + 1] == Constants.
                           REFERENCE_WALL)
12                         lebendeNachbarn++;
13                     if (level.getLevel()[x][y - 1] == Constants.
                           REFERENCE_WALL)
14                         lebendeNachbarn++;
15                     if (lebendeNachbarn == 3 && level.getLevel()[x
                           ][y] == Constants.REFERENCE_FLOOR)
16                         tempLevel.changeField(x, y, Constants.
                               REFERENCE_WALL);
17                     else if ((lebendeNachbarn < 1 ||
                           lebendeNachbarn <= 3)
18                             && level.getLevel()[x][y] == Constants
                                   .REFERENCE_WALL) {
19                         tempLevel.changeField(x, y, Constants.
```

```
                                REFERENCE_FLOOR);
20                      }
21
22                  }
23              }
24
25          }
26          level = tempLevel.copyLevel();
27      }
```

7.1.15. remove unrechable floors

```
1   private void removeUnreachableFloors(final CodedLevel lvl) {
2           lvl.resetList();
3           for (int x = 0; x < lvl.getXSize(); x++) {
4               for (int y = 0; y < lvl.getYSize(); y++) {
5                   if (lvl.getLevel()[x][y] == Constants.
                        REFERENCE_FLOOR) {
6                       if (!isReachable(lvl, x, y))
7                           lvl.changeField(x, y, Constants.
                                REFERENCE_WALL);
8                   }
9               }
10          }
11      }
```

7.2. CodedLevel

7.2.1. changeField

```
1   public void changeField(int x, int y, char s) {
2           if (this.level[x][y] == Constants.REFEERNCE_EXIT)
3               this.exit = null;
4
5           else if (this.level[x][y] == Constants.REFERENCE_START)
6               this.start = null;
7
8           if (s == Constants.REFEERNCE_EXIT) {
9               if (this.hasExit())
10                  s = Constants.REFERENCE_FLOOR;
11              else {
12                  this.exit=new int[2];
13                  this.exit[0] = x;
14                  this.exit[1] = y;
15              }
16          } else if (s == Constants.REFERENCE_START) {
17              if (this.hasStart())
18                  s = Constants.REFERENCE_FLOOR;
19              else {
```

```
20                  this.start=new int[2];
21                  this.start[0] = x;
22                  this.start[1] = y;
23              }
24          }
25          this.level[x][y] = s;
26      }
```

7.3. LevelParser

7.3.1. parseLevel

```
1  public Level parseLevel(final CodedLevel level) {
2          ISurface[][] lvl = new ISurface[level.getXSize()][level.
               getYSize()];
3
4          for (int x = 0; x < level.getXSize(); x++) {
5              for (int y = 0; y < level.getYSize(); y++) {
6                  if (level.getLevel()[x][y] == Constants.
                       REFERENCE_WALL)
7                      lvl[x][y] = new Wall();
8                  else if (level.getLevel()[x][y] == Constants.
                       REFERENCE_FLOOR)
9                      lvl[x][y] = new Floor();
10                 else if (level.getLevel()[x][y] == Constants.
                       REFERENCE_START)
11                     lvl[x][y] = new Start();
12                 else if (level.getLevel()[x][y] == Constants.
                       REFEERNCE_EXIT)
13                     lvl[x][y] = new Exit();
14
15             }
16         }
17
18         return new Level(level.getXSize(), level.getYSize(), lvl);
19     }
```

7.3.2. place

```
1  public void placeMonster(final Level lvl, final Monster monster,
       final ISurface surfaceToPutOn) throws Exception {
2          ArrayList<ISurface> checkedSurfaces = new ArrayList<
               ISurface>();
3          while (checkedSurfaces.size() < lvl.getXSize() * lvl.
               getYSize()) {
4              int x = (int) Math.random() * lvl.getXSize();
5              int y = (int) Math.random() * lvl.getYSize();
6              if (!checkedSurfaces.contains(lvl.getLevel()[x][y])
```

```
7                       && lvl.getLevel()[x][y].getClass() ==
                            surfaceToPutOn.getClass()
8                       && lvl.getLevel()[x][y].setMonsterOnSurface(
                            monster))
9                   return;
10              else
11                  checkedSurfaces.add(lvl.getLevel()[x][y]);
12
13          }
14          throw new Exception("Level voll");
15      }
```

7.3.3. Texturen Generieren

```
1       public boolean generateTextureMap(final Level lvl, final
            String path, final String name) {
2           try {
3               BufferedImage img1;
4               BufferedImage joinedImgLine = null;
5               BufferedImage joinedImgComplete = null;
6               img1 = ImageIO.read(new File(lvl.getLevel()[0][0].
                    getTexture()));
7               for (int y = 0; y < lvl.getYSize(); y++) {
8                   for (int x = 1; x < lvl.getXSize(); x++) {
9                       BufferedImage img2 = ImageIO.read(new File(lvl
                            .getLevel()[x][y].getTexture()));
10                      joinedImgLine = JoinImage.
                            joinBufferedImageSide(img1, img2);
11                      img1 = joinedImgLine;
12                  }
13                  if (y == 0)
14                      joinedImgComplete = joinedImgLine;
15                  else
16                      joinedImgComplete = JoinImage.
                            joinBufferedImageDown(joinedImgComplete,
                            joinedImgLine);
17                  img1 = ImageIO.read(new File(lvl.getLevel()[0][y].
                        getTexture()));
18              }
19              boolean success = ImageIO.write(joinedImgComplete, "
                    png", new File(path + "\\" + name + ".png"));
20              System.out.println("saved success? " + success);
21              if (!success)
22                  return false;
23
24          } catch (IOException e) {
25              e.printStackTrace();
26              return false;
27          }
28          return true;
29      }
30  }
```

```
    public static BufferedImage joinBufferedImageSide(final
        BufferedImage img1, final BufferedImage img2) {
        int wid = img1.getWidth() + img2.getWidth();
        int height = Math.max(img1.getHeight(), img2.getHeight());
        BufferedImage newImage = new BufferedImage(wid, height,
            BufferedImage.TYPE_INT_ARGB);
        Graphics2D g2 = newImage.createGraphics();
        Color oldColor = g2.getColor();
        g2.setPaint(Color.WHITE);
        g2.fillRect(0, 0, wid, height);
        g2.setColor(oldColor);
        g2.drawImage(img1, null, 0, 0);
        g2.drawImage(img2, null, img1.getWidth(), 0);
        g2.dispose();
        return newImage;
    }

    public static BufferedImage joinBufferedImageDown(final
        BufferedImage img1, final BufferedImage img2) {
        {
            int wid = Math.max(img1.getWidth(), img2.getWidth());
            int height = img1.getHeight() + img2.getHeight();
            BufferedImage newImage = new BufferedImage(wid, height
                , BufferedImage.TYPE_INT_ARGB);
            Graphics2D g2 = newImage.createGraphics();
            Color oldColor = g2.getColor();
            g2.setPaint(Color.WHITE);
            g2.fillRect(0, 0, wid, height);
            g2.setColor(oldColor);
            g2.drawImage(img1, null, 0, 0);
            g2.drawImage(img2, null, 0, img1.getHeight());
            g2.dispose();
            return newImage;
        }
    }
}
```

```
    public static BufferedImage joinBufferedImageSide(final
        BufferedImage img1, final BufferedImage img2) {
        int wid = img1.getWidth() + img2.getWidth();
        int height = Math.max(img1.getHeight(), img2.getHeight());
        BufferedImage newImage = new BufferedImage(wid, height,
            BufferedImage.TYPE_INT_ARGB);
        Graphics2D g2 = newImage.createGraphics();
        Color oldColor = g2.getColor();
        g2.setPaint(Color.WHITE);
        g2.fillRect(0, 0, wid, height);
        g2.setColor(oldColor);
        g2.drawImage(img1, null, 0, 0);
        g2.drawImage(img2, null, img1.getWidth(), 0);
        g2.dispose();
        return newImage;
```

```
14      }
15
16          public static BufferedImage joinBufferedImageDown(final
                BufferedImage img1, final BufferedImage img2) {
17          {
18              int wid = Math.max(img1.getWidth(), img2.getWidth());
19              int height = img1.getHeight() + img2.getHeight();
20              BufferedImage newImage = new BufferedImage(wid, height
                    , BufferedImage.TYPE_INT_ARGB);
21              Graphics2D g2 = newImage.createGraphics();
22              Color oldColor = g2.getColor();
23              g2.setPaint(Color.WHITE);
24              g2.fillRect(0, 0, wid, height);
25              g2.setColor(oldColor);
26              g2.drawImage(img1, null, 0, 0);
27              g2.drawImage(img2, null, 0, img1.getHeight());
28              g2.dispose();
29              return newImage;
30          }
31      }
32  }
```

7.3.4. Room ZUsamensetzer

```
1   package generator;
2
3   import java.util.ArrayList;
4
5   public class ConnectedRoomGenerator {
6
7       public CodedLevel generateLevelWithRooms(final int xSize,
            final int ySize) {
8
9           CodedLevel level = new CodedLevel(new char[ (int)(xSize *
                2.5)][ (int)(ySize * 2.5)], (int)(xSize * 2.5), (int)(
                ySize * 2.5));
10          ArrayList<CodedRoom> rooms = new ArrayList<CodedRoom>();
11
12          for (int x = 0; x < level.getXSize(); x++)
13              for (int y = 0; y < level.getYSize(); y++)
14                  level.getLevel()[x][y] = Constants.REFERENCE_EMPTY
                        ;
15          int fe = xSize * ySize;
16          LevelGenerator lg = new LevelGenerator();
17          while (fe > 0) {
18              System.out.println("gen Level");
19              int xp = (int) (Math.random() * xSize);
20              int yp = (int) (Math.random() * ySize);
21              if (xp < Constants.MINIMAL_XSIZE)
22                  xp = Constants.MINIMAL_XSIZE;
23              if (yp < Constants.MINIMAL_YSIZE)
24                  yp = Constants.MINIMAL_YSIZE;
25              CodedLevel rl = lg.generateLevel(xp, yp, 2, 1, 2, 2);
```

```
26              CodedRoom r = new CodedRoom(rl.getLevel(), rl.getXSize
                    (), rl.getYSize());
27              r.changeField(rl.getStart().x, rl.getStart().y,
                    Constants.REFERENCE_START);
28              r.changeField(rl.getExit().x, rl.getExit().y,
                    Constants.REFEERNCE_EXIT);
29              rooms.add(r);
30              fe -= xp * yp;
31          }
32
33          // Bestimme Raum in den Start und Exit sind
34          int startRoom = (int) (Math.random() * rooms.size());
35          int exitRoom = (int) (Math.random() * rooms.size());
36
37          // remove start and exit for each room
38          for (CodedLevel room : rooms) {
39              if (rooms.indexOf(room) != startRoom)
40                  room.changeField(room.getStart().x, room.getStart
                        ().y, Constants.REFERENCE_WALL);
41              if (rooms.indexOf(room) != exitRoom)
42                  room.changeField(room.getExit().x, room.getExit().
                        y, Constants.REFERENCE_WALL);
43          }
44
45          // Platziere rooms
46          level = placeRooms(rooms, level);
47
48
49          ArrayList<CodedRoom> notConnected = new ArrayList<
                CodedRoom>();
50          notConnected.add(rooms.get(0));
51          for (int j = 0; j < rooms.size() - 1; j++) {
52              CodedRoom room = notConnected.get(j);
53              int abstand = Integer.MAX_VALUE;
54              int index = j + 1;
55              for (int i = 0; i < rooms.size(); i++) {
56                  if (!notConnected.contains(rooms.get(j))) {
57                      int tempabstand = Math.abs(room.getMidPoint().
                            x - rooms.get(j).getMidPoint().x)
58                              + Math.abs(room.getMidPoint().y -
                                    rooms.get(j).getMidPoint().y);
59                      if (abstand > tempabstand) {
60                          abstand = tempabstand;
61                          index = j;
62                      }
63                  }
64
65              }
66              notConnected.add(rooms.get(index));
67          }
68          this.connectRoomsWithSquareFloors(level, notConnected);
69
70          // leere Felder mit Walls auffüllen
71          for (int x = 0; x < level.getXSize(); x++)
72              for (int y = 0; y < level.getYSize(); y++)
73                  if (level.getLevel()[x][y] == Constants.
```

```
                    REFERENCE_EMPTY)
74                   level.changeField(x, y, Constants.
                        REFERENCE_WALL);
75
76          return level;
77
78      }
79
80      private CodedLevel placeRooms(ArrayList<CodedRoom> rooms,
            CodedLevel lvl) {
81          CodedLevel level = lvl.copyLevel();
82          int counter = 0;
83          int placed = 0;
84          while (placed < rooms.size() && counter < 100) {
85              System.out.println("place room");
86              CodedRoom room = rooms.get(placed);
87              int yp;
88              int xp;
89              do {
90                  xp = (int) (Math.random() * (level.getXSize() -
                        room.getXSize()));
91                  yp = (int) (Math.random() * (level.getYSize() -
                        room.getYSize()));
92              } while (xp + room.getXSize() > level.getXSize() || yp
                     + room.getYSize() > level.getYSize());
93
94              boolean free = true;
95
96              // checken ob kein andere raum im weg ist
97              for (int x = xp; x < xp + room.getXSize() - 1; x++)
98                  for (int y = yp; y < yp + room.getYSize() - 1; y
                        ++)
99                      if (level.getLevel()[x][y] != Constants.
                            REFERENCE_EMPTY)
100                         free = false;
101
102             if (free) {
103
104                 for (int x = xp; x < xp + room.getXSize(); x++) {
105
106                     for (int y = yp; y < yp + room.getYSize(); y
                            ++) {
107
108                         level.changeField(x, y, room.getLevel()[x
                                - xp][y - yp]);
109                     }
110
111                 }
112                 room.setUpperLeftCorner(xp, yp);
113                 room.setUpperRightCorner(xp + room.getXSize(), yp)
                        ;
114                 room.setLowerLeftCorner(xp, yp + room.getYSize());
115                 room.setLowerRightCorner(xp + room.getXSize(), yp
                        + room.getYSize());
116                 room.setMidPoint(xp + room.getXSize() / 2, yp +
                        room.getYSize() / 2);
```

```
117                 placed++;
118             }
119
120             counter++;
121         }
122
123         if (counter >= 100)
124             return placeRooms(rooms, lvl);
125         else
126             return level;
127
128     }
129
130         private void createReachableList(final CodedLevel level,
                final int x, final int y) {
131         if (level.getLevel()[x][y] != Constants.REFERENCE_FLOOR &&
                level.getLevel()[x][y] != Constants.REFEERNCE_EXIT
132                 && level.getLevel()[x][y] != Constants.
                    REFERENCE_START)
133             throw new IllegalArgumentException("Surface must be a
                    floor Is " + level.getLevel()[x][y]);
134
135         level.getReachableFloors().add(x + "_" + y);
136
137         if ((level.getLevel()[x - 1][y] == Constants.
                REFERENCE_FLOOR
138                 || level.getLevel()[x - 1][y] == Constants.
                    REFEERNCE_EXIT
139                 || level.getLevel()[x - 1][y] == Constants.
                    REFERENCE_START)
140                 && !level.getReachableFloors().contains((x - 1) +
                    "_" + y))
141             createReachableList(level, x - 1, y);
142
143         if ((level.getLevel()[x + 1][y] == Constants.
                REFERENCE_FLOOR
144                 || level.getLevel()[x + 1][y] == Constants.
                    REFEERNCE_EXIT
145                 || level.getLevel()[x + 1][y] == Constants.
                    REFERENCE_START)
146                 && !level.getReachableFloors().contains((x + 1) +
                    "_" + y))
147             createReachableList(level, x + 1, y);
148
149         if ((level.getLevel()[x][y - 1] == Constants.
                REFERENCE_FLOOR
150                 || level.getLevel()[x][y - 1] == Constants.
                    REFEERNCE_EXIT
151                 || level.getLevel()[x][y - 1] == Constants.
                    REFERENCE_START)
152                 && !level.getReachableFloors().contains(x + "_" +
                    (y - 1)))
153             createReachableList(level, x, y - 1);
154
155         if ((level.getLevel()[x][y + 1] == Constants.
                REFERENCE_FLOOR
```

```
156                 || level.getLevel()[x][y + 1] == Constants.
                        REFEERNCE_EXIT
157                 || level.getLevel()[x][y + 1] == Constants.
                        REFERENCE_START)
158                 && !level.getReachableFloors().contains(x + "_" +
                        (y + 1)))
159             createReachableList(level, x, y + 1);
160     }
161
162     private void connectRoomsWithSquareFloors(CodedLevel level,
            ArrayList<CodedRoom> rooms) {
163
164         for (int i = 0; i < rooms.size() - 1; i++) {
165             CodedRoom r1 = rooms.get(i);
166             CodedRoom r2 = rooms.get(i + 1);
167
168             int xAbstand = r2.getMidPoint().x - r1.getMidPoint().x
                    ;
169             int yAbstand = r2.getMidPoint().y - r1.getMidPoint().y
                    ;
170             System.out.println(xAbstand);
171             System.out.println(yAbstand);
172             if (yAbstand < 0) {
173                 // links oben
174                 if (xAbstand < 0) {
175                     for (int y = Math.abs(yAbstand); y > 0; y--) {
176                         level.changeField(r1.getMidPoint().x, r1.
                                getMidPoint().y - y, Constants.
                                REFERENCE_FLOOR);
177                     }
178                     for (int x = Math.abs(xAbstand); x > 0; x--) {
179
180
181             if(level.getLevel()[r1.getMidPoint().x - x][ r1.
                    getMidPoint().y + yAbstand]!=Constants.
                    REFERENCE_START && level.getLevel()[r1.getMidPoint
                    ().x - x][ r1.getMidPoint().y + yAbstand]!=
                    Constants.REFEERNCE_EXIT)
182                         level.changeField(r1.getMidPoint().x - x,
                                r1.getMidPoint().y + yAbstand,
183                                     Constants.REFERENCE_FLOOR);
184                     }
185
186                     // recht oben
187                 } else {
188                     for (int y = Math.abs(yAbstand); y > 0; y--) {
189                         level.changeField(r1.getMidPoint().x, r1.
                                getMidPoint().y - y, Constants.
                                REFERENCE_FLOOR);
190                     }
191                     for (int x = 0; x <= xAbstand; x++) {
192
193                         if(level.getLevel()[r1.getMidPoint().x + x
                                ][ r1.getMidPoint().y + yAbstand]!=
                                Constants.REFERENCE_START && level.
                                getLevel()[r1.getMidPoint().x + x][ r1.
```

```
                                getMidPoint().y + yAbstand]!=Constants.
                                REFEERNCE_EXIT)
194                         level.changeField(r1.getMidPoint().x + x,
                                r1.getMidPoint().y + yAbstand,
195                                 Constants.REFERENCE_FLOOR);
196                     }
197
198                 }
199
200             } else {
201                 // links unten
202                 if (xAbstand < 0) {
203                     for (int y = 0; y <= yAbstand; y++) {
204                         level.changeField(r1.getMidPoint().x, r1.
                                getMidPoint().y + y, Constants.
                                REFERENCE_FLOOR);
205                     }
206
207                     for (int x = Math.abs(xAbstand); x > 0; x--) {
208
209                         if(level.getLevel()[r1.getMidPoint().x - x
                                ][ r1.getMidPoint().y + yAbstand]!=
                                Constants.REFERENCE_START && level.
                                getLevel()[r1.getMidPoint().x - x][ r1.
                                getMidPoint().y + yAbstand]!=Constants.
                                REFEERNCE_EXIT)
210                         level.changeField(r1.getMidPoint().x - x,
                                r1.getMidPoint().y + yAbstand,
211                                 Constants.REFERENCE_FLOOR);
212                     }
213
214                     // recht unten
215                 } else {
216
217                     for (int y = 0; y <= yAbstand; y++) {
218                         level.changeField(r1.getMidPoint().x, r1.
                                getMidPoint().y + y, Constants.
                                REFERENCE_FLOOR);
219                     }
220
221                     for (int x = 0; x <= xAbstand; x++) {
222                         if(level.getLevel()[r1.getMidPoint().x + x
                                ][ r1.getMidPoint().y + yAbstand]!=
                                Constants.REFERENCE_START && level.
                                getLevel()[r1.getMidPoint().x + x][ r1.
                                getMidPoint().y + yAbstand]!=Constants.
                                REFEERNCE_EXIT)
223                         level.changeField(r1.getMidPoint().x + x,
                                r1.getMidPoint().y + yAbstand,
224                                 Constants.REFERENCE_FLOOR);
225                     }
226                 }
227             }
228
229         }
230
```

```
231     }
232
233     public static void main(String[] args) {
234         ConnectedRoomGenerator cr = new ConnectedRoomGenerator();
235         LevelParser p = new LevelParser();
236         for (int i = 0; i < 10; i++)
237             p.generateTextureMap(p.parseLevel(cr.
                    generateLevelWithRooms(50, 50)), "./results/img", "
                    test_" + i);
238
239     }
240 }
```

7.3.4.1. Erweiterung um Räume und Flure

Um Level in verschiedene unteräume auzuteilen, ohne auf zufällige erzeugung, Raum ähnlicher Sturkturen zu hoffen, kann der Algorithmus erweitert werden. Hierzu werden die bisher erzeugten Level, als Raum interpretiert, zufällig im Level verteilt und mithilfe von Tunneln verbunden. Um Verbindungen zwischen den einzelnen Räume zu erzeugen, wird von jeden Raum aus, in unterschiedlicher Reihenfolge, jede Richtung nach angrenzenden Nachbar Räumen abgesucht, wird ein Raum gefunden, wird auf direkten Wege eine Verbindung hergestellt. Der so erzeugte Flur kann wiederum auch als Raum interpretiert werden und von anderen Räumen als anschlusspunkt genutzt werden. Es werden wieder zufällig Levelstart und Levelende platziert. Sollte ein Raum vom Start aus nicht erreichbar sein, weil sich in seiner Reichweite kein weiterer Raumbefindet, wird der nächste Verbunde Raum gesucht und eine Verbindung hergestellt.

Die Zusammensetzung der Level könnte auch mithilfe von GAs umgesetzt werden, wird in dieser Implementierung allerdings nicht gemacht. # Zusammenfassung

7.4. Fazit

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

7.5. Ausblick

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum

dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Literatur

A. Appendix 1: Some extra stuff

A.1. Bilder

wuppie!

A.2. Code

fluppie?

A.3. Tabellen

foo! bar!

B. Wuppie

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als entsprechend kenntlich gemacht.

Ich habe die Arbeit bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Das PDF-Exemplar stimmt mit den eingereichten Exemplaren überein.

Minden, den 05. März 2020