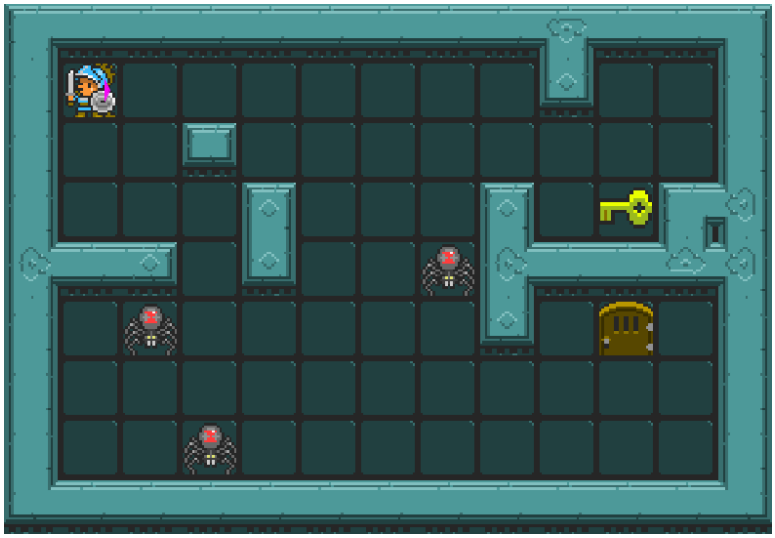
MSE | MASTER OF SCIENCE IN ENGINEERING

Lucerne University of Applied Sciences and Arts
HOCHSCHULE LUZERN

Master of Science in Engineering
Vertiefungsmodul I

Prozedurale Generierung einer Spielmechanik für Videospiele



Studierender: Manuel Jordan
manuel.jordan@stud.hslu.ch

Advisor: Prof. Dr. Thomas Koller
thomas.koller@hslu.ch

Abgabedatum: 20.08.2018

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema: „Prozedurale Generierung einer Spielmechanik für Videospiele“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum	Unterschrift

Danksagung

Hiermit möchte ich mich bei allen Personen und Organisationen bedanken, die mich während der Vertiefungsarbeit unterstützt und mir hilfreiche Ratschläge gegeben haben.

Ein besonderer Dank geht an meinen Betreuer Prof. Dr. Thomas Koller, der mich während dieser Zeit unterstützt hat.

Des Weiteren möchte ich mich auch bei Prof. Dr. Thilo Stadelmann und meinem Studienkollegen Patrik Deke für ihre Anregungen während der Arbeit danken.

Management Summary

Im Rahmen des Vertiefungsmoduls I des Master of Science in Engineering beschäftigt sich die vorliegende Arbeit mit der automatischen Erzeugung von Inhalten für Videospiele. Diese sogenannte prozedurale Generierung wird in Videospielen breit eingesetzt, die Anwendung auf Spielmechaniken und komplette Spiele ist jedoch noch sehr eingeschränkt. Die erfolgreichsten Ansätze basieren auf evolutionären Algorithmen. Bei dieser von der Natur inspirierten Methode wird eine Population von Spielen stetig weiterentwickelt. Die Evaluation von einzelnen Spielen erfolgt meistens mit einer Simulation des Spiels.

In dieser Arbeit wurde ein genetischer Algorithmus entwickelt, um mit dem GVGAI-Framework Spiele zu generieren, welche in der Video Game Description Language kodiert sind. Erste Tests bewiesen grundsätzlich die Funktion des Generators. Durch Optimierung der Hyperparameter konnte die Performance verbessert werden.

Die erzeugten Spiele erfüllen grundsätzlich die gestellten Anforderungen, sind jedoch ohne Nacharbeit für einen Menschen nur eingeschränkt spielbar. Der Algorithmus besitzt aktuell noch viele Einschränkungen, er dient lediglich zur Demonstration des Konzeptes.

Unter Verwendung einer komplexeren Kodierung und grösserer Rechenleistung könnten zukünftig umfangreichere Spiele erzeugt werden.

Abkürzungsverzeichnis

ASP Answer set programming. 4, 6

BSP Binary Space Partitioning. 4

GAN generative adversarial networks. 7

GDL Game Description Language. 8, 9

GGP General Game Playing. 8

GVGAI General Video Game AI. III, 18, 19, 27

KI Künstliche Intelligenz. 7, 13, 18, 19

MCTS Monte Carlo tree search. 18, 19

OLETS Open Loop Expectimax Tree Search. 18–20

PCG Procedural Content Generation. 2, 3, 7

RHEA Rolling Horizon Evolutionary Algorithms. 18, 19

VGDL Video Game Description Language. III, 9, 10, 12, 14, 15, 18, 27, 30, 35

Inhaltsverzeichnis

Teil I.

Hauptteil

1. Einleitung

1.1. Videospiel

Als ein Videospiel, Computerspiel, E-Game oder umgangssprachlich auch nur Game, bezeichnet man ein interaktives Medium, bei welchem der Benutzer, durch Regeln eingeschränkt, versucht einen künstlich erzeugten Konflikt zu lösen. Durch überlegtes Eingreifen in das Spielgeschehen versucht er einen gewünschten Zustand zu erreichen (z.B. Sieg).[5]

Ein Videopiel besitzt folgich folgende Elemente:

- Spielmechanik, Regeln
- Geschichte, Handlung (nicht immer)
- Benutzerschnittstelle
  - Grafik
  - Sound
  - Steuerung

Videospiele können in veschiedene Genres unterteilt werden. Die gebräuchlichsten sind:[26][3]

- Abenteuer
- Action / Arcade
- Rennspiele
- RPG
- Jump 'n' Run
- Shooter
- Simulation
- Sport
- Strategie

Die Übergänge zwischen den Genres sind zum Teil fliessend, des weiteren können einige Spiele auch mehreren Genres zugeordnet werden. Normalerweise werden auch interaktive Filme zu den Spielen gezählt, da sie grundsätzlich alle Elemente eines Videospiels enthalten, wenn auch teils in sehr einfacher Form.

1.2. Ausgangslage

Seit knapp 40 Jahren wird Procedural Content Generation (PCG) für Videospiele eingesetzt. Als solche wird die algorithmische Erzeugung von Spielinhalten mit indirektem oder eingeschränktem Benutzereingriff bezeichnet. Ursprünglich wurde PCG vor allem wegen beschränkten Speicherkapazitäten eingesetzt. Heute wird sie auch für die Erzeugung riesiger Spielwelten ohne Einsatz menschlicher Designer verwendet. [21]

Nachfolgend sind einige bekannte Spiele genannt, welche PCG einsetzen:

Rogue Eines der ersten Spiele mit prozedural generierten Inhalten

Minecraft Wohl eines der bekanntesten Spiele mit PCG. Es können theoretisch bis zu 2^{64} verschiedene Welten erzeugt werden, welche in die reale Welt umgerechnet je rund die 8-fache Erdoberfläche einnehmen[13]

No Man's Sky Ein aktuelles Spiel mit massivem Einsatz von PCG. Es wird eine Galaxie mit 2^{64} verschiedene Planeten samt Flora und Fauna prozedural generiert.

Abbildung 1.1.: Prozedural generiertes Dorf in Minecraft

1.3. Ziel

Das Ziel dieser Arbeit ist, ein mögliches Konzept eines Spielegenerators zu definieren, mit dem bisher unbekannte / ungewohnte Spielmechaniken erzeugt werden können. Für die Evaluation des Konzeptes soll eine Tech Demo entwickelt werden, mit der die grundsätzliche Funktionsfähigkeit des Generators nachgewiesen werden kann.

1.4. Anforderungen

Die vom Generator erzeugten Spiele sollen spielbar (gewinnbar) und nicht-trivial sein. Ein menschlicher Spieler soll nach Vertraut machen mit dem Spiel in der Lage sein, das Spiel zu gewinnen.

Es werden vom Generator keine hochwertigen Spiele erwartet, er soll aber zeigen, dass solche Konzepte grundsätzlich möglich sind.

2. Stand der Technik

2.1. Methoden

Methoden für die prozedurale Generierung von Spielinhalten lassen sich grundsätzlich in vier Gruppen aufteilen [16][27, p. 157ff]:

Konstruktive Methoden Die Eigenschaften der erzeugten Inhalte sind im Vorfeld bestimmt worden. Der Generator kombiniert verschiedene Eigenschaften ohne Anwendung eines Suchverfahrens. Es findet im Normalfall keine Überprüfung der Erzeugten Inhalte statt, es wird vorausgesetzt, dass der Generator nur gültige Ergebnisse produziert.

Löserbasierte Methoden Inhalte werden unter Verwendung eines Solvers oder anderen logischen Methoden erzeugt z.B. mit Answer set programming (ASP).

Suchbasierte Methoden Es werden evolutionäre Algorithmen oder andere stochastische Optimisierungsalgorithmen verwendet. Wichtiger Bestandteil ist die Evaluierungsfunktion, welche bestimmt, wie „gut“ ein bestimmtes Erzeugnis ist. Die meisten Evaluierungsfunktionen für komplette Spiele sind simulationsbasiert, d.h. die Spiele müssen gespielt werden, um sie zu evaluieren.

Machine Learning Der Generator wird mit bestehendem Inhalt trainiert, um anschliessend weitere Inhalte mit ähnlichen Eigenschaften zu erzeugen.

2.1.1. Konstruktive Methoden

2.1.1.1. Raumpartitionierung

Bei der Raumpartitionierung wird das Spielfeld rekursiv in mehrere Teile unterteilt. Die gebräuchlichste Variante ist die Binary Space Partitioning (BSP), hierbei wird jeder Bereich in jeweils zwei Teilbereiche unterteilt. Der resultierende BSP-Baum kann unter anderem für die Erzeugung eines Dungeons verwendet werden (siehe Abbildung 2.1). [18] [12]

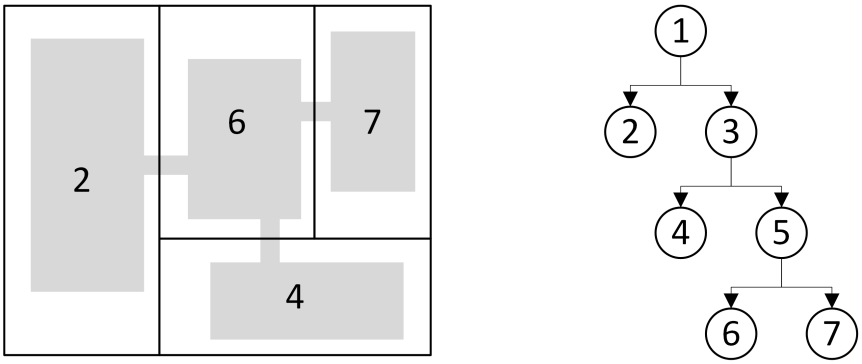


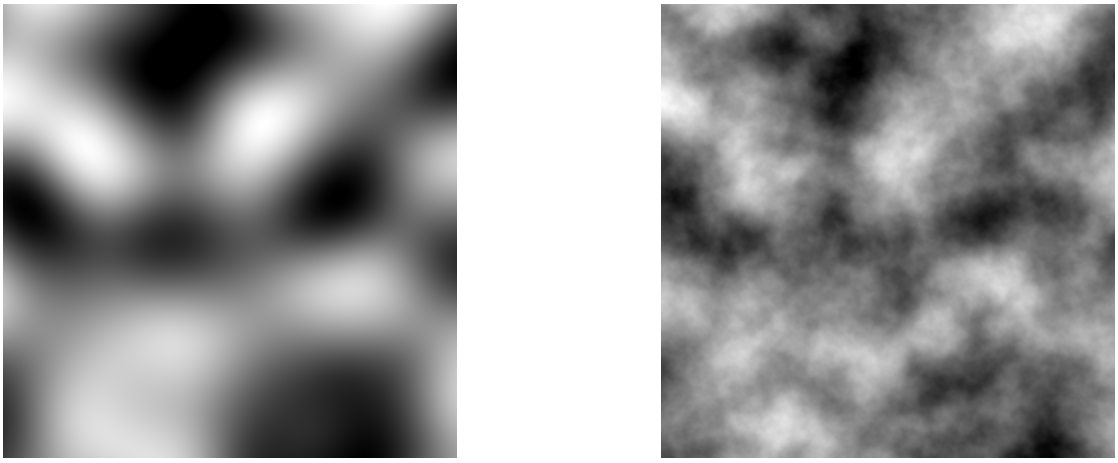
Abbildung 2.1.: Mit BSP erzeugter Dungeon

2.1.1.2. Fraktale

Fraktale Strukturen lassen sich unter anderem durch skalierte Überlagerung einer Grundfunktion erzeugen. Für Landschaften oder auch Texturen kann z.B. eine gradientenbasierte Rauschfunktion verwendet werden. Bekannte Beispiele hierfür sind die Perlin-Noise und die Simplex-Noise Funktionen. Eine populäre Variante ist das $\frac{1}{f}$-Rauschen:[19] [27, p. 169ff]

$$P_{xy} = \sum_{i=0}^{n} \frac{Noise(x * 2^i, y * 2^i)}{2^i} \tag{2.1}$$

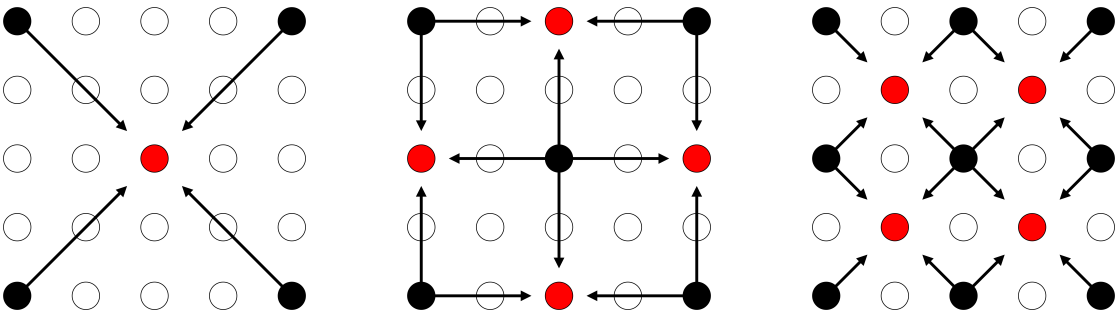
Abbildung 2.2 zeigt die Anwendung auf den Simplex-Noise. Im Anhang unter Kapitel A.1 finden sich weitere Beispiele.



(a) Simplex-Noise (b) $\frac{1}{f}$ Simplex-Noise

Abbildung 2.2.: Umwandlung des Simplex-Noise in ein $\frac{1}{f}$-Rauschen

Eine einfache alternative Möglichkeit, zweidimensionale fraktale Strukturen zu erzeugen, ist der Diamond-Square-Algorithmus [27, p. 169ff]. Bei diesem werden, ausgehend von vier Eckpunkten, die Mitte des äussersten Quadrates bestimmt. Dieser entspricht dem Mittelwert der Eckpunkte plus ein Zufallswert. Dieser Schritt wird abwechslungsweise diagonal und orthogonal für die neu entstandenen Quadrate durchgeführt (siehe Abbildung 2.3).



(a) Diamond-Schritt (b) Square-Schritt (c) Diamond-Schritt

Abbildung 2.3.: Ablauf des Diamond-Square-Algorithmus

2.1.1.3. Generative Grammatik

Eine formale Grammatik bestehend aus Startsymbol, Terminalsymbolen, Nichtterminalsymbolen und Produktionsregeln kann ebenfalls zum Generieren von Spielinhalten verwendet werden. Die wohl bekannteste Klasse von Grammatiken sind L-Systeme, welche das Wachstum von organischen System modellieren. [20]

Weitere Varianten sind Shape- und Graph-Grammatiken, bei welchen nicht mit Zeichenketten, sondern mit geometrischen Formen respektive Graphen gearbeitet wird. Mit Graph-Grammatiken lassen sich unter anderem Missionsketten und Handlungsstränge erzeugen, Shape-Grammatiken können für Dungeons verwendet werden. [12]

2.1.1.4. Zellulärer Automat

Ein zellulärer Automat ist ein diskretes Berechnungsmodell bestehend aus einem N-dimensionalen Gitter, einer Menge möglicher Zustände und einer Menge Überführungsregeln. Jede Zelle befindet sich in einem der möglichen Zustände, die Überführungsregeln bestimmen den Zustand der Zelle in Abhängigkeit ihrer direkten Nachbarn. Der Automat bestimmt in diskreten Schritten jeweils gleichzeitig der neue Zustand jeder einzelnen Zelle.[18]

Mit zellulären Automaten lassen sich unter anderem organische Strukturen nachbilden. Abbildung 2.4 zeigt einen zweidimensionalen Automaten mit nur zwei Zellenzuständen (schwarz / weiss). Die Zellen werden mit einer zufälligen Farbe initialisiert. Die Überführungsregeln färbt jede Zelle mit 6-8 schwarzen Nachbarn schwarz ein, jene mit 1-2 werden weiss, die anderen behalten ihre Farbe bei.

Abbildung 2.4.: Erzeugung eines Höhlensystems mit einem zellulären Automaten, Quelle: [8]

2.1.2. Löserbasierte Methoden

Bei löserbasierten Methoden werden verschiedene Bedingungen in einer logischen Programmiersprache kodiert. Häufig wird das entstandene Problem auf ein SAT-Problem reduziert, welches mit einem SAT-Solver gelöst wird.[27, p.161]

Ein Ansatz für die prozedurale Generierung ist Answer set programming (ASP), bei welchem umfangreiche Bedingungen in einer Prolog-ähnlichen Sprache spezifiziert werden. Ein ASP-Solver kann nun eine Konfiguration der Spielwelt (z.B. Positionierung der Wände für ein Labyrinth) suchen. [17][14]

2.1.3. Suchbasierte Methoden

Suchbasierte Methoden durchsuchen den Raum aller möglichen Spiele mit evolutionären Algorithmen oder anderen stochastischen Optimierungsalgorithmen. Diese Methoden lassen sich auf

praktisch alle Elemente eines Spiels anwenden, sind jedoch relativ langsam. Generell lassen sich auch Laufzeiten nur schwer abschätzen. [27, p. 161] Neben dem Suchalgorithmus und der Kodierung ist die Evaluations- oder Fitnessfunktion ein wichtiger Bestandteil von suchbasierten Methoden. Im Bereich von PCG unterscheidet man zwischen drei Arten von Evaluationsfunktionen: [23] [27, p. 159f]

direkt: Die Fitnessfunktion kann die erzeugten Inhalte direkt auf eine Fitness oder einen Qualitätswert abbilden.

simulationsbasiert: Eine KI versucht, durch den erzeugten Inhalt zu spielen. Aus dem Verhalten der KI kann auf die Qualität und Spielbarkeit der generierten Inhalte geschlossen werden.

interaktiv: Die Qualität der Erzeugnisse wird durch Interaktion mit Menschen bestimmt. Der Mensch liefert durch Spielen der Spiele implizit oder explizit ein Feedback.

2.1.4. Machine Learning

Ein in der PCG-Forschung noch junges Feld sind Methoden basierend auf Machine Learning. Hierbei werden Generatoren mit bestehenden Inhalten trainiert, um neue Inhalte mit ähnlichen Eigenschaften zu erzeugen. Vor allem generative neuronale Netze wie generative adversarial networks (GAN)s und Autoencoder haben hier Aufmerksamkeit erlangt. Diese Algorithmen eignen sich gut zur Erzeugung von Bildern und Musik, stossen bei Spielen jedoch auf Probleme. Viele Spielinhalte besitzen starke Einschränkungen, so kann z.B. ein generiertes Level optisch einen guten Eindruck machen, verunmöglicht aber durch zu grosse Hindernisse einen Sieg.[27, p. 171f]

2.2. Erzeugen der Spielinhalte

2.2.1. Überblick

Grundsätzlich lassen sich alle Elemente eines Spiels generieren. Eine (nicht abschliessende) Übersicht liefert die Tabelle 2.1, welche aufzeigt, mit welchen vorgestellten Methoden die verschiedenen Inhalte generiert werden können.

	Terrain	Dungeon / Level	Quest / Story	Spielmechanik	Texturen	Pflanzen
Evolutionäre Algorithmen	X	X	X	X	X	X
Generative Grammatik		X	X	?		X
Generative Neuronale Netze	X	X	?	?	X	?
Fraktale	X				X	X
Zellulärer Automat	X	X			X	
Raumpartitionierung		X			?	
Answer Set Programming		X	?	X		

Tabelle 2.1.: Übersicht der erzeugbaren Inhalte

In vielen Spielen werden jedoch meist nur einzelne Elemente erzeugt. Ebenso lassen sich nur die wenigsten Methoden für komplette Spiele einsetzten.

2.2.2. Ansätze für komplette Spiele

Nachfolgen werden zwei existierende Systeme vorgestellt, welche bereits erfolgreich Spiele komplett prozedural generieren.

2.2.2.1. Ludi

Ludi ist ein System zur automatischen Generierung von Brettspielen. Diese werden mit der Ludi GDL (siehe Kapitel 2.3.1) kodiert. Ludi verwendet für die Erzeugung einen genetischen Algorithmus. Eine Besonderheit ist, dass vor der eigentlichen Evaluation der einzelnen Spiele eine Überprüfung der Spielregeln durchgeführt wird. Spiele mit widersprüchlichen Regeln, wie z.B. Unterteilung eines dreieckigen Spielfeldes in Quadrate, werden aussortiert.

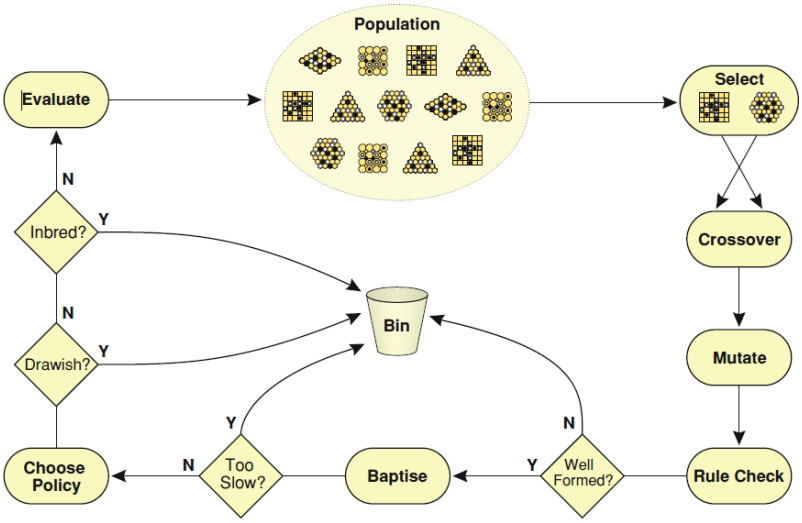


Abbildung 2.5.: Ablauf des genetischen Algorithmus von Ludi, Quelle: [2, p. 39]

2.2.2.2. Angelina

Angelina[1] ist ein laufendes Projekt und eines der fortgeschrittensten Generatoren für Videospiele. Auch dieses System basiert auf evolutionären Algorithmen. Angelina entwickelt das Level, die Regeln und die Platzierung der Entitäten getrennt von einander in einer kooperativen Co-Evolution. Die Entwicklung eines Spiels dauert meist mehrere Tage oder gar Wochen, denn Angelina erstellt z.B. auch über Twitter Umfragen und lässt die Ergebnisse in die Entwicklung einfliessen. Angelina versucht auch selbständig die Spiele zu benennen und passende Musik und Grafiken auszuwählen. [27, p. 195] [22]

2.3. Game Description Language

Als Game Description Language (GDL) wird eine meist höhere Sprache zur Beschreibung von Spielen bezeichnet. Die Notwendigkeit einer solchen Sprache entstand durch das General Game Playings (GGPs). In diesem Feld der künstlichen Intelligenz werden Programme entwickelt, welche sich nicht nur auf ein Spiel fokussieren, sondern mehrere Spiele erfolgreich meistern können. Diese Spiele werden mit einer GDL repräsentiert.

[1] http://www.gamesbyangelina.org/

2.3.1. GDL für Brettspiele

Für Brettspiele und kombinatorische Spiele werden aktuell vor allem drei Sprachen verwendet [2, p. 11ff]:

Stanford GDL ist eine logische Programmiersprache, die rein deklarativ ist.[25]

Zillions ZRF ist ein Format mit Lisp-ähnlicher Syntax und wird für die kommerzielle Spielesammlung „Zillions of Games“ verwendet. [9]

Ludi GDL besitzt ein höheres Level als die anderen GDLs (siehe Abbildung 2.2), jedoch ist auch der Umfang aller möglichen beschreibbaren Spiele kleiner.

	Ludi GDL	ZRF	Stanford GDL
Tic Tac Toe	19	88	384
Schach	325	528	4392

Tabelle 2.2.: Anzahl benötigter Token im Vergleich, Quelle: [2, p. 14]

```
(game Tic-Tac-Toe
   (players White Black)
   (board
      (tiling square i-nbors)
      (size 3 3)
   )
   (end (All win (in-a-row 3)))
)
```

Abbildung 2.6.: Tic Tac Toe beschrieben mit der Ludi GDL

2.3.2. VGDL

Die Video Game Description Language (VGDL) ist eine in [4] definierte Sprache zur Beschreibung von 2D-Videospielen. Bei der Definition der Sprache wurden sechs notwendige Kriterien identifiziert:

Versändlich: Der menschliche Betrachter soll bestehende Beschreibungen einfach verstehen können und neue schnell selber definieren können.

Eindeutig: Die Spielbeschreibungen sollen schnell und unkompliziert in funktionierende Spiele übersetzt werden.

Durchsuchbar: Das Spiel wird in einer Baumstruktur repräsentiert, um generative Algorithmen wie genetisch Programmieren anwenden zu können.

Ausdrucksstark: Die Sprache muss eine angemessene Expressivität besitzen, um Inhalte und Mechaniken klassischer 2D-Videospiele repräsentieren zu können.

Erweiterbar: Es können jederzeit neue Typen und Effekte hinzugefügt werden.

Brauchbar für zufällig erzeugte Spiele: Alle Spielelemente besitzen gut abgestimmte Standartwerte, um möglichst gut mit anderen, zufällig erzeugten Elementen zusammenarbeiten zu können.

Eine Repräsentation eines Spiels mit der VGDL besteht aus zwei Teilen: die Spielmechanik und das Level. Die Spielmechanik wird in vier Gruppen aufgeteilt:

SpriteSet: Listet alle im Spiel vorkommenden Sprites auf. Jedem Sprite-Typ können verschiedene Eigenschaften zugewiesen werden.

LevelMapping: Jedem Zeichen der Levelbeschreibung wird ein Sprite zugewiesen.

InteractionSet: Auflistung aller möglichen Events mit ihren Auswirkungen bei aufeinandertreffen von jeweils zwei Sprites

TerminationSet: Auflistung aller Bedingungen für Sieg und Niederlage

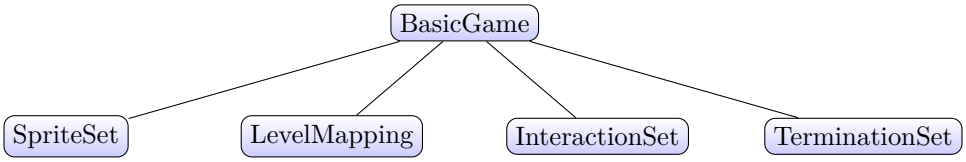


Abbildung 2.7.: Struktur der Spielbeschreibung

Die VGDL besitzt ein Python-ähnliche Syntax und unterstützt fakultative Klammern und Kommata für die bessere Lesbarkeit. Zentrales Element ist der Zuweisungsoperator „>“. Dieser stellt eine Verbindung her zwischen:

- Zeichen und Objektnamen
- Objektnamen und Objekteigenschaften
- Kollisionspaaren und ihren Effekten

```
1 1.........................
2 000.......................
3 000.......................
4 .........................
5 .........................
6 .........................
7 .........................
8 ....000......000000.....000...
9 ...00000....00000000...00000..
10 ...0...0....00....00...00000..
11 ..............A............
```

Abbildung 2.8.: Beschreibung eines Level für „Aliens“

```
BasicGame square_size=32
    SpriteSet
        background > Immovable img=oryx/space1 hidden=True
        base   > Immovable  color=WHITE img=oryx/planet
        avatar > FlakAvatar  stype=sam img=oryx/spaceship1
        missile > Missile
            sam  > orientation=UP  color=BLUE singleton=True img=oryx/bullet1
            bomb > orientation=DOWN color=RED speed=0.5 img=oryx/bullet2
        alien  > Bomber      stype=bomb  prob=0.01 cooldown=3 speed=0.8
            alienGreen > img=oryx/alien3
            alienBlue > img=oryx/alien1
        portal > invisible=True hidden=True
            portalSlow > SpawnPoint  stype=alienBlue cooldown=16 total=20 img=portal
            portalFast > SpawnPoint  stype=alienGreen cooldown=12 total=20 img=portal

    LevelMapping
        . > background
        0 > background base
        1 > background portalSlow
        2 > background portalFast
        A > background avatar

    TerminationSet
        SpriteCounter    stype=avatar            limit=0 win=False
        MultiSpriteCounter stype1=portal stype2=alien limit=0 win=True

    InteractionSet
        avatar EOS  > stepBack
        alien  EOS  > turnAround
        missile EOS > killSprite

        base bomb > killBoth
        base sam > killBoth scoreChange=1

        base  alien > killSprite
        avatar alien > killSprite scoreChange=-1
        avatar bomb > killSprite scoreChange=-1
        alien sam  > killSprite scoreChange=2
```

Abbildung 2.9.: Beschreibung des Spiels „Aliens“

3. Konzept

3.1. Grundidee

Um Spielmechaniken zu generieren, müssen diese codiert werden [22]. Für eine breite Palette von Videospielen bietet sich die VGDL an, welche bereits für die Erzeugung von Spielen benutzt wurde [10]. Die VGDL ist aktuell auch die einzige Sprache, welche das generelle Erzeugen von Spielmechaniken für klassische 2D-Spiele ermöglicht [4].

Bei der Erzeugung einer Spielmechanik gibt es im Zusammenhang mit dem Level verschiedene Möglichkeiten:

1. Fixe Regeln, Level generieren
2. Fixes Level, Regeln generieren
3. Zuerst Level generieren, dann Regeln generieren
4. Zuerst Regeln generieren, dann Level generieren
5. Regeln und Level gleichzeitig generieren

Die erste Möglichkeit fällt für diese Arbeit weg, dieser Ansatz resultiert in einem reinen Levelgenerator. Wenn hingegen das Level fixiert wird, schränkt man den Suchraum der erzeugbaren Spiele massiv ein. Will man Regeln und Level nacheinander generieren, stösst man vor allem bei evolutionären Algorithmen auf das Problem, dass die Fitnessfunktion nicht klar definiert werden kann. Es ist nicht klar, auf was z.B. die Regeln optimiert werden sollen, sie können aufgrund des fehlenden Levels nicht getestet werden. Die einzig sinnvolle Variante ist das gleichzeitige Generieren von Spielmechanik und Spielwelt, so wie es auch im Projekt Angelina eingesetzt wird (siehe Kapitel 2.2.2.2).

Diese Idee kann noch weitergeführt werden, indem man anstatt eines einzelnen Levels einen kompletten Levelgenerator erzeugt. Es muss aber sichergestellt werden, dass für die Regeln eine Menge von Levels existiert, mit denen das Spiel überhaupt spielbar ist. Ein möglicher Ansatz ist:

1. Regeln und Level gleichzeitig generieren
2. Regeln fixieren und Levelgenerator erzeugen

Für den ersten Schritt kommen vor allem evolutionäre Algorithmen in Frage, diese lassen sich am flexibelsten einsetzen. Methoden wie Machine Learning lassen sich kaum implementieren, da hierfür viel zu wenig Daten vorhanden wären. Somit wäre ein Overfitting sehr wahrscheinlich und die Chance, dass völlig neue Spiele generiert werden, ist eher gering. Der zweite Schritt könnte unter anderem mit einer Neuroevovlution eines Neuronalen Netzes durchgeführt werden. Als mögliche Fitness kann z.B. der Anteil der spielbaren Levels im Verhältnis zur Gesamtheit aller erzeugten Levels verwendet werden.

Diese Arbeit behandelt vor allem den ersten Schritt, die Entwicklung eines Levelgenerator-Generators würde über den Rahmen dieser Arbeit hinausgehen.

3.2. Bewertung der Spielbarkeit

Das Level und die Spielregeln sind isoliert betrachtet nicht vollständig bewertbar. Es können nur offensichtliche Widersprüche und Syntaxfehler erkannt werden. Beide Elemente können nur in Kombination getestet werden, es muss folglich eine simulationsbasierte Evaluation, wie in Kapitel 2.1.3 beschrieben, verwendet werden.

3.2.1. Kriterien

Aus Sicht eines menschlichen Spielers lassen sich verschiedene Kriterien für ein gutes Spiel nennen:

1. Spielspass
2. Herausforderung
3. Abwechslung
4. ...

Diese sind jedoch teils sehr subjektiv und kaum messbar. Der Spielspass könnte sich unter anderem an der Schwierigkeit eines Spiels messen lassen. Man kann davon ausgehen, dass zu einfache und zu schwierige Spiele weniger Spass machen. [15]
Für den Generator lassen sich in Bezug auf die simulationsbasierte Evaluation also folgende Kriterien ableiten:

1. Mindestens eine Künstliche Intelligenz (KI) muss das Spiel gewinnen, das Spiel soll überhaupt gewinnbar sein.
2. Eine gute KI soll in dem Spiel gut abschneiden.
3. Eine schlechte KI soll in dem Spiel schlecht abschneiden. Das Spiel darf nicht zu einfach / trivial sein

Als Evaluation können n_w verschiedene „Winner“-KIs und n_l „Loser“-KIs gegeneinander antreten. Je höher die Gewinnchance und die Punkte der „Winner“ im Vergleich zu den „Loser“, desto besser wird das Spiel bewertet. Eine einfache Fitnessfunktion könnte so aussehen:

$$Fitness = \frac{1}{n_w}\sum_{i=0}^{n_w} win_{w_i} - \frac{1}{n_l}\sum_{j=0}^{n_l} win_{l_j} \tag{3.1}$$

Um bessere Ergebnisse zu erzielen, kann für die „Winner“ auch nur der beste Controller betrachtet werden. Ein Vorteil ist auch, dass nicht alle n_w Controller das Spiel durchspielen müssen, es kann nach dem ersten Sieg abgebrochen werden. Um die grundsätzliche Spielbarkeit zu beweisen, genügt es, wenn nur ein Controller gewinnt. Für die „Loser“ muss aber zwingend ein Mittelwert berechnet werden, um einzelne Ausreisser nach oben zu ignorieren. Es kann nämlich sein, dass auch eine schlechte KI zufälligerweise ein Spiel gewinnt. Der Grossteil der „Loser“ darf aber nicht gewinnen.

Ein Problem einer solchen Fitnessfunktion ist aber die Tatsache, dass unspielbare Spiele nicht unterschieden werden können, alle erhalten die gleiche Bewertung. Die Fitness bleibt somit lange bei 0, inkrementelle Verbesserungen der Spiele sind nicht messbar. Es ist nicht feststellbar, ob ein Spiel „fast“ spielbar ist.

Als Lösungsansatz könnte die Fitnessfunktion mehrere Teilpunkte vergeben wie z.B. für:

1. kein sofortiges Game Over
2. erreichte Punkte während Spiel (Score), auch wenn das Spiel nicht gewonnen werden kann

3.2.2. Fitnessfunktion

Aus den obigen Überlegungen und unter Berücksichtigung bereits existierenden Evaluationskonzepten [10] [11] wird eine Fitnessfunktion definiert, welche 0-4 Punkte vergeben kann:

1 Punkt kein sofortiges Game Over

1 Punkt Sieg und Niederlage möglich (nicht alle Controller gewinnen / verlieren ausschliesslich)

0-1 Punkt „Winner“ W erreichen mehr Punkte als „Loser“ L

0-1 Punkt Gewinnchancen von L sind kleiner als die von W

Die daraus resultierende Fitnessfunktion lässt sich folgendermassen darstellen:

$$Fitness = NoInstantGameOver + WinAndLose + ScoreDiff + WinDiff \tag{3.2}$$

$$NoInstantGameOver = \begin{cases} 0 & \text{wenn sofortiges Game Over} \\ 1 & \text{sonst} \end{cases} \tag{3.3}$$

$$WinAndLose = \begin{cases} 0 & \text{wenn alle Controller ausschliesslich gewinnen bzw. verlieren} \\ 1 & \text{sonst} \end{cases} \tag{3.4}$$

$$ScoreDiff = \frac{max(Score_{w_i}) - \frac{1}{n_l}\sum_{j=0}^{n_l} Score_{l_j}}{max(Score_{w_i})} \tag{3.5}$$

$$WinDiff = max(win_{w_i}) - \frac{1}{n_l}\sum_{j=0}^{n_l} win_{l_j} \tag{3.6}$$

3.3. Genetischer Algorithmus

Der Ablauf eines genetischen Algorithmus gliedert sich in folgende Schritte:

1. Population aus N Individuen zufällig erstellen
2. Fitness aller Individuen berechnen
3. jeweils zwei Individuen aus der Population für die Fortpflanzung auswählen
4. Nachkommen erzeugen mit Genen der Eltern
5. zufällige Mutation der Nachkommen
6. Nachkommen ersetzen die alte Population
7. ab Schritt 2 wiederholen

3.3.1. Codierung

Für die Verwendung eines genetischen Algorithmus muss die Spielbeschreibung kodiert werden. Es bietet sich folgende Unterteilung an:

Phänotyp Das übersetzte / geparste Spiel, ausführbar

Genotyp die Beschreibung des Spiels mit der VGDL

Chromosom einzelnes Set der VGDL (z.B SpriteSet)

Gen einzelner Eintrag in einem Set (z.B. Definition des Avatar-Sprites)

Ein Gen entspricht somit einem String und ein Chromosom stellt eine Liste von Strings dar. Der Genotyp umfasst insgesamt fünf Chromosome, für jedes der vier Sets eines plus ein weiteres für das Level.

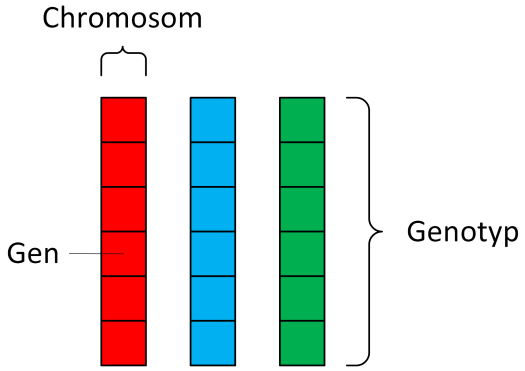


Abbildung 3.1.: Übersicht der Begriffe eines genetischen Algorithmus

Alternativ liesse sich die VGDL auch durch kontextfreie Grammatik definieren (siehe Anhang Kapitel A.2), bei welcher die einzelnen Produktionen die Gene darstellen.

3.3.2. Hyperparameter

Für den gewählten genetischen Algorithmus lassen sich folgende Hyperparameter definieren:

- Populationsgrösse
- Anzahl Generationen
- Auswahlfunktion
- Elitismus (Bestes Individuum wird unverändert übernommen)
- Crossover Operator
- Mutationsrate
- Fitnessfunktion
  - Zykluszeit
  - Anzahl Zyklen
  - Anzahl Controller

Diese werden in den nachfolgenden Kapiteln näher betrachtet.

3.3.3. Auswahlfunktion

Für die Auswahl von Individuen für die Fortpflanzung gibt es Zahlreiche Funktionen. Einige der bekanntesten sind: [1]

Tournament Selection: Aus der Population werden t Individuen zufällig ausgewählt. Aus diesen t Individuen wird das beste ausgewählt. Hierbei entspricht t der Tournamentgrösse.

Proportional Selection: Die Auswahlwahrscheinlichkeit eines Individuums ist proportional zu seiner Fitness.

Linear Ranking Selection: Die Auswahlwahrscheinlichkeit eines Individuums ist eine lineare Funktion seines Rangs in der Population.

Aufgrund seiner Einfachheit und Flexibilität wird die Tournament Selection verwendet.

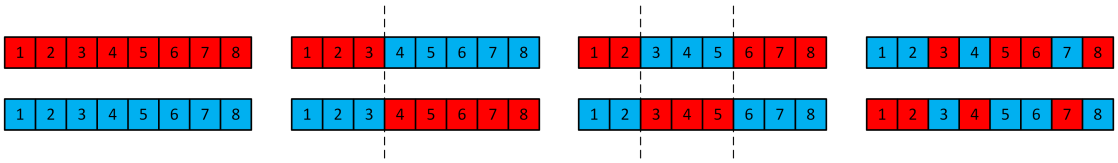
3.3.4. Crossover

Bei der Fortpflanzung werden den Nachkommen einen Mix aus den Genen der Eltern zugewiesen. Die populärsten Crossover Operatoren sind: [24]

Single Point Crossover: Es wird ein zufälliger Index ausgewählt. Die Chromosome werden an dieser Stelle aufgeteilt und der Nachkomme erhält von seinen Eltern je einen Teil.

Multi Point Crossover: Die Verallgemeinerung des Single Point Crossover teilt die Chromosome an mehreren Stellen auf. Der Nachkomme erhält von seinen Eltern abwechslungsweise je einen Teil.

Uniform Crossover: Für jedes Gen wird zufällig Entschieden, von welchem Elternteil es übernommen wird.



(a) Elternchromosome (b) Single Point Crossover (c) Multi Point Crossover (d) Uniform Crossover

Abbildung 3.2.: Vergleich der verschiedenen Crossover-Operationen

3.3.5. Mutation

Für die Mutation von Individuen lassen sich folgende Varianten mit aufsteigendem Ausmass definieren:

- Veränderung einzelner Parameter in einem Gen
- Komplett neues Gen
- Komplett neues Chromosom
- Komplett neuer Genotyp

Je grösser das Ausmass einer Mutation, desto seltener tritt sie auf. Mutationraten liegen üblicherweise im einstelligen Prozentbereich.

Eine Mutation soll auch die Anzahl der Gene verändern können (z.B. die Anzahl Interaktionsregeln). Für jedes Chromosom ist eine Normalverteilung vorgesehen, anhand derer sich mutierte Anzahl orientiert. Die einzelnen Chromosome werden bei der erstmaligen Erstellung mit einer Genzahl aus der Normalverteilung initialisiert. Eine Veränderung der Anzahl erfolgt während Mutationen schrittweise mit folgender Formel:

$$n_{Gene_{t+1}} = n_{Gene_t} + signum(Zufallszahl(MEAN, STDDEV) - n_{Gene_t}) \tag{3.7}$$

Je weiter eine Genzahl vom Mittelwert entfernt ist, desto grösser ist die Chance, dass sie in die entsprechend andere Richtung korrigiert wird.

3.3.6. Fitnessfunktion

Die Evaluation einzelner Individuen erfolgt simulationsbasiert gemäss den Überlegungen aus Kapitel 3.2.2.

3.4. Einschränkungen

Grundsätzlich soll der Generator so wenig wie möglich eingeschränkt werden, um möglichst vielfältige Ergebnisse erzielen zu können. Nichts desto trotz bedarf es einiger Beschränkungen, welche die Geschwindigkeit des Algorithmus massiv verbessern:

- Es sollte immer mindestens ein Spieler-Avatar existieren. Ohne Avatar ist das Spiel nicht spielbar. Wird die Definition eines Avatar erzwungen, können zu Beginn der Berechnungen des genetischen Algorithmus mehrere Generationen eingespart werden.
- Für jedes Zeichen im Level sollte auch ein entsprechendes Level Mapping und folglich auch eine Spritebeschreibung vorhanden sein. Doppelte Einträge können entweder direkt unterbunden werden und bei der Bereinigung des erzeugten Spiels entfernt werden.

Um die zweite Bedingung sicher zu gewährleisten, kann folgende jede Möglichkeit in Betracht gezogen werden: Alle Sets teilen sich eine fixe Anzahl von Spritenamen. Das SpriteSet muss zwingend für jeden Namen eine Spritedefinition aufführen. Die anderen Sets können nun bei der Erstellung / Veränderung von Genen einen Spritenamen aus dieser Liste auswählen, dieser ist in jedem Fall definert. Dasselbe Vorgehen kann auch für die Zeichen eines Levels verwendet werden. Das Level stellt für einen fixen Zeichensatz ein Mapping zur Verfügung, die Levelbeschreibung kann aus diesen Zeichen ein Level zusammenstellen.

Ein Nachteil dieser Vereinfachung ist, dass die Anzahl verschiedener Sprites während des gesamten Evolutionsprozesses fix bleiben und somit die Anzahl möglicher Spiele einschränkt.

4. Implementierung

4.1. Eingesetzte Mittel

Der in Kapitel 3 vorgestellte Algorithmus wurde in Java basierend auf einem einfachen genetischen Algorithmus [7] und dem Framework General Video Game AI (GVGAI)-Competition [1]. Zur Visualisierung der Ergebnisse wird JFreeChart [2] verwendet.

4.2. GVGAI Framework

Das GVGAI Framework ist eine Game Engine, welche für in der VGDL definierten Spiele ausgelegt ist. Das Framework bietet eine Vielzahl bereits definierter Spiele und stellt auch mehrere fortgeschrittene Algorithmen bereit, mit der eine KI die Spiele durchspielen kann. Die vielversprechendsten sind: [6]

Rolling Horizon Evolutionary Algorithms (RHEA) erzeugen eine Sequenz von Aktionen, von welchen aber nur die ersten paar im eigentlichen Spiel ausgeführt werden.

Monte Carlo tree search (MCTS) ist ein sehr populärer Suchalgorithmus, welcher bei mehrfachem Durchspielen des Spiels jeweils die vielversprechendsten Aktionen weiterverfolgt. [27, p. 45f]

Open Loop Expectimax Tree Search (OLETS) ist eine Variante von MCTS ohne Monte Carlo Simulationen.

4.3. Sprites

Da die Bilder der einzelnen Sprites beim Generieren nicht sinnvoll zugeordnet werden können, werden spezielle Debug-Texturen verwendet (siehe Abbildung 4.1). Diese sind farblich klar unterscheidbar und besitzen einen farbigen, versetzten Registerpunkt, um auch mehrere übereinanderliegende Sprites unterscheiden zu können.

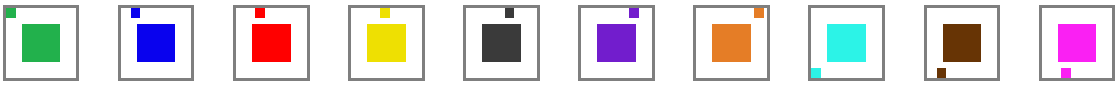


Abbildung 4.1.: Debug-Texturen

4.4. Fitnessfunktion

4.4.1. Performance

Die Berechnung der Fitnessfunktion ist der mit grossem Abstand zeitaufwendigste Schritt des genetischen Algorithmus. Die maximale Berechnungsdauer kann mit folgender Formel abgeschätzt

[1] http://www.gvgai.net
[2] http://www.jfree.org/jfreechart/

werden:

$$T_{max} \approx N_{cyc} * T_{cyc} * N_{gen} * N_{pop} * \left\lceil \frac{N_{contr}}{N_{proc}} \right\rceil \tag{4.1}$$

T_{max} = maximale Berechnungsdauer
N_{cyc} = maximale Anzahl Zyklen pro Spiel
T_{cyc} = Zykluszeit
N_{gen} = Anzahl Generationen
N_{pop} = Populationsgrösse
N_{contr} = Anzahl Controller
N_{proc} = Anzahl Prozessoren

So erwartet man z.B. $N_{cyc} = 1000$, $T_{cyc} = 10ms$, $N_{gen} = 50$, $N_{pop} = 50$ und bei $N_{contr} = N_{proc}$ eine maximale Berechnungszeit von rund 8 Stunden.

Da das GVGAI Framework kein Multithreading für einzelne Controller implementiert hat, werden mehrere Spielsimulationen parallel gestartet. Es ist somit sinnvoll, die Anzahl der Controller N_{contr} auf ein ganzzahliges Vielfaches der Prozessorzahl des ausführenden Computers einzustellen.

Je nach Spiel erreichen die Controller eine Gewinnrate von nur 20% [6]. Bei 8 Controllern erhält man gemäss Gleichung 4.3 zu 83.22% mindestens ein Sieg. Wird das Spiel 16 Mal gespielt, kann die Wahrscheinlichkeit auf 97.19% erhöht werden, jedoch folglich mit einer bis zu zweifachen Berechnungsdauer.

$$P(X = k) = \binom{n}{k} * p^k * (1-p)^{n-k} \tag{4.2}$$

$$P(X > 0) = 1 - P(X = 0) = 1 - (1-p)^n \tag{4.3}$$

4.4.2. Qualität

Um die in Kapitel 3.2.2 definierte Fitnessfunktion zu überprüfen, wird sie an den bestehenden Spielen des GVGAI Frameworks getestet. Insgesamt sind 112 Spiele mit jeweils 5 verschiedenen Leveln definiert.

Als gute KI („Winner“) stehen RHEA, MCTS und OLETS zur Verfügung, als schlechte KI („Loser“) werden ein Random-Controller (wählt ein jedem Schritt eine zufällige Aktion aus) und ein DoNothing-Controller verwendet. All diese Controller sind bereits im GVGAI Framework implementiert.

Jeder Controller spielt jede Spiel-Level-Kombination 8 Mal und erhält pro Zyklus 10ms Zeit für die Berechnung. Die Verteilung der erreichten Fitnesswerte sind in Abbildung 4.2. Als vierte Variante ist eine Kombination von allen guten Controllern gelistet, welche für jedes Spiel jeweils die beste Fitness aller Controller auswählt.

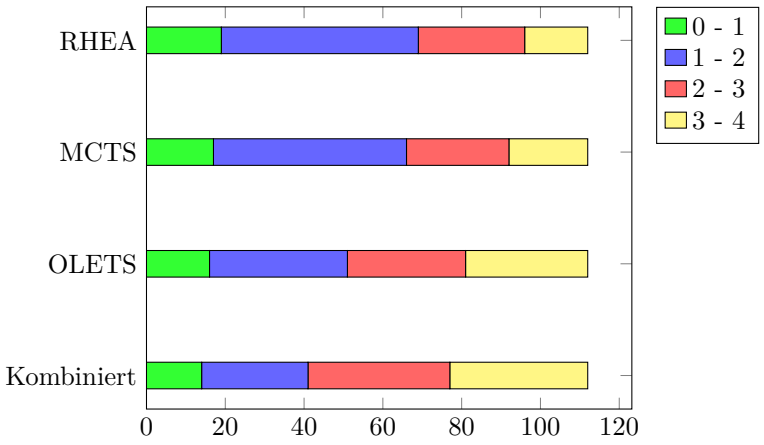


Abbildung 4.2.: Verteilung der erreichten Fitnesswerte mit verschiedenen Controllern

Es zeigt sich, dass der beste Controller, der OLETS, lediglich rund ein Viertel der Spiele mit einer hohen Fitness (gelb) bewertet. Durch die Kombination aller Controller verbessert sich das Ergebnis nur schwach. Daraus lässt sich schliessen, dass gewisse Spiele für alle Controller unspielbar (grün) sind.

In einer weiteren Messreihe wird der Einfluss der zur Verfügung stehenden Zykluszeit der Controller auf deren Performance untersucht. Mit dem OLETS-Controller wird nochmals jede Spiel-Level-Kombination 8 Mal gespielt. Die Berechnungen werden einmal 10ms und einmal mit 50ms Zykluszeit durchgeführt.

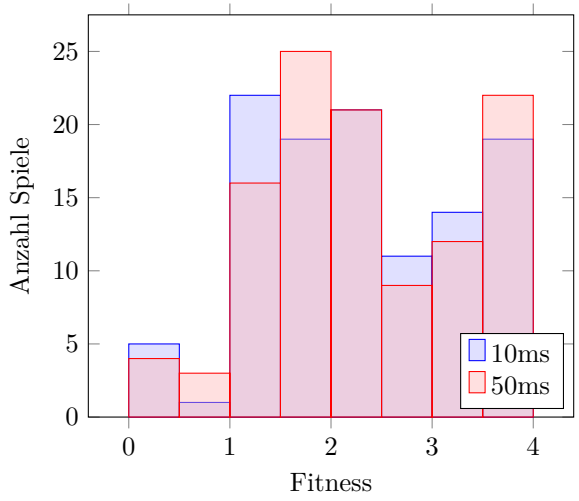


Abbildung 4.3.: Verteilung der erreichten Fitnesswerte mit verschiedenen Zykluszeiten

Durch die Erhöhung kann zwar die durchschnittliche Fitness erhöht werden, diese steht jedoch in keinem Verhältnis zur Verfünfachung der Berechnugsdauer. In Abbildung 4.3 ist ausserdem zu Erkennen, dass in den Kategorien 0-2 Punkte und 2-4 Punkte die Anzahl Spiele nicht verändert hat. Daraus lässt sich schliessen, dass Spiele, welche bei 10ms weniger als 2 Punkte erhalten haben, auch bei höheren Zykluszeiten stets weniger als 2 Punkte erreichen.

Eine letzte Messreihe mit nur einem Spiel, bei welchem aus 100 Simulationen eine durchschnittliche Gewinnrate errechnet wurde, zeigt den begrenzten Einfluss der Zykluszeit. Für jede Zeit wurde fünf Mal eine Gewinnrate berechnet. Die Resultate mit empirischer Standardabweichung sind in Abbildung 4.4 dargestellt. Bis 10ms Zyklus steigert sich die Gewinnrate linear, danach bleibt sie bis 50ms relativ konstant bei rund 90%.

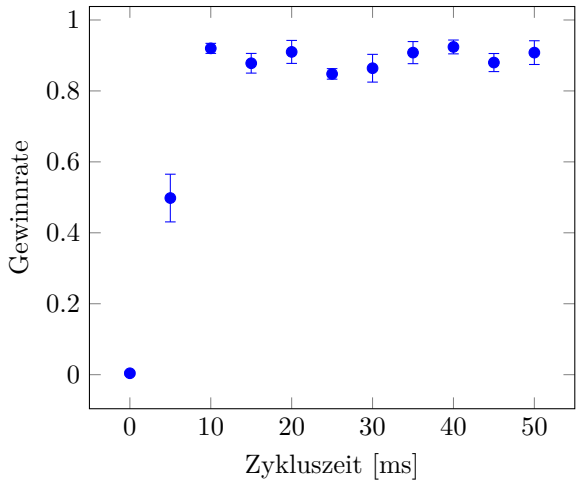


Abbildung 4.4.: Einfluss der Zykluszeit auf die Gewinnchance

5. Resultate

5.1. Parameter Tuning

Um die Performance des genetischen Algorithmus zu testen, müssen die Hyperparameter optimiert werden.

5.1.1. Einfluss der Hyperparameter

Es lässt sich grundsätzlich sagen, dass alle Parameter ein Trade-off zwischen Geschwindigkeit und Qualität des Resultates sind.

Populationsgrösse grösser -> mehr Diversität -> grössere Chance auf globales Optimum, aber langsamer

Auswahlfunktion härter -> kleinere Diversität -> kleinere Chance auf globales Optimum, aber schneller

Mutation grösser -> mehr globale Optima werden „übersprungen“, aber schneller

Zykluszeit länger -> bessere Evaluation -> weniger globale Optima werden „übersehen“, aber langsamer

Anzahl Zyklen mehr -> bessere Evaluation -> weniger globale Optima werden „übersehen“, aber langsamer

Anzahl Controller mehr -> bessere Evaluation -> weniger globale Optima werden „übersehen“, aber langsamer

5.2. Vorgehen

Der Generator wird mit verschiedenen Einstellungen jeweils fünf Mal getestet. Die Beschränkung auf lediglich fünf Testversuche wird mit der sehr langen Berechnungsdauer eines einzelnen Versuchs begründet. In Kapitel 4.4.1 wurde bereits festgestellt, dass ein einziger Durchlauf bereits mehrere Stunden dauern kann.

Als Startkonfiguration des Generators werden folgende Einstellungen genommen:

- Populationsgrösse = 50
- Anzahl Generationen = 50
- Elitismus = 1
- Tournament Selection = 5
- Mutationsrate = 0.3 / 0.1 / 0.01 (normale / starke / sehr starke Mutation)
- Zykluszeit = 10ms
- Anzahl Zyklen pro Spiel = 1000
- Levelgrösse = 6x6

Ein Vergleich verschiedener Einstellungen soll anschliessend Aufschluss über die optimalen Parameterwerte geben. Für diesen Vergleich werden folgende Werte gemessen:

- maximal Erreichte Fitness
- Dauer bis zu einer guten Fitness (> 3.75)
- Monotonie der Fitnesswerte, sind die Werte im grösser als in der vorherigen Generation?
- Geschwindigkeit des Verlusts der Diversität, nach wie vielen Generationen übernimmt eine Spezies die komplette Population?

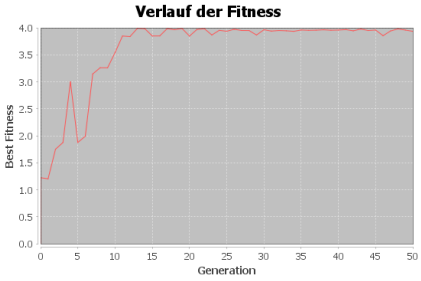
5.3. Ergebnisse

Bereits die erste Konfiguration zeigt fünf sehr unterschiedliche Verhaltensweisen, welche sehr stark von den generierten Spielen in der Population abhängen. In Abbildung 5.3 sind jeweils der Verlauf der Fitness und die Zusammensetzung der Population aufgeführt. Es wird im Schnitt nach rund 20 Generationen eine gute Lösung gefunden, die Streuung ist jedoch sehr gross. Bei einem von fünf Durchgängen wurde selbst nach 50 Generationen keine vernünftige Lösung gefunden. Bei allen Versuchen wurde ausserdem festgestellt, dass die Fitness regelmässig kurz einbricht. Dies lässt sich dadurch erklären, dass die Evaluation der Spiele auch einer Streuung unterliegt und dass somit dasselbe Spiel auch eine schlechtere Fitness erhält als in der vorherigen Generation.

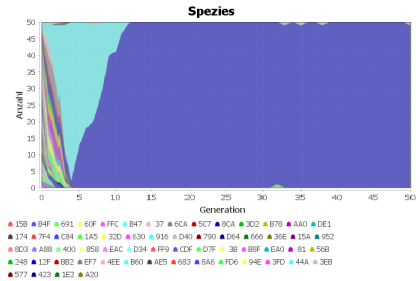
Eine Erhöhung der Populationsgrösse auf 100 verringert Streuung der Konvergenzdauer. Im Schnitt wurde nach 15 Generationen eine gute Lösung gefunden, spätestens aber nach 20. Bei einer Reduktion der Populationsgrösse auf 10 konnte nie eine Fitness grösser als zwei erreicht werden. Folglich sind für die Populations Werte im Bereich von 50 bis 100 zu empfehlen.

Der zweite wichtige Parameter ist die Tournament Grösse. Sie beeinflusst vor allem, wie schnell die Diversifikation abnimmt, also wie schnell eine Spezies die gesamte Population übernimmt. In der ersten Konfiguration mit einer Tournament Grösse von 5 kann eine Spezies innerhalb von weniger als fünf Generationen die Population übernehmen. Man stellt fest, dass solche Populationen sich nur noch schwach entwickeln können. Eine Verbesserung der Fitness wird so über mehrere Generationen hinweg sehr unwahrscheinlich. Wird nun die Tournament Grösse auf 3 reduziert, scheint dieser Übergang etwas langsamer abzulaufen. Eine massgebliche Änderung der Ergebnisse findet allerdings nicht statt.

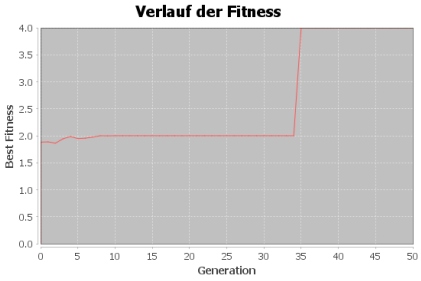
Die Ergebnisse der angepassten Konfigurationen sind im Anhang unter Kapitel A.3 zu finden.



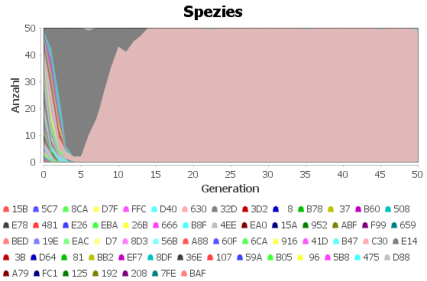
(a) 1. Versuch Fitness



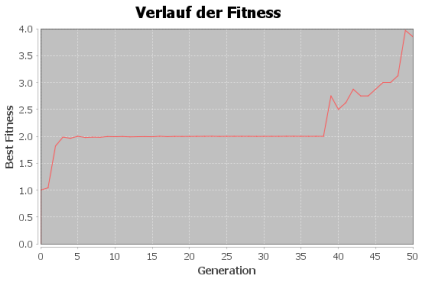
(b) 1. Versuch Spezies



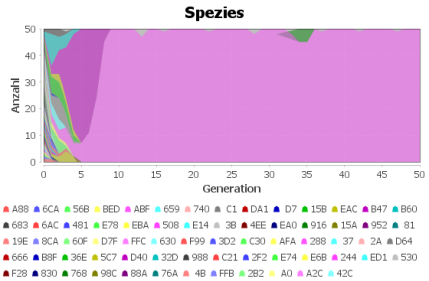
(c) 2. Versuch Fitness



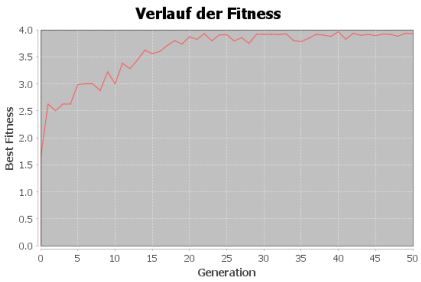
(d) 2. Versuch Spezies



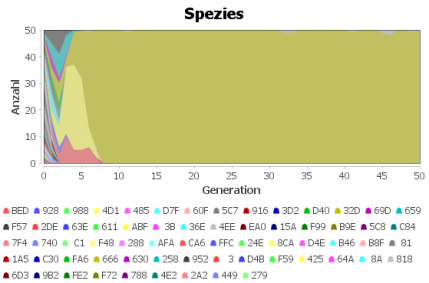
(e) 3. Versuch Fitness



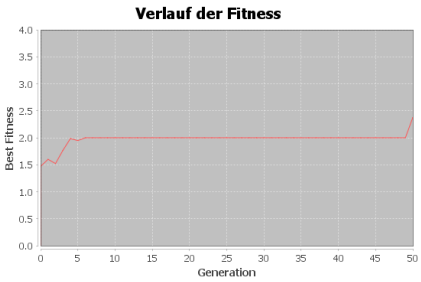
(f) 3. Versuch Spezies



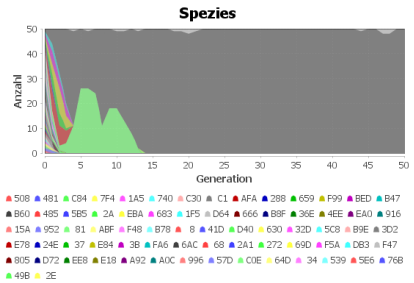
(g) 4. Versuch Fitness



(h) 4. Versuch Spezies



(i) 5. Versuch Fitness



(j) 5. Versuch Spezies

Abbildung 5.1.: Resultate der ersten Konfiguration

5.4. Bewertung

Aufgrund der begrenzten Testläufe ist es schwierig, Aussagen über bestimmte Konfigurationen zu tätigen. Es wurde bei allen Einstellungen gute wie schlechte Ergebnisse festgestellt. Bereits die Startkonfiguration liefert aber meist brauchbare Ergebnisse, die verschiedenen Einstellungen scheinen nur einen Begrenzten Einfluss zu haben.

Die erzeugten Spiele sind sehr abstrakt und für menschliche Spieler ohne weiteres nur sehr eingeschränkt spielbar. Abbildung 5.4 zeigt ein Beispiel eines generierten Spiels, bei welchem nach kurzem Ausprobieren durchaus einfach gewonnen werden. Der grüne Sprite stellt der Spieler-Avatar dar, welcher regelmässig einen Sprung nach vorne macht. Ein Blick in die Spielregeln (Abbildung 5.2) zeigt, das man für einen Sieg in einem Zug durch sechs gelbe Sprites hindurchspringen muss. Hierbei enstehen sechs blaue Sprites, welche die Siegesbedingung erfüllen.

```
1  BasicGame
2    SpriteSet
3      sprite0 > CarAvatar stype=sprite2 color=GREEN cooldown=21 cons=3 prob=0.01
           speed=0.8 frameRate=8 randomtiling=0.9 img=debug/1
4      sprite1 > FlakAvatar stype=sprite3 color=LIGHTGRAY cooldown=14 cons=2 prob=0.01
           speed=0.8 frameRate=8 randomtiling=0.9 img=debug/2
5      sprite2 > FlakAvatar stype=sprite1 color=ORANGE cooldown=26 cons=5 prob=0.01
           speed=0.8 frameRate=8 randomtiling=0.9 img=debug/3
6      sprite3 > Missile stype=sprite3 color=YELLOW cooldown=23 cons=7 prob=0.01 speed=0.8
           frameRate=8 randomtiling=0.9 img=debug/4
7      sprite4 > RandomMissile stype=sprite3 color=GOLD cooldown=4 cons=1 prob=0.01
           speed=0.8 frameRate=8 randomtiling=0.9 img=debug/5
8
9    LevelMapping
10     0 > sprite3
11     1 > sprite3
12     2 > sprite4
13     3 > sprite4
14     4 > sprite0
15
16   InteractionSet
17     sprite1 sprite1 > killIfFrontal
18     sprite1 sprite4 > spawn stype=sprite1
19     sprite4 sprite0 > killIfFromAbove scoreChange=1
20     sprite0 sprite3 > spawnAbove stype=sprite1
21
22   TerminationSet
23     MultiSpriteCounter stype1=sprite1 stype2=sprite1 stype3=sprite2 limit=6 win=true
24     SpriteCounter stype=sprite4 limit=0 win=false
```

Abbildung 5.2.: Beschreibung des erzeugten Spiels

```
204123
213024
204131
234014
024012
410400
440332
```

Abbildung 5.3.: Levelbeschreibung des erzeugten Spiels

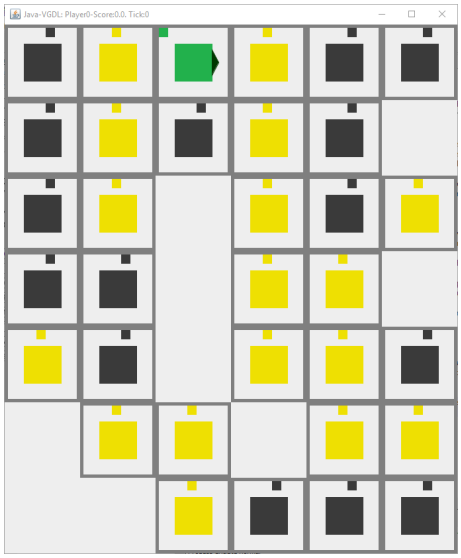


Abbildung 5.4.: Visualisierung des erzeugten Spiels

Es wurde festgestellt, dass manche Spiele bei der Evaluation unterbewertet werden, weil nach dem ersten Sieg eines guten Controllers die Spiele der anderen guten Controller abgebrochen werden, um Zeit zu sparen. Es ist gut möglich, dass der erste Controller weniger Punkte holt als die anderen. Als Lösungsansatz könnte man stets alle Controller das Spiel beenden lassen, was aber die Berechnungszeit wieder vergrössert.

6. Schlussbetrachtung

6.1. Zusammenfassung

Die prozedurale Generierung wird in Videospielen breit eingesetzt. Die Anwendung auf Spielmechaniken und komplette Spiele ist jedoch noch sehr eingeschränkt. Die erfolgreichsten Ansätze basieren auf evolutionären Algorithmen. Die Evaluation von einzelnen Spielen erfolgt meistens mit einer Simulation des Spiel.

In dieser Arbeit wurde ein genetischer Algorithmus entwickelt, um mit dem GVGAI-Framework Spiele zu generieren, welche in der Video Game Description Language kodiert sind. Erste Tests bewiesen grundsätzlich die Funktion des Generators. Durch Optimierung der Hyperparameter konnte die Performance etwas verbessert werden.

Die erzeugten Spiele erfüllen grundsätzlich die gestellten Anforderungen, sind jedoch ohne Nacharbeit für einen Menschen nur eingeschränkt spielbar. Der Algorithmus besitzt aktuell noch viele Einschränkungen, er dient lediglich zur Demonstration des Konzeptes.

6.2. Ausblick

Das vorgestellte Konzept unterliegt aktuell noch vielen Einschränkungen. Am Generator können für zukünftige Anwendungen diverse Verbesserungen vorgenommen werden:

Bessere Kodierung der Spiele. Ermöglichen einer Baum-Struktur für das SpriteSet, beliebige Form und Grösse für das Level

Parallelisierung: Die Evaluation der Spiele kann weiter optimiert werden, indem Berechnung auch auf der Grafikkarte durchgeführt werden. Dies führt zu kürzeren Rechenzeiten und / oder genaueren Evaluationen.

Levelgenerator: Als Ergänzung kann das in Kapitel 3.1 vorgestellte Konzept mit der Erzeugung eines Levelgenerator anstatt ein statischen Level umgesetzt werden. Dadurch werden Spielmechaniken genauer evaluiert, benötigen aber deutlich Berechnungsaufwand.

6.3. Fazit

Die Funktion des Konzepts konnte grundsätzlich bewiesen werden. Die Qualität der erzeugten Spiele hängt stark von der Fitnessfunktion ab. Ein Problem hierbei ist die subjektive Wahrnehmung der Spiele. Für jeden Menschen bereiten verschiedene Spiele unterschiedlichen Spielspass. Des weiteren sind die generierten Spiele noch sehr abstrakt und für den Menschen ohne weiteres nur eingeschränkt spielbar.

Für die Erzeugung etwas komplexerer 2D-Spiele müssen die Probleme der Skalierbarkeit und der Performance gelöst werden.

Literaturverzeichnis

[1] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.

[2] Cameron Browne. *Evolutionary Game Design*. Springer London, London, 2011.

[3] Valve Corporation. Steam shop. `store.steampowered.com`, 2018. [Online; Zugriff am 7. Mai 2018].

[4] Marc Ebner, John Levine, Simon M. Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a Video Game Description Language. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 85–100. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013.

[5] Nicolas Esposito. A short and simple definition of what a videogame is. In *DiGRA Conference*, 2005.

[6] H. Horn, V. Volz, D. Pérez-Liébana, and M. Preuss. Mcts/ea hybrid gvgai players and game difficulty estimation. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, Sept 2016.

[7] Lee Jacobson. Creating a genetic algorithm for beginners. `www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3`, 2012. [Online; Zugriff am 21. Mai 2018].

[8] Jeremy Kun. The cellular automaton method for cave generation. `https://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/`, 2012. [Online; Zugriff am 16. Juli 2018].

[9] Jeff Malletty and Mark Lefler. Zillions of games. `http://www.zillions-of-games.com`, 1998. [Online; Zugriff am 14. Mai 2018].

[10] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson. Towards generating arcade game rules with vgdl. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 185–192, Aug 2015.

[11] Thorbjorn S. Nielsen, Gabriella A. B. Barros, Julian Togelius, and Mark J. Nelson. General video game evaluation using relative algorithm performance profiles. In *EvoApplications*, 2015.

[12] Marco Niemann. Constructive generation methods for dungeons, 2015. `www.wi.uni-muenster.de/sites/wi/files/users/mpreu_02/games-material/ba-vm-ss2015/marco_niemann_-_constructive_generation_methods_for_dungeons_-_thesis.pdf`.

[13] Markus Persson. The world is bigger now. `notch.tumblr.com/post/443693773/the-world-is-bigger-now`, 2010. [Online; Zugriff am 7. Mai 2018].

[14] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, Sept 2011.

[15] Nathan Sorenson and Philippe Pasquier. Towards a generic framework for automated video game level creation. In *EvoApplications*, 2010.

[16] Julian Togelius, Mark Nelson, and Antonios Liapis. Characteristics of generatable games. In *Proceedings of the 5th Workshop on Procedural Content Generation in Games*, United States, 2014. Association for Computing Machinery.

[17] Julian Togelius, Noor Shaker, and Mark J. Nelson. Asp with applications to mazes and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 143–157. Springer, 2016.

[18] Julian Togelius, Noor Shaker, and Mark J. Nelson. Constructive generation methods for dungeons and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 31–55. Springer, 2016.

[19] Julian Togelius, Noor Shaker, and Mark J. Nelson. Fractals, noise and agents with applications to landscapes. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 57–72. Springer, 2016.

[20] Julian Togelius, Noor Shaker, and Mark J. Nelson. Grammars and l-systems with applications to vegetation and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 73–98. Springer, 2016.

[21] Julian Togelius, Noor Shaker, and Mark J. Nelson. Introduction. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 1–15. Springer, 2016.

[22] Julian Togelius, Noor Shaker, and Mark J. Nelson. Rules and mechanics. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 99–121. Springer, 2016.

[23] Julian Togelius, Noor Shaker, and Mark J. Nelson. The search-based approach. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 17–30. Springer, 2016.

[24] tutorialspoint. Genetic algorithms - crossover. `www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm`, 2018. [Online; Zugriff am 18. Juli 2018].

[25] Stanford University. Game definition language. `http://games.stanford.edu/games/gdl.html`, 2013. [Online; Zugriff am 14. Mai 2018].

[26] Unterhaltungssoftware Selbstkontrolle (USK). Die Genres der USK. `www.usk.de/pruefverfahren/genres/`, 2018. [Online; Zugriff am 6. August 2018].

[27] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018. `http://gameaibook.org`.

Abbildungsverzeichnis

Tabellenverzeichnis

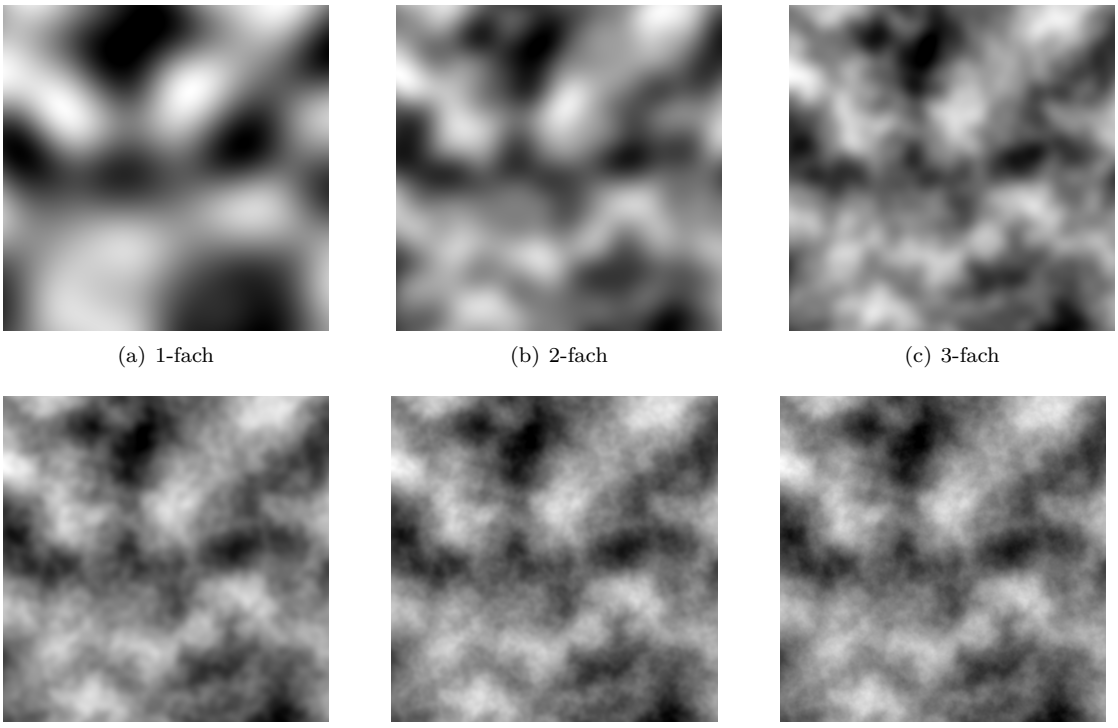
Teil II.

Anhang

A. Anhang

A.1. Fraktale Landschaftserzeugung

Nachfolgend wird gezeigt, wie ein Simplex-Rauschen schrittweise einem $\frac{1}{f}$-Rauschen angenähert wird.

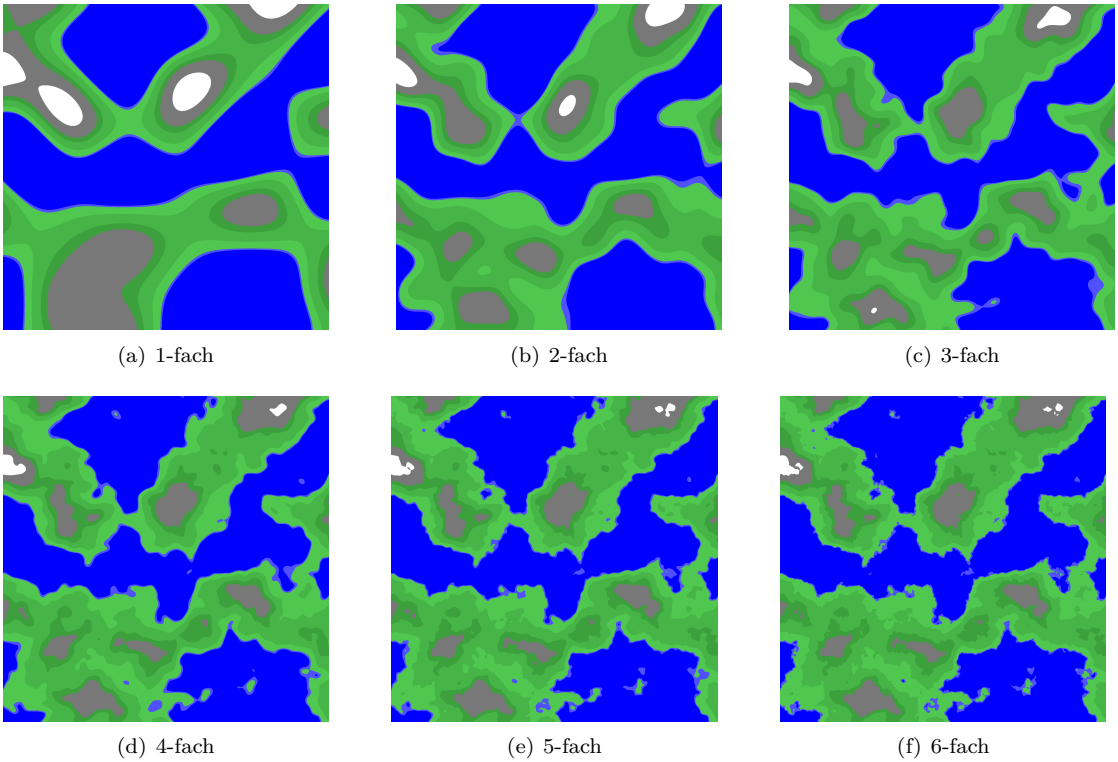


(a) 1-fach (b) 2-fach (c) 3-fach

(d) 4-fach (e) 5-fach (f) 6-fach

Abbildung A.1.: Vergleich von verschieden überlagerten Simplex-Rauschen

Durch eine diskrete Farbzuweisung lassen sich sehr einfach Landschaften erzeugen.



(a) 1-fach (b) 2-fach (c) 3-fach

(d) 4-fach (e) 5-fach (f) 6-fach

Abbildung A.2.: Vergleich von verschieden überlagerten Simplex-Rauschen mit Farbzuweisung

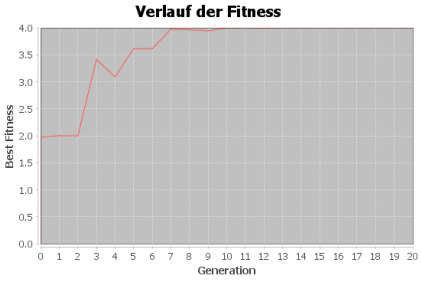
A.2. VGDL als Grammatik

```
1  BasicGame -> SpriteSet "\n" TerminationSet "\n" InteractionSet "\n" LevelMapping
2
3  SpriteSet -> "SpriteSet \n" SpriteSetEntry
4  SpriteSetEntry -> SpriteSetEntry "\n" SpriteSetEntry | SpriteSetEntry "\n\t"
       SpriteSetEntry | SpriteName ">" Type Property
5  Type -> "" | "Immovable" | "Door" | "ShootAvatar stype=" SpriteName | "RandomNPC
       cooldown=" Int | ...
6  Property -> Property Property | "color=" Color | "img=" Image | ...
7
8  LevelMapping -> "LevelMapping \n" LevelMappingEntry
9  LevelMapping -> LevelMappingEntry "\n" LevelMappingEntry | LevelChar ">" LSpriteName
10 LSpriteName -> SpriteName | LSpriteName LSpriteName
11
12 InteractionSet -> "InteractionSet \n" InteractionSetEntry
13 InteractionSetEntry -> InteractionSetEntry "\n" InteractionSetEntry | SpriteName
       ISpriteName ">" Action
14 ISpriteName -> SpriteName | "EOS"
15 Action -> Action Action | "stepBack" | "{killSprite" | "scoreChange=" Int |
       "transformTo stype=" SpriteName | ...
16
17 TerminationSet -> "TerminationSet \n" TerminationSetEntry
18 TerminationSetEntry -> TerminationSetEntry "\n" TerminationSetEntry | Condition ">
       win=" Bool
19 Condition -> MCondition TSpriteName1 TSpriteName2 TSpriteName3 "limit=" Int |
       SCondition "stype=" SpriteName "limit=" Int | "Timeout limit=" Int
20 MCondition -> "MultiSpriteCounter" | "MultiSpriteCounterSubTypes" | "StopCounter"
21 SCondition -> "SpriteCounter" | "SpriteCounterMore"
22 TSpriteName1 -> "stype1=" SpriteName
23 TSpriteName2 -> "stype2=" SpriteName | ""
24 TSpriteName3 -> "stype3=" SpriteName | ""
```

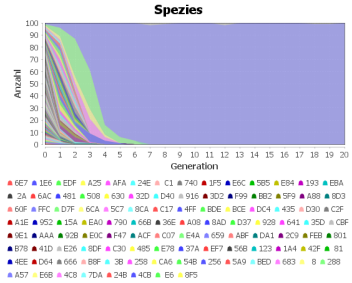
Abbildung A.3.: Beschreibung der VGDL als kontextfreie Grammatik

A.3. Resultate der Testläufe

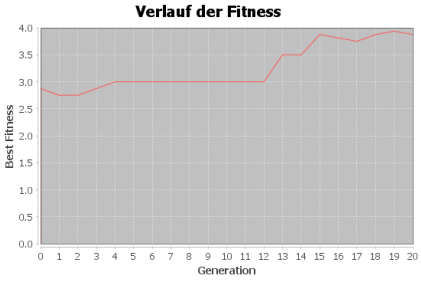
Nachfolgend sind die Ergebnisse verschiedener Versuchsreihen ausgehend von der Startkonfiguration. In Abbildung A.3 wurde die Populationsgrösse von 50 auf 100 erhöht, in Abbildung A.3 auf 10 reduziert. In A.3 wurde die Tournament Grösse auf 3 reduziert.



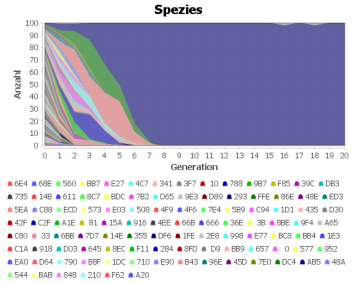
(a) 1. Versuch Fitness



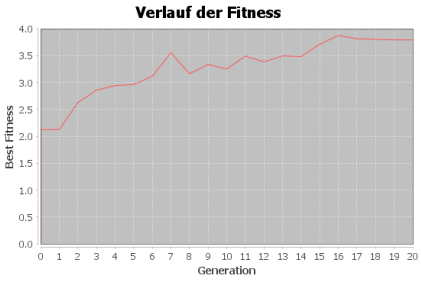
(b) 1. Versuch Spezies



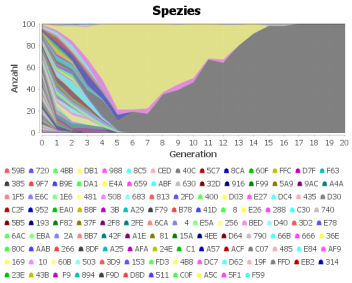
(c) 2. Versuch Fitness



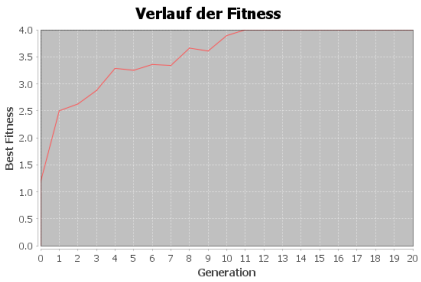
(d) 2. Versuch Spezies



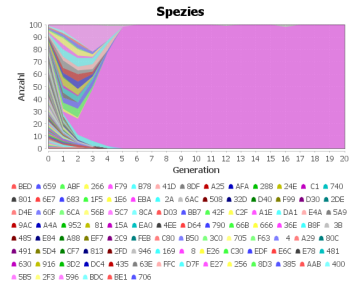
(e) 3. Versuch Fitness



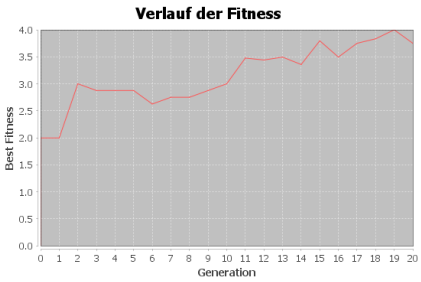
(f) 3. Versuch Spezies



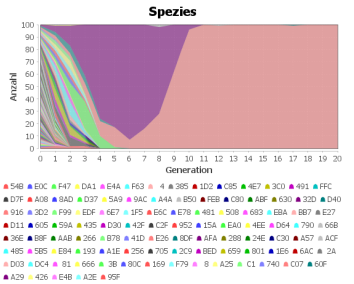
(g) 4. Versuch Fitness



(h) 4. Versuch Spezies

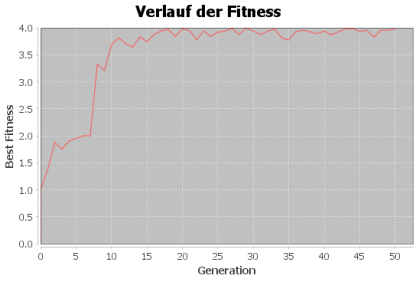


(i) 5. Versuch Fitness

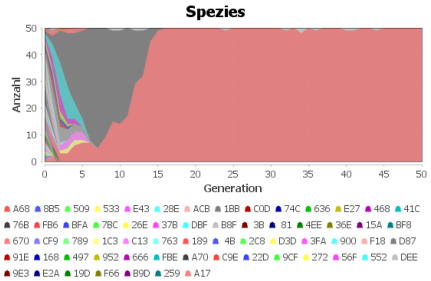


(j) 5. Versuch Spezies

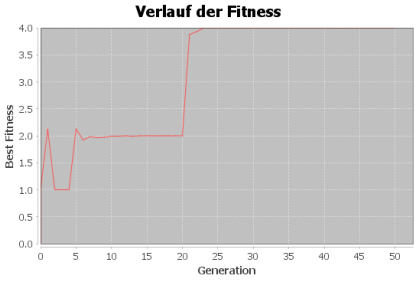
Abbildung A.4.: Populationsgrösse = 100



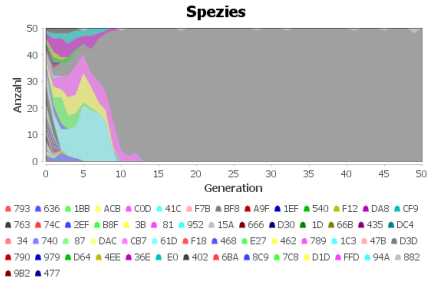
(a) 1. Versuch Fitness



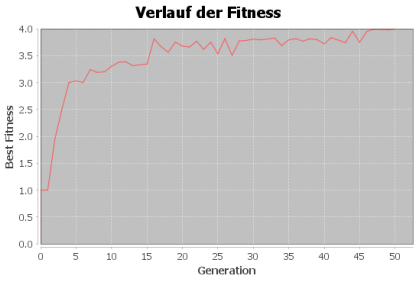
(b) 1. Versuch Spezies



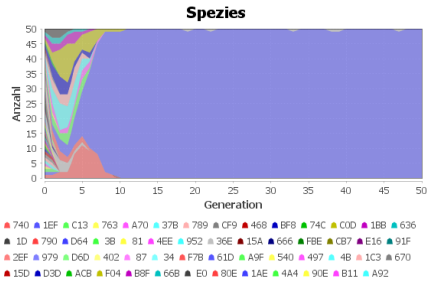
(c) 2. Versuch Fitness



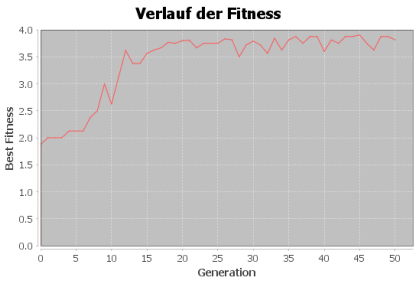
(d) 2. Versuch Spezies



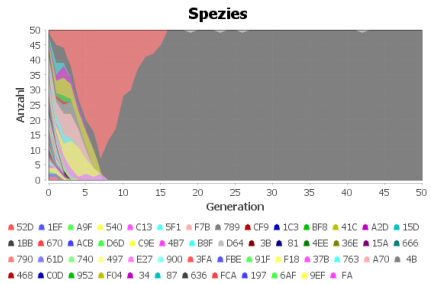
(e) 3. Versuch Fitness



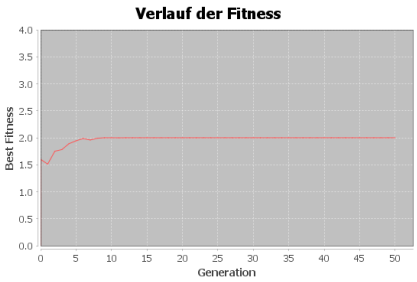
(f) 3. Versuch Spezies



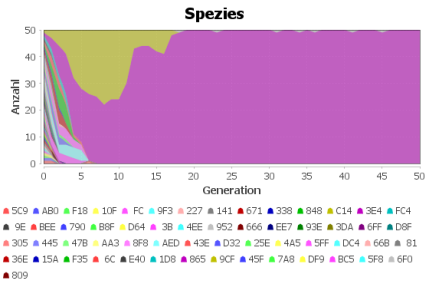
(g) 4. Versuch Fitness



(h) 4. Versuch Spezies

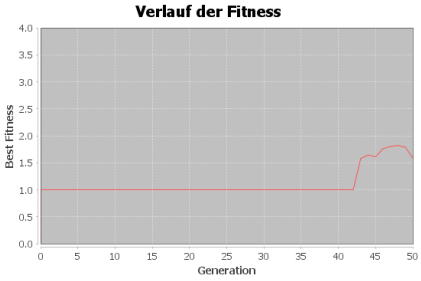


(i) 5. Versuch Fitness

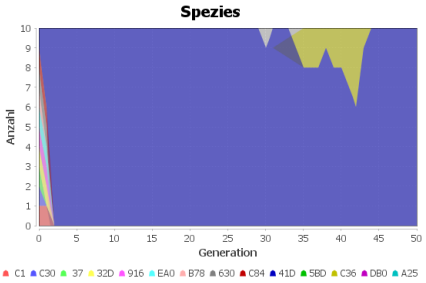


(j) 5. Versuch Spezies

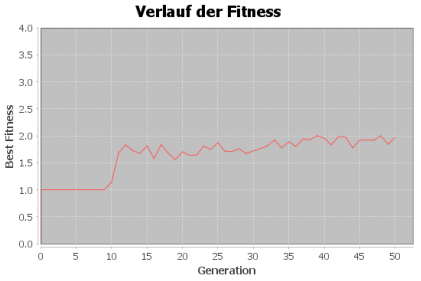
Abbildung A.5.: Tournament Grösse = 3



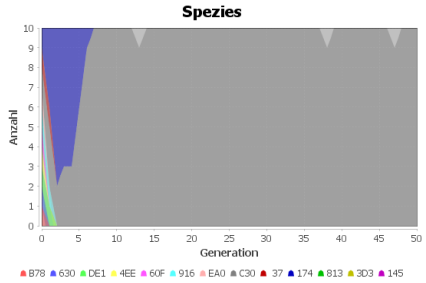
(a) 1. Versuch Fitness



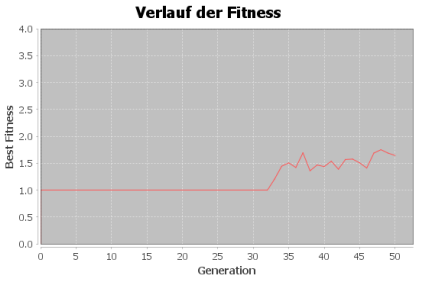
(b) 1. Versuch Spezies



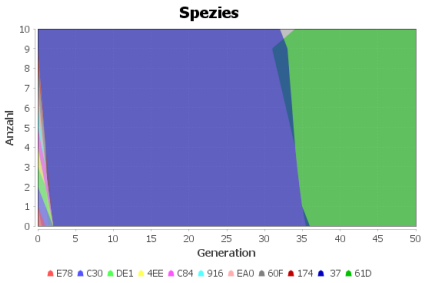
(c) 2. Versuch Fitness



(d) 2. Versuch Spezies



(e) 3. Versuch Fitness



(f) 3. Versuch Spezies

Abbildung A.6.: Populationsgrösse = 10