

# Implementierung eines Roguelike-Videospiels mit adaptiven Gegnern

**Verfasser:** ???

**Betreuungsperson:** ???

**Koreferent:** ???

??, ??

## **Zusammenfassung**

In diesem Projekt wurde ein Roguelike-Videospiel mit adaptiven Gegnern entwickelt und evaluiert, die Reinforcement Learning und genetische Algorithmen nutzen. Dabei wurde analysiert, wie sich das Verhalten der Gegner in einer 2D-Kampfsimulation durch Q-Learning und genetische Algorithmen verändert. Das Spiel wurde in der Godot-Engine entworfen und später auf Itch.io veröffentlicht. In dem Spiel lernten mehrere Agenten, ihre Kampfstrategien über aufeinanderfolgende Angriffswellen hinweg anzupassen. Die Leistung der verschiedenen Konfigurationen wurde anhand der Überlebenszeit, des Schadens am Spieler und der Ausweichfähigkeit der Gegner bewertet. Die Ergebnisse zeigen, dass Q-Learning in Verbindung mit bestimmten Belohnungen zu einem konsistenteren und fitnesshöheren Verhalten führt, während der genetische Algorithmus nur sehr langsam Änderungen hervorbringt. Die Kombination beider Methoden könnte zu einem stabileren Lernen in dynamischen Umgebungen führen. Zukünftige Arbeiten umfassen die Optimierung der Hyperparameter sowie das Ausprobieren weiterer Algorithmen.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>3</b>
1.1 Motivation . . . . .	3
1.2 Zielsetzung . . . . .	3
<b>2 Theoretischer Hintergrund</b>	<b>4</b>
2.1 Roguelike Spiele . . . . .	4
2.1.1 Charakterisierung . . . . .	4
2.1.2 Untergenre von Roguelike und Inspiration . . . . .	4
2.2 Gegnerische KI . . . . .	6
2.2.1 Übliche KI-Implementierung - Hardcoded KI . . . . .	6
2.2.2 Adaptive KI . . . . .	6
2.3 Adaptive KI-Algorithmen . . . . .	8
2.3.1 Reinforcement Learning, Q-Learning . . . . .	8
2.3.2 Genetische Algorithmen . . . . .	9
2.4 Spiel-Engine . . . . .	10
2.4.1 Unity . . . . .	11
2.4.2 Godot . . . . .	11
2.4.3 Unreal Engine . . . . .	12
2.4.4 Ohne Spiel-Engine/Pygame . . . . .	12
<b>3 Methodik und Implementierung</b>	<b>13</b>
3.1 Auswahl der Spiel-Engine . . . . .	13
3.2 Spiel Design . . . . .	13
3.2.1 Untergenre . . . . .	13
3.2.2 Ziel und Setting des Spiels . . . . .	14
3.2.3 Implementierung . . . . .	14
3.3 KI-Design . . . . .	18
3.3.1 Grundlagen zum Gegner . . . . .	18
3.3.2 KI-Algorithmus . . . . .	18
3.3.3 Implementierung . . . . .	20
3.4 Bewertung . . . . .	23
3.4.1 Bewertung des Spassfaktors des Spiels . . . . .	23

3.4.2	Bewertung der KI . . . . .	23
<b>4</b>	<b>Ergebnisse und Diskussion</b>	<b>26</b>
4.1	Ergebnisse . . . . .	26
4.1.1	Ziel 1 und 3: Spiel und Spass . . . . .	26
4.1.2	Ziel 2: Lernende KI . . . . .	28
4.2	Künftige Arbeit . . . . .	31
4.2.1	Mehr Experimente und KI-Verbesserungen . . . . .	31
4.2.2	Gameplay-Verbesserungen . . . . .	31
4.3	Fazit & Reflexion . . . . .	32
<b>5</b>	<b>Anhang</b>	<b>33</b>
5.1	Wichtige Codeausschnitte . . . . .	33
5.2	Python Server und Sockets . . . . .	35
5.2.1	Spielarchitektur . . . . .	36
5.3	Hyperparameter . . . . .	37

# Kapitel 1

## Einleitung

### 1.1 Motivation

Die Wahl des Themas beruhte auf meinem Interesse an KI-Technologien, einem Bereich der Programmierung, der in letzter Zeit einige Durchbrüche verzeichnen konnte. Allerdings braucht jede Technologie auch einen Nutzen. Diesen habe ich in meinen Spielerfahrungen entdeckt. Ich möchte Gegner in einem Spiel programmieren, die aus den Handlungen der Spieler lernen können. Meine Programmiererfahrungen und der Wunsch, dieses Projekt umzusetzen, waren der letzte Anstoss, den ich brauchte.

### 1.2 Zielsetzung

Das Ziel besteht also darin, ein Spiel zu programmieren, bei dem die Gegner aus den Handlungen der Spieler lernen und somit die Wiederspielbarkeit des Spiels erhöhen. Passenderweise wird das Genre „Roguelike“ gewählt, in dem die Wiederspielbarkeit an erster Stelle steht. Die KI sollte so trainiert werden, dass sie sich im Spiel bemerkbar macht und zu einer der Hauptmechaniken wird. Insgesamt sollen also folgende Hauptziele erreicht werden:

- Ein Basisspiel zu erstellen, das Spass macht und bei dem die KI Möglichkeiten hat, zu lernen
- Algorithmen finden und implementieren, die es den Gegnern erlauben, während des Spiels zu lernen
- Die KI so zu regulieren, dass das Spiel Spass macht.

# Kapitel 2

## Theoretischer Hintergrund

Bevor mit der Umsetzung begonnen werden kann, müssen zunächst alle Fachbegriffe definiert und das Projekt konkretisiert werden. Den Anfang machen Roguelike-Spiele.

### 2.1 Roguelike Spiele

#### 2.1.1 Charakterisierung

Roguelike-Spiele sind ein Subgenre der Videospiele, das sich durch wiederholbare Durchläufe, permanente Charaktertode („Permadeath“) und variierende Herausforderungen auszeichnet. Charakteristisch ist, dass jeder Durchlauf (Run) von vorne beginnt und der Spieler im Falle eines Misserfolgs von vorne startet. Fortschritt entsteht somit nicht durch das Speichern von Spielständen, sondern durch das Lernen aus vergangenen Versuchen und das schrittweise Verfeinern der eigenen Strategien. (9)

Diese Lernorientierung ist ein zentrales Merkmal des Genres. Kanagawa und Kaneko (2019) beschreiben Roguelikes als Systeme, in denen Lernen und Anpassung durch Variation und Wiederholung gefördert werden. Das Spiel zwingt die Spielenden, sich fortlaufend an neue Situationen anzupassen, wodurch sowohl mechanisches Können als auch strategisches Denken gefordert sind. (6)

#### 2.1.2 Untergenre von Roguelike und Inspiration

Das Roguelike-Genre ist jedoch recht allgemein gehalten. Um das Spiel genauer zu beschreiben, werden Untergenres oder Kategorien verwendet. In diesem Abschnitt werden diese anhand von Beispielen beliebter Roguelikes vorgestellt. Die Liste ist natürlich sehr begrenzt und zeigt hauptsächlich die Spiele, die mein Produkt inspiriert haben.

#### Vampire Survivors

*Vampire Survivors* ist ein Roguelike, das den sogenannten Bullet Heaven-Stil populär gemacht hat. Es zeichnet sich durch automatische Angriffe und massenhafte Gegner aus. Die Variabilität

entsteht durch zufällige Waffen-Upgrades und unterschiedliche Gegner. Die Runs dauern etwa 30 Minuten und bieten durch Meta-Upgrades eine langfristige Motivation. (7)



Abbildung 2.1: Screenshot aus *Vampire Survivors*, Quelle: Steam-Seite (7)

### Enter the Gungeon

*Enter the Gungeon* ist ein Top-Down-Shooter-Roguelike, der im Gegensatz zu Vampire Survivors das Bullet-Hell-Genre populär gemacht hat. Die Spieler durchqueren 5 bis 10 Stockwerke, finden neue Gegenstände und Waffen, bekämpfen Feinde und „töten“ ihre Vergangenheit, um zu gewinnen. (10)

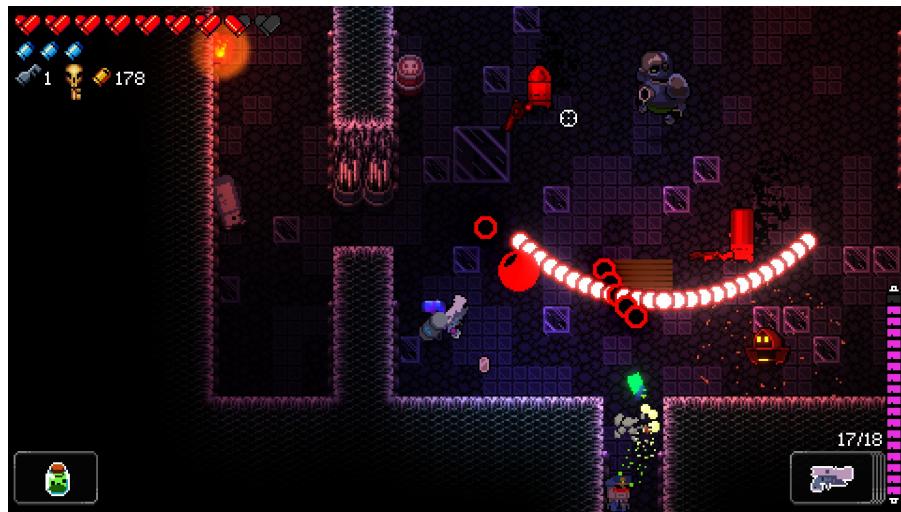


Abbildung 2.2: Screenshot aus *Enter the Gungeon*, Eigene Aufnahme

## 2.2 Gegnerische KI

Um das Verständnis des Spielkonzepts weiter zu vertiefen, werden im folgenden Abschnitt die Hintergründe der künstlichen Intelligenz in Spielen erläutert. Dabei werden die fest programmierte und die adaptive KI erklärt und verglichen. Der darauffolgende Abschnitt befasst sich mit den technischen Aspekten der adaptiven KI anhand von Algorithmen.

### 2.2.1 Übliche KI-Implementierung - Hardcoded KI

Die übliche Methode, Intelligenz in Spielen zu imitieren, besteht darin, sie fest zu programmieren. Das Verhalten der Gegner in den oben genannten Spielen dient als Beispiel dafür.

In Vampire Survivors verhalten sich die meisten Gegner wie folgt: Gegner erscheinen im Spiel und bewegen sich sofort auf den Spieler zu. Wenn der Spieler den Gegner nicht rechtzeitig tötet oder sich von ihm entfernt, wird der Gegner den Spieler verletzen.

Ähnliches gilt für die Gegner in Enter the Gungeon, von denen die meisten sich dem Spieler nähern und direkt auf ihn schießen. Einige Gegner zeigen komplexeres Verhalten, z. B. indem sie Tische benutzen, um die Kugeln des Spielers abzuwehren und sich dahinter zu verstecken. Obwohl dieses Verhalten clever erscheint, kann es dem Spieler oft helfen: Die Tische blockieren auch die Kugeln der Gegner, und manchmal schießen die Gegner auf den Spieler, woraufhin ein anderer Gegner den Tisch aktiviert und den gesamten Angriff auf den Spieler blockiert. Es ist klar, dass die Gegner nicht dynamisch auf den Spieler reagieren, sondern jedes Mal, wenn das Spiel gespielt wird, das gleiche Verhalten zeigen.

Im Allgemeinen ist es einfach, die oben genannten Verhaltensweisen zu programmieren. Die Gegner erhalten Informationen über die Position des Spielers und verfügen über „if“-Anweisungen, um ihre nächste Aktion zu entscheiden. Diese KI wird oft als „hard-coded“ bezeichnet.



Abbildung 2.3: Screenshot aus Enter the Gungeon, Eigene Aufnahme. Einer der Gegner versteckt sich hinter dem Tisch

### 2.2.2 Adaptive KI

Adaptive KI in Videospielen bezeichnet künstliche Intelligenzsysteme, die sich dynamisch an das Verhalten der Spieler anpassen. Anders als traditionelle KI, die vordefinierte, festgelegte Verhaltensmuster nutzt, lernt Adaptive KI fortlaufend aus den Aktionen des Spielers — z. B. wie er kämpft, wie er Ausweichmanöver macht, welche Strategie er nutzt — und verändert daraufhin ihre Taktiken, Schwierigkeit und Reaktionen in Echtzeit oder über Spielabschnitte hinweg. (2)

## Adaptive KI in Spielen

Im Gegensatz zu fest programmierte KI ist adaptive KI in Spielen selten und im Roguelike-Genre noch seltener. Der Grund dafür liegt in den Vor- und Nachteilen der dynamischen KI – sie ist unvorhersehbar. Spieler können die Muster ihrer Gegner nicht lernen und sie dann wie gewohnt dominieren. Adaptive KI ist außerdem viel schwieriger zu implementieren und erfordert eine Menge Rechenleistung. Trotz dieser Probleme gibt es immer noch Spiele, die adaptive KI für ihre Gegner verwenden. Nachfolgend finden Sie eine Liste der beliebtesten Spiele, die diese Technologie nutzen.

### Alien: Isolation

*Alien: Isolation* ist ein Stealth-Horror-Spiel, in dem der Spieler in einer Raumstation mit cleveren Taktiken einem furchterregenden Ausserirdischen ausweichen muss. KI-Einsatz: Der Xenomorph (feindlicher Ausserirdischer) nutzt ein zweistufiges KI-System: Eine Ebene weiss, wo sich der Spieler befindet, die andere nicht - so wirkt der Ausserirdische intelligent und unberechenbar. Er „lernt“ die Verhaltensmuster des Spielers und passt sich an, indem er Geräusche oder Bewegungen mit der Zeit aggressiver untersucht. ([2](#); [1](#))



Abbildung 2.4: Screenshot aus *Alien: Isolation*, Quelle: Steam-Seite ([1](#))

### Middle-earth: Shadow of Mordor

*Middle-earth: Shadow of Mordor* ist ein Action-Adventure-Spiel, das in Tolkiens Universum spielt, in dem sich die Gegner je nach deinen Handlungen weiterentwickeln. Einsatz von KI: Bekannt für das Nemesis-System, bei dem die Ork-Anführer sich an den Spieler erinnern, stärker werden und ihre Taktik anpassen, wenn sie eine Begegnung überleben. Dadurch entstehen personalisierte Rivalitäten und eine dynamische Entwicklung der Gegner. ([2](#); [8](#))



Abbildung 2.5: Screenshot aus *Middle-earth: Shadow of Mordor*, Quelle: Steam-Seite ([8](#))

## 2.3 Adaptive KI-Algorithmen

Es gibt viele adaptive KI-Algorithmen. In diesem Abschnitt werden die für das Produkt relevantesten Algorithmen vorgestellt und kurz erläutert. Die tatsächliche Implementierung wird im Abschnitt „Implementierung“ behandelt. Viele der Informationen zu Algorithmen sind aus dem Buch „AI and Games“ von Yannakakis und Togelius ([12](#)) paraphrasiert.

### 2.3.1 Reinforcement Learning, Q-Learning

Beim Reinforcement Learning (verstärkendes Lernen) handelt es sich um einen Ansatz der adaptiven KI. Dabei lernen Agenten durch positive oder negative Belohnungen aus der Umgebung, Entscheidungen zu treffen. In einem bestimmten Zustand  $s$  nimmt der Agent eine spezifische Aktion  $a$  aus den verfügbaren Aktionen für diesen Zustand und erhält eine Belohnung  $r$ . ([12](#), S. 71-74)

#### Q-Learning

Q-Learning ist ein modellfreier Algorithmus für verstärktes Lernen, der eine tabellarische Darstellung von  $Q(s, a)$  verwendet. Informell ausgedrückt gibt  $Q(s, a)$  an, wie gut es für den Agenten ist, Aktion  $a$  im Zustand  $s$  auszuführen. Formal beschreibt  $Q(s, a)$  die erwartete diskontierte Belohnung, die der Agent erhält, wenn er Aktion  $a$  im Zustand  $s$  wählt und dann die bestmöglichen Entscheidungen trifft.

Das Ziel des Q-Learning-Agenten ist es, die Belohnung zu maximieren, indem er die richtige Aktion für den Zustand ausführt. Die Belohnung besteht aus einer Summe der erwarteten Werte der zukünftigen Belohnungen. Der Q-Learning-Algorithmus ändert die Q-Werte nach und nach. Am Anfang sind in der Q-Tabelle Werte, die der Entwickler festgelegt hat. Wenn der Agent Aktion  $a$  bei Zustand  $s$  auswählt, besucht er den Zustand  $s'$ , erhält eine Belohnung  $r$  und

aktualisiert seinen  $Q(s, a)$ -Wert folgenderweise:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \left[ r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a) \right] \quad (2.1)$$

wobei  $\alpha \in [0; 1]$  die Lernrate und  $\gamma \in [0; 1]$  der Diskontierungsfaktor ist. Die Lernrate bestimmt, inwieweit die neue Schätzung für  $Q$  die alte Schätzung überschreibt. Der Diskontierungsfaktor gewichtet die Bedeutung früherer gegenüber späteren Belohnungen; je näher  $\gamma$  an 1 liegt, desto grösser ist das Gewicht, das zukünftigen Verstärkungen beigemessen wird. (12, S.74-75)

Der Algorithmus lässt sich im Allgemeinen anhand der folgenden Vorlage zusammenfassen (12, S. 52-53):

#### **Algorithmus 1: Q-Learning**

Gegeben ist eine unmittelbare Belohnungsfunktion  $r$  und eine Tabelle von  $Q(s, a)$ -Werten für alle möglichen Aktionen in jedem Zustand:

1. Initialisiere die Tabelle mit beliebigen  $Q$ -Werten; z. B.  $Q(s, a) = 0$ .
2.  $s \leftarrow$  Startzustand.
3. Solange nicht beendet\*:
  - (a) Wähle eine Aktion  $a$  basierend auf einer aus  $Q$  abgeleiteten Politik (z. B.  $\varepsilon$ -greedy).
  - (b) Führe die Aktion aus, wechsle in den Zustand  $s'$  und erhalte eine unmittelbare Belohnung  $r$ .
  - (c) Aktualisiere den Wert von  $Q(s, a)$  gemäss 2.1 .
  - (d)  $s \leftarrow s'$ .

\*Die am häufigsten verwendeten Abbruchbedingungen betreffen die *Geschwindigkeit* des Algorithmus — d. h. Abbruch nach einer bestimmten Anzahl von Iterationen — oder die *Qualität* der Konvergenz — d. h. Abbruch, wenn die gefundene Politik zufriedenstellend ist.

### **2.3.2 Genetische Algorithmen**

Genetische Algorithmen (GA) sind Optimierungsalgorithmen, die von der darwinistischen Evolution inspiriert sind. Im Allgemeinen besteht die Idee hinter GA darin, eine Population von Lösungen zu erstellen und dann durch Auswahl die guten Lösungen beizubehalten und zu reproduzieren und die schlechten zu verwerfen. Eine weitere Idee, die aus der realen Welt für GA übernommen wurde, ist die Kreuzung oder Rekombination (entspricht der sexuellen Fortpflanzung in der realen Welt). Zwei oder mehr Elternteile erzeugen eine neue Lösung, die die Eigenschaften der Eltern kombiniert. Die Idee dahinter ist, dass wenn zwei Elternteile fit sind, ihre Kinder möglicherweise noch fitter oder zumindest fit genug sind. Allerdings funktioniert die Kreuzung nicht immer gut und muss gut geplant und umgesetzt werden. (12, S. 52)

Der Algorithmus lässt sich im Allgemeinen anhand der folgenden Vorlage zusammenfassen (12, S. 52-53):

### Algorithmus 2: Genetische Algorithmen

1. *Initialisierung:* Die Population wird mit N zufällig erzeugten Lösungen gefüllt. Bekannte Lösungen mit hoher Fitness können ebenfalls zu dieser Anfangspopulation hinzugefügt werden.
2. *Evaluation:* Die Fitnessfunktion wird verwendet, um alle Lösungen in der Population zu bewerten und ihnen Fitnesswerte zuzuweisen.
3. *Auswahl der Eltern:* Auf der Grundlage der Fitness werden diejenigen Populationsmitglieder ausgewählt, die für die Fortpflanzung verwendet werden sollen. Zu den Auswahlstrategien gehören Methoden, die direkt oder indirekt von der Fitness der Lösungen abhängen, darunter Roulette-Rad (proportional zur Fitness), Rangfolge (proportional zum Rang in der Population) und Turnier.
4. *Reproduktion:* Nachkommen entstehen durch Kreuzung der Eltern, durch einfache Kopie der elterlichen Merkmale oder durch eine Kombination dieser Methoden.
5. *Variation:* Die Mutation wird auf einige oder alle Elternteile und/oder Nachkommen angewendet.
6. *Ersatz:* In diesem Schritt wählen wir aus, welche Eltern und/oder Nachkommen es in die nächste Generation schaffen. Beliebte Ersatzstrategien der aktuellen Population sind unter anderem die **Generationsstrategie** (Eltern sterben, Nachkommen ersetzen sie), die **Steady-State-Strategie** (Nachkommen ersetzen die schlechtesten Eltern, wenn und nur wenn sie besser sind) und die **Elitismusstrategie** (Generationsstrategie, aber die besten % der Eltern überleben).
7. *Termination:* Sind wir fertig? Entscheiden Sie anhand der Anzahl der durchlaufenen Generationen oder Bewertungen (**Erschöpfung**), der höchsten von einer Lösung erreichten Fitness (**Erfolg**) und/oder einer anderen Abbruchbedingung.
8. Zurück zu Schritt 2.

## 2.4 Spiel-Engine

Um Spiele schnell entwickeln zu können, wird oft eine sogenannte Spiel-Engine verwendet. Eine Spiel-Engine ist ein System mit einer Reihe vorprogrammierter Tools, die die Entwicklung beschleunigen und vereinfachen. In diesem Abschnitt werden die beliebtesten Engines betrachtet.

Die Vor- und Nachteile werden mithilfe von Artikel (5) zusammengefasst. Im Abschnitt zur Methodik wird dann eine Engine ausgewählt.

### 2.4.1 Unity

Unity ist eine beliebte Spiel-Engine mit einer grossen Community. Sie verwendet C# als Programmiersprache und ermöglicht die Erstellung von 2D- und 3D-Spielen. Sie unterstützt eine Vielzahl von Plattformen und ist ideal für Indie-Entwickler. Allerdings gab es in letzter Zeit einige Probleme mit der Lizenzierung. Außerdem ist sie recht ressourcenintensiv. (5, S.57-58)(11)

- **Vorteile:**

- Grosse Community und viele Tutorials
- Plattformübergreifende Entwicklung (PC, Mobile, Web, etc.)
- Umfangreicher Asset Store mit vielen Erweiterungen

- **Nachteile:**

- Proprietäre Software, kein Zugriff auf den Quellcode
- Relativ hoher Ressourcenverbrauch bei grösseren Projekten

### 2.4.2 Godot

Godot ist eine kostenlose Open-Source-Spiel-Engine, die sich ideal für ressourceneffiziente Projekte eignet. Sie benötigt nur wenige Ressourcen und unterstützt sowohl 2D- als auch 3D-Spiele. Sie unterstützt C#, C++ und Gdscript, eine Python-ähnliche Sprache. Da es sich um eine Open-Source-Engine handelt, können Entwickler sie selbst modifizieren, was zum Experimentieren anregt. Da sie jedoch relativ neu ist, fehlen ihr noch einige Funktionen, insbesondere im 3D-Bereich. Godot verwendet ein sogenanntes Knotensystem. Knoten sind spezielle Objekte, die über spezielle Funktionen verfügen und einer bestimmten Hierarchie folgen. (5, S.56)(4)

- **Vorteile:**

- Komplett kostenlos und Open-Source (MIT-Lizenz)
- Leichtgewichtig und schnell, ideal für kleine bis mittlere Projekte
- Einfache, gut lesbare Scriptsprache (GDScript)

- **Nachteile:**

- 3D-Funktionen noch nicht so ausgereift wie bei Unity oder Unreal
- Kleinere Community und weniger Assets als bei anderen Engines
- Manche Funktionen (z.B. komplexe Physik oder Networking) erfordern eigene Implementierungen

### 2.4.3 Unreal Engine

Unreal Engine ist eine leistungsstarke und umfassende Spiel-Engine. Sie ermöglicht die Erstellung wunderschöner 3D-Grafiken und wird häufig für AAA-Spiele verwendet. Sie unterstützt visuelle Programmiersysteme sowie C++. Für Indie-Spiele oder kleine Projekte kann sie jedoch zu komplex und ressourcenintensiv sein. (5, S.58)(3)

- **Vorteile:**

- Extrem leistungsfähig, besonders für realistische 3D-Grafik
- Visuelles Scripting mit Blueprints erleichtert den Einstieg
- Umfangreiche integrierte Tools (z. B. für Animation, KI, Netzwerk)

- **Nachteile:**

- Hohe Systemanforderungen und lange Build-Zeiten
- Komplexe Engine-Struktur, besonders bei Nutzung von C++
- Für einfache oder 2D-Projekte oft überdimensioniert

### 2.4.4 Ohne Spiel-Engine/Pygame

Kleine Spiele können auch ohne Engine erstellt werden. Stattdessen kann man für einige Sprachen sehr leichtgewichtige Bibliotheken verwenden (z. B. PyGame für Python) und einen Teil der Spiel-Engine-Funktionalität selbst erstellen. Dies gibt vollständige Programmierfreiheit, erfordert jedoch mehr Zeit und Wissen.

# Kapitel 3

## Methodik und Implementierung

### 3.1 Auswahl der Spiel-Engine

Bevor mit der Arbeit an dem Spiel begonnen werden konnte, musste eine endgültige Entscheidung hinsichtlich der Spiel-Engine getroffen werden. Aus der Auswahl an Spiel-Engines im Abschnitt „Theoretischer Hintergrund“ wurde Godot ausgewählt. Dafür gibt es mehrere Gründe, die im Folgenden aufgeführt sind:

- Das Basisspiel für KI-Experimente sollte nicht zu komplex sein, da einfachere Mechaniken das Lernen für die KI erleichtern. Die Unreal Engine ist hierfür sicherlich überdimensioniert, Godot hingegen ist perfekt geeignet.
- Es wird entschieden, dass das Spiel 2D sein wird (siehe nächster Abschnitt). Godot ist dafür bekannt, dass man einfach und schnell 2D Spiele machen kann.
- Das Spiel hat einen experimentellen Charakter, daher wäre es gut, wenn man den Quellcode der Engine einsehen und bei Bedarf ändern kann.
- Godot verwendet GdScript, eine Sprache, die Python sehr ähnlich ist. Ich hatte bereits ziemlich viel Erfahrung mit Python, daher wird es mir nicht schwerfallen, GDScript zu lernen.

### 3.2 Spiel Design

Nachdem ich die Spiel-Engine ausgewählt hatte, konnte ich mit der Arbeit beginnen. Um die Entwicklung des Spiels zu vereinfachen, mussten jedoch einige Konkretisierungen vorgenommen werden. Die wichtigsten sind nachfolgend aufgeführt:

#### 3.2.1 Untergenre

- Das Spiel wird in 2D sein, was bisher noch nicht ausdrücklich erwähnt wurde, aber am sinnvollsten ist. 2D-Spiele sind einfacher zu erstellen als 3D-Spiele und bieten eine einfache Umgebung für Gegner (Bewegung mit nur 2 Koordinaten statt 3). Roguelikes

(Roguelikes) sind meist in 2D, und es gibt keinen Grund, gegen diesen Trend zu verstossen.

- Das Spiel wird ein Subgenre des Top-Down-Horde-Survivor-Shooters sein (ähnlich wie Vampire Survivors). Dadurch lassen sich genetische Algorithmen relativ einfach im Spiel implementieren. Diese Genreklassifizierung, insbesondere das Horde-Survivor-Element, bedeutet, dass die Gegner im Mittelpunkt stehen und immer präsent sind, was für die KI nützlich ist.

### 3.2.2 Ziel und Setting des Spiels

#### Ziel

Das Ziel des Spiels ist es, die Gegner zu neutralisieren, Gegenstände zu sammeln und generell stark genug zu werden, um ein Tor zu zerstören und aus der „Arena“ zu entkommen. Natürlich können sich die Spieler auch andere Ziele setzen, beispielsweise wie lange sie überleben können.

#### Setting

Die Setting ist relativ einfach und für die Durchführung der Experimente nicht wirklich notwendig, macht das Spiel aber unterhaltsamer. Sie wurde von gefundenen Sprites (Bildern von Spielobjekten) inspiriert und lässt sich wie folgt beschreiben:

Der Spieler ist ein Ritter mit einer Familie. Eines Tages, während er fernsieht, wird der Spieler von einem sehr bösen Goblin in sein Lager teleportiert. Der Spieler muss sich nun irgendwie verteidigen und aus dem Lager fliehen. Dazu sieht er in der Nähe eine Waffe, nämlich einen Stock. Er sieht auch die Tore und dass sie zerbrochen werden können. Die Goblins lassen jedoch nicht zu, dass ihre Tore zerbrochen werden, und greifen den Spieler in Wellen an. Die Goblins sind zunächst dumm: Sie benutzen Stöcke als Waffen, schwingen diese eher willkürlich und haben im Allgemeinen keine Strategie. Mit jeder Welle lernen sie jedoch, bessere Waffen herzustellen, Spieler statt Luft anzugreifen und gemeinsam eine Strategie gegen Spieler zu entwickeln.

### 3.2.3 Implementierung

Kommen wir nun zur eigentlichen Umsetzung. In diesem Abschnitt werden die wichtigsten Objekte und Mechaniken des Spiels beschrieben.

#### Spieler

Zuerst habe ich den Spieler erstellt. Dazu habe ich den Knoten „**CharacterBody2D**“ verwendet. Dieser Knotentyp ist für Entitäten vorgesehen und enthält die folgenden Funktionen:

- Bewegung mit Geschwindigkeitsparameter
- Kollisions- und Neigungserkennung mit der Methode move\_and\_slide()

- Berechnung des Eingabevektors

Um jedoch Kollisionen, Animationen und Angriffe für Spieler zu implementieren, benötigt dieser Knoten mehrere zusätzliche untergeordnete Knoten. Diese sind in der Abbildung 3.1 dargestellt und werden im Folgenden näher erläutert. Der erste untergeordnete Knoten ist **CollisionShape2D**. Wie der Name schon sagt, beschreibt er die Kollisionsform von CharacterBody2D. Da das Sprite ebenso einfach ist, reicht hier ein einfaches Rechteck aus. Der zweite Knoten ist **AnimatedSprite2D**. Der Name beschreibt, wie er funktioniert, genau wie beim letzten Knoten, nämlich dass er Animationen mithilfe von Sprite-Sheets erstellt.

Im Abbildung 3.2 ist das genutztes Sprite-Sheets dargestellt. Der dritte Knoten ist **Weaponholder**, ein selbstgemachter Knoten, der für Waffen verwendet wird. Er kann auch als „Waffenkomponente“ bezeichnet werden, da er erneut für Feinde verwendet wird. So gibt es auch **Timer Knoten** und zwar zwei. Sie sind für Abklingzeiten verantwortlich: Eine misst, wann der Spieler einen „Sprint“ ausführen kann, die andere, wann der Spieler wieder verletzt werden kann.



Abbildung 3.2: Die Spieler-Idle-Animationsframes. Die Animation bewegt sich vom linken Sprite nach rechts bis zum Ende und wiederholt sich dann mit einer Geschwindigkeit von 5 FPS (d. h. 5 dieser Sprites pro Sekunde). Die Animation dauert somit eine Sekunde. Sie zeigt hauptsächlich die Atmung.

## Arena

Danach wurde die Umgebung (Arena) für den Spieler erstellt. Zunächst war die Umgebung sehr einfach: nur eine Kachelkarte für den Boden ohne Wände. Später jedoch wurde es komplexer mit der Wände, Spawners und Toren. In Abbildung 3.4 ist die letzte Version dargestellt.

In Godot habe ich den für die Arena verantwortlichen Knoten „**Objekte**“ genannt. Die Struktur ist in der Abbildung 3.3 zu sehen. Sie enthält hauptsächlich einfache Node2D-Knoten (blaue Kreise), die wiederum andere komplexere Knoten enthalten. Der Knoten „**Tiles**“ enthält beispielsweise 6 TileMapLayers, die die folgenden Objekte zeichnen: Boden, Wände, Pflanzen, Schatten, Requisiten und unsichtbare Kollisionssubjekte. Der **Spawners**-Knoten enthält 3 Spawner-Objekte, in denen Gegner erscheinen. **Worldboundary** ist ein sogenannter Area2D-Knoten, der für Siegbedingungen vorgesehen ist und vier rechteckige CollisionShapes enthält. Diese werden an den Grenzen jeder Seite der

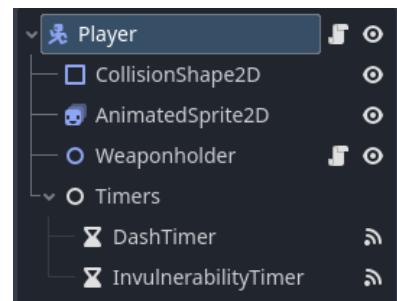


Abbildung 3.1: Die Spieler-Knoten-Struktur

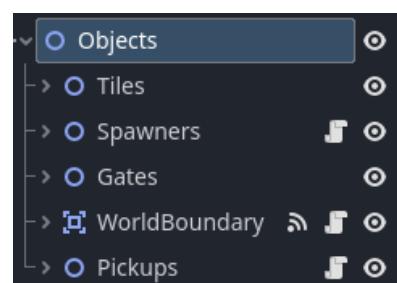


Abbildung 3.3: Die Arena-Knoten-Struktur

Arena platziert und überprüfen, ob Spieler die Arena verlassen haben. Der **Pickups**-Knoten enthält einfach die Waffen und Gegenstände, die auf dem Boden liegen. Der **Gates**-Knoten enthält zwei Tore. Diese werden in nächster Sektion noch genauer betrachtet.

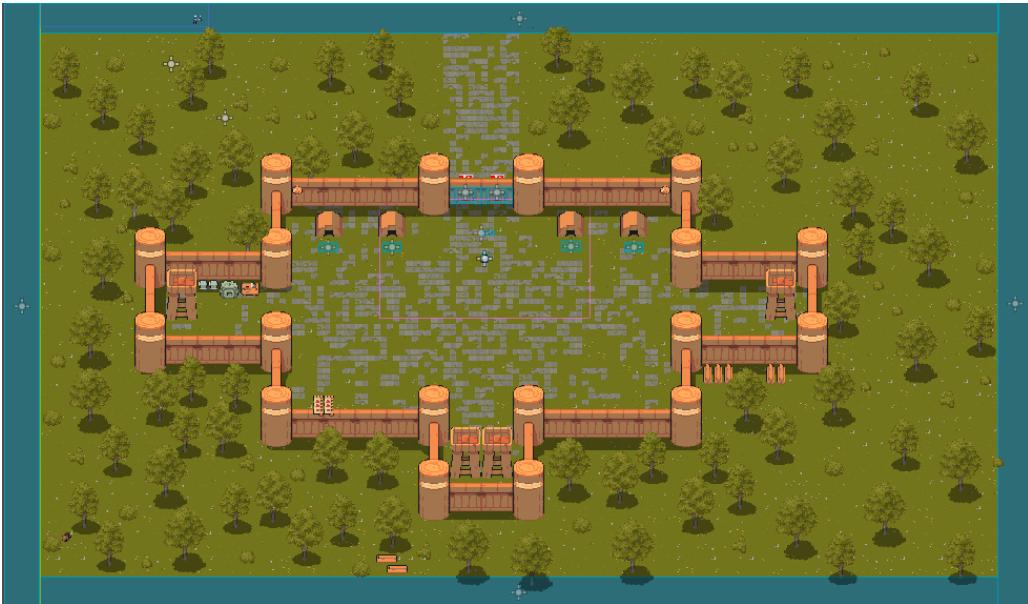


Abbildung 3.4: Die Arena

## Tore

Das Ziel des Spiels ist es, die Tore zu zerstören. Tore sind spezielle Objekte, die im Gegensatz zu Mauern zerstört werden können. Sie verfügen über HP-Komponenten (LP – Lebenspunkte). Wenn die HP aufgrund von Angriffen des Spielers (oder des Gegners) auf null sinken, wird das Tor zerstört und die Kollisionsfigur ändert sich so, dass der Spieler hindurchgehen kann. Dies ist jedoch nicht einfach zu erreichen, da sich die HP jede Sekunde regenerieren. Genauer gesagt sind es 20 HP pro Sekunde, das Tor hat maximal 100 HP und die Regeneration erfolgt diskret, wenn der Timer eine Sekunde zählt (d. h. der Gegner hat 70 HP. Eine Sekunde passt, dann hat er sofort 90 HP). Um die Tore zu zerstören, muss man dem Tor mehr Schaden zufügen, als es regenerieren kann. Dies zwingt dem Spieler dazu, Gegenstände und bessere Waffen zu finden, wenn er gewinnen will.

## Gegner Spawning

Nachdem die Umgebung erstellt war, war es an der Zeit, die Gegner ins Spiel zu bringen. Die Feinde selbst werden im Abschnitt „KI-Design“ behandelt, hier beschreiben wir jedoch die Mechanik des Feind-Spawnens. Das Spawning erfolgt wie zuvor beschrieben an Spawn-Punkten und hängt von mehreren Variablen ab. Ein Gegner spawnt an einem zufälligen Spawn-Punkt in der Arena, wenn die Welle aktiv ist und die Gegnerbegrenzung pro Welle nicht überschritten wurde. Die Geschwindigkeit, mit der Feinde spawnen, und die Begrenzung der Gegner pro Welle hängen von der Welle ab.

## Waffen

Waffen wurden schon recht früh implementiert, aber sie wurden vielen Änderungen unterzogen. Letztendlich wurden drei Klassen (Typen) für Waffen erstellt: eine abstrakte Klasse „**Weapon**“, die eine allgemeine, nicht existierende Waffe beschreibt, aber dazu dient, Code-Wiederholungen zu vermeiden, und zwei von der ersten Klasse abgeleitete Klassen: **Melee-Weapon** und **RangedWeapon**. Die Namen der Klassen machen den Unterschied zwischen den beiden Klassen ziemlich deutlich: Die Klasse MeleeWeapon erzeugt Nahkampfwaffen (Beispiel: Schwert, siehe Abbildung 3.5 (a), während die Klasse RangedWeapon Projektiler erzeugt, die Schaden verursachen (Beispiel: SMG, siehe Abbildung 3.5 (b)).

Sowohl Spieler als auch Gegner können Waffen tragen. Spieler können diese aufheben und dann verwenden. Die Art der Waffen, mit denen Gegner erscheinen, hängt von der Welle ab, ebenso wie die Anzahl der Gegner. Die Seltenheit der Waffe, d. h. ob es sich um einen Stock oder ein Schwert handelt, ist zufällig. Jeder Gegner hat eine Chance, seine Waffe fallen zu lassen, wenn er getötet wird. Auf diese Weise kann der Spieler bessere Waffen finden, um mehr Schaden zu verursachen.



(a) Beispiel der MeleeWeapon



(b) Beispiel der RangedWeapon

Abbildung 3.5: Die Waffenklassen

## Level

Alle benannten Objekte können während des Spiels in hochrangigen Knotenpunkten gefunden werden, die als „Levels“ bezeichnet werden. Diese werden instanziert, wenn man auf „Play“ klickt. Die Knotenpunktstruktur ist in Abbildung 3.6 dargestellt. Die Knoten „Projektil“ und „Entitäten“ enthalten die Projektiler und Entitäten, die während des Spiels entstehen. Zu Beginn sind sie leer (sie haben keine untergeordneten Knoten). Der UI-Knoten verfügt über ein HUD (LP, Anzahl der Gegner, aktuelle Welle sowie aktuelle Gegenstände und Waffen), ein Pausenmenü sowie Upgrade- und Tutorial-Fenster.

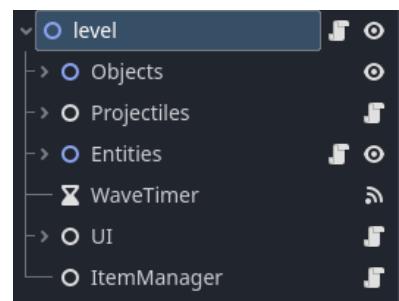


Abbildung 3.6: Die Level-Knoten-Struktur

## 3.3 KI-Design

### 3.3.1 Grundlagen zum Gegner

Bevor KI verstanden werden kann, muss eine Grundlage für den Gegner geschaffen werden. Der Gegner ist dem Spieler sehr ähnlich; die Knotenstruktur ist fast identisch (siehe Abbildung 3.7). Ein Unterschied besteht darin, dass er über Area2D verfügt, mit dem die Angriffs-Hitbox konfiguriert wird. Die Knoten sind gelb, da sie von Enemy Scene geerbt wurden. Dies wurde zuvor verwendet, wenn nicht alle Gegner über eine Waffe verfügten; sie konnten dem Spieler nur durch Kontakt Schaden zufügen. Diese Szene bzw. Klasse wird nur als Abstraktion und übergeordnete Klasse verwendet.

Damit KI verstehen kann, welche Aktionen möglich sind, müssen diese definiert werden. Gegner können Aktionen aus einer Liste ausführen, die im Code unter 3.1 zu finden ist.

Code 3.1: Mögliche Aktionen

```
1 var valid_actions = ["move_forward", "strafe_left", "strafe_right", "retreat", "use_weapon"]
```

Was genau diese Aktionen bewirken, ist in der Funktion „execute\_action“ definiert. Dies ist im Anhang in Code 5.1 dargestellt. Die Umsetzung der Aktionen ist ziemlich einfach. Die meisten Aktionen ändern lediglich die Richtung des Geschwindigkeitsvektors. Die Bewegung wird mit der integrierten Funktion „move\_and\_slide“ umgesetzt. Diese Funktion ist nur für CharacterBody2D verfügbar und bewegt das Objekt mit definierter Geschwindigkeit. Das wurde auch für den Spieler genutzt. Die letzte Aktion „use\_weapon“ nutzt die Waffe, wenn der Gegner sie hat. Die Funktion „add\_reward\_event“ wird im nächsten Abschnitt im Zusammenhang mit Q-Learning behandelt.

### 3.3.2 KI-Algorithmus

Nachdem die Grundlagen des Gegners bekannt sind, kann ein Algorithmus zur Berechnung seiner Intelligenz entwickelt werden. Dafür werden sowohl Q-Learning als auch genetische Algorithmen genutzt, wodurch dieser neue Algorithmus hybrid wird.

#### Q-Learning Verwendung

Um den Algorithmus zu nutzen, wird jeder Gegner eine Q-Tabelle als Dictionary (bzw. Hashmap) besitzen. Diese enthält als Schlüssel den Zustand des Gegners und als Wert ein Dictionary mit Aktionen als Schlüsseln und Q-Werten als Werten. Anhand eines Beispiels wird dieses Format in der Implementierung klarer. Zu Beginn ist die Q-Tabelle leer, später wird sie jedoch

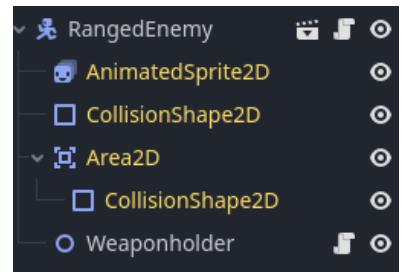


Abbildung 3.7: Die Gegner-Knoten-Strukture

ausgefüllt und modifiziert. Die Gegner suchen ihren Zustand in der Q-Tabelle. Falls sie diesen Zustand bereits erlebt haben (es gibt einen Q-Wert für jede Aktion der Tabelle), wählen sie die beste mögliche Aktion (die Aktion mit dem höchsten Q-Wert). Anschliessend führt der Gegner die Aktion im Spiel aus und erhält die Belohnung. Diese Belohnung aktualisiert den Q-Wert der ausgeführten Aktion mithilfe der Q-Learning-Formel (siehe Gleichung: 2.1). Dieser Prozess wird in jedem Frame wiederholt und nur beendet, wenn der Gegner stirbt.

### **Genetischer Algorithmus Verwendung**

Nachdem alle Gegner besiegt wurden, endet die Welle. Die resultierenden Q-Tabellen der Gegner können mithilfe von GA zur Gegneroptimierung genutzt werden. Dazu muss jedoch jede Q-Tabelle anhand der Performance der Gegner evaluiert werden. Das heisst, die Fitness jedes Gegners muss berechnet werden. Der berechnete Fitness-Wert wird in einem Dictionary mit der Gegner-ID als Schlüssel gespeichert – nicht in der Gegner-Instanz, da diese gelöscht wird. Mithilfe des bekannten Fitness-Werts kann sich die Population (die Gegner) nun selektiv reproduzieren, um eine neue Q-Tabelle zu erstellen. Aus dieser werden alle Gegner der nächsten Welle initialisiert und nicht, wie bei der ersten Welle, aus einem leeren Dictionary. So werden die Gegner mit jeder Welle intelligenter.

### **Resultierender Algorithmus**

Wenn man die beiden Algorithmen kombiniert, erhält man folgenden hybriden, spezifischen Algorithmus zum Spiel:

#### **Algorithmus 3: Genetisches Q-Learning**

1. *Initialisierung:* Die Welle beginnt. Die Population (die Gegner) initialisiert ihre Q-Tabelle aus der Shared Q-Tabelle, sofern diese vorhanden ist. Andernfalls wird sie als leer gesetzt.
2. *Leben:* Für jede Instanz in der Population:
  - (a)  $s \leftarrow$  Startzustand.
  - (b) Solange nicht tot:
    - i. Wähle eine Aktion  $a$  basierend auf den q-Werten und dem Zustand  $s$  aus.
    - ii. Führe  $a$  aus, wechsle in den Zustand  $s'$  und erhalte eine unmittelbare Belohnung  $r$ .
    - iii. Aktualisiere den Wert von  $Q(s, a)$  gemäss Gleichung 2.1 .
    - iv.  $s \leftarrow s'$ .
  - (c) *Evaluation:* Wenn tot, berechne die eigene Fitness und speichere sie.

- 3. Auswahl der Eltern:** Auf der Grundlage der Fitness werden diejenigen Populationsmitglieder ausgewählt, die für die Fortpflanzung verwendet werden sollen.
- 4. Reproduktion:** Nachkommen entstehen durch Kreuzung der Eltern, durch einfache Kopie der elterlichen Merkmale oder durch eine Kombination dieser Methoden.
- 5. Variation:** Die Mutation wird mit einer bestimmten Wahrscheinlichkeit auf einige Nachkommen angewendet.
- 6. Ersatz:** Ersatz erfolgt de facto durch die Generationsstrategie. Die Eltern sterben während der Welle, sie reproduzieren nach deren Ende.
- 7. Zurück zu Schritt 1.**

### 3.3.3 Implementierung

In diesem Abschnitt wird genau beschrieben, wie der oben beschriebene Algorithmus implementiert und in das Spiel integriert wurde.

*Initialisierung.* Die Welle beginnt und die Spawner spawnen die Gegner. Die Gegner beginnen mit leeren Q tabellen.

*Leben.* Jeder Gegner berechnet seinen Zustand als String. Ein Zustand sieht beispielsweise so aus: „**wt0d4a0bd-1ba-1**“. Die Zustandsberechnung ist einer der wichtigsten Teile für die lernende KI. Der Zustand beschreibt die Situation der Umwelt und erfüllt damit die gleiche Funktion wie die sensorischen Organe. Daher wird nun die Berechnung und Wahl der Parameter erklärt.

#### Zustandsberechnung

Zur Veranschaulichung der Zustandsberechnung betrachten wir noch einmal den Zustand:

**"wt0d4a0bd-1ba-1"**

Der Zustand wird durch eine Folge aus Buchstaben und Zahlen dargestellt. Zunächst kommen ein paar Buchstaben, dann folgt eine Zahl. Die Bedeutung ist folgende: Die Buchstaben geben den verkürzten Namen des Parameters an, die Zahl seinen Wert. Ein Beispiel für den ersten Parameter ist „**wt0**“. Der vollständige Name lautet „`weapon_type`“ und gibt an, ob der Gegner eine Nahkampfwaffe (0) oder eine Fernkampfwaffe (1) hat. Die Formatierung und Bedeutung jedes Parameters wird klarer, wenn wir uns den eigentlichen Code der Funktion „`get_state()`“ anschauen. Was die Funktion zurückgibt, ist in Code 3.2 dargestellt.

Code 3.2: Wiedergabe der Funktion „get\_state”.

```
1  return "wt{wt}d{d}a{a}bd{bd}ba{ba}".format({  
2      "wt": weapon_type, "d": dist, "a": angle, "bd": bullet_dist, "ba":  
3      bullet_angle  
4  })
```

Bevor die Parameter aufgelistet werden, sollte Folgendes erwähnt werden: Die möglichen Werte der Parameter sollten klein und diskret gewählt werden. Andernfalls würde die Kombination der Parameter zu viele Zustände erzeugen, sodass die Gegner fast nie denselben Zustand erreichen würden. Dies würde das Lernen sehr erschweren, da die Situationen immer unterschiedlich wären.

Jeder Parameter wird anhand dessen erklärt, was er beschreibt, warum er für das Lernen wichtig ist und welche möglichen Werte er annehmen kann:

- „wt“ – Waffentyp. Wie im Beispiel erläutert, gibt dieser Parameter an, welche Waffe der Gegner trägt. Mögliche Werte sind: (-1): keine Waffe, (0): Nahkampfwaffe und (1): Fernkampfwaffe. Der Grund für die Nutzung dieses Parameters sollte ziemlich klar sein. Nahkampfgegner und Fernkampfgegner sollen sich unterschiedlich verhalten. Nahkampfgegner können dem Spieler nur schaden, wenn sie ihm nah sind, während Fernkampfgegner keinen Grund haben, ihm sehr nah zu sein.
- „d“ – Distanz zum Spieler. Mögliche Werte sind durch den Bereich (0 – 4) beschrieben, wobei 0 bedeutet, dass sich der Spieler direkt daneben befindet, und 4, dass er sich sehr weit entfernt befindet. Dieser Parameter zeigt dem Gegner direkt an, ob ein Angriff sinnvoll ist.
- „a“ – Der Winkel zum Spieler. Mögliche Werte sind durch den Bereich (0 – 3) beschrieben, wobei jeder Wert eine Richtung (N, S, W oder E) zum Spieler hin angibt. Das hilft dem Gegner, eine Flanking-Strategie zu entwickeln. Das bedeutet beispielsweise, die schwache Seite des Spielers anzugreifen oder ihn von allen Seiten anzugreifen, damit er nicht fliehen kann.
- „bd“ – Distanz zur nächsten Kugel. Die möglichen Werte sind dieselben wie bei der Distanz zum Spieler, jedoch wird (-1) addiert, wenn es keine Kugel gibt. Dieser Parameter ist wichtig, damit die Gegner den Kugeln des Spielers ausweichen können.
- „ba“ – Der Winkel zur nächsten Kugel. Die möglichen Werte sind dieselben wie beim Winkel zum Spieler. Es wird jedoch (-1) addiert, wenn keine Kugel vorhanden ist. Dies ist für die Ausweichmechanik hilfreich.

Auf diese Weise werden die Zustände aller Gegner berechnet und in ihrer Instanz als „current\_state“ gespeichert.

## Aktionwahl

Die Gegner können nun den bestimmten Zustand als Schlüssel zur Q-Tabelle nutzen, um die Aktionen mit ihren Q-Werten zu ermitteln. Aus diesen Aktion-Q-Wert-Paaren wählen sie nun die Aktion mit dem höchsten Q-Wert oder falls der Zufall es so will, eine andere Aktion zur Exploration. Eine zufällige Aktion wird auch gewählt, wenn es für den Zustand noch keinen Q-Wert gibt. So werden für alle Gegner gewählte Aktionen in einem Dictionary gespeichert, wobei die Gegner-ID der Schlüssel und die Aktion der Wert ist.

## Durchführung und Belohnung

Sobald die Aktionen ausgewählt wurden, können die Gegner sie nun im Spiel ausführen. Welche Aktionen ausgewählt werden können und wie sie definiert sind, wurde bereits in Code 5.1 erläutert. Gemäss Q-Learning muss die Aktion nach der Ausführung belohnt werden. Dies wird für jede Aktion mit einfachen Zahlen definiert. Die genauen Zahlen sind in Code 3.3 aufgeführt.

Code 3.3: Die Belohnungen

```
1     self.REWARDS = {  
2         "TOOK_DAMAGE": -5,  
3         "TIME_ALIVE": 0.0,  
4         "HIT_PLAYER": 10,  
5         "RETREATED": -0.2,  
6         "WASTED_MOVEMENT": -0.05,  
7         "MOVED_CLOSER": 0.05,  
8         "MISSED": -0.2,  
9         "DIED": -5,  
10        "STUCK": -1,  
11        "DODGED_BULLET": 6  
12    }  
13
```

Die Zustandsberechnung, die Auswahl der Aktion, die Ausführung und die anschliessende Belohnung werden auf diese Weise in jedem Frame wiederholt, bis der Gegner stirbt. Nach dem Tod des Gegners findet eine **evaluation** statt, d. h. eine Fitnessberechnung. Um die Fitness zu berechnen, speichern die Gegner während des Lebensprozesses, wie viel Schaden sie dem Spieler zugefügt haben, wie lange sie gelebt haben und wie vielen spielerischen Kugeln sie ausgewichen sind. Diese Parameter werden nun verwendet, um die Fitness der Gegner zu berechnen. Dies erfolgt gemäss Gleichung 3.1.

$$f(L, D, B) = 7D + 0.2L + 1B \quad (3.1)$$

Dabei bezeichnet  $D$  den verursachten Schaden am Spieler,  $L$  die Überlebenszeit in Sekunden und  $B$  die Anzahl der ausgewichenen Kugeln.

Der nächste implementierte Schritt ist die **selektion**. Dabei werden zwei Gegner die meist

berechneten Fitness haben. Danach folgt die Reproduktion mit Kreuzung. Hierbei kommt eine klassische Kreuzungsstrategie zum Einsatz: Für die Erstellung einer neuen Q-Tabelle werden die Werte jedes Elternteils mit einem 50-Perzentil gewählt. Genauer gesagt wird für jeden Zustand der gesamte Aktions-Q-Wert-Dictionay von zufällig gewählten Eltern übernommen. In diesem Schritt findet die **Mutation** statt. Jeder Q-Wert hat eine Wahrscheinlichkeit von 5 Prozent, sich um einen zufällig gewählten Wert von -1 bis 1 zu ändern.

Diese Schritte finden am Ende einer Welle statt. Anschliessend beginnt die neue Welle, diesmal jedoch mit Gegnern, die aus der letzten Welle reproduziert wurden. Die genauen Werte für die Belohnungen und die Fitnessfunktion wurden durch Ausprobieren ermittelt und sind wahrscheinlich nicht optimal. Eine bessere Methode zur Ermittlung dieser Werte wird im Abschnitt über zukünftige Arbeiten erläutert.

## 3.4 Bewertung

Um die Vor- und Nachteile des Spiels und der KI besser zu erkennen und somit zu sehen, wo es Verbesserungspotenzial gibt, wurde beschlossen, das Spiel und vor allem die KI zu testen.

### 3.4.1 Bewertung des Spassfaktors des Spiels

Andere spielen das Spiel und sammeln Feedback, um zu überprüfen, wie viel Spass das Spiel macht und ob die Spieler die Verbesserungen der KI während des Spiels bemerken. Zu diesem Zweck veröffentlichen Entwickler ihre Spiele und es wurde beschlossen, dass auch mein Spiel veröffentlicht werden sollte. Es gibt viele Anbieter für die Veröffentlichung von Spielen, aber die Wahl fiel hier nicht schwer. Itch.io bietet eine kostenlose Veröffentlichung, zeigt das Spiel Hunderten von Menschen und ist entwicklerfreundlich.

Spassfaktor wird nur qualitativ anhand informeller Fragen, Kommentaren und meiner eigenen Meinung bewertet.

### 3.4.2 Bewertung der KI

Um den Erfolg der Ziele 3 und 4 zu überprüfen, musste ein Massstab für das KI-Lernen festgelegt werden. Zu diesem Zweck wird die Effizienz von Q-Learning, genetischen Algorithmen und Baselines einzeln und gemeinsam bewertet. Im Allgemeinen werden die folgenden Metriken der Gegner untersucht: Lebensdauer, dem Spieler zugefügter Schaden und Anzahl. Die ersten beiden Metriken werden mit der Fitnessfunktion (siehe Gleichung 3.1) kombiniert, um einen Fitnesswert zu ermitteln.

#### Testumgebung

Die Testumgebung wird ganz anders aussehen als das Spiel, sie wird einfacher sein. Alle Gegner werden die gleiche Waffe, nämlich die „Handgun“, tragen. Der Spieler wird diese Waffe ebenfalls tragen und über unendliche Munition verfügen. Es wird keine Items geben, die die Spieler

stärker machen. Die Spieler selbst werden durch Bots ersetzt, sodass die Tests leicht automatisierbar sind. Der Spieler ist unsterblich, sodass das Spiel bis zu einer bestimmten Wellenanzahl weitergeht und dann schliesst. Die Wellen werden jeweils acht Gegner beinhalten, diese Zahl wird im Gegensatz zum normalen Spiel mit jeder Welle nicht erhöht. Auch die Werte der Gegner bleiben mit jeder Wellenanzahl konstant. Das Spiel wird ohne Menüs sofort gestartet. Zudem wird ein Seed eingegeben, sodass die Experimente reproduzierbar sind. All diese Änderungen wurden vorgenommen, damit das Spiel weniger Varianz aufweist und die Gegner effektiv lernen können. Die wichtigsten Parameter sind auch in Tabelle 3.1 notiert.

## Technische Seite der Tests

Die Tests fanden ausserhalb von Godot in Python statt, sodass man viele Tests mit verschiedenen Parametern parallel starten kann. Mit Python lassen sich auch die Plots leicht mit matplotlib erstellen. Die genaue Funktionsweise der Kommunikation zwischen Python und Godot wird im Anhang erläutert.

Zur Datenerfassung werden sogenannte Log-Typen erstellt. Die Bedeutung ist einfach: Art der Datensammlung. Während des Spiels werden zwei dieser Log-Typen erstellt:

- „Wave Snapshot“: Dieser passiert konstant in gleichen Zeitintervallen von einer Sekunde. Dabei werden alle relevanten Daten aller lebenden Gegner erfasst, durchschnittlich berechnet und an Python geschickt.
- „Death Log“: Dieser passiert, wenn die Gegner sterben. Dabei werden alle relevanten Daten über den jeweiligen Gegner erfasst und an Python geschickt.

Python speichert dann die Daten als Eintrag in einer CSV-Datei. Nun folgen die Testkonfigurationen.

## Kontrollgruppe

Zur Überprüfung der Effizienz anderer Algorithmen wird eine Kontrollgruppe erstellt. In dieser werden gegnerische Q-Tabellen mit zufälligen Q-Werten während der Welle gefüllt. Wenn der Gegner denselben Zustand erreicht, wird dieser zufällige Wert angezeigt. Insgesamt funktioniert der Prozess ähnlich wie der Q-Learning-Algorithmus, jedoch ohne Belohnung. Im Gegensatz zu genetischen Algorithmen wird hier zudem keine Tabelle zwischen den Wellen gespeichert. Die Hypothese wäre, dass die Fitness während der Welle leicht erhöht ist, da die Lebensdauer in die Fitnesskalkulation eingeht, und dass die Fitness zwischen den Wellen konstant bleibt.

## Test der Q-Learning Effizienz

In der ersten Welle beginnt jeder Gegner mit einer leeren Q-Tabelle. Diese wird gemäss dem normalen Q-Learning-Verfahren (Algorithmus 1) gefüllt. Eine Übertragung von Tabellen zwischen den Wellen findet nicht statt. Eine Hypothese wäre eine rasche Fitnesserhöhung während der Welle und eine konstante Fitness zwischen den Wellen. In beiden Grafiken sollte die Fitness höher sein als die der Kontrollgruppe.

## Test der genetischen Algorithmen

In diesem Test wird die Verbesserung der Gegner durch Wellen unter Verwendung eines genetischen Algorithmus (Algorithmus 2) untersucht. Praktisch beginnt der Test wie eine Kontrollgruppe, also mit zufälligen Q-Werten. Nach der Welle werden jedoch die zwei fittesten Gegner selektiert und gekreuzt. Zudem wird es Mutationen geben, sodass Variationen entstehen. Die Hypothese lautet, dass die Fitness zunächst niedrig ist, während der Welle konstant bleibt und mit jeder Welle langsam steigt.

## Test der genetischen Q-Learning

Dieser Test wird einen früher entwickelten genetischen Q-Learning-Algorithmus (Algorithmus 3) unter die Lupe nehmen. Eine Hypothese wäre, dass die Fitness intra-wave wie beim Q-Learning rasch ansteigt und dass die Fitness mit jeder Welle langsam ansteigt. Es soll daher auch die höchste Fitness aufweisen.

## Experimentalmatrix & Parameter

Jede Konfiguration wird mehrfach mit unterschiedlichen Zufallssamen ausgeführt und statistisch ausgewertet. Insgesamt können alle Testkonfigurationen und Testvariablen in einer Experimentalmatrix 3.1 erfasst werden. Die Hyperparameter der Algorithmen sind im Anhang in Tabelle 5.1 aufgeführt.

Tabelle 3.1: Experimentalmatrix: Übersicht der getesteten Konfigurationen und Messgrößen

ID	Konfiguration	Q-Learning	GA	Anmerkung / Ziel
A	base	aus	aus	Basislinie: zufälliges Verhalten, Untergrenze
B	q_only	an	aus	Isolieren des intra-Wellen-Lernens
C	ga_only	aus	an	Effekte der GA
D	gen_q_learning	an	an	Voller Hybrid (Q + GA)

VersuchsvARIABLEN	Werte / Beschreibung
Repeats	N_rep = 5
Seed-Strategie	Je repeat ein neuer Zufallssamen; protokolliert für Reproduzierbarkeit
Waves pro Lauf	bis 10
Terminierung	Erreichen der maximalen Wellenanzahl
Logging	Pro Gegner: damage, lifespan, fitness; pro Welle: sekündliche Screenshots (mean_fitness_alive)
Bewertungsfunktion	$f(L, D, B) = 7D + 0.2L + 1B$
Anzahl Gegner pro Welle	15
Wellenzeitzlimit	30 sec

# Kapitel 4

## Ergebnisse und Diskussion

### 4.1 Ergebnisse

In diesem Abschnitt wird das resultierende Spiel mitsamt seiner Gegner-KI anhand der zuvor festgelegten Kriterien und Ziele bewertet. Alle relevanten Dateien zum Produkt, die Ergebnisse und der Update-Verlauf wurden in einem GitHub-Repository unter dem folgenden Link gespeichert: <https://github.com/Jokolto/Matura>.

#### 4.1.1 Ziel 1 und 3: Spiel und Spass

##### Eigene Bewertung

Nach stundenlangem Playtesting während der Entwicklung und dem Spielen des fertigen Produkts kann ich es wie folgt bewerten:

Das Spiel ist bestenfalls **durchschnittlich**. Dabei macht es durchaus Spass und verfügt über einige interessante Mechaniken. Aus zeitlichen Gründen konnten jedoch einige Probleme nicht gelöst werden. Die Progression ist fast vollständig dem Zufall überlassen. Zwar ist Skill wichtig, aber Glück ist viel wichtiger. (Was in Roguelikes eigentlich ziemlich oft passiert.)

Ausserdem bietet das Spiel nur eine Spielzeit von etwa 15 Minuten sowie eine geringe Variabilität bei den Items und Waffen. Die Items sind ziemlich einfach umgesetzt, meist gibt es nur eine einfache Steigerung der Werte. Dadurch ähneln sich alle Runs stark und der Spieler kann nur wenig an seiner Strategie ändern.

Trotz all dieser Probleme bin ich mit dem Spiel **zufrieden**. Für mein erstes Godot-Spiel und mein erstes veröffentlichtes Spiel, das ich innerhalb der gegebenen Frist fertiggestellt habe und das auch KI-Mechaniken beinhaltet, hatte ich nicht viel mehr erwartet.

##### itch.io-Rezeption

Die Resonanz auf itch.io war besser als erwartet: Viele Leute haben das Spiel tatsächlich gespielt, einige haben Verbesserungsvorschläge gemacht und ihre Unterstützung angeboten. Negative Rezensionen gab es kaum – das Itch.io-Auditorium scheint zu verstehen, dass es sich eher

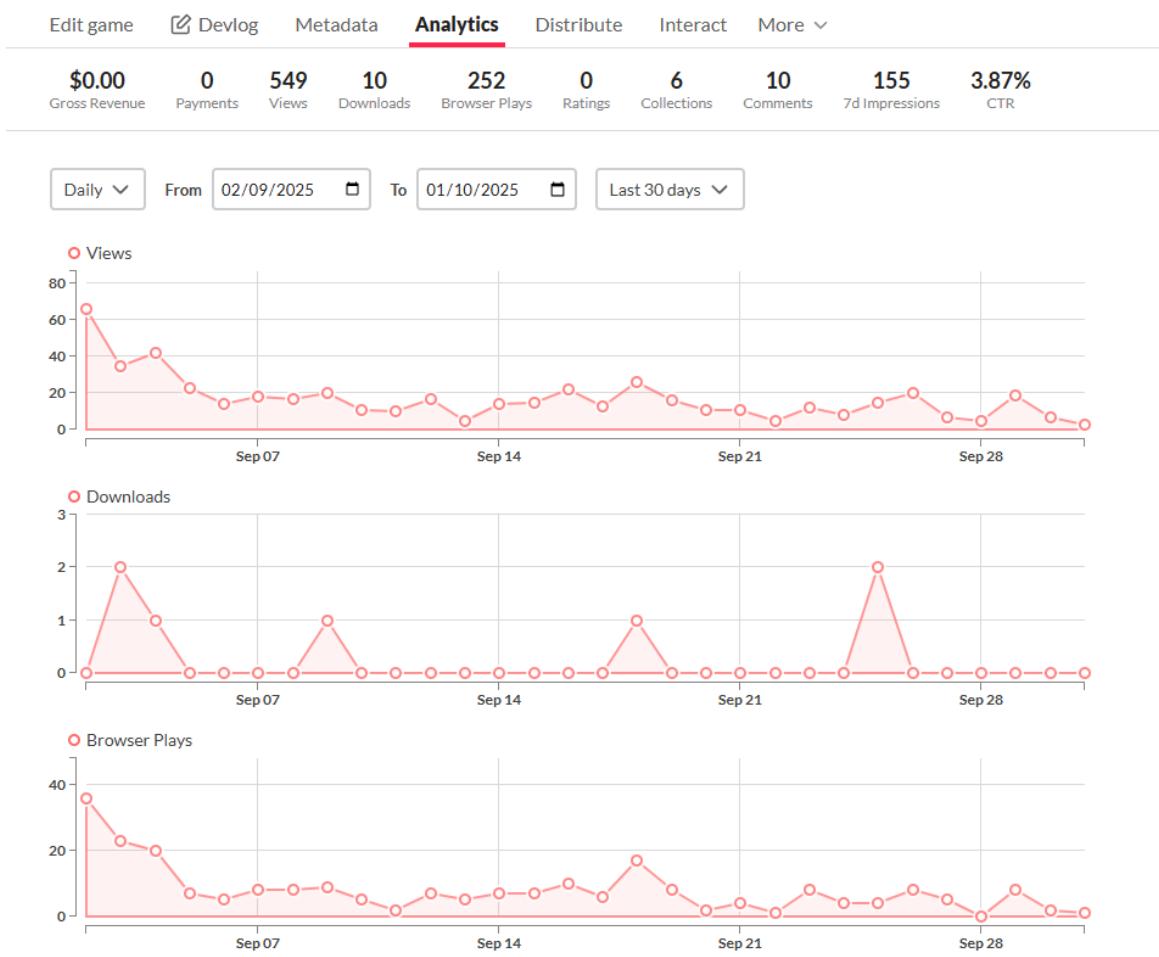


Abbildung 4.1: Analytics-Werkzeug von Itch.io

um ein Probierprojekt handelt. Genauere Zahlen zu den Spiel- und Ansehenszeiten können mit Hilfe der Itch.io-Analytics ermittelt werden. Diese sind in Abbildung 4.1 zu sehen. Die Daten eines Monats werden nach der Veröffentlichung angezeigt. Die drei wichtigsten Zahlen sind auch in Grafiken für jeden Tag eines Monats dargestellt und sind folgende:

1. Views: Wie viele Leute haben die Spielseite gesehen?
2. Browser Plays: Wie viele Leute haben das Spiel eigentlich gespielt?
3. Downloads: Wie viele Leute haben das Spiel heruntergeladen und offline gespielt?

Eine schnelle Analyse zeigt, dass das Spiel für 252 Spieler interessant genug war, um es auszuprobiert. Nur zehn Spieler haben offline gespielt, da es einfacher ist, das Spiel im Browser zu spielen. Schliesslich wurde die Spiel Seite 549 Mal aufgerufen, was auf ein relativ erfolgreiches Marketing schliessen lässt. Die anderen Werte werden nicht behandelt, da sie nicht besonders relevant sind.

Die Veröffentlichung auf Itch.io war insgesamt erfolgreich und nützlich. Dabei wurde einiges an Feedback gegeben, wodurch sich das Spiel verbessert hat. Das Spiel wird auch in Zukunft auf Itch.io unter dem Link <https://jokolto.itch.io/lcamp> verfügbar sein und so mehr Spieler erreichen.

## 4.1.2 Ziel 2: Lernende KI

### Qualitative Bewertung

In diesem Abschnitt wird qualitativ analysiert, inwiefern es den Gegnern gelungen ist, sich adaptiv zu verhalten und die im Abschnitt „Zustandsberechnung“ beschriebenen Taktiken zu erlernen. Die Analyse fand dabei nicht in der Testumgebung, sondern im veröffentlichten Spiel statt.

**Positionierung & Distanzhaltung:** Während des Spiels wird schnell klar, dass sich die Gegner anders als gewohnt bewegen. Ihre Bewegungen sind jedoch nicht zufällig, denn sie bewegen sich auf den Spieler zu und greifen ihn bereits in der ersten Welle an. Sie haben also schnell gelernt, dass sie die Distanz zum Spieler verringern müssen, um effektiv angreifen zu können. Diese Strategie bleibt für sie fast immer die beste, da sie nur so effektiv angreifen können.

Erscheinen jedoch immer mehr Feuerwaffen, muss die Strategie geändert werden. Theoretisch wäre es für Gegner mit Feuerwaffen am besten, die Distanz zu halten, da dies sicherer ist und sie nicht nah beim Spieler sein müssen, um ihm Schaden zuzufügen. Praktisch ist diese Strategie nur selten im Spiel zu sehen. Es könnte mehrere Gründe dafür geben:

1. **Schadenpriorität.** Wenn der Gegner in unmittelbarer Nähe zum Spieler schießt, kann dieser der resultierenden Kugel nicht ausweichen. Das Spiel priorisiert den verursachten Schaden gegenüber dem Überleben. Das heißt, ein Gegner, der zehnmal geschossen hat und überlebt hat, ist weniger wichtig als ein Gegner, der einmal aus nächster Nähe auf den Spieler geschossen hat, ihn dabei verletzt hat und sofort gestorben ist. Das war nicht ganz intendiert, ist aber eine effektive Taktik, wenn es viele Gegner gibt.
2. **Kugelkollision.** Die Kugel ist so programmiert, dass sie bei einer Kollision entfernt wird. Dabei ist es egal, ob sie mit einem Spieler, einer Wand oder einem Gegner kollidiert. Wenn es ein Spieler ist, wird er Schaden nehmen; wenn es ein Gegner ist, wird er einfach aus der Welt entfernt. Das kann zu Problemen führen, wenn andere Gegner (vor allem Nahkampfgegner) den Weg von Feuerwaffen versperren. So müssen Gegner mit Feuerwaffen nah herankommen, um Schaden zu verursachen.

**Kugelausweichen:** In den Zuständen der Gegner gibt es einige Parameter, die ihnen das Ausweichen von Kugeln ermöglichen sollen. Im Spiel ist dies jedoch nicht eindeutig erkennbar. Zwar weichen die Gegner die Kugel manchmal aus, diese Taktik wird jedoch nicht allgemein erlernt. Das heißt, die Gegner lernen zwar individuell, Kugeln auszuweichen, aber nur kurzfristig. Danach vergessen sie die Taktik wieder. Mögliche Gründe dafür sind:

1. **Unoptimale Belohnungen und Zustände:** Der Zustandsraum und die Belohnungen sind möglicherweise nicht optimal für das Erlernen von Ausweichmechanismen geeignet.
2. **Schwache Ausweichbedingung:** In der letzten Arbeitswoche wurde schnell eine Ausweichbelohnung eingeführt und dafür auch eine Bedingung festgelegt. Diese scheint zu funktionieren, wurde jedoch nur wenig getestet.

Insgesamt ist ein gewisses Adaptivitätsverhalten erkennbar, jedoch nicht in dem gewünschten Ausmass. Einige Ausweichmanöver und Positionierungsänderungen finden zwar nicht immer statt, existieren aber bereits.

## Quantitative Bewertung

In diesem Abschnitt werden die quantitativen Ergebnisse der im Abschnitt „KI-Bewertung“ beschriebenen Tests präsentiert und erörtert. Zur besseren Verständlichkeit wurden die resultierenden CSV-Tabellen in Liniengraphen visualisiert. Alle Parameter und Konfigurationsbedeutungen wurden im selben Abschnitt definiert. Zunächst wird die Fitness-Intra-Welle analysiert.

**Intra-Welle-Lernen:** Da das Intrawellenlernen nur im Q-Lernen stattfindet, wird nur dieses mit der Kontrollgruppe verglichen. Dieser Vergleich ist in Abbildung 4.2 zu sehen.

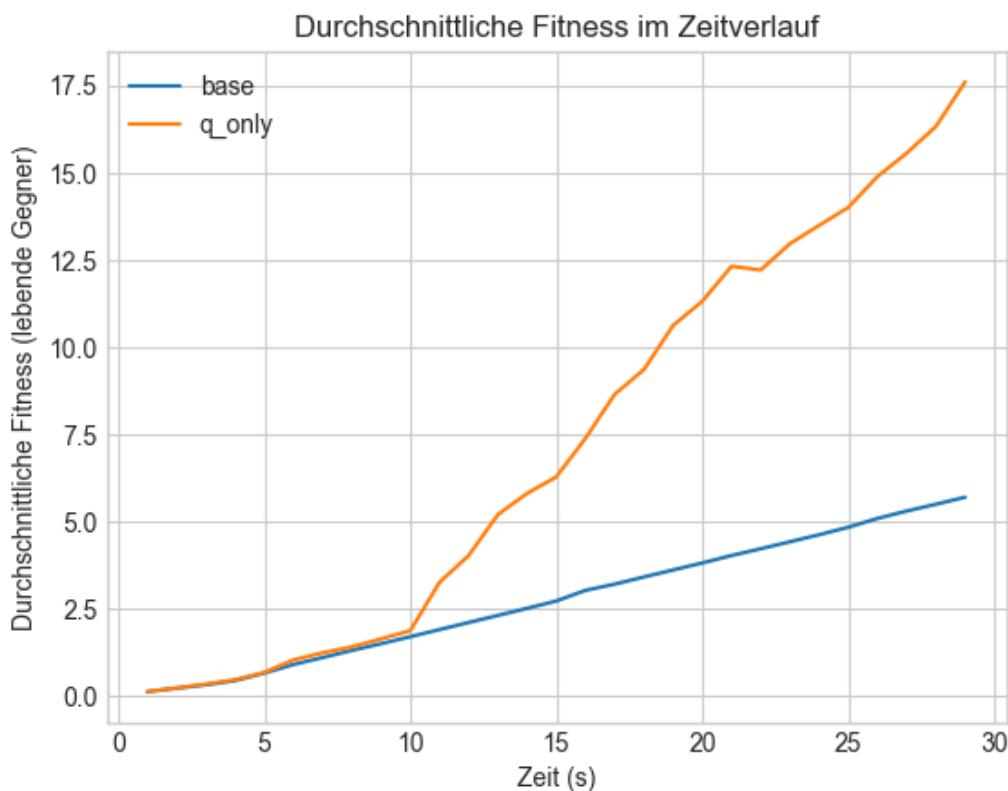


Abbildung 4.2: Durchschnittliche Fitness im Zeitverlauf

Wie in der Abbildung zu sehen ist, beginnen beide Konfigurationen gleich, nach zehn Sekunden jedoch nicht mehr. Während die Q-Learning-Konfiguration auf eine rasche Fitnesssteigerung hindeutet, ist bei der Kontrollgruppe eine langsame, lineare Steigerung zu beobachten. Dies lässt sich wie folgt erklären: Die adaptiven Gegner spawnen, lernen schnell, die Distanz zum Spieler zu reduzieren, und greifen ihn dann an. Die Angriffsphase scheint bei 10 s zu beginnen, was die schnelle Fitnesserhöhung erklärt. Die Gegner der Kontrollgruppe lernen hingegen nicht, sich dem Spieler anzunähern. Das bedeutet, dass ihre Fitness nur durch ihre Lebensdauer erhöht wird. Da die Lebensdauererhöhung komplett linear ist, ist auch die Fitnesserhöhung linear.

Insgesamt steht fest, dass Q-Lernen tatsächlich funktioniert und die Fitness der Gegner

sich dadurch schnell verbessert. Als Nächstes soll die Fitness zwischen den Wellen analysiert werden.

**Zwischenwelle-Lernen:** Hier werden alle Konfigurationen miteinander verglichen. Der Vergleich ist in Abbildung 4.3 dargestellt.



Abbildung 4.3: Durchschnittliche Fitness pro Welle

Wie die Abbildung zeigt, sind die Fitnesswerte sowohl bei der Kontrollgruppe als auch beim genetischen Algorithmus nahezu konstant niedrig. Während der Trend der Kontrollgruppe wie erwartet ausfiel, war dies beim genetischen Algorithmus nicht der Fall. Es wurde eine kleinere Steigung erwartet, da nur die besten Gegner jeder Welle selektiert wurden. Eine mögliche Erklärung dafür ist, dass die Anzahl der Wellen mit zehn niedrig ist. Genetische Algorithmen – wie auch die Evolution in der realen Welt – benötigen sehr viele Generationen, um effektiv zu werden. Weitere Experimente mit einer grösseren Anzahl von Wellen wären hier hilfreich.

Die beiden Algorithmen mit Q-Learning weisen wie erwartet eine relativ hohe Fitness auf. Bei Q-Learning kommt es zu grossen Schwankungen, da die Gegner ihre Q-Tabellen immer neu auffüllen müssen. In diesem Prozess gibt es Zufallselemente (Exploration, wenn kein Zustand gegeben ist), die immer zu einem anderen Fitnessniveau führen. Genetisches Q-Learning scheint stabiler zu sein, da die zuerst gelernten Taktiken zwischen den Wellen erhalten bleiben. Es ist jedoch keine Steigerung der Fitness mit jeder Welle zu erwarten. Dies ist jedoch leicht erklärbar, wenn der genetische Algorithmus selbst nicht richtig funktioniert.

Insgesamt ist klar, dass Q-Learning eine gute Lernkurve aufweist, während der genetische Algorithmus entweder sehr langsam oder fehlerhaft arbeitet. Um die Effektivität des GA zu

steigern, sind eindeutig mehr Experimente erforderlich.

## 4.2 Künftige Arbeit

Die Ergebnisse zeigen, dass es noch viel zu tun gibt. Um einen besseren Überblick zu erhalten, wird die künftige Arbeit in zwei Kategorien unterteilt. Für jede Kategorie ist ein eigener Abschnitt vorgesehen.

### 4.2.1 Mehr Experimente und KI-Verbesserungen

Natürlich gibt es aber auch bei der gegnerischen KI viel zu verbessern. Und diese Verbesserung kann nur durch mehr und bessere Experimente erreicht werden.

#### Experimente

Das Spiel bietet grosses Experimentierpotenzial, insbesondere im Bereich GA. Im Vergleich zu anderen Arbeitsbereichen ist die für diese Arbeit aufgewendete Zeit für die Experimente relativ gering. Die Experimente wurden rasch und unkompliziert durchgeführt. Dabei wurde viel Zeit in die Automatisierung und die Erstellung der Controller investiert. Mit den fertigen Controllern kann man jeden Parameter der genutzten Algorithmen optimieren und auch neue Algorithmen ausprobieren. Die Tatsache, dass der Server in Python geschrieben ist, hilft dabei enorm. Mit Bibliotheken wie Pytorch ist der nächste mögliche Schritt, neuronale Netze auszuprobieren.

#### Hyperparameter-Optimierung

Wie das Ergebnis zeigt, sind die Hyperparameter nicht optimal eingestellt. Es ist nahezu unmöglich, alle Parameter perfekt einzustellen. Es sind schlicht zu viele, um sie jedes Mal einzeln zu kontrollieren – insbesondere, da sie sich gegenseitig beeinflussen und gemeinsam geändert werden müssen. Die Suche nach den optimalen Parametern kann jedoch automatisiert werden. Dazu kann eine systematische Suche über ein vordefiniertes Gitter von Hyperparameter-Kombinationen oder eine GA mit Hyperparameter-Kombinationen-Population durchgeführt werden. Man könnte auch die Variation der Hyperparameter während des Spiels hinzufügen, beispielsweise die Mutationsrate mit jeder Welle reduzieren, sodass die Gegner früher mutierte Taktiken beibehalten. Die Möglichkeiten sind unbegrenzt.

### 4.2.2 Gameplay-Verbesserungen

Das Gameplay hat grosses Verbesserungspotenzial. Einige Ideen sind:

#### Weniger Zufall, mehr Können

In dem aktuellen Spiel dreht sich alles vor allem um Waffen, die von Gegnern fallengelassen werden, sowie um Items, die nach den einzelnen Wellen erscheinen. All das ist zufällig. In den

Wellen selbst kann der Spieler sein Können nicht wirklich unter Beweis stellen. Gefragt sind lediglich gutes Timing beim Angriff, das Ausweichen von Gegnerprojektilen, gutes Aim, wenn der Spieler Feuerwaffen hat, und die richtige Wahl der Items. Diese Mechaniken sind jedoch sehr schnell zu meistern, wodurch sie mit der Zeit langweilig werden. Um dies zu ändern, können neue Mechaniken eingeführt werden. Ein Beispiel hierfür sind aktive Fähigkeiten, die nach einer Abklingzeit erneut genutzt werden können.

### Mehr Items, mehr Waffen, mehr Stats

Das Spiel ist als Roguelike konzipiert. Das bedeutet, dass es sehr variantenreich sein soll. Das jetzige Spiel verfügt über acht Waffenvarianten, was eigentlich nicht wenig ist. Ein mehr problematischer Aspekt des Spiels sind die Items. Es gibt nicht genug davon und sie dienen hauptsächlich der Steigerung der Werte. Dadurch wird die Strategie sehr begrenzt, was weniger Spass macht. Darüber hinaus gibt es nicht viele Synergien, die den strategischen Aspekt des Spiels interessant machen würden. Die Lösung wäre, einige der jetzigen Items beizubehalten – einige Statssteigerungs-Items sind notwendig – und die anderen interessanter zu gestalten. Das Spiel wäre mit speziellen neuen Mechaniken wie der Auto-Aim-Kugel oder der Grössenerhöhung der Knockback-Wirkung bei Nahkampfwaffen bestimmt interessanter geworden.

## 4.3 Fazit & Reflexion

### Fazit

In dieser Arbeit wurde ein 2D-Topdown-Roguelike entwickelt, in dem die Gegner mithilfe von Q-Learning und genetischen Algorithmen lernen. Die Ziele waren, ein interessantes Spiel zu kreieren, die Gegner adaptiv zu gestalten und die resultierende KI anschliessend zu bewerten. Diese beiden Ziele wurden in unterschiedlichem Ausmass erreicht. Das entwickelte Spiel wurde auf Itch.io gut aufgenommen. Die KI der Gegner war jedoch nicht optimal. Während der Q-Learning-Algorithmus eine gute Lernkurve zeigte, war dies beim genetischen Algorithmus nicht der Fall. Dies ist vermutlich auf die unoptimalen Hyperparameter und die unzureichende Anzahl an Wellen zurückzuführen. Die Verbesserung dieser Algorithmen sowie des allgemeinen Gameplays ist Gegenstand der zukünftigen Arbeit.

### Reflexion

Dieses Projekt war das bisher grösste, an dem ich programmiertechnisch gearbeitet habe. Mit rund 3.900 Codezeilen in 64 verschiedenen Dateien und zwei Programmiersprachen war es sehr anspruchsvoll. Dabei musste ich schnell viele verschiedene Rollen übernehmen, unter anderem die des Spielentwicklers, des KI-Programmierers und des Datenanalytikers. An einigen Stellen kam es daher zu Fehlern. Vor allem die Bereiche Planung, Zeitmanagement und Priorisierung haben Verbesserungspotenzial. Insgesamt ist Folgendes zu sagen: Die Maturaarbeit war spannend. Sie hat mir viel beigebracht.

# Kapitel 5

## Anhang

### 5.1 Wichtige Codeausschnitte

Code 5.1: Aktionen Definition

```
1 func execute_action(action: String):
2     dir = (player.global_position - global_position).normalized()
3     var shooting_dir = player.global_position
4     last_action = action
5     match action:
6         "move_forward":
7             velocity = dir * move_speed
8             add_reward_event(GlobalConfig.RewardEvents["MOVED_CLOSER"])
9         "retreat":
10            velocity = -dir * move_speed
11            add_reward_event(GlobalConfig.RewardEvents["RETREATED"])
12        "strafe_left":
13            velocity = dir.rotated(-PI/2) * move_speed
14            add_reward_event(GlobalConfig.RewardEvents["WASTED_MOVEMENT"])
15        "strafe_right":
16            velocity = dir.rotated(PI/2) * move_speed
17            add_reward_event(GlobalConfig.RewardEvents["WASTED_MOVEMENT"])
18        "use_weapon":
19            if weapon_instance and weapon_instance.is_ready():
20                weapon_instance.use_weapon(shooting_dir)
21                weapon_instance.store_state(current_state, action)
22                # gets its reward from bullet if it hits player
23            _:
24                velocity = Vector2.ZERO
25            move_and_slide()
```

Code 5.2: Die wichtigste Methode aus der Klasse „Q-Learner”.

```

1
2     def get_q_value(self, state: str, action: str) -> float:
3         if state not in self.q_table:
4             self.q_table[state] = {}
5         if action not in self.q_table[state]:
6             self.q_table[state][action] = 0.0
7         return self.q_table[state][action]
8
9     def apply_reward(self, reward, new_state, action_to_reward, state_to_reward):
10        if action_to_reward is None and state_to_reward is None:
11            if not self.pending_actions:
12                logging.debug("No pending actions to apply reward to.", self.
13 pending_actions)
14                return # Nothing to apply reward to
15            state, action = self.pending_actions.popleft()
16            old_value = self.get_q_value(state, action)
17            state, action = (state_to_reward, action_to_reward)
18            old_value = self.get_q_value(state, action)
19            # Calculate max future Q-value for the new state
20            if new_state in self.q_table and self.q_table[new_state]:
21                max_future_q = max(self.q_table[new_state].values())
22            else:
23                max_future_q = 0.0
24            new_value = old_value + self.learning_rate * (reward + self.discount_factor
25 * max_future_q - old_value) # Update Q-value according to Q-learning formula
26            self.q_table[state][action] = new_value
27
28    def choose_action(self, state: str, valid_actions: list[str], epsilon: float =
29                      0.1, random_q: bool = False) -> str: # last parameter is true in base and
30      ga_only configs
31      self.last_state = state
32      if random_q:
33          epsilon = 0.0
34          if state not in self.q_table:
35              self.q_table[state] = {action: random.uniform(-1, 1) for action in
36 valid_actions} # random q values for actions that persists for agents
37          # Explore if epsilon hits or state is unknown
38          if random.random() < epsilon or state not in self.q_table or not self.
39 q_table[state]:
40              action = random.choice(valid_actions)
41          else:
42              action = max(self.q_table[state], key=self.q_table[state].get)
43          # Store for later reward
44          self.pending_actions.append((state, action))
45          if len(self.pending_actions) > MAX_PENDING:
46              self.pending_actions.popleft()
47      return action
48

```

Code 5.3: Die wichtigste Methode für GA aus der Klasse „Shared-Q-Learner“

```
1 def per_state_crossover(self, parents, mutation_prob=0.05, mutation_range=0.1):
2     ''' new crossover, where top candidates individual with highest fitness
3         produce new q table. For each state random parents q values are taken '''
4     # Collect all states across all candidates
5     all_states = set()
6     for candidate in parents:
7         all_states.update(candidate.q_table.keys())
8
9     for state in all_states:
10        # Candidates that have this state
11        candidates_with_state = [c for c in parents if state in c.q_table]
12        # Pick one randomly
13        parent = random.choice(candidates_with_state)
14        self.q_table[state] = parent.q_table[state].copy()
15
16        # Apply mutation
17        for action, value in self.q_table[state].items():
18            if random.random() < mutation_prob:
19                # Add or subtract a small random number
20                self.q_table[state][action] += random.uniform(-mutation_range,
mutation_range)
21
```

## 5.2 Python Server und Sockets

In der Praxis wurde entschieden für Experimente, die Q-Tabelle, die Aktionswahl, die Belohnung und die Fitnesswahl zu isolieren und separat in Python ausserhalb von Godot zu realisieren. Für Kommunikation zwischen den Godot und Python Server wird TCP Socket genutzt. Wegen der Nutzung der externen zum Godot Umgebung, wird der Projekt und Spiel komplexer. Das macht jedoch mehr Experimenten möglich, Python verfügt über zahlreiche KI- und Plotting-Bibliotheken, darunter Pytorch und matplotlib.

Der Prozess des Spieles wäre dann folgender:

Server und dann der Spiel startet.

Zustandsberechnung mit resultierenden Dictionay  $enemy_id : zustand$  (Godot)

Dictionay wird zu JSON und dann durch TCP Socket zu Server geschickt (Godot)

Server bekommt den JSON und dekodiert es (Python Server)

Der Server berechnet die Aktionen aller Gegner, verpackt sie in ein JSON (Python Server) und schickt es zurück

Godot erhält den JSON-Text und dekodiert ihn. Die Gegner führen die dekodierte Aktion aus. (Godot)

Godot erstellt noch eine JSON-Datei zur Belohnung der ausgeführten Aktionen. Dieses enthält Gegner-ID, Aktion zur Belohnung, aktuellen Zustand, neuen Zustand und Belohnungstyp.

Der Server erhält die Belohnungen und wendet sie auf jede Gegner-Q-Tabelle an.

### 5.2.1 Spielarchitektur

Im Allgemeinen lässt sich die Spielarchitektur in einer Abbildung 5.1 darstellen.

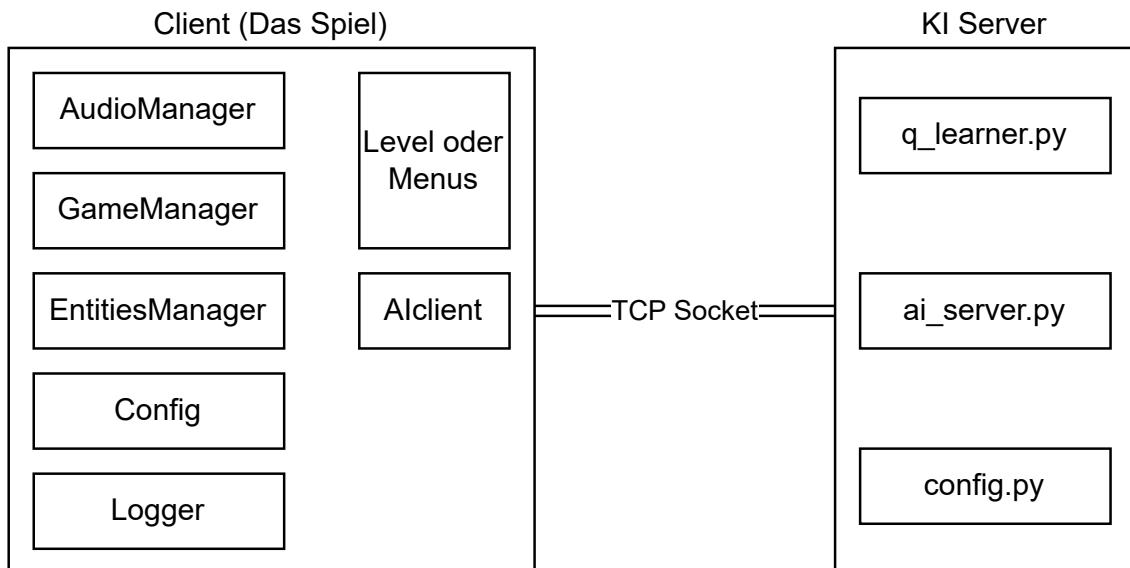


Abbildung 5.1: Grobe Architektur des Spiels

## 5.3 Hyperparameter

Tabelle 5.1: Verwendete Hyperparameter in den Experimenten.

Kategorie	Parameter	Wert / Beschreibung
Q-Learning	Lernrate ( $\alpha$ )	0,2
	Diskontierungsfaktor ( $\gamma$ )	0,9
	Explorationsrate ( $\epsilon$ )	0,2
Belohnungen	Treffer auf Spieler	+10,0
	Eigener Schaden	-5,0
	Überlebenszeit	+0,0 pro Sekunde
	Tod	-5,0
	Bewegung ohne Nutzen	-0,05
	Rückzug	-0,2
	Fehlgeschlagener Schuss	-0,2
	Bewegung in Richtung Spieler	+0,05
	Feststecken	-1,0
	Ausweichmanöver	+6,0
GA	Populationsgrösse	15 (pro Welle)
	Auswahlstrategie	Die zwei besten Individuen werden für die Reproduktion ausgewählt
	Kreuzung	Nachkomme entsteht durch Mittelwertbildung der Q-Werte beider Eltern
	Mutationsrate	5 % pro Q-Wert
	Mutationsänderung	Zufälliger Wert zwischen -1,0 und +1,0

# Literaturverzeichnis

- [1] Creative Assembly. Alien: Isolation on steam, 2014. Zugriff am 8. August 2025. URL: [https://store.steampowered.com/app/214490/Alien\\_Isolation/](https://store.steampowered.com/app/214490/Alien_Isolation/).
- [2] Blocktunix. Adaptive ai in video games: Shaping the future, 2024. URL: <https://blocktunix.com/adaptive-ai-in-video-games-shaping-future/>.
- [3] Epic Games. *Unreal Engine*. Epic Games, 2025. Version 5.x. URL: <https://www.unrealengine.com/>.
- [4] Godot Engine contributors. *Godot Engine*. Godot Engine, 2025. Version 4.x. URL: <https://godotengine.org/>.
- [5] Mohammad Hossain, Md. Hasan, and Mahmudur Rahman. Analyzing strengths and weaknesses of modern game engines. *International Journal of Computer Theory and Engineering (IJCTE)*, 15(1):54–60, 2023. URL: <https://www.ijcte.org/vol15/IJCTE-V15N1-1330.pdf>.
- [6] Kazuhisa Kanagawa and Takuya Kaneko. Rogue-gym: A new challenge for generalization in reinforcement learning. *IEEE Conference on Games (CoG)*, 2019. URL: [https://www.researchgate.net/publication/336101419\\_Rogue-Gym\\_A\\_New\\_Challenge\\_for-Generalization\\_in\\_Reinforcement\\_Learning](https://www.researchgate.net/publication/336101419_Rogue-Gym_A_New_Challenge_for-Generalization_in_Reinforcement_Learning), doi:10.1109/CIG.2019.8848075.
- [7] poncle. Vampire survivors on steam, 2022. Zugriff am 5. August 2025. URL: [https://store.steampowered.com/app/1794680/Vampire\\_Survivors/](https://store.steampowered.com/app/1794680/Vampire_Survivors/).
- [8] Monolith Productions. Middle-earth: Shadow of mordor on steam, 2014. Zugriff am 8. August 2025. URL: [https://store.steampowered.com/app/241930/Middleearth\\_Shadow\\_of\\_Mordor/](https://store.steampowered.com/app/241930/Middleearth_Shadow_of_Mordor/).
- [9] Roguelike Development Conference. Berlin interpretation of roguelike games. [https://www.roguebasin.com/index.php/Berlin\\_Interpretation](https://www.roguebasin.com/index.php/Berlin_Interpretation), 2008. Roguelike Development Conference 2008, abgerufen am 12. August 2025.
- [10] Dodge Roll. Enter the gungeon on steam, 2016. Zugriff am 5. August 2025. URL: [https://store.steampowered.com/app/311690/Enter\\_the\\_Gungeon/](https://store.steampowered.com/app/311690/Enter_the_Gungeon/).
- [11] Unity Technologies. *Unity Real-Time Development Platform*. Unity Technologies, 2025. Version 2022.3 LTS. URL: <https://unity.com/>.

- [12] Georgios N Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018. doi:10.1007/978-3-319-63519-4.

## Redlichkeitserklärung

Ich, Oleh Chekhovych, 4B,

bestätige mit meiner Unterschrift, dass die eingereichte Arbeit selbstständig und ohne unerlaubte Hilfe Dritter verfasst wurde. Die Auseinandersetzung mit dem Thema erfolgte ausschliesslich durch meine persönliche Arbeit und Recherche. Es wurden keine unerlaubten Hilfsmittel benutzt.

Ich bestätige, dass ich sämtliche verwendeten Quellen sowie Informanten/-innen im Quellenverzeichnis bzw. an anderer da-für vorgesehener Stelle vollständig aufgeführt habe. Alle Zitate und Paraphrasen (indirekte Zitate) wurden gekennzeichnet und belegt. Sofern ich Informationen von einem KI-System wie bspw. ChatGPT verwendet habe, habe ich diese in meiner Maturaarbeit gemäss den Vorgaben im Leitfaden zur Maturaarbeit korrekt als solche gekennzeichnet, einschliesslich der Art und Weise, wie und mit welchen Fragen die KI verwendet wurde.

Ich bestätige, dass das ausgedruckte Exemplar der Maturaarbeit identisch mit der digitalen Version ist.

Ich bin mir bewusst, dass die ganze Arbeit oder Teile davon mittels geeigneter Software zur Erkennung von Plagiaten oder KI-Textstellen einer Kontrolle unterzogen werden können.

Datum..... Unterschrift.....