

Titel der Maturaarbeit

Zusammenfassung

Kurzfassung der Arbeit.

Inhaltsverzeichnis

1 Einleitung	3
1.1 Motivation	3
1.2 Zielsetzung	3
2 Theoretischer Hintergrund	4
2.1 Roguelike Spiele	4
2.1.1 Charakterisierung	4
2.1.2 Untergenre von Roguelike und Inspiration	5
2.2 Gegnerische KI	6
2.2.1 Übliche KI-Implementierung - Hardcoded KI	7
2.2.2 Adaptive KI	7
2.3 Adaptive KI-Algorithmen	8
2.3.1 Reinforcement Learning, Q-Learning	9
2.3.2 Genetische Algorithmen	10
2.3.3 Neuronale Netze	11
2.4 Spiel-Engine	12
2.4.1 Unity	12
2.4.2 Godot	12
2.4.3 Unreal Engine	13
2.4.4 Ohne Spiel-Engine/Pygame	13
3 Methodik und Implementierung	14
3.1 Auswahl der Spiel-Engine	14
3.2 Spiel Design	15
3.2.1 Untergenre	15
3.2.2 Ziel und Setting des Spieles	15
3.2.3 Implementierung	15
3.3 KI-Design	19
3.3.1 Grundlagen zum Gegner	19
3.3.2 KI-Algorithmus	20
3.3.3 Implementierung	22
3.4 Bewertung	24
3.4.1 Bewertung des Spassfaktors des Spiels	24
3.4.2 Bewertung der KI	25

4	Ergebnisse und Diskussion	26
4.1	Ergebnisse	26
4.1.1	Ziel 1 und 3: Spiel und Spass	26
4.1.2	Ziel 2: Lernende KI	26
4.2	Fazit	26
4.3	Künftige Arbeit	26
4.4	Reflexion	26
5	Zusatz	27

Kapitel 1

Einleitung

1.1 Motivation

Die Wahl des Themas beruhte auf meinem Interesse an KI-Technologien, einem Bereich der Programmierung mit kürzlichen Durchbrüchen. Allerdings braucht jede Technologie auch einen Nutzen. Diesen Nutzen habe ich in meinen Spielerfahrungen gefunden - die Gegner in einem Spiel, die aus den Handlungen der Spieler lernen können. Die Programmiererfahrungen und der Wunsch waren die letzte Motivation, die ich brauchte, um dieses Projekt umzusetzen.

1.2 Zielsetzung

Ziel ist es also, ein Spiel zu programmieren, bei dem die Gegner aus den Handlungen des Spielers lernen und so die Wiederspielbarkeit des Spiels verbessern. Das Genre des Spiels wird dann passenderweise als Roguelike gewählt, ein Genre, in dem Wiederspielbarkeit an erster Stelle steht. Die KI sollte so trainiert werden, dass sie sich im Spiel bemerkbar macht, also zu einer der Hauptmechaniken des Spiels wird. Folgende Hauptziele sollen also erreicht werden:

- Ein Basisspiel zu erstellen, 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

2.1 Roguelike Spiele

2.1.1 Charakterisierung

Roguelike-Spiele zeichnen sich durch grosse Variabilität aus, meist durch prozedural generierte Inhalte. Das Spielerlebnis ist dadurch bei jedem Durchlauf anders. Diese Dynamik führt dazu, dass man das Spiel über lange Zeit spielen kann – bis entweder die Variabilität ausgeschöpft ist oder der Spieler die Spielbalance als unbefriedigend empfindet. Die Spielschleife sieht oft folgendermassen aus:

Der Spieler beginnt den Durchlauf, er ist sehr schwach



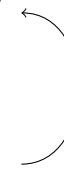
Der Spieler steht vor einer Herausforderung (meist Gegner), die er überwinden muss.



Der Spieler bekommt eine Belohnung zur Auswahl, die ihn stärker macht.



Der Spieler trifft auf eine höhere Herausforderung, wobei die Belohnung ihm auf einzigartige Weise hilft



Es wiederholt sich viele Male, immer mit einer gewissen Variabilität



Der Spieler trifft auf die letzte Herausforderung, die testet, wie gut der Aufbau und die erlernten Fähigkeiten des Spielers sind.

Normalerweise wird der letzte Schritt - das Gewinnen - zunächst nicht erreicht. Erst durch mehrere Durchläufe lernt der Spieler das Spiel, die Gegner und die Belohnungsmechanismen kennen, so dass er schliesslich gewinnen kann. Danach wird der Spieler durch die Variabilität dazu verleitet, weiterzuspielen und erneut zu gewinnen. Wie die Variabilität in meinem Spiel umgesetzt wird, wird in Methodik erklärt.

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.

Vampire Survivors

Vampire Survivors (1) ist ein Roguelite, 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.



Abbildung 2.1: Screenshot aus *Vampire Survivors*, Quelle: Steam-Seite (1)

Slay the spire

Slay the spire (2) ist ein Roguelite, bei dem die Entscheidungen der Spieler das Wichtigste sind. Es handelt sich um ein rundenbasiertes Deckbau-Spiel. Spieler und Gegner sind abwechselnd an der Reihe, und nach einem Sieg wählt der Spieler eine Karte für sein Deck aus. Auch erhält er oft Gegenstände, die mit bestimmten Karten synergieren. In diesem Spiel müssen die Spieler stets planen und sich anpassen, um einen guten Deckaufbau zu erzielen. Das Freischalten neuer Charaktere, Karten und Gegenstände erhöht die Variabilität.



Abbildung 2.2: Screenshot aus *Slay the spire*, Eigene Aufnahme

Enter the Gungeon

Enter the Gungeon (3) Enter the Gungeon ist ein Top-Down-Shooter-Roguelite, 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. Diese Liste ist natürlich sehr begrenzt



Abbildung 2.3: Screenshot aus *Enter the Gungeon*, Eigene Aufnahme

und zeigt hauptsächlich die Spiele, die mein Produkt inspiriert haben.

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

2.2.2 Adaptive KI

need citation, for adaptive ki defintion <https://blocktunix.com/adaptive-ai-in-video-games-shaping-future/> maybe

Adaptive KI in Spielen

Im Gegensatz zu fest programmierten 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 (4) 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.



Abbildung 2.5: Screenshot aus *Alien: Isolation*, Quelle: Steam-Seite (4)

F.E.A.R

F.E.A.R (5) ist ein Ego-Shooter, der Horrorelemente und intelligentes Gegnerverhalten kombiniert und so taktische und fesselnde Kämpfe ermöglicht. KI-Einsatz: Die gegnerischen Soldaten nutzen zielorientierte Aktionsplanung (GOAP), um basierend auf der Position des Spielers und der Umgebung dynamisch intelligente Verhaltensweisen (Flankieren, Rückzug, Deckung suchen) zu wählen.

Middle-earth: Shadow of Mordor

Middle-earth: Shadow of Mordor (6) 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.

2.3 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



Abbildung 2.6: Screenshot aus *F.E.A.R.*, Quelle: Steam-Seite ([5](#))



Abbildung 2.7: Screenshot aus *Middle-earth: Shadow of Mordor*, Quelle: Steam-Seite ([6](#))

wird im Abschnitt „Implementierung“ behandelt. Viele der Informationen zu Algorithmen sind aus dem Buch „AI and Games“ von Yannakakis und Togelius ([7](#)) 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 . ([7](#), 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.

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

Der Algorithmus lässt sich im Allgemeinen anhand der folgenden Vorlage zusammenfassen (7, 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. ε -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. (7, S. 52)

Der Algorithmus lässt sich im Allgemeinen anhand der folgenden Vorlage zusammenfassen (7, 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 durchlauften Generationen oder Bewertungen (**Erschöpfung**), der höchsten von einer Lösung erreichten Fitness (**Erfolg**) und/oder einer anderen Abbruchbedingung.
8. Zurück zu Schritt 2.

2.3.3 Neuronale Netze

Ein neuronales Netzwerk ist ein Algorithmus, der wie andere Algorithmen von der Natur inspiriert ist und die Neuronen im Gehirn nachahmt. Sie können Muster in grossen und komplexen nicht-diskreten Datensätzen finden. Sie haben viele Anwendungsbereiche, wie beispielsweise Bilderkennung, Sprachverarbeitung und manchmal auch KI in Spielen. Damit ein neuronales Netzwerk gut funktioniert, muss es oft mit vielen Daten trainiert werden. Da die neuronalen Netze in diesem Artikel in erster Linie zu Vergleichszwecken verwendet werden, werden sie hier nicht näher erläutert. Eine ausführlichere Erklärung findet sich jedoch in „AI and Games“ von Yannakakis und Togelius (7, S. 59-65).

2.4 Spiel-Engine

Um ein Spiel schnell entwickeln zu können, verwenden sie oft eine sogenannte Spiel-Engine. Eine Spiel-Engine ist ein System mit einer Reihe vorprogrammierter Tools, die die Entwicklung schneller und einfacher machen. Das bedeutet in der Regel, dass man weniger Freiheit bei der Programmierung hat, aber das ist meist nur bei grossen oder hochspezialisierten Spielen der Fall. In diesem Abschnitt wird die beliebtesten Engines und Entwicklungsmethoden angeschaut, und im Abschnitt zur Methodik wird eine Methode ausgewählt.

2.4.1 Unity

Unity (8) 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.

- **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
- Lizenzänderungen haben in der Community für Unsicherheit gesorgt

2.4.2 Godot

Godot (9) 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.

- **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 (10) 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.

- **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 Programmfreiheit, 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. Unreal Engine ist hier sicherlich ein Overkill, Godot hingegen ist perfekt geeignet.
- Es wird entschieden, dass das Spiel 2D sein wird (siehe nächste Sektion). Godot ist dafür bekannt, dass man einfach und schnell 2D Spiele machen kann.
- Das Spiel hat einen experimentellen Charakter, daher wäre es gut, wenn Sie den Quellcode der Engine einsehen und bei Bedarf ändern könnten.
- 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.

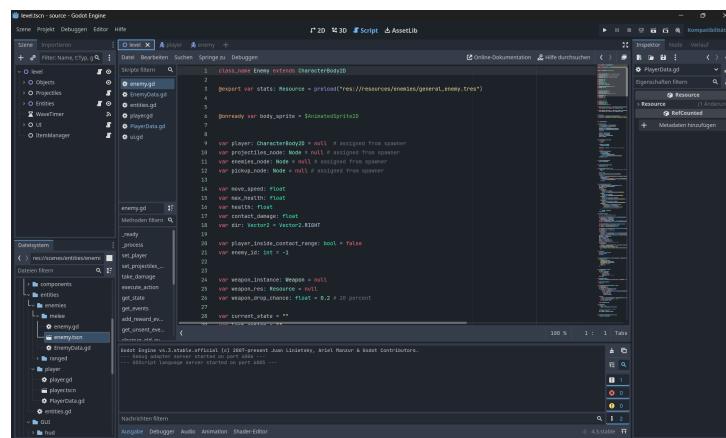


Abbildung 3.1: Godot Umgebung

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 einfachere Umgebung für Gegner (Bewegung mit nur 2 Koordinaten statt 3). Roguelikes (Roguelites) 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 Spieles

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 anfangs etwas dumm, lernen aber mit der Zeit dazu.

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 3.2 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 3.3 ist das genutztes Sprite-Sheet 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 cooldowns verantwortlich: eine messt wenn kann der Spieler „Dash“ ausführen, die andere wenn der Spieler noch mal verletzt sein kann.

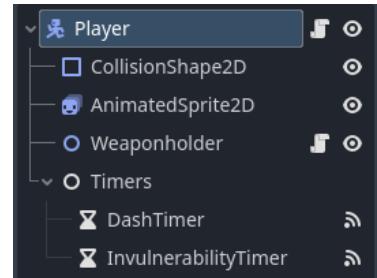


Abbildung 3.2: Die Spieler-Knoten-Struktur



Abbildung 3.3: Die Spieler-Idle-Animationsframes

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 den Wänden, Spawners und Toren. In Abbildung 3.5 ist die letzte Version dargestellt.

In Godot habe ich die für die Arena verantwortliche Knoten „**Objekte**“ genannt. Die Struktur ist in der Abbildung 3.4 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,

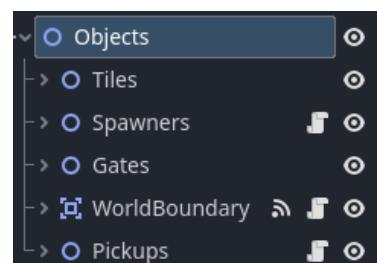


Abbildung 3.4: Die Arena-Knoten-Struktur

Pflanzen, Schatten, Requisiten und unsichtbare Kollisionsobjekte. 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 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.

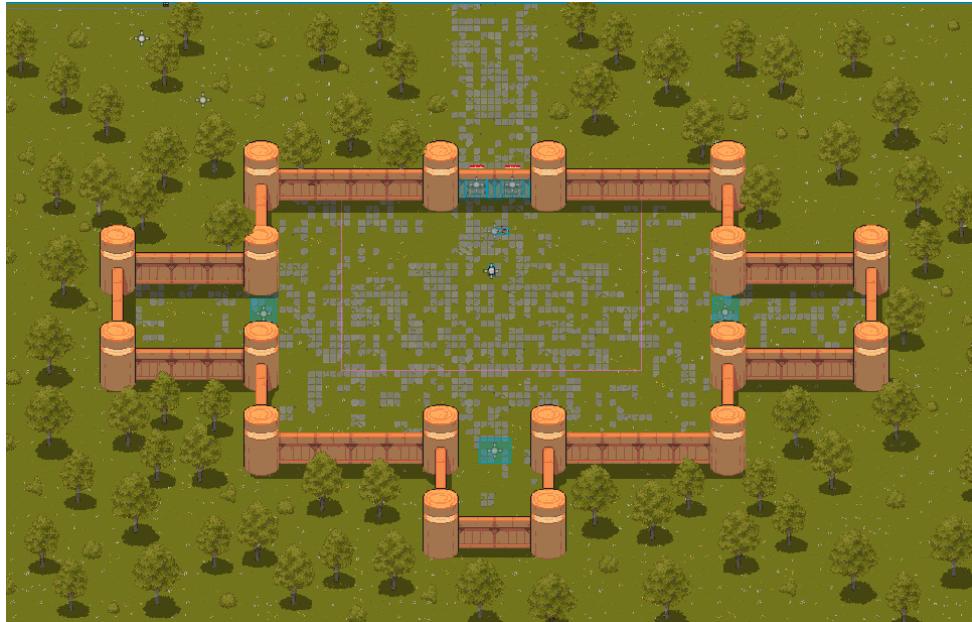


Abbildung 3.5: Die Arena

Toren

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. 70–1 Sekunde–90). 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 Spawnen 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: **MeleeWeapon** und **RangedWeapon**. Die Namen der Klassen machen den Unterschied zwischen den beiden Klassen ziemlich deutlich: Die Klasse MeleeWeapon erzeugt Nahkampfwaffen (Beispiel: Schwert, siehe Abbildung ??) a), während die Klasse RangedWeapon Projektiler erzeugt, die Schaden verursachen (Beispiel: SMG, siehe Abbildung ??) 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 3.6: 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 3.7 dargestellt. Die Knoten „Projektile“ und „Entitäten“ enthalten die Projektile 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.

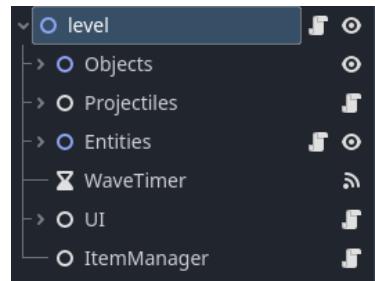


Abbildung 3.7: Die Level-Knoten-Strukture

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 3.8). 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. In der neuesten Version des Spiels wird diese Szene oder Klasse nur als Abstraktion und übergeordnete Klasse verwendet.

Damit KI verstehen kann, welche Aktionen möglich sind, müssen diese definiert werden. In der neuesten Version des Produkts können Gegner Aktionen aus einer Liste in Code 3.1 ausführen.

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 in Code 3.2 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.



Abbildung 3.8: Die Gegner-Knoten-Strukture

Code 3.2: 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             else:
24                 velocity = Vector2.ZERO
25     move_and_slide()
26

```

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-Lernen 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 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. 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 einer aus Q-Tabelle und den Zustand s .
 - ii. Führe die Aktion aus, wechsle in den Zustand s' und erhalte eine unmittelbare Belohnung r .
 - iii. Aktualisiere den Wert von $Q(s, a)$ gemäss 2.1 .
 - iv. $s \leftarrow s'$.
 - (c) *Evaluation:* Wenn Tot, berechne eigenes Fitness und speichere es.
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 auf einige oder alle Elternteile und/oder Nachkommen angewendet.
6. *Ersatz:* Ersatz erfolgt de facto durch die Generationsstrategie. Die Eltern sterben während ihren Leben 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-1ad-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-1ad-1"

Der Zustand wird durch eine Folge aus Buchstaben und Zahlen dargestellt. Zunächst kommen ein paar Buchstaben, dann folgt eine Zahl. So war es auch programmiert. 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 3.3 dargestellt.

Code 3.3: Wiedergabe der Funktion „get_state“.

```
1     return "wt{wt}d{d}a{a}bd{bd}ba{ba}ad{ad}".format({
2         "wt": weapon_type, "d": dist, "a": angle, "bd": bullet_dist, "ba":
3             bullet_angle, "ad": dist_ally, "aa": angle_ally
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. Nun zur Liste. 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.
- „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, um das Ausweichen von Projektilen durch die Spieler zu ermöglichen.
- „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.
- „ad“ – Die Distanz zur nächsten Gegner. Die möglichen Werte sind dieselben wie bei der Distanz zur Kugel. Dieser Parameter ist wichtig, um eine Mehr-Gegner-Strategie zu entwickeln.
- „aa“ – der Winkel zur nächsten Gegner. Die möglichen Werte sind dieselben wie beim Winkel zur Kugel. Dieser Parameter hilft eine Mehr-Gegner-Strategie entwickeln.

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, 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 3.2 erläutert. Laut Q-Learning muss die Aktion nach der Ausführung belohnt werden. Dies wird für jede Aktion mit einfachen Integer-Werten definiert. Die genauen Zahlen sind in Code 3.4 aufgeführt.

Code 3.4: Die Belohnungen

```

1  var REWARDS := {
2      "TOOK_DAMAGE": -2.0,
3      "TIME_ALIVE": 0.05,
4      "HIT_PLAYER": 14.0,
5      "RETREATED": -0.2,
6      "WASTED_MOVEMENT": -0.1,
7      "MOVED_CLOSER": 0.05,
8      "MISSED": -0.2,
9      "DIED": -7,
10     "STUCK": -1
11 }
12

```

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 und wie lange sie gelebt haben. Diese Parameter werden nun verwendet, um die Fitness der Gegner zu berechnen. Dies erfolgt gemäß Gleichung 3.1.

$$f(L, D) = 5D + 0.2L \quad (3.1)$$

Dabei bezeichnet D den verursachten Schaden am Spieler und L die Überlebenszeit in Sekunden. Der letzte implementierte Schritt ist die *selektion*. Dabei werden die Gegner anhand der berechneten Fitness ausgewählt. Es wurde ein globaler Rekombinationsprozess gewählt, bei dem alle Gegner zur Erzeugung eines neuen Gegners beitragen. Der Einfluss jedes Elternteils auf die Eigenschaften des Nachkommens ist proportional zu seiner individuellen Fitness. Dieser Schritt findet am Ende der Welle statt. Danach beginnt die neue Welle, diesmal jedoch mit Gegnern, die aus der letzten Welle ausgewählt 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 mein Spiel das gleiche Schicksal erleiden 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 Entfernung zum Spieler. Die ersten beiden Metriken werden mit der Fitnessfunktion (siehe Gleichung 3.1) kombiniert, um einen Fitnesswert zu ermitteln.

Test der Q-Learning Effizienz

Jeder Gegner beginnt in der ersten Welle mit einer leeren Q-Tabelle. Alle tragen die gleichen Waffen, die Spieler verhalten sich in grundlegenden und Q-Lernfällen ungefähr gleich, und das Experiment wird mehrmals wiederholt, um statistische Störfaktoren zu eliminieren. Während der Welle werden in festen Zeitintervallen Zustandsinformationen und Leistungsmetriken aufgezeichnet.

Metriken Die Fitness wird als eine der Testmetriken verwendet, für die wiederum die Fitnessfunktion zum Einsatz kommt. Die kumulative Belohnung und die Anzahl jeder Aktion werden ebenfalls gemessen. **Basisfall**. Um die Effizienz der Algorithmen quantitativ messbar zu machen, wurde ein Basisfall definiert. In diesem Fall wählen die Gegner ihre Aktionen zufällig, ohne dazuzulernen.

Test der genetischen Algorithmen

Dieser Test untersucht die Verbesserung der Gegner durch Wellen unter Verwendung eines genetischen Algorithmus. Zu diesem Zweck wird ein **Basisfall** erstellt, in dem Q-Lernen ohne Q-Tabellenübertragung zwischen den Wellen stattfindet.

Metriken Die Fitness wird erneut verwendet, jedoch nicht in Zeitintervallen während der Welle gemessen, sondern nach Ende der Wellen. Somit wird die Wellenzahl statt Zeit als unabhängige Variable für Diagramme verwendet.

Kapitel 4

Ergebnisse und Diskussion

4.1 Ergebnisse

4.1.1 Ziel 1 und 3: Spiel und Spass

4.1.2 Ziel 2: Lernende KI

4.2 Fazit

4.3 Künftige Arbeit

4.4 Reflexion

Kapitel 5

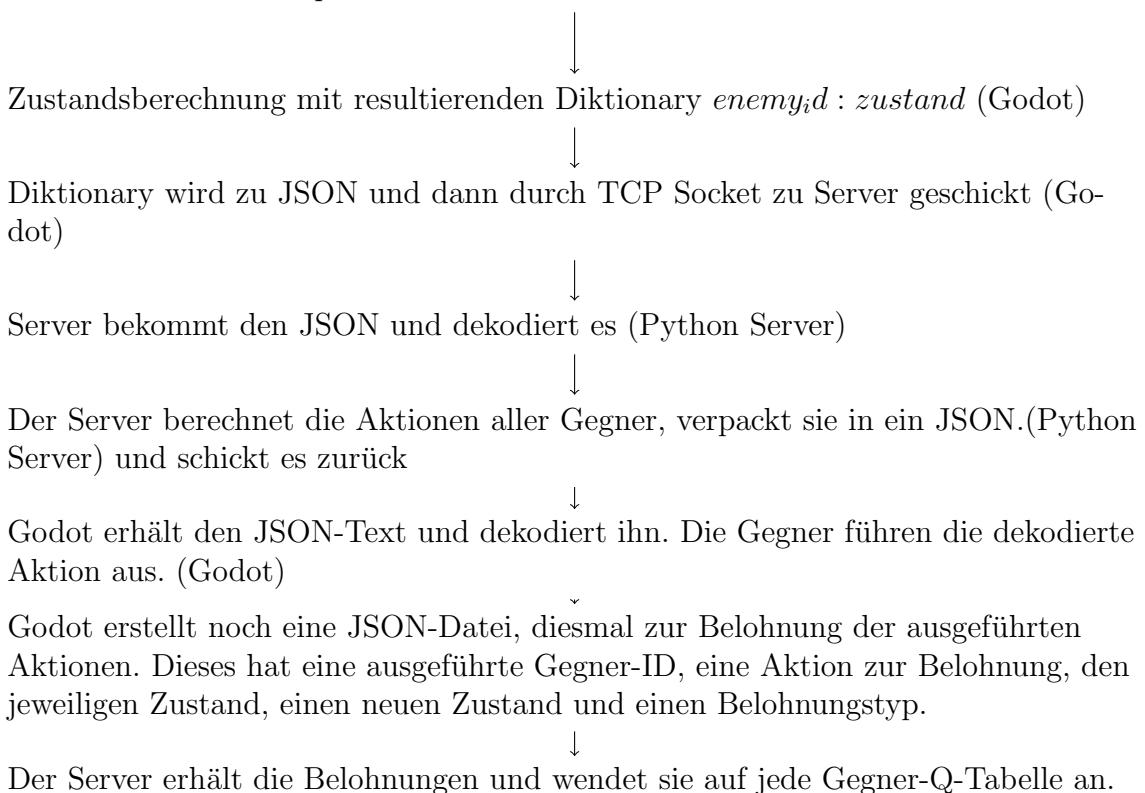
Zusatz

Python Server und Sockets

In der Praxis wurde entschieden, 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.



Spielarchitektur

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

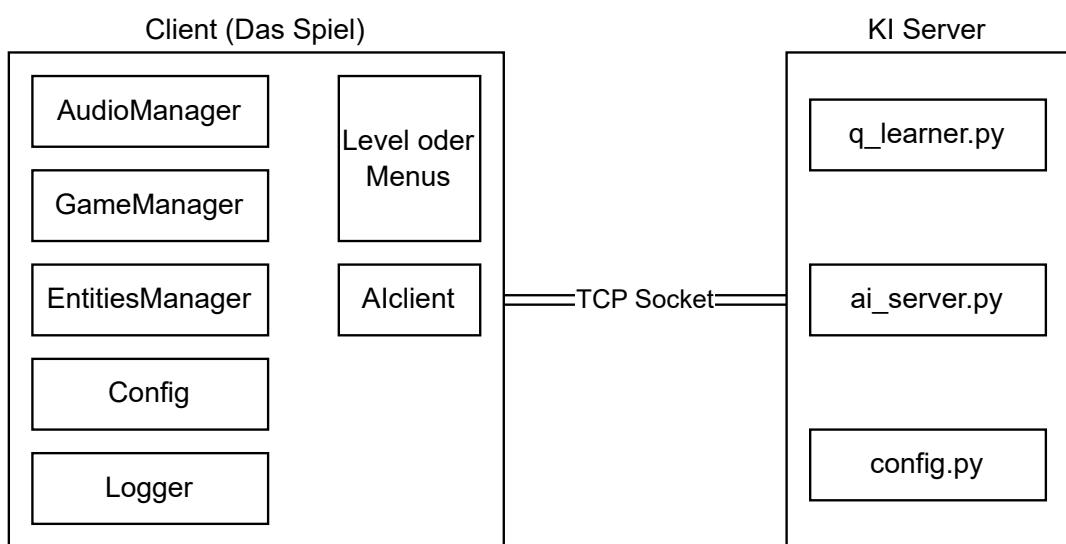


Abbildung 5.1: Grobe Architektur des Spiels

Literaturverzeichnis

- [1] poncle. Vampire survivors on steam, 2022. Zugriff am 5. August 2025. URL: https://store.steampowered.com/app/1794680/Vampire_Survivors/.
- [2] MegaCrit. Slay the spire on steam, 2019. Zugriff am 5. August 2025. URL: https://store.steampowered.com/app/646570/Slay_the_Spire/.
- [3] Dodge Roll. Enter the gungeon on steam, 2016. Zugriff am 5. August 2025. URL: https://store.steampowered.com/app/311690/Enter_the_Gungeon/.
- [4] Creative Assembly. Alien: Isolation on steam, 2014. Zugriff am 8. August 2025. URL: https://store.steampowered.com/app/214490/Alien_Isolation/.
- [5] Monolith Productions. F.e.a.r. on steam, 2005. Zugriff am 8. August 2025. URL: <https://store.steampowered.com/app/21090/FEAR/>.
- [6] 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/.
- [7] Georgios N Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018. doi:[10.1007/978-3-319-63519-4](https://doi.org/10.1007/978-3-319-63519-4).
- [8] Unity Technologies. *Unity Real-Time Development Platform*. Unity Technologies, 2025. Version 2022.3 LTS. URL: <https://unity.com/>.
- [9] Godot Engine contributors. *Godot Engine*. Godot Engine, 2025. Version 4.x. URL: <https://godotengine.org/>.
- [10] Epic Games. *Unreal Engine*. Epic Games, 2025. Version 5.x. URL: <https://www.unrealengine.com/>.